Analysis of Stateflow models using mCRL2

by
R.C. van Cann

Supervisors:
J.F. Groote (TU/e)
J. Hooman (ESI)

Eindhoven, June 2008
Preface

After graduating for my bachelor degree in Electrical Engineering, I started the master programme Computer Science and Engineering at Eindhoven University of Technology. During this study I discovered that I did not want to distance myself too much from Electrical Engineering. When I attended the lectures of the course “Requirement Analysis, Design and Verification” I became enthusiastic about the verification of systems, both digital electronics and software. At this point I decided that I wanted to do my master project in the field of system verification.

After getting in contact with Jan Friso and Jozef, we decided that my master project would be about the verification of Stateflow models using the modelling language mCRL2. During the execution of the project, I learned a lot about the semantics of Stateflow and the creation of mCRL2 models. I also had the chance to gain experience at working on research oriented projects.

In the future I want to work at the development of electronics and software for embedded systems, instead of performing research to new techniques. I hope that the knowledge gained during the master project about mCRL2 is helpful for the analysis of those systems.

Remko van Cann
Breda, June 2008

Acknowledgements  I would like to thank the following persons: the supervisors, Jan Friso Groote for his help and patience and Jozef Hooman for giving me the opportunity to work at my master project at the Embedded Systems Institute. I also want to thank Jeroen van der Wulp for helping me out whenever I encountered a problem with the mCRL2 toolset. Finally I want to thank my parents for their support and for financing my studies during the last ten years.
Summary

At the Embedded Systems Institute, research is performed on the run-time verification of embedded systems, using a model that runs in parallel with the system. These models are currently made using Stateflow. In order to correctly verify the systems, the models must behave correctly. During this master project, a technique to verify the Stateflow models using the mCRL2 modelling language is developed.

First a conversion from Stateflow models to mCRL2 models is made. This conversion is able to convert all basic building blocks from Stateflow models to mCRL2 models. Some small parts of the existing Stateflow models have to be manually converted. During the development of the conversion technique, a case study of the sound behaviour of a TV was converted. The converted model is used to verify the requirements for the sound behaviour model using model checking.

After the verification of the converted model, the mCRL2 model is extended to verify the model using a range of values for a constant data object. This extension to the mCRL2 conversion allows the exploration of the limits of the initialised values for the model.

After the construction of the conversion technique, another case study of the TV behaviour is converted and partially verified. This case study is significantly larger than the sound behaviour model and it demonstrates the possibility to apply the techniques on bigger, real-life Stateflow models.

Although the verification of the larger TV behaviour model is much harder to complete than the sound behaviour model, it is possible to verify both. More knowledge of the theory and the mCRL2 toolset is needed to generate the state space of the TV model.

Another observation is that the construction of modal formulas for the converted Stateflow models is not always straightforward. The requirements that were verified on the models assumed certain states in the systems, but these states do not automatically correspond completely to states in Stateflow. This is caused by the fact that there are situations where a group of Stateflow states are all inactive. When the conversion technique is used on regular basis, it might be useful if the construction of modal formulas for Stateflow models is explored.
## Contents

1 Introduction .......................... 1  
  1.1 Embedded Systems Institute ........ 1  
  1.2 Project description .................. 2  
  1.3 Document overview .................. 2  

2 Modelling and verification ........... 3  
  2.1 Introduction ......................... 3  
  2.2 The mCRL2 language .................. 3  
  2.2.1 Data types ........................ 3  
  2.2.2 Actions and multi-actions ........ 6  
  2.2.3 Sequential processes .............. 7  
  2.2.4 Parallel processes ............... 8  
  2.2.5 Timed processes ................... 11  
  2.3 mCRL2 toolset ........................ 11  
  2.3.1 Types of files ..................... 11  
  2.3.2 Tools from the mCRL2 toolset .... 13  
  2.3.3 3rd party toolset: CADP .......... 18  
  2.4 Model checking ....................... 18  
  2.4.1 Regular Alternation-free $\mu$-calculus 18  
  2.4.2 Classification of properties ...... 21  
  2.4.3 Evaluator ......................... 22  

3 Case study: on-screen display ...... 23  
  3.1 Introduction ......................... 23  
  3.1.1 On screen display ................. 23  
  3.2 System description ................... 24  
  3.2.1 Status OSD ......................... 25  
  3.2.2 Preset OSD ......................... 25  
  3.3 mCRL2 specification .................. 26  
  3.3.1 Data types ........................ 26  
  3.3.2 Actions ............................ 26  
  3.3.3 Processes .......................... 27  
  3.3.4 Initiation .......................... 30  
  3.4 Evaluating the state space ........ 31  
  3.4.1 Statespace reduction ............... 31  
  3.4.2 Ltsgraph ............................ 32  
  3.5 Formal requirements and verification 32  
  3.5.1 Creating patterns for modal formulas 32
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2 Verifying formal requirements</td>
</tr>
<tr>
<td>3.5.3 Feedback to the specification and the requirements</td>
</tr>
<tr>
<td>3.6 Conclusion</td>
</tr>
<tr>
<td>4 Stateflow</td>
</tr>
<tr>
<td>4.1 Concepts and notation</td>
</tr>
<tr>
<td>4.1.1 Stateflow charts</td>
</tr>
<tr>
<td>4.1.2 Actions and action sequences</td>
</tr>
<tr>
<td>4.1.3 States</td>
</tr>
<tr>
<td>4.1.4 Transition and junction objects</td>
</tr>
<tr>
<td>4.1.5 Other objects</td>
</tr>
<tr>
<td>4.2 Example 2: sound behaviour</td>
</tr>
<tr>
<td>4.2.1 The system and the environment</td>
</tr>
<tr>
<td>4.2.2 System description</td>
</tr>
<tr>
<td>4.3 Stateflow semantics</td>
</tr>
<tr>
<td>4.3.1 Executing an event</td>
</tr>
<tr>
<td>4.3.2 Executing a chart</td>
</tr>
<tr>
<td>4.3.3 Executing a state</td>
</tr>
<tr>
<td>4.3.4 Executing a set of flow graphs</td>
</tr>
<tr>
<td>4.3.5 Entering a state</td>
</tr>
<tr>
<td>4.3.6 Exiting a state</td>
</tr>
<tr>
<td>5 Stateflow conversion to mCRL2</td>
</tr>
<tr>
<td>5.1 Introduction</td>
</tr>
<tr>
<td>5.2 General structure</td>
</tr>
<tr>
<td>5.2.1 Stateflow structure</td>
</tr>
<tr>
<td>5.2.2 mCRL2 conversion structure</td>
</tr>
<tr>
<td>5.3 Data objects</td>
</tr>
<tr>
<td>5.3.1 State information</td>
</tr>
<tr>
<td>5.4 Events</td>
</tr>
<tr>
<td>5.5 Performing actions and action sequences</td>
</tr>
<tr>
<td>5.5.1 Event broadcasts</td>
</tr>
<tr>
<td>5.5.2 Data manipulation</td>
</tr>
<tr>
<td>5.5.3 Matlab functions</td>
</tr>
<tr>
<td>5.5.4 Action sequences</td>
</tr>
<tr>
<td>5.6 The Stateflow process</td>
</tr>
<tr>
<td>5.7 Executing an active state</td>
</tr>
<tr>
<td>5.7.1 Executing a set of flow graphs</td>
</tr>
<tr>
<td>5.7.2 State execution</td>
</tr>
<tr>
<td>5.7.3 Example</td>
</tr>
<tr>
<td>5.8 Entering a state</td>
</tr>
<tr>
<td>5.8.1 mCRL2 conversion</td>
</tr>
<tr>
<td>5.8.2 Example</td>
</tr>
<tr>
<td>5.9 Exiting a state</td>
</tr>
<tr>
<td>5.9.1 Example exit process</td>
</tr>
<tr>
<td>5.10 mCRL2 initialisation</td>
</tr>
<tr>
<td>5.11 Case study: sound behaviour</td>
</tr>
<tr>
<td>5.11.1 Custom conversions to mCRL2</td>
</tr>
<tr>
<td>5.12 Conclusions</td>
</tr>
</tbody>
</table>
CONTENTS

6 Case study: sound behaviour
   6.1 Introduction ................................................. 75
   6.2 System requirements ....................................... 75
      6.2.1 System overview .................................. 75
      6.2.2 Requirements ..................................... 76
   6.3 Verification ................................................ 77
      6.3.1 Global requirements verification .................. 77
      6.3.2 Unmuted requirements verification ............... 78
      6.3.3 Muted requirements verification ................. 83
   6.4 DisplayMin analyses ....................................... 85
   6.5 Evaluating data object values ............................ 86
   6.6 Conclusions ............................................... 87

7 Case study: TV behaviour
   7.1 Introduction ............................................... 89
   7.2 Stateflow implementation ................................. 89
      7.2.1 TV environment .................................. 89
      7.2.2 Top level states .................................. 90
      7.2.3 Video control ................................... 92
      7.2.4 Audio control ................................... 92
      7.2.5 OSD control ...................................... 93
   7.3 mCRL2 implementation ...................................... 95
      7.3.1 Custom conversions ................................ 95
      7.3.2 Optimising the model ............................... 95
   7.4 System requirements ....................................... 96
      7.4.1 System overview .................................. 97
      7.4.2 Requirements ..................................... 97
   7.5 Verification ................................................ 98
      7.5.1 Global requirements verification .................. 98
      7.5.2 Unmuted requirements verification ............... 99
      7.5.3 Muted requirements verification ................. 103
   7.6 Conclusions ............................................... 106
      7.6.1 Final state space .................................. 106

8 Concluding remarks
   8.1 Introduction ............................................... 109
   8.2 Converting Stateflow models to mCRL2 .................... 109
      8.2.1 Creating a complete conversion .................... 109
      8.2.2 Automating the conversion ......................... 109
      8.2.3 Evaluating different settings ...................... 110
   8.3 Verifying the mCRL2 models ............................... 110
      8.3.1 Standardised generation of modal formulas .......... 110
      8.3.2 Modal formula generation tool ..................... 110
      8.3.3 Evaluating model checker feedback ............... 111
   8.4 mCRL2 toolset ............................................. 111
      8.4.1 Warnings ......................................... 111

A mCRL2 operators

B On-screen display
C mCRL2 specification of example 2
D mCRL2 specification of example 3
Chapter 1

Introduction

1.1 Embedded Systems Institute

The goal of the Embedded System Institute (ESI) is to perform research on high-tech embedded system engineering. This research is performed by combining knowledge from the industry and academia, leading to methods and tools that can be applied during development, testing and the run-time of high-tech embedded systems.

There are a number of research projects at the ESI. These projects cover different subjects like system engineering, performance, evolvability and reliability. At different research projects at the ESI, realistic case studies from industry are used. The projects result in a proof of concept.

Trader project: reliability

With the increase of complexity and the decrease of product life cycles in embedded systems, it becomes harder to maintain reliability of the systems. The objectives of the Trader project are to increase the reliability, focusing on high-tech consumer applications. An important part of the project is minimising the system errors that are exposed to the user. The project should result into methods and tools that increase the reliability of embedded systems.

Run-time awareness

Part of the Trader projects focuses on the run-time awareness of embedded systems. When detecting errors during the run-time of systems, the system can respond to the error. This is done in three steps: detecting an error, locating the source of the error and reacting to the error in order to prevent wrong feedback to the users.

One way to detecting an error is model-based error detection. Model-based error detection runs a model of the system, parallel to the system itself. When the model and the system shows different outputs, an error has occurred. In order to gain a reliable results from model based error detection, the models have to be correct. If the models are not correct, non-existing errors can be detected.

The Trader project uses a digital TV system as case study.
1.2 Project description

There are currently models of the TV system available at the ESI. These models are created using Stateflow (see chapter 4 for more details about Stateflow). Currently, the models are simulated to verify the correctness. Using simulation, the correctness of the models cannot be guaranteed.

Goals

The goal of this master project is to check if the mCRL2 modelling language can be used to verify Stateflow models. If the models can be described in mCRL2, model checking as well as the available mCRL2 tools can be applied to the model. During the project, techniques used to verify Stateflow models with the mCRL2 modelling language are applied to the existing TV models.

Besides the verification of the Stateflow models, feedback about the mCRL2 toolset can be used to improve the toolset.

1.3 Document overview

First the mCRL2 language and toolbox are described in chapter 2. This knowledge is applied to a small case study in chapter 3. The case study shows how a design is implemented in mCRL2 and verified using modal formulas.

After the introduction of mCRL2, the syntax and semantics of Stateflow models are explained in chapter 4. This chapter also introduces the second case study, which is used to show the conversion techniques from Stateflow models to mCRL2 models in chapter 5. Finally the second case study is verified in chapter 6.

Being able to convert a Stateflow model to mCRL2, a third case study is introduced, converted and partially verified in chapter 7. This third and last case study is significantly bigger than the previous case studies, showing the possibility of the conversion to be used on larger models.

Finally chapter 8 highlights the problems occurred during the project and suggests ideas for future work.
Chapter 2

Modelling and verification

2.1 Introduction

Before modelling and verifying a system, an introduction of mCRL2 and the toolset is given. First the mCRL2 syntax is described, followed by an description of the different tools available. Next creating modal formulas and verifying these formulas on a mCRL2 specification is discussed.

2.2 The mCRL2 language

mCRL2 stands for milli Common Representation Language 2 and it is the successor of µCRL [2]. mCRL2 can be used to specify the behaviour of distributed systems.

This section will explain the basics of the mCRL2 syntax and it will give an idea of the possibilities of mCRL2 and allow you to understand the mCRL2 specifications in this report. The complete syntax is described in “The Formal Specification Language mCRL2” [3].

2.2.1 Data types

mCRL2 has predefined datatypes and type constructors as well as a mechanism to define new types. Before the standard types are explained, the mechanism to define new sorts is explained.

**basic datatype definition mechanism**

**sort & cons** The keyword **sort** is used to declare sorts. Sorts are non-empty sets with data elements. the keyword **cons** can be used to define constructor functions on a sort that denote exactly all elements in the sort. To give an example, the sort of all natural numbers **Nat** can be defined by:

```ml
sort Nat;
cons zero : Nat;
successor : Nat → Nat;
```
The sort $\text{Nat}$ has an element $\text{zero}$ and successors of a natural number $\text{successor}$. The $\text{successor}$ function can be applied to another $\text{successor}$ or to $\text{zero}$. The elements created by the constructors are not necessarily different.

Sorts do not have to contain any constructor function (in case the data elements in a sort do not need to be denotable by a term), it cannot be empty either. If a sort has a collection of constructor functions that cannot be combined to create at least one element the sort is illegal.

**map, var and eqn** The keyword `map` can be used to declare functions.

```plaintext
map plus : Nat × Nat;
```

declares a function `plus` with two parameters of sort `Nat`. The behaviour of this function is declared with an equation using the keyword `eqn`. Variables can be used using the keyword `var`.

```plaintext
var n, m : Nat;
```

defines the variables $n$ and $m$ of type `Nat`, and

```plaintext
eqn plus(n, zero) = n;
plus(n, successor(m)) = successor(plus(n, m));
```

defines the equations for the `plus` function on natural numbers. They state that $n + 0 = n$ and that $n + (m + 1) = (n + m) + 1$ always hold. The variables $n$ and $m$ can take any value of their sort while using these equations.

Equations do not have to be complete. Not every possible situation for every function has to be defined. If there is a situation for which you do not know the outcome of the function, you leave it undefined and a calculation with the specification assumes it can hold any possible outcome.

Equation may also have conditions. The equation

```plaintext
eqn m ≈ zero → plus(n, m) = n;
```

defines the same as

```plaintext
eqn plus(n, zero) = n;
```

would.

As mentioned before, data elements of a sort are not necessarily different. The only sort where the elements are considered different are booleans. In order to prove that two elements of sort $S$ are different from each other, "reductio at absurdum" is used. For example the function `less` defined as:

```plaintext
map less : Nat × Nat → Bool;
var n, m : Nat;
eqn less(n, zero) = false;
less(zero, successor(m)) = true;
less(successor(n), successor(m)) = less(n, m);
```

holds `false` for `less(zero, zero)` and `true` for `less(zero, successor(zero))` and therefore `zero` and `successor(zero)` are different elements because true and false are considered different elements. If the elements are the same they would be interchangeable as parameters in all function mappings.
2.2. THE MCRL2 LANGUAGE

Predefined datatypes and type constructors

Booleans and numbers  Within μCRL booleans had to be defined in every specification. In mCRL2 the booleans as well as positive natural numbers, natural numbers, integers and reals are predefined. The standard operators on booleans and numbers are also predefined. An overview of the available operators is given in tables A.2 and A.1 of appendix A.

While working with numbers implicit type conversion is applied. Positive numbers can be used as natural numbers, natural numbers as integers, which in turn can be used as real numbers.

Lists  A list with elements of sort $A$ are declared by the sort expression $List(A)$. All lists consist of the constructors $[\ ]$, the empty list, and $\triangleleft$, putting an element of sort $A$ in front of the list. All standard operators on list are also mapped on lists. An overview of the operators on lists is given in appendix A table A.3.

Sets and bags  A set or a bag with elements of sort $A$ are declared by the sort expression $Set(A)$ or $Bag(A)$. The empty set and the empty bag are represented by the empty enumeration $\{\}$. Sets can be declared by $\{a_0, \ldots, a_n\}$ where $a_i \in A$ or by $\{x : A | P(x)\}$ where the set consist of all elements $x$ of sort $A$ for which predicate $P(x)$ holds. $P(x)$ is an expression of sort $Bool$.

Bags can be declared by $\{a_0 : m_0, \ldots, a_n : m_n\}$ where $a_i \in A$ and $m_i \in Nat$. Another way to declare bags is $\{x : A | f(x)\}$ where the bag in which all elements $x$ of sort $A$ occurs $f(x)$ holds. $f(x)$ is an expression of sort $Nat$.

All operators on sets and bags are listed in table A.4.

Function types  Another way of creating sorts is the use of function types. In the next example $F$ is the sort of functions from natural numbers to natural numbers, $G$ is the sort of functions from real numbers and natural numbers to real numbers. Function types are right associative and therefore $R$ is the sort of functions from real numbers to functions from $F$ to $List(G)$.

$$\begin{align*}
\text{sort} & \quad F = Nat \rightarrow Nat \\
& \quad G = Real \times Nat \rightarrow Real \\
& \quad R = Real \rightarrow F \rightarrow List(G)
\end{align*}$$

Functions for these sorts can be made using lambda abstraction and application. $\lambda n : Nat. n^2$ denotes a function from Nat to Nat that yields for each argument $n$ its square. Using round brackets the function can be applied to a argument, i.e. $(\lambda n : Nat. n^2)(4)$ equals 16.

The operators on function types are listed in table A.5 in appendix A.

Structured data types  Structured data types, also called functional or recursive types, consists of a number of constructors. These constructors characterise different elements.

$$\begin{align*}
\text{sort} & \quad S = \text{struct} \quad c_1 | \ldots | c_n
\end{align*}$$
denotes the structure \( S \) with \( n \) constructors.

Constructors may be extended with recognisers. Recognisers are functions mappings from \( S \) to Booleans and they yield true if they are applied to the constructor they belong to.

\[
\text{sort } S = \text{struct } c_1|\ldots|c_n
\]
denotes the same structure \( S \) with recognisers.

Constructors may depend on sorts. They may even depend on themself to create recursive data types such as lists and trees.

\[
\text{sort } S = \text{struct } c_1(A_1, \ldots, A_k)|\ldots|c_n(B_1, \ldots, B_m)
\]
declares the structure \( S \) where all constructors depends on a number of sorts.

When constructors are depending on sorts, projection functions can be assigned to these sorts.

\[
\text{sort } S = \text{struct } c(\text{get}_1 : A_1, \ldots, \text{get}_k : A_k)
\]
declares the structure \( S \) where the projection functions \( \text{get}_i \) are assigned to the sorts \( A_i \) where \( 1 \leq i \leq k \) holds. When the projection function is applied to the constructor it belongs to, you’ll get the element of the sort to which the projection function was declared, i.e. \( \text{get}_i(c(a_1, \ldots, a_k)) = a_i \).

The operators on structured types are listed in table A.6.

### 2.2.2 Actions and multi-actions

Since behaviour of systems will be described in terms of actions and interactions, the basic elements of the mCRL2 language are actions.

**Actions**

Actions are the basic elements when describing processes. All actions happen atomically in time. This means that actions are always executed interleaved unless indicated otherwise. Actions can be declared using the \texttt{act} keyword.

\[
\text{act } a, b;
\]
\[
c : \text{Nat};
\]
\[
d : \text{Bool} \times \text{Nat};
\]

In this example \( a \) and \( b \) are actions, \( c \) is an action with a data parameter of type \( \text{Nat} \) and \( d \) is an action with data parameters of types \( \text{Bool} \) and \( \text{Nat} \).

**Multi-actions**

Multi-actions represent a collection of actions that occur at the same time. Multi-actions are constructed according to the following BNF:

\[
\alpha ::= \tau \mid a \mid a(\overrightarrow{\text{d}}) \mid (\alpha \mid \beta)
\]

where \( a \) denotes an action and \( \overrightarrow{\text{d}} \) a vector of data parameters. \( \tau \) is an empty multi-action which contains no actions, it is also called an internal action. \( \alpha \mid \beta \) consists of two multi-actions that are executed simultaneously.
2.2.3 Sequential processes

In order to describe systems in terms of actions and interactions, operators are used to combine multi-actions into processes. The basic process consists of only one multi-action or a deadlock. New processes can be combined using operators on processes. For example, the sequential composition operator $p.q$ indicates that process $q$ is executed after process $p$ has terminated. A list of all possible basic processes and process operators is given below.

- $\alpha$ A simple process consists of only one multi-action. This process can be combined with other processes to describe complex behaviour.

- $\delta$ This is a special process called deadlock. This process cannot be executed and will never terminate. Although the deadlock is almost never used in a process to describe the behaviour of a system, it can be generated by other operators on processes.

- $p \cdot q$ This is the sequential composition operator. This means that process $q$ is executed after process $p$ terminates. The sequential composition terminates when $q$ terminates.

- $p + q$ This is the choice operator. This means that either process $p$ or $q$ can be chosen. The choice between the processes is made by the first action that is executed in either process $p$ or $q$.

- $c \rightarrow p \diamond q$ This is the conditional operator. The conditional operator checks the value of boolean $c$ and executes either process $p$ or $q$. $p$ is executed if $c$ is true, else $q$ is executed.

- $\sum_{d: D} p(d)$ This is the sum operator. The sum operator is a generalisation of the choice operator. A choice can be made between all possible instances of process $p$. All variables $d$ of sort $D$ that occur in one instance of process $p$ are replaced with the same element of sort $D$. This allows process $p$ to be executed with any of the possible elements of sort $D$ substituted for $d$.

Recursive processes

Recursive specifications are introduced by the means of process variables. A process variable is a identifier with a process assigned to it. Process variables are often just called processes. Processes are declared using the keyword proc.

```plaintext
act set, alarm, reset;

proc P = set.Q;
    Q = reset.P + alarm.Q;
```

This specification describes the behaviour of a simple alarm clock. Process $P$ describes the behaviour of the clock when it is turned off. The only possible
action in $P$ is to set the alarm. When the alarm is set process $Q$ is started. Process $Q$ has two possible actions, reset and alarm. This clock keeps executing the action alarm (starting process $Q$ again) until the reset action is executed and process $P$ is started.

If a set of equations defining a process has a single process variable at the left side of the equality sign it is considered a recursive specification. A recursive specification is considered a guarded recursive specification if every occurrence of a defined process variables at the right side of the equality sign is preceded by at least one action.

Process variables may contain parameters that can be used to keep track of the current status. A process variable with parameters is a function from the parameters to a process. In the next example $R$ is a function from integers to processes.

\[
\text{act} \quad \text{up, down}; \\
\text{read} : \text{Int}; \\
\text{proc} \quad R(n : \text{Int}) = \\
\quad \text{up}.R(n + 1) + \\
\quad \text{down}.R(n - 1) + \\
\quad \text{read}(n);
\]

$R$ is a counter that can count up, down or return the current value. $R(5)$ is a process as shown below.

\[
R(5) = \text{up}.R(6) + \text{down}.R(4) + \text{read}(5)
\]

The parameters of a process may not be of sort process themself.

**Initiation**

The keyword \texttt{init} can be used to define the initial state of a process. The initial process for the counter $R$ can be:

\[
\text{init} \quad R(0);
\]

### 2.2.4 Parallel processes

With sequential processes it is possible to describe processes that can interact with their environment. The next step is to execute multiple processes in parallel and force interactions between these processes. To create parallel processes the following six operators on processes are introduced.

\[
p \parallel q \quad \text{This is the parallel composition of process } p \text{ and } q. \text{ The parallel composition allows you to execute actions from process } p \text{ and } q \text{ interleaved or simultaneously. The parallel composition terminates successfully when process } p \text{ and } q \text{ both terminate successfully.}
\]

Two processes consisting of only one action in parallel behaves as:

\[
a \parallel b = a.b + b.a + a|b
\]

The actions within one process have to be executed in the same order as normal.
2.2. THE MCRL2 LANGUAGE

Γ\textsubscript{C}p This is the communication operator. The set \( C \) contains allowed communications of the form \( a_0|...|a_n \to c \) with \( n \geq 1 \) and \( a_i \) and \( c \) as action names. In all multi-actions containing actions \( a_0 \) to \( a_n \), these actions are replaced with the action \( c \). The data parameters have to be the same for all actions in an allowed communication. This operator only replaces the parts where the communication is valid, it doesn’t force the communication to happen. Executions of the original independent actions \( a_0 \) to \( a_n \) are still possible.

∇\textsubscript{V}p This is the allow operator. The set \( V \) contains multi-actions that are allowed to be executed in process \( p \). Set \( V \) cannot include the internal action \( \tau \), but \( \tau \) is still allowed to execute. All actions in \( p \) not occurring in \( V \) are replaced with a deadlock \( \delta \). This operator can be used to enforce a communication between two actions by not allowing the actions to happen individually.

∂\textsubscript{B}p This is the block or encapsulation operator. The set \( B \) contains action names that are not allowed to execute in process \( p \). All these actions occurring in process \( p \) are replaced by a deadlock \( \delta \). Any multi-action containing one of the blocked actions is also blocked.

ρ\textsubscript{R}p This is the rename operator. The set \( R \) contains renamings of the form \( a \to b \). All occurrences of action \( a \) in process \( p \) are replaced with the action \( b \). To avoid unclarities, every action name may only occur once on the left hand side of the arrow in set \( R \). All renamings are applied simultaneously.

τ\textsubscript{I}p This is the hide operator. The set \( I \) contains action names that should be hidden in process \( p \). All these actions occurring in process \( p \) are replaced by the hidden action \( \tau \). Any multi-action containing one of the hidden actions is executed without this action.

Communication between processes Because communication between processes is one of the most important parts of the mCRL2 specification a small example is given. Consider the mCRL2 specification:

\begin{verbatim}
act sMessage, rMessage, cMessage;
proc P = sMessage;
proc Q = rMessage;
init P || Q;
\end{verbatim}

In this mCRL2 specification are three actions, \( sMessage \), \( rMessage \) and \( cMessage \). Process \( P \) sends a message by executing action \( sMessage \) and process \( Q \) receives a message by executing action \( rMessage \). After the processes have sent or received a message, they terminate successfully. These processes are executed in parallel.

The goal of this example is to enforce communication between these processes. Each time process \( P \) sends a message process \( Q \) has to receive that
message. Currently the specification allows the send and receive actions to be executed either sequentially or simultaneously. This is shown in figure 2.1.

Figure 2.1: Parallel processes

State 0 is the initial state. Now the actions can be executed either sequentially or simultaneously ending in state 3 where the Terminate action indicates successful termination. First we rename all the transitions where the communication between the processes behaves correctly. This is done by extending the initiation of the specification with the communication operator:

\[
\text{init } \Gamma_{\{sMessage|\text{rMessage} \rightarrow cMessage\}}(P \parallel Q);
\]

Now all multi-actions \(sMessage|\text{rMessage}\) are replaced with action \(cMessage\) leading to the new situation in figure 2.2.

Figure 2.2: Communication between processes

The new specification behaves like the old specification, but now we are able to differentiate between the correct and incorrect executions of the send and receive actions. By blocking all incorrect executions of actions \(sMessage\) and \(rMessage\) the specification is forced to execute the send and receive actions simultaneously. The following initiation leads to the situation of figure 2.3.
2.3. MCRL2 TOOLSET

\[
\text{init } \partial\{sMessage,rMessage\} (\Gamma\{sMessage|rMessage\rightarrow cMessage\} (P \parallel Q) );
\]

Figure 2.3: Forced communication between processes

In this specification send actions are indicated with a name starting with a \(s\), receive actions starts with a \(r\) and the communication action starts with a \(c\). This naming convention is also used during the mCRL2 specifications in this report.

2.2.5 Timed processes

In mCRL2 it is also possible to include timing in the specifications. Including timing in a specification will lead to giant state spaces that are very hard to verify. Therefore this is outside the scope of this section and will not be discussed or used during this project.

2.3 mCRL2 toolset

In figure 2.4 an overview of the mCRL2 toolset is given. In this figure the ovals represent types of files, the rectangles represent types of actions and the arrows indicate the types of files that can be used as input or output.

Section 2.3.1 describes the different types of files used by the mCRL2 toolset. After that the different tools available will be described.

2.3.1 Types of files

This section describes the different types of files used by the mCRL2 toolset.

mCRL2 specifications

The mCRL2 specifications are described in section 2.2. mCRL2 specifications can be used to generate linear processes.

Modal formula

A modal formula is a formal description of requirements that can be checked on specifications. In section 2.4.1 the construction of modal formulas is described.
CHAPTER 2. MODELLING AND VERIFICATION

Linear process specifications

A LPS is a restricted form of the mCRL2 specification which does not contain any parallel operators, encapsulations and hidings. A LPS is an intermediate step towards a labelled transition systems (LTS). Besides generating a LTS or PBES from a LPS it is also possible to simulate or manipulate a LPS.

The lineariser: mcrl22lps

The lineariser is called mcrl22lps and it will transform a correct mCRL2 specification into a Linear process specification (LPS) with the option of using different techniques and simplifications. If there are errors in the mCRL2 specification feedback about the errors is returned.

Labelled transition systems

The basis of modelling and verification are labelled transition systems (LTS). A LTS consists of states and transitions between states. The transitions carry labels to identify them.

A labelled transition system is a tuple of the form \( < S, \land, \to > \) where \( S \) is a set of states, \( \land \) is a set of transition labels and \( \to \) is a set of transitions of the form \( S \times \land \times S \). If \( p \) and \( q \) are states and \( a \) is a label then the transition \( (p, a, q) \) is presented as \( p \xrightarrow{a} q \).

By default a LTS does not contain any state variables. Some labelled transition systems extend the standard definition and add information to the states.

There are different formats of LTSs available. The standard format generated by lps2lts is the mCRL2 *.svc. Table 2.1 shows an overview of the different LTS formats.

The Aldebaran format is used by the CADP toolset. The three other formats are used by the mCRL2 toolset. The .fsm format allows the use of state variables.
### 2.3. MCRL2 TOOLSET

<table>
<thead>
<tr>
<th>format</th>
<th>ext.</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aldebaran</td>
<td>.aut</td>
<td></td>
</tr>
<tr>
<td>Finite state machine</td>
<td>.fsm</td>
<td></td>
</tr>
<tr>
<td>$\mu$CRL SVC</td>
<td>.svc</td>
<td></td>
</tr>
<tr>
<td>mCRL2 SVC</td>
<td>.svc</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: LTS formats

**LTS generator**  The tool *lps2lts* is used to generate a LTS from a LPS. Just like the lineariser different techniques and an optional reduction can be used. It is also possible to check the generated statespace for deadlocks and return a user defined number of traces that lead to a deadlock.

**(Parameterised) Boolean Equation Systems**

A PBES is an intermediate step towards a Boolean Equation Systems (BES) but it can be solved directly. The PBES will yield true if the modal formula is true in the initial state of the LPS and it will yield false otherwise.

**PBES generator**  When the mCRL2 toolset is used for model checking, the LPS and a modal formula can be transformed into a Parameterised Boolean Equation Systems (PBES) using the tool *lps2pbes*.

**BES generator**  A PBES can be transformed to a BES with the tool *pbes2bes*.

#### 2.3.2 Tools from the mCRL2 toolset

**Simulating LPSs: xsim**

Linear processes can be simulated using the tool *xsim*. A screenshot of *xsim* simulating a specification of a counter is shown in figure 2.5. The tool shows a list of all actions that can be executed from the current state. Behind the actions all state variables that will change by performing the actions are listed with their new values. Below the actions a list of all state variables with their current value is given. Clicking on an action will execute it and the lists of actions and state variables are updated.

If an error is suspected, *xsim* could be used to verify traces of actions through a statespace. With very small state spaces it can also be used to verify the complete system, although this is not common practice.

**Manipulation of LPSs**

There are different manipulations possible on a LPS. These manipulations read a LPS, modify it according to the tools description and then return a new, simplified LPS. Other tools might run faster/more efficiently after these simplifications. Note that some of these simplifications are already performed when running the linearisation tool *mcrl22lps*, but they can be disabled using flags. An overview of these tools can be found at the mCRL2 website [1].
Solving PBESs

The PBESs that are generated from the LPS and the modal formula can be solved using the tool \texttt{pbessolve}. The outcome will be either \textit{true} or \textit{false}. The PBES can also be transformed into a boolean equation system (bes) using the tool \texttt{PBES2bes}.

Manipulation of LTSs

\texttt{ltsconvert}  The two main functions of the \texttt{ltsconvert} tool is the conversion between different LTS formats and the conversion from a LTS to an equivalent LTS using one of the equivalent relations.

\textit{ltsconvert} supports the four input formats shown in table 2.1 Conversions from and to different formats are possible. Besides the LTS formats \texttt{ltsconvert} can also generate .dot graphics.

Besides converting a LTS to exactly the same LTS of a different format, \texttt{ltsconvert} can be used to generate a LTS using one of the following four types of equivalence relations.

- Strong bisimulation - the same sequences of traces with the same branching structure are possible
- Branching bisimulation - the same sequences of traces with the same branching structure are possible. Internal $\tau$ actions are removed if this has no effect on the branching structure
- Trace equivalence - the same sequences of actions are possible, the branching structure may be lost
- Weak trace equivalence - the same sequences of actions are possible. Internal $\tau$ actions are removed
**ltscompare**  The tool *ltscompare* can check if two LTSs are equivalent to each other using one of these equivalence relations.

**Visualisation of LTSs**

**Ltsgraph**  Ltsgraph is a tool that generates a graph of the statespace. The layout of the graph can be changed manually or through a force directed graph drawing method. A screenshot of the tool is given in figure 2.6. On the left of the screen the graph is drawn while on the right of the screen information and options are displayed.

By changing the parameters of the force directed drawing method the layout of the graph can be changed. Individual states can also be locked to their location and positioned manually. The appearance of the graph can be changed by changing state labels, colour and size.

Ltsgraph could be used practically to show state spaces up to fifty states.

**Ltsview**  Ltsview generates a three dimensional graph of the statespace as in figure 2.7. In this graph sets of states can be coloured based on the values of state variables. It is also possible to simulate the system while keeping track of the current state in the graph.

Ltsview could be used practically to show large state spaces, but it will run slowly when the state spaces are large.

**Diagraphica**  Diagraphica divides the states in groups depending on the values of state variables. When dividing the states, only the selected state variables are used. This way the values of particular variables can be compared. Figure 2.8 shows diagraphica.

In the upper left corner all state variables are shown. Directly below it are the values of the selected state variable.

In the upper right corner is a graph representation of all states divided in two groups depending on the selected state variable. Below the graph is a simulator that allows you to step forward and backward through the statespace indicating in which cluster of states the current state of the simulator is.

Another part of diagraphica can be used to draw a abstract representation of the system using shapes like rectangles and circles. Parameters of these shapes can be changed in size, colour, position and opacity depending on values of state variables. This representation can be seen in the lower left corner and as states in the simulator.

Diagraphica can be used to examine the values and combination of values of state variables. When a forbidden combination of values is found it is easy to backtrack a trace of actions from that group of states to the initial state of the LTS, allowing the error to be found in the mCRL2 code. When a abstract representation of the system is created it can be used to present the behaviour of the system to other people.

A downside of using diagraphica is that it heavily depends on state variables. In mCRL2 state space reductions are made by changing some of the values of state variables. This could lead to wrong representations of the LTS. Diagraphica can be used to examine very large state spaces.
Figure 2.6: Ltsgraph
2.3. MCRL2 TOOLSET

Figure 2.7: Ltsview

Figure 2.8: Diagraphica
2.3.3 3d party toolset: CADP

CADP is a third party toolset to generate, manipulate, simulate and visualise LTSs. From a mCRL2 point of view, the main use for this toolset is the model checker Evaluator. The section about model checking will discuss the use of this tool.

2.4 Model checking

A very attractive alternative to the simulation and visualisation of state spaces is the approach of formal verification. Model checking is formal verification method of properties on a model. Two of the advantages are:

1. Model checking is fully automatic.

2. When the design fails to satisfy a property, the model checker can give a counter example of a situation where the property fails.

This section will deal with the formulation and verification of properties on models using the model checker Evaluator from the CADP toolset.

2.4.1 Regular Alternation-free $\mu$-calculus

In order to describe properties that the model must satisfy, a modal formula of each property must be constructed. $\mu$-calculus is a temporal logic that can be used to formally specify properties. The version of the $\mu$-calculus syntax used by the model checker “Evaluator” is Regular Alternation-free $\mu$-calculus, which is simply called $\mu$-calculus in this report.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>action formula</td>
</tr>
<tr>
<td>R</td>
<td>regular formula</td>
</tr>
<tr>
<td>F</td>
<td>state formula</td>
</tr>
</tbody>
</table>

Table 2.2: Formula types

The logic is build from three types of formulas, described in table 2.2. This section will explain the syntax of the different formulas.

Action Formulas

Action formulas specify which actions are allowed to be executed. Actions are allowed to execute if they satisfy the formula. Action formulas are created according to the following BNF:
2.4. MODEL CHECKING

\[ A ::= \begin{align*}
& true \\
| & false \\
| & a \\
| & \neg A \\
| & A_1 \land A_2 \\
| & A_1 \lor A_2 \\
| & A_1 \rightarrow A_2 \\
| & A_1 \equiv A_2 \\
\end{align*} \]

The operators are written down in order of precedence and all operators are left-associative. The semantics of these operators are:

- \( true \) is satisfied by any action.

- \( false \) is never satisfied by an action.

- \( a \) is satisfied iff the action label equals \( a \).

- \( \neg A \) is satisfied iff it does not satisfy \( A \).

- \( A_1 \land A_2 \) is satisfied iff it satisfies both \( A_1 \) and \( A_2 \).

- \( A_1 \lor A_2 \) is satisfied iff it satisfies \( A_1 \) or it satisfies \( A_2 \).

- \( A_1 \rightarrow A_2 \) is satisfied iff it does not satisfy \( A_1 \) or it satisfies \( A_2 \).

- \( A_1 \equiv A_2 \) is satisfied iff either it satisfies both \( A_1 \) and \( A_2 \), or none of them.

In the plain text syntax of modal formulas the action labels \( a \) can be described using strings enclosed in quotes or a Unix regular expressions. For this reason action labels stated in this report are enclosed in quotes. When defining pattern of modal formulas the capital letters \( P \), \( Q \), \( R \) and \( S \) are used.

**Regular Formulas**

Regular formulas allow traces of zero or more actions to be executed. A trace of actions can be executed if it satisfies the regular formula. Regular formulas are created according to the following BNF:

\[ R ::= \begin{align*}
& \text{nil} \\
| & A \\
| & R_1.R_2 \\
| & R_1|R_2 \\
| & R^* \\
| & R^+ \\
\end{align*} \]

Where "nil" is the empty operator, "." is the concatenation operator, "|" is the choice operator, "\( ^* \)" is the transitive and reflexive closure operator, and "\( ^+ \)" is the transitive closure operator.

A sequence of LTS transitions satisfies:
• *nil* iff there are no transitions possible.

• *A* iff it is a single transition with an action that satisfies *A*.

• *R₁.R₂* iff it is the concatenation of two sequences satisfying *R₁* and *R₂*, respectively.

• *R₁|R₂* iff it satisfies *R₁* or it satisfies *R₂*.

• *R*⁺ iff it is the concatenation of zero or more sequences satisfying *R*.

• *R*⁻⁺ iff it is the concatenation of one or more sequences satisfying *R*.

**State Formulas**

A state formula is a logical formula built from boolean, modal, and fixed point operators, according to the grammar below:

\[
F ::= \text{true} \\
| \text{false} \\
| \neg F \\
| F₁ \lor F₂ \\
| F₁ \land F₂ \\
| F₁ \rightarrow F₂ \\
| F₁ \equiv F₂ \\
| < R > F \\
| [R]F \\
| @(R) \\
| X \\
| \mu X.F \\
| \nu X.F
\]

A state satisfies:

• *true* is satisfied by any state.

• *false* is not satisfied by any state.

• \(\neg F\) iff it does not satisfy *F*.

• *F₁ \lor F₂* iff it satisfies *F₁* or it satisfies *F₂*.

• *F₁ \land F₂* iff it satisfies both *F₁* and *F₂*.

• *F₁ \rightarrow F₂* iff it does not satisfy *F₁* or it satisfies *F₂*.

• *F₁ \equiv F₂* iff either it satisfies both *F₁* and *F₂*, or none of them.

• *< R > F* iff there is at least one transition sequence starting at the state, satisfying *R*, and leading to a state satisfying *F*. 

2.4. MODEL CHECKING

- $[R]F$ iff all transition sequences starting at the state and satisfying $R$ are, leading to states satisfying $F$.

- $@([R])$ iff there is a transition sequence starting at the state and being an infinite concatenation of sequences that satisfy $R$.

- $\mu X.F$ iff it belongs to the minimal solution of the fixed point equation $X = F(X)$ where the propositional variable $X$ denotes a set of LTS states.

- $\nu X.F$ iff it belongs to the maximal solution of the fixed point equation $X = F(X)$ where the propositional variable $X$ denotes a set of LTS states.

- $X$ iff it is an element of the set $X$.

2.4.2 Classification of properties

When formulating the requirements of a system, these requirements can be divided in safety and liveness properties. When translating the requirements to modal formulas these different kinds of requirements will follow different structures. Next examples of the different structures are given.

Safety properties

Safety properties state that something bad never happens. This can be formulated in different ways. For instance, if an $error$ action is bad, the formula

$[true^*."error"]false$

checks if all traces ending with an error action end in a state where false holds. Every trace ending in an error action that are able to execute will end in a state where true holds and therefore this formula will only yield true if there are no such traces possible.

Another way of constructing a safety property is by stating that in every state a good property holds. For example, a system with a deadlock means that there is a state where no actions are possible. In the formula

$[true^*]<true>true$

is checked that in every state there is at least one action possible.

Liveness properties

Liveness properties state that something good eventually happens. Typical liveness properties are actions or traces that always have the potential of being executed or will inevitably be executed. The formula

$<true^*."send">true$

states that there exists a path leading to a send action. and the formula

$\mu X.(<true>\land["send"]X)$

states that the "send" action is executed inevitably.
**Fairness properties** Sometimes it is desirable to only check fair traces when checking for liveness properties. For instance, the formula

\[
((\neg \text{"send"})^* \cdot \text{"send"}) \mu X. (\text{true} \land (\neg \text{"receive"})[X])
\]

checks if a receive action is inevitably executed after every send action. If there is a loop of action on any path from the send action to the receive action the model checker will return false because the loop of action can be executed infinitely often, preventing the receive action from being reached. If the loop is only allowed to be executed a finite number of times the formula might hold after all. To embed this fairness constraint in the formula it can be adapted. The formula

\[
[\text{true}^* \cdot \text{"send"} \cdot (\neg \text{"receive"})^*] < (\neg \text{"receive"})^* \cdot \text{"receive"} > \text{true}
\]

checks for the same property ignoring all loops on the path to the receive action. Another way of formulating this property would be:

\[
[\text{true}^* \cdot \text{"send"}] \nu Y. \mu X. ((\text{"wait"})[Y \land \text{true} \land (\neg \text{"receive"})[X])
\]

Now only loops consisting of wait actions are ignored. A problem with this formula is that the model checker Evaluator cannot verify formulas that has nested minimum and maximum fixedpoints.

### 2.4.3 Evaluator

**Model checking with the mCRL2 toolset**

The mCRL2 toolset has a modelchecker of its own. This model checker is currently under development and therefore it is not yet able to use this tool with the same results as Evaluator.

The main problem with the mCRL2 model checker is the limited feedback. It will only tell if the formula is valid or not. No traces of actions are returned.

The upside of this modelchecker is that nested fixedpoints of different kinds can be verified.
Chapter 3

Case study: on-screen display

3.1 Introduction

In the previous chapter a description of mCRL2 and the toolset was given. In this chapter we are going to look at a small example where we use mCRL2 and the toolset to verify a part of the on screen display (OSD) controller of a TV.

3.1.1 On screen display

The OSD control is responsible for all the information display on the TV screen. It handles when the information is visible, what the information should be and the position of the information. There are six types of OSDs.

**Status OSD**  This is a set of OSDs that displays the current system status after a user request.

**Preset OSD**  This is a set of OSDs that displays the current channel number after each channel change or when the TV set wakes up.

**Mute OSD**  This is a OSD that displays the mute status of the system.

**Message OSD**  This is a set of OSDs that display some message/information.

**Single Control OSDs (SCOSD)**  These are the OSDs that react to the user triggers by changing the system parameters that it represents. Examples of such an OSD is the Volume.

**Menu OSD**  Each Menu fragment here is identified as an Application.

Different types of OSDs behave differently to each other. During this example the behaviour of two types are modeled and verified.
3.2 System description

The system that is modeled for this example consists of the Status OSD and the Preset OSD. These OSDs have to communicate with each other and with the environment. As shown in figure 3.1, the environment of the system consists of the Single Control OSD, active applications, user input and the TV display.

![Figure 3.1: Overview of the processes with their communication](image)

The user input consists of three signals. The keyDisplay signal tells the system that status OSD has to be displayed or removed. The changeChan signal indicates that the channel of the TV has been changed. Finally, the Mode signal indicates if the OSD has to behave in minimal mode or not, true stands for minimal mode and false stands for normal mode.

The SCOSD can be suppressed by the system and it can suppress parts of the system. Through the two signals one part can tell the other to be removed and wait for a confirmation.

Applications sends a boolean to tell the system if there is an application active or not, true stands for an active application. The display receives two Booleans from the system that indicates if the status and preset OSDs have to be shown or not.

Internally the system consists of two processes, one for the behavior of each OSD. The behavior of these processes towards each other and the environment is described in the next two sections.

Besides the behavior of the two processes the requirement in table 3.1 must hold at any time.

<table>
<thead>
<tr>
<th>R00</th>
<th>Preset OSD cannot be shown at the same time as status OSD</th>
</tr>
</thead>
</table>

Table 3.1: Global requirements
3.2 SYSTEM DESCRIPTION

3.2.1 Status OSD

When a user presses the info button on the remote controller, the status of the system is shown on the screen. This information includes the information of the Preset OSD. The status is removed from the screen when the info button is pressed again.

Status OSD may co-exist with Mute OSD and Message OSD. When Status OSD is activated Preset OSD is immediately suppressed. When an application is activated Status OSD is removed from the screen. All SCOSDs are removed each time Status OSD is activated, but if an SCOSD is activated it will remove Status OSD.

The behaviour of the status OSD is rewritten as eight requirements in table 3.2.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>Status OSD is activated when the info button is pressed</td>
</tr>
<tr>
<td>R02</td>
<td>Status OSD is deactivated when the info button is pressed</td>
</tr>
<tr>
<td>R03</td>
<td>Status OSD is deactivated when SCOSD becomes active</td>
</tr>
<tr>
<td>R04</td>
<td>Status OSD is deactivated at most 10 seconds after it is activated</td>
</tr>
<tr>
<td>R05</td>
<td>Status OSD is deactivated when an application becomes active</td>
</tr>
<tr>
<td>R06</td>
<td>Status OSD cannot be activated when an application is active</td>
</tr>
<tr>
<td>R07</td>
<td>Status OSD will deactivate SCOSD when it becomes active</td>
</tr>
<tr>
<td>R08</td>
<td>Status OSD will suppress preset OSD when it becomes active</td>
</tr>
</tbody>
</table>

Table 3.2: Status OSD requirements

3.2.2 Preset OSD

The Preset OSD is responsible for the display of channel information. Preset OSD can operate in two modes, normal mode and minimal mode. When in normal mode the OSD is shown permanently. When in minimal mode the OSD is only shown for a limited time after the channel is changed. The mode of the Preset OSD is determined in another part of the system. When an application is activated Preset OSD will operate in minimal mode.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R09</td>
<td>Preset OSD is active when the TV set wakes up</td>
</tr>
<tr>
<td>R10</td>
<td>Preset OSD is activated when the channel is changed</td>
</tr>
<tr>
<td>R11</td>
<td>Preset OSD is deactivated after a timeout when the OSD mode is minimal</td>
</tr>
<tr>
<td>R12</td>
<td>Preset OSD is deactivated when an application is activated</td>
</tr>
<tr>
<td>R13</td>
<td>Preset OSD is suppressed when status OSD is activated</td>
</tr>
<tr>
<td>R14</td>
<td>Preset OSD is deactivated when SCOSD is activated</td>
</tr>
<tr>
<td>R15</td>
<td>Preset OSD is not deactivated after a timeout when the OSD mode is normal</td>
</tr>
<tr>
<td>R16</td>
<td>Preset OSD behaves as in minimal mode when an application is active</td>
</tr>
</tbody>
</table>

Table 3.3: Preset OSD requirements

Preset OSD can co-exist with all other types of OSDs except Status OSD. When Status OSD appears on the screen the Preset OSD is suppressed, when
Status OSD disappears. Preset OSD is shown again.

The behaviour of the preset OSD is rewritten as eight requirements in table 3.3.

### 3.3 mCRL2 specification

In this section the different parts of the mCRL2 specification of the system are explained. First the datatypes and actions are described. Next the structure of the processes are described followed by the initiation of the system.

#### 3.3.1 Data types

Most of the OSD types could be divided in four different states, inactive, active, suppressed and removed. The processes store their current state in a process variable. The type `state` is a structure with the four possible states of the OSDs as elements including recognisers for each element. The benefit of using a structured type is the predefined difference between the elements eliminating the need for function mappings on this type to force the elements to be different. The only function mappings needed in this specification are the recognisers to be able to determine in which state the process currently is.

**Inactive**  Inactive means that the OSD is not visible on screen and it is waiting for a signal to become visible.

**Active**  Active means that it is shown on screen.

**Suppressed**  When the OSD is suppressed it is supposed to be visible, but it is not shown on the screen as long as the suppressing OSD is active.

**Removed**  Removed OSDs are not visible and will not be able to become active or suppressed as long as they are removed.

\[
\text{sort } state = \text{struct } \text{active?is\_active} \quad \mid \text{suppressed?is\_suppressed} \quad \mid \text{inactive?is\_inactive} \quad \mid \text{removed?is\_removed}
\]

#### 3.3.2 Actions

The processes of the system communicate with each other and the environment as shown in figure 3.2. Each of the incoming or outgoing signals in the overview is considered to be an action that can be executed. Communications between processes follow the naming convention used in the example in section 2.2.4.

Besides the actions previously shown in figure 3.2 there are two more actions declared as the timeout signals of the status and preset OSD, `statusTime` and `presetTime`. These signals can be executed when preset or status OSD is active.
3.3. MCRL2 SPECIFICATION

Figure 3.2: Overview of the processes with their communication

and can be deactivated by timeout. The meaning of the other actions is given in table 3.4. The declaration of the actions is shown below

\[
\text{act} \quad \text{keyDisplay, changeChan;}
\]

\[
\text{rMode : Bool;}
\]

\[
\text{showStatus, showPreset : Bool;}
\]

\[
\text{presetTime, statusTime;}
\]

\[
\text{sStatus, rStatus, cStatus : Bool;}
\]

\[
\text{sPreset, rPreset, cPreset;}
\]

\[
\text{rApp, cApp : Bool;}
\]

\[
\text{sS2SC, rSC2S;}
\]

\[
\text{sP2SC, rSC2P;}
\]

During the rest of this chapter \{all actions\} is the set off all the single actions defined in this specification. Multi-actions not specifically declared, like sPreset|rPreset, are not included in this set.

3.3.3 Processes

Both processes of the system have the same structure. This section will explain the mCRL2 code for the preset OSD process. The complete mCRL2 code for the mCRL2 specification of this example is listed in appendix [3].

The following code shows the global structure of the process presetOSD. After the keyword proc the name and parameters are listed. stPreset of type
### Action names | Parameter | Meaning
--- | --- | ---
keyDisplay |  | The user presses the info button.
changeChan |  | The user changes the channel.
rMode | Bool | The user changes the mode to Bool.
showStatus | Bool | Bool indicates if status OSD is shown.
showPreset | Bool | Bool indicates if preset OSD is shown.
presetTime |  | Internal timeout signal for preset OSD.
statusTime |  | Internal timeout signal for status OSD.
sStatus,rStatus, | Bool | Communication from status to preset OSD.
cStatus | sPreset,rPreset, | Communication from preset to Status OSD.
cPreset | rApp,cApp | Bool indicates if an application is active. Both process receives this value, therefore the receiving actions must communicate with each other.
sS2SC,rSC2S |  | Communication between status and SCOSD.
sP2SC,rSC2P |  | Communication between preset and SCOSD.

<table>
<thead>
<tr>
<th>Table 3.4: Meaning of the actions</th>
</tr>
</thead>
</table>

state keeps track of the current state of this process, it holds either inactive, active, suppressed or removed. The parameter status is a boolean that keeps track of the visibility of the status OSD, the boolean app tells if there is an application active and the boolean min tells if the OSD mode is minimal.

\[
\text{proc } \text{presetOSD}(\text{stPreset : state, status : Bool, app : Bool, min : Bool}) = \\
\quad \text{is_inactive}(\text{stPreset}) \rightarrow (A) \\
\quad + \text{is_active}(\text{stPreset}) \rightarrow (B) \\
\quad + \text{is_suppressed}(\text{stPreset}) \rightarrow (C) \\
\quad D
\]

After the process name if-statements determine the state of the process and allow the corresponding actions to be executed. Note that the state “removed” is not checked, the preset OSD is never in this state. The actions that can be executed at any time are placed behind the checks for the current state at
Position D.

\[
\text{changeChan}.\text{status} \rightarrow ( \\
\text{presetOSD}(\text{suppressed}, \text{status}, \text{app}, \text{min}) \\
\diamond \text{showPreset}(\text{true}), \text{presetOSD}(\text{active}, \text{status}, \text{app}, \text{min}) \\
) \\
+ \sum_{b: \text{Bool}} r\text{Status}(b).s\text{Preset}.\text{presetOSD}(st\text{Pres}et, b, \text{app}, \text{min})
\]

Position A contains the actions that can be executed if the preset OSD is inactive. These actions are shown above. There is a choice between the actions changeChan and \(r\text{Status}(b)\). The \(r\text{Status}(b)\) action can be executed with either true or false substituted for \(b\). After this action is received a confirmation \(s\text{Preset}\) is send back and the process \(\text{presetOSD}\) is executed again with the value of the parameter \(\text{status}\) changed to the received value \(b\).

When a changeChan action is executed there are two possibilities. If the process variable status equals true the process \(\text{presetOSD}\) is executed with the value \(\text{suppressed}\) for parameter \(st\text{Pres}et\), else \(\text{showPreset}(\text{true})\) is send and the process \(\text{presetOSD}\) is executed with the value \(\text{active}\) for parameter \(st\text{Pres}et\).

\[
( r\text{Status}(\text{true}) \\
. \text{showPreset}(\text{false}) \\
. s\text{Preset} \\
. \text{presetOSD}(\text{suppressed}, \text{true}, \text{app}, \text{min}) \\
) \\
+ ( r\text{Status}(\text{false}) \\
. s\text{Preset} \\
. \text{presetOSD}(\text{active}, \text{false}, \text{app}, \text{min}) \\
) \\
+ ( rSC2P \\
. \text{showPreset}(\text{false}) . sP2SC \\
. \text{presetOSD}(\text{inactive}, \text{status}, \text{app}, \text{min}) \\
) \\
+ ( \text{changeChan} \\
. \text{presetOSD}(st\text{Pres}et, \text{status}, \text{app}, \text{min}) \\
) \\
+ ( \text{min} \lor \text{app} ) \rightarrow ( \\
. \text{presetTime} \\
. \text{showPreset}(\text{false}) \\
. \text{presetOSD}(\text{inactive}, \text{status}, \text{app}, \text{min}) \\
)
Position B contains the actions that can be executed if the preset OSD is active. These actions are shown above. The action \texttt{rStatus(false)} can be executed at any time and when the action \texttt{rStatus(true)} is executed preset OSD is removed from the screen and the parameter \texttt{stPreset} becomes suppressed.

If \texttt{rSC2P} is executed the preset OSD is removed from the screen and the parameter \texttt{stPreset} becomes inactive. The action \texttt{changeChan} can be executed at any time as long as the parameter \texttt{stPreset} is active.

If either the parameter \texttt{min} or \texttt{app} is true, preset OSD can be timed out with the action \texttt{presetTime}. After this action \texttt{showPreset(false)} is executed and the parameter \texttt{stPreset} becomes inactive.

\[
\texttt{rStatus(true).sPreset.presetOSD(suppressed, true, app, min)} + \texttt{rStatus(false).sPreset.showPreset(true)} \cdot \texttt{presetOSD(active, false, app, min)} + \texttt{changeChan.presetOSD(stPreset, status, app, min)} + (\texttt{min} \lor \texttt{app}) \rightarrow \texttt{presetTime.presetOSD(inactive, status, app, min)}
\]

Position C contains the actions that can be executed if the preset OSD is suppressed. These actions are shown above. The action \texttt{rStatus(true)} can be executed at any time and when the action \texttt{rStatus(false)} is executed preset OSD is shown on the screen and the parameter \texttt{stPreset} becomes active. The actions \texttt{changeChan} and \texttt{presetTime} behave the same as when the system is active.

\[
+ \sum_{b: Bool} \texttt{rMode(b).presetOSD(stPreset, status, app, b)} + \sum_{b: Bool} \texttt{rApp(b).presetOSD(stPreset, status, b, min)};
\]

The actions \texttt{rMode(b)} and \texttt{rApp(b)} can be executed regardless of the state of presetOSD. All they do is receive a boolean and substitute the appropriate process parameter with this boolean. This parameter can influence the behaviour of the process.

### 3.3.4 Initiation

The initiation of the specification consist of the two processes with the communication, allow and hiding operator applied to it. The communication operator add the actions \texttt{cStatus}, \texttt{cApp} and \texttt{cPreset} to the specification based on possible
multi-actions.

\[
\text{init} \quad \tau_I(\nabla_V (\Gamma_C (\text{presetOSD}(active, false, false, false) \parallel \text{statusOSD}(inactive) )));
\]

\[
C = \{s\text{Status}|r\text{Status} \rightarrow c\text{Status}, r\text{App}|r\text{App} \rightarrow c\text{App}, \text{sPreset}|r\text{Preset} \rightarrow c\text{Preset}\}
\]

\[
V = \{\text{all_actions}\}/\{s\text{Status}, r\text{Status}, r\text{App}, s\text{Preset}, r\text{Preset}\}
\]

\[
I = \emptyset
\]

The allow operator is used to enable all action except the actions \(s\text{Status}, r\text{Status}, r\text{App}, s\text{Preset}\) and \(r\text{Preset}\). These are the actions that provided the communication between the parallel processes. Because they are no longer allowed to execute, these actions have to be executed through the actions added by the communication operator. This will force the communication between the parallel processes.

Finally the hide operator can be used to declare actions as internal actions. This way the statespace generated from the specification can be reduced. Note that using the block operator instead of the allow operator can give different results. This is caused by the actions from parallel processes that are executed simultaneously. These multi-actions have to be blocked separately, which is done automatically by the allow operator and has to be done explicitly by the block operator. Currently the hiding operator does not hide any actions. This action is included for future changes.

### 3.4 Evaluating the statespace

During the creation of this specification different tools were used to gain information about the current behaviour of the system. The main tool is ltsgraph. Ltsgraph shows the (inter)actions of the system. When a possible error is found, xsim can be used to verify the error and find a trace in which the error occurs. This trace can be used to find the part of the mCRL2 specification that is responsible for the error.

#### 3.4.1 Statespace reduction

When visualising the state space using ltsgraph the number of states and transitions has to be checked. If the statespace is too big the graph will not be helpful. Instead of showing the entire statespace you can focus on a part of the behaviour while ignoring the rest of the actions.

The initiation of the mCRL2 specification can be used to hide actions using the hiding operator. The new initiation of the specification hides all actions except \(s\text{Status}, s\text{Preset}, c\text{Status}\) and \(c\text{Preset}\). This way a graph can be made showing the behaviour of the visibility of the preset and status OSDs combined with the communication between the two OSDs. The empty set \(I\) of the original initiation is replaced by:

\[
I = \{\text{all_actions}\}/\{s\text{Status}, s\text{Preset}, c\text{Status}, c\text{Preset}\}
\]
Next the statespace with hidden actions can be used as an input for \texttt{ltsconvert} to generate an equivalent reduced statespace. The original statespace consists of 74 states and 263 transitions, while the weak trace equivalent state space consist of 8 states and 10 transitions.

### 3.4.2 Ltsgraph

Using the reduced statespace from the previous section the following graph can be generated.

Although this graph does not show the complete behaviour, the communication between the two processes can be seen very clearly. In the initial state 0 the preset OSD is shown, now either a internal (hidden) action like a time-out can cause the preset OSD to disappear ending in state 6, or a message from the status OSD process can tell the preset OSD process that preset OSD has to disappear. Now from state 5 (upper left state) preset OSD responds by disappearing and sending a confirmation back to the status OSD process that reacts by showing the status OSD ending in state 1 (upper right state). From state 1 status OSD can disappear enabling either status or preset OSD to become visible again. This looks like the correct behaviour for the communication between the processes, allowing us to continue with the remainder of the specification.

As you can see this method of verification only allows small specifications or small parts of large specifications to be evaluated.

### 3.5 Formal requirements and verification

First all the requirements have to be transformed to $\mu$-calculus formulas. Next the model checker can verify if the specification satisfies all these requirements. Those requirements that are not met by the specification have to be evaluated and changes have to be made in the specification and/or the formulas. Now the specification can be verified again until the specification meets all requirements. The formal requirements are listed in table 3.5.

#### 3.5.1 Creating patterns for modal formulas

Most requirements can be described/formalised using the same patterns of modal formulas. In this section three patterns that fits to most of the requirements are discussed.
3.5. FORMAL REQUIREMENTS AND VERIFICATION

No \( P \) after \( Q \) until \( R \)

“No \( P \) after \( Q \) until \( R \)” means that after a \( Q \) is executed no \( P \) is allowed to execute until a \( R \) is executed.

\[
[true^*.Q.(\neg R)^*.P]false
\]

This formula states that for every path satisfying the action sequence \( true^*.Q.(\neg R)^*.P \) has to end in a state where false holds. Because false never holds the only way to satisfy this formula is if no such paths exist. The \( true^* \) enables you to reach every state in the statespace, then you try to execute a \( Q \) from all these states ending in all states directly reachable from a transition labeled with a \( Q \). Next you go from these states to all states reachable without performing a \( R \) action. Now you can be in any state between a \( Q \) and a \( R \) action. If it is possible to execute a \( P \) action from any of these states there exists a path that satisfies the formula \( true^*.Q.(\neg R)^*.P \) and then the model checker will return false, else it will not be possible to execute a \( P \) action after a \( Q \) until a \( R \) action.

Requirement R00, R06 and R15 are following this pattern. Note that R00 has to follow this pattern twice to exclude each other.

\( R \) responds to \( P \) after \( Q \)

“\( R \) responds to \( P \) after \( Q \)” means that after each execution of \( Q \) an \( R \) will respond to a \( P \). As soon as the \( R \) is executed a \( R \) does not have to respond to a \( P \) anymore until a new \( Q \) executes.

\[
[true^*.Q.(\neg R)^*.P]\mu X. < true > true \land [\neg R].X
\]

This formula states that for every path satisfying the action sequence \( true^*.Q.(\neg R)^*.P \) has to end in a state that belongs to the set of states generated by \( \mu X. < true > true \land [\neg R].X \). The action sequence is the same sequence as the previous pattern, but now it is allowed to execute this path as long as you end in a state where the execution of action \( R \) is inevitable. This way it holds that for all paths where action \( P \) does happen, action \( R \) will respond to it.

The minimal fixed point \( \mu X \) starts with the empty set of states \( X \) and it will add new states to \( X \) each iteration until no new states can be added anymore. Each iteration it will add all states that satisfy the conditions \( < true > true \land [\neg R].X \). The \( < true > true \land [\neg R].X \) states that there is at least one transition leaving the state and \( [\neg R].X \) states that all transitions that leaves the state are labeled with an \( R \) or will end up in a state already included in \( X \).

Requirements R01 to R05, R10, R12 to R14 matches this pattern or a variation of this pattern. In requirements R01 and R12 the \( R \) should also respond to \( P \) after the initial state. This is indicated by “\( R \) responds to \( P \) after init” and the formula above should be extended by:

\[
\land[([\neg R)^*.P]\mu X. < true > true \land [\neg R].X
\]

S and T precedes P

“S and T precedes P” means that before a \( P \) is executed a S and a T are executed. The S must be executed before the T.
The formula searches for all paths that end with a P where either no S is executed, or where a S is executed but there was no T after the S and before the P action. Note that the pattern assumes that both S and T precedes a P action, but only where the S precedes T as well.

Requirements R07 and R08 matches this pattern.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
</table>
| R00 | No “showPreset(true)” after “showStatus(true)” until “showStatus(false)”  
No “showStatus(true)” after “showPreset(true)” until “showPreset(false)” |
| R01 | “showStatus(true)” responds to “keyDisplay” after “showStatus(false)”  
“showStatus(true)” responds to “keyDisplay” after “init” |
| R02 | “showStatus(false)” responds to “keyDisplay” after “showStatus(true)” |
| R03 | “showStatus(false)” responds to “rSC2S” after “showStatus(true)” |
| R04 | “showStatus(false)” responds to “statusTime” after “showStatus(true)” |
| R05 | “showStatus(false)” responds to “cApp(true)” after “showStatus(true)” |
| R06 | No “showStatus(true)” after “cApp(true)” until “cApp(false)” |
| R07 | “sS2SC” and “rSC2S” precedes “showStatus(true)” |
| R08 | “cStatus(true)” and “cPreset” precedes “showStatus(true)” |
| R09 | < (¬“showPreset(true)”)*.”presetTime” > true |
| R10 | “showPreset(true)” responds to “keyDisplay” after “showPreset(false)”  
“showPreset(false)” responds to “cApp(true)” after “init” |
| R11 | [true *(“rMode(true)”).(¬“rMode(false)”) *.”showPreset(true)”]  
[“showPreset(true)”*(¬“showPreset(false)”)*.”rMode(true)”]  
< (¬ (“showPreset(false)”∨“rMode(false)”))*.”presetTime” > true |
| R12 | “showPreset(false)” responds to “cApp(true)” after “showPreset(true)”  
“showPreset(false)” responds to “cApp(true)” after “init” |
| R13 | “showPreset(false)” responds to “cStatus(true)” after “showPreset(true)” |
| R14 | “showPreset(false)” responds to “rSC2P” after “showPreset(true)” |
| R15 | No “presetTime” after “rMode(false)” until “rMode(true)” |
| R16 | mode minimal  
[true *.”cApp(true)”.(not “cApp(false)”)*.”showPreset(true)”]  
< (¬ (“showPreset(false)”∨“cApp(false)”))*.”presetTime” > true |

Table 3.5: Formal OSD requirements
3.5.2 Verifying formal requirements

The results of the model checker are listed in table 3.6. The three main reasons why the modal formulas did not hold were the existence of unfair paths, unimplemented behaviour and too strict requirements. This section will discuss the solutions of these three problems.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00</td>
<td>True</td>
</tr>
<tr>
<td>R01</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R02</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R03</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R04</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R05</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R06</td>
<td>True</td>
</tr>
<tr>
<td>R07</td>
<td>False (Not implemented)</td>
</tr>
<tr>
<td>R08</td>
<td>True</td>
</tr>
<tr>
<td>R09</td>
<td>True</td>
</tr>
<tr>
<td>R10</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R11</td>
<td>True</td>
</tr>
<tr>
<td>R12</td>
<td>False (Not implemented)</td>
</tr>
<tr>
<td>R13</td>
<td>True</td>
</tr>
<tr>
<td>R14</td>
<td>False (Fairness)</td>
</tr>
<tr>
<td>R15</td>
<td>False (Minimal mode and applications)</td>
</tr>
<tr>
<td>R16</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 3.6: Verified requirements

Unfair paths

When a modal formula requires an action to be executed eventually and when this action is stalled by another process infinitely, the model checker will return false. The counter example for this formula is given in the form of a graph, see picture 3.3 for an example.

The solution for this problem is to weaken the formula in such a way that the infinite loops are ignored when looking for a finite path. In the previous chapter, examples of modal formulas considering fairness are given. The pattern:
CHAPTER 3. CASE STUDY: ON-SCREEN DISPLAY

\[ [\text{true}^* \cdot Q. (\neg R)^* \cdot P] \mu X. < \text{true} > true \land \neg \neg R] X \]
can be written as:

\[ [\text{true}^* \cdot Q. (\neg R)^* \cdot P. \neg (R)^*] < \neg (R)^* \cdot R > true \]

When considering fairness.

Too strict requirements: minimal OSD mode and applications

Requirement R15 states that if the OSD mode is normal, the preset OSD cannot be timed out. Requirement R16 states that when an application is active, preset OSD act as in minimal mode. The modal formula for requirement 15 only looks at the \( rMode(true) \) and \( rMode(false) \) actions, while the \( cApp(true) \) and \( cApp(false) \) have as much impact. To counter this problem when model checking, the modal formula describing requirement R15 has to be weakened to still accept time outs when an application is active. The new modal formula will be:

\[
[\text{true}^*.\text{"rMode(false)"}.(\neg\text{"rMode(true)"}).\ast.\text{"cApp(false)"})\]
\[
|\text{"cApp(false)"}.(\neg\text{"cApp(true)"}).\ast.\text{"rMode(false)"})\]
\[
\text{\"rMode(true)"}.(\neg\text{"cApp(true)"}).\ast.\text{"presetTime"})]\false
\]

Unimplemented behaviour

Some required behaviour is not modelled at all. The solution to this is very simple, implement it.

3.5.3 Feedback to the specification and the requirements

After changing the mCRL2 specification to implement R07 and R12 and change the modal formula’s to check fair paths, most requirements holds. Requirement R12 also failed because of unfair paths and had to be adapted to ignore them. Now all requirements hold.

3.6 Conclusion

During this chapter we discussed an example from start to end. First the requirements were discussed. Next a mCRL2 specification was made according to the requirements. Although there is no description of errors found with ltsgraph, it is used intensively during the coding of the specification. It also enables the user to view properties of the specification that were not considered while making the requirements.

When the specification is finished model checking can be used to verify the requirements. Even after verifying the specification using other methods like ltsgraph, errors are found. The feedback of the model checker allows you to correct the code and/or (formal) requirements until you are satisfied with the result.
Chapter 4

Stateflow

This chapter will introduce the notation and semantics of Stateflow models. The next chapters will use the semantics of Stateflow models to generate a conversion method from Stateflow to mCRL2 models.

First the notation is explained, followed by the introduction of the running example of this chapter and the next chapters. Finally, the semantics of Stateflow models is explained.

4.1 Concepts and notation

Simulink is a toolbox of Matlab tool that allows the user to model, simulate and analyse designs. A Simulink design consists of multiple blocks that communicate with each other. A Stateflow chart is one of the blocks that can be used in Simulink.

4.1.1 Stateflow charts

Stateflow charts consist of a set of objects. There are different types of objects to specify different behaviour. The different types of objects and their behaviour and properties are explained in this section.

Hierarchy All objects are organised in a hierarchy. This hierarchy is based on containment. Every object is contained by another object or the chart itself. This leads to a tree of objects with the chart itself as the root.

Not all objects are able to contain other objects. The possible containment of objects by the different object types is described later this section. Charts can contain any type of objects.

Graphical representation Some objects are displayed in a graphical representation of the model, while other objects are defined on the background, to be used by the chart.

Data

Data objects can store numeric values to be used in charts. Other objects can read and/or write the values of data objects. These objects are not shown in
the graphical representation of the Stateflow chart.

Data objects have a property that determines the scope of the data value. The scope of the data can either be local or external. Local data can only be accessed by the chart itself and external data can be modified and/or read by components outside the chart. External data can be used to communicate with the environment of the chart.

**Events**

Event objects are responsible for triggering charts. When an event is broadcast, all receiving objects are executed by that event. Event objects are not shown in the graphical representation of the Stateflow chart.

Events have a scope which is either local, input or output. Local events are broadcast and received by the chart itself. Input events are generated by an external Simulink block and received by the chart. Output events are generated by the chart and sent to other Simulink blocks.

**Implicit events** Implicit events are pre-defined events that are automatically broadcast during the execution of the chart. They are generated by various actions that occur during the execution of a chart, i.e. when a value is assigned to a data object $D$, the event $\text{change}(D)$ can be generated.

Besides the data objects, implicit events can be generated by state objects (see section 4.1.3) and the chart itself. For instance, the chart generates a $\text{wakeup}$ event when the simulation starts and the chart is initialised. The implicit events generated by state objects are explained in section 4.1.3.

**Functions**

Function objects can be defined and used in Stateflow charts. These functions can accept parameters and return values in the same way Matlab and C functions work. Besides the functions defined within Stateflow, charts are also able to use functions defined elsewhere, such as Matlab and C functions.

Function objects are passive objects, they are not executed until another object calls it.

**4.1.2 Actions and action sequences**

Some objects can perform actions during the execution of the chart. Besides the use of standard mathematical operators, actions may be used to alter data objects, execute functions and broadcast events. Only these three kinds of actions are explained in this section. A complete description of Stateflow actions can be found in chapter 8 of the *Stateflow user’s guide* [4].

**Alter data objects** Using actions new values can be assigned to data objects. The current values of data objects can be read at any time.

**Broadcasting events** A local event can be generated by an action. This event will halt the execution of the current event, execute the chart itself, and then finish the execution of the chart by the initial event.
4.1. CONCEPTS AND NOTATION

Executing functions Functions can be executed during an action. If the result of the function is returned to the chart, it can be used by other operations during the same action.

Action sequences

Instead of performing a single action, some objects allow sequences of multiple actions to be performed. These sequences consist of a series actions separated by a semicolon. These actions are performed one at a time.

4.1.3 States

State objects are either active or inactive, the combination of active states in the model denote the current status of the system. During the execution of the chart, states may change from active to inactive and back. Figure 4.1 shows five state objects. The names of the states are shown in the upper left corner of the states.

![Figure 4.1: An example of the state decomposition used in Stateflow](image)

Hierarchy

States are able to contain objects. When a state contains objects, that state is considered the parent of those objects. States are also able to contain other states. This way all states are connected in a tree structure. In figure 4.1 states B1 and B2 are children of state A. States C1 and C2 are children of state B1.

State types

There are two types of states, parallel and exclusive states.

Parallel states Parallel states are all active as long as their parent is active. These states are represented in the graph by a dashed line, i.e. states B1 and B2 are parallel states. When state A is active, so are B1 and B2. When A is inactive, B1 and B2 will be inactive as well.
**Exclusive states**  When the parent of a set of exclusive states is active, one of the exclusive states will be active as well. It is not possible for more than one exclusive state with the same parent to be active. When the parent is inactive, all exclusive children are inactive as well.

Exclusive states are represented by a solid line. States A, C1 and C2 are exclusive states. When state B1 is active, either C1 or C2 will be active as well.

**State decomposition**  A state has either a parallel decomposition or an exclusive decomposition. This means that either all child states are parallel states or exclusive states. It is never possible for a state to have both parallel and exclusive states as children at the same time.

**Alternative representation**

In this report we will use an alternative way to denote the structure of states. Here the states will be represented as a tree. Figure 4.2 shows the tree representation of the Stateflow model in figure 4.1.

![Figure 4.2: Tree representation of the example of state decomposition](image)

The root of the tree is the chart itself, all other nodes are the states. Parallel composition is shown through lines from the parent to the child state connected with a horizontal line. State B1 and state B2 are parallel child states of state A. Exclusive decomposition is shown as single lines from the parent to the child state, i.e. state C1 and state C2 are exclusive children of state B1.

**Entering and exiting states**

During the execution of a chart, states can be entered and exited. When an inactive state is entered it becomes active and when an active state is exited it becomes inactive. Besides becoming active or inactive, action sequences can be performed and parent and child states may be entered or exited as well.

**State actions types**  States can contain actions sequences. During the execution of a chart, different situations can cause states to perform these action sequences. State B2 in the example in figure 4.1 contains three action sequences
4.1. CONCEPTS AND NOTATION

named \( \text{entB}2 \), \( \text{durB}2 \) and \( \text{exitB}2 \). These sequences are preceded by the trigger of the actions. This trigger indicates at what stage of the chart execution the action sequences are performed. Table 4.1 shows the different triggers for actions sequences contained by states.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Action are performed if:</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry:</td>
<td>The state is entered.</td>
</tr>
<tr>
<td>exit:</td>
<td>The state is deactivated.</td>
</tr>
<tr>
<td>during:</td>
<td>The state is executed and no transitions cause the state to be deactivated.</td>
</tr>
<tr>
<td>on_event</td>
<td>The state is executed as a result of a specific event and no transitions cause the state to be deactivated.</td>
</tr>
</tbody>
</table>

Table 4.1: Action sequence triggers for states

**Implicit events** When a state is entered or exited, an implicit event can be generated. For instance, when state B2 is entered, the event \( \text{enter(B}2) \) can be broadcasted. When state B2 is exited, the event \( \text{exit(B}2) \) can be broadcasted.

4.1.4 Transition and junction objects

Transition objects can be used to connect two state objects to each other. One of these state objects is the source and the other is the destination of the transition. Every time the chart is executed and the source state is active, the transition object may exit the source state and enter the destination state. This may cause a change of the status of the system.

Figure 4.3 shows a transition object from state B (source) to state C (destination). This is indicated by the arrow pointing to the destination state.

![Figure 4.3: An example of a transition from state B to state C.](image)

**Transition labels**

A transition object can be specified by a transition label. This label can describe the events and conditions that are required before the transition object becomes valid. The label can also contain action sequences that have to be executed when the conditions are true or when the transition is performed.
Valid transitions  Before a transition object may cause a transition from one state to another, it must be valid. Transition objects are considered valid when the source state is active and all requirements listed in the transition label are met.

A transition label has the following format:

\[ event[condition][\{condition\_actions\}] / transition\_actions \]

**Event and condition**  The event part of the label indicates which event object has to trigger the transition object to get a valid transition. It is allowed to specify multiple events that validate the transition by using the OR logical operator (\( \mid \)). When no event is specified, any event may trigger the transition.

**Condition**  The condition is a boolean expression that has to evaluate to true to validate the transition. The condition expression is enclosed in square brackets. Conditions are not required.

**Condition actions**  Whenever a transition is checked and found valid (i.e. the event matches and the condition is true), the condition actions are immediately performed. The condition actions are denoted between curly brackets.

**Transition actions**  When a transition is valid and executed, the transition actions are performed after deactivating the source state and before activating the destination state. Transition actions are denoted after a slash in the transition label.

![Figure 4.4: A default transition.](image)

Default transitions  When a state or chart with an exclusive decomposition is entered, one of the child states has to be activated. Default transitions indicates which of these child’s is activated. A default transition starts with a dot and ends in a state or a junction. When this transition is executed it must always result in a child state that is entered. Figure 4.4 shows a default transition ending in state C1.
4.1. CONCEPTS AND NOTATION

Junctions
Besides state objects, the source and/or destination of a transition could also be junction objects. Transitions ending in a junction can be combined with a transition starting at that junction, leading to paths of transition objects from one state to another. These are called transition paths. When all transitions of a transition path are valid, the transition path itself is valid. Valid transition paths can be performed as it was a single transition.

Figure 4.5 shows an example of a set of transition objects and a junction object.

Flow graphs
Some transition objects are of one of the types of flow graphs. There are three types of flow graphs, outer, inner and default flow graphs.

- **Outer flow graphs** Outer flow graphs of a state are all transition segments that start at that state and reside partially outside that state.

- **Inner flow graphs** Inner flow graphs of a state are all transition segments that start at that state and reside completely inside that state.

- **Default flow graphs** Default flow graphs of a state are all default transition segments that have that state as its parent.

Sets of flow graphs From these flow graphs sets of transition objects can be generated. A set of a particular type of flow graphs consists of all transition segments of that type with the same source state and all transition segments that can be reached through junctions from segments already in the set of flow graphs.

4.1.5 Other objects

Notes
Note objects can be used to add comments to a Stateflow model. Notes do not contribute to the behaviour of the Stateflow model.
Boxes

Box objects are used to graphically group a number of objects. Besides this organisational use, boxes do not contribute to the behaviour of the Stateflow model. Figure 4.6 shows two states that are grouped by a box object.

![Box Diagram]

Figure 4.6: Two states grouped by a box object.

History junctions

History junction objects can be used to extend the functionality of a state or chart. When a state or chart with an exclusive decomposition and containing a history junction is exited, the active child state is stored. Next time the state is entered, the stored child state becomes active again, thereby overriding any default transitions. A history junction is represented by a circle with an $H$ inside it. This symbol is placed inside the state that has the history junction.

Hierarchy of objects

Figure 4.7 shows a complete overview of the hierarchy of the chart and its objects. The chart and state objects are able to contain all objects that are drawn inside it.

4.2 Example 2: sound behaviour

This example will illustrate the different concepts described in the previous section. It will also be used as a running example in the next chapters.

4.2.1 The system and the environment

The model represents the behaviour of the sound of a TV. This includes the volume of the sound and messages regarding to the current status. The focus of this model is on the possible delays between applying the input and the change in the output. An overview of the system is shown in figure 4.8.

System inputs

The system has three inputs, *Volume up*, *Volume down* and *MuteOnOff*. These inputs are buttons on the remote control of the TV. The buttons should respectively increase, decrease and mute the sound.

Besides the normal inputs the Stateflow implementation of the system has a *Clk* event that is broadcast regularly.
4.2. EXAMPLE 2: SOUND BEHAVIOUR

System outputs
The system has two outputs, volume and display. Volume represents the current volume of the TV speakers and display indicates the current message that is displayed on the TV screen. Only one message can be shown at a time, either the Mute or the Unmute message. It is also possible that no message is shown.

4.2.2 System description
The states structure of the Stateflow implementation of the system is shown in figure 4.9. The functionality of the system can be divided into three parts. These parts are represented by the states and their descendants Process Keys, ActiveAudio and OnScreenDisplay. The remaining states TV and Active run these three parts in parallel.

The Stateflow model of the states TV and Active are shown in figures 4.10 and 4.11. When the chart is woken up, the entry actions of state TV are
Figure 4.9: Simulink state structure.
performed. These two actions call the Matlab functions `buffer` and `rcgui` that will initialise a gui of the remote control and the buffer that temporarily stores the remote control inputs. These are both functions that are a part of the environment of the system.

```
TV
entry: ml.buffer('init'); ml.rcgui('init');
```

Figure 4.10: State TV.

```
ProcessKeys  Active
  
  Active
  ActiveAudio
  OnScreenDisplay
```

Figure 4.11: State Active.

**ProcessKeys**

State *ProcessKeys* read inputs from the remote control and generates events that match that input. As shown in figure 4.12 state *Process Keys* has an exclusive decomposition and two child states, *AwaitKey* and *Delay*. The default transition leads to state *AwaitKey*.

When *AwaitKey* is active the buffer containing the inputs from the remote control is read every clock cycle. If an input is read, a corresponding event is executed in the chart. When the data object *delay* is set to a value larger than 0 and state *AwaitKey* is active, a transition to state *Delay* is performed.

A transition from state *Delay* back to state *AwaitKey* is made as soon as a number of seconds equal to the value of data object *delay* has passed. When state *AwaitKey* is active again, new inputs from the remote controller can be received.
CHAPTER 4. STATEFLOW

ActiveAudio

State ActiveAudio controls the volume of the TV. As shown in figure 4.13, state ActiveAudio has an exclusive decomposition and two child states, On and Mute. The default transition leads to state On.

If state On is active, there are three possible transitions triggered by three different events, VolumeUp, VolumeDown and MuteOnOff. The transitions triggered by VolumeUp and VolumeDown respectively increase and decrease data object vlevel, as long as vlevel does not exceed the volume limits 0 and MaxVol. After data object vlevel is changed, delay is set to VolDelayMax and state On and VolChangeDelay are entered. When delay is set to 0 in state ProcessKeys the transition from VolChangeDelay to UnMute is performed and volume is set to the value of vlevel. State VolChangeDelay represents the time the system
needs to change the volume.

The \textit{MuteOnOff} event triggers a transition from state \textit{On} to state \textit{Mute}. During this transition \textit{delay} is set to \textit{MuteDelayMax}. State \textit{MuteDelay} represents the time it takes to mute the sound. When \textit{delay} is set to 0 by state \textit{ProcessKeys}, the sound is muted, a \textit{Mute} event is send to state \textit{OnScreenDisplay} and a transition to state \textit{MuteState} is performed. The \textit{Mute} event indicates that the mute message is should be displayed on the TV.

When state \textit{Mute} is active, there are five transitions to state \textit{ShortestUnmute} that are triggered by three different events, \textit{MuteOnOff}, \textit{VolumeUp} and \textit{VolumeDown}. The transitions triggered by \textit{VolumeUp} and \textit{VolumeDown} respectively increase and decrease data object \textit{vlevel}, as long as \textit{vlevel} does not exceeds the volume limits 0 and \textit{MaxVol}.

The time needed by the system to set volume to the value of \textit{vlevel} is represented by two states, \textit{ShortestUnmute} and \textit{PossibleMute}. The transitions from these states generates the events \textit{FirstUnmute} and \textit{LastUnmute}. These events represents the start and end time of the unmmute message.

\textbf{OnScreenDisplay}

State \textit{OnScreenDisplay} controls the sound information display of the TV. As shown in figure 4.14 state \textit{OnScreenDisplay} has an exclusive decomposition and three child states, \textit{noDisplay}, \textit{on} and \textit{Display}. The default transition leads to state \textit{On}.

![Figure 4.14: State OnScreenDisplay.](image)

The current message that is displayed is indicated by a data object storing an integer. A one indicates that the mute message is shown, a two indicates that the unmute message is shown and a zero indicates that no message is shown at all.

States \textit{noDisplay}, \textit{On} and \textit{display} sets the data object to 0, 1 and 2 respectively when the corresponding state is entered. When state \textit{noDisplay} or \textit{DisplayUnmute} is active and a \textit{mute} event executes it, a transition to state \textit{on} is performed, thereby showing the mute message.
When state on is active and is executed by the FirstUnmute event, a transition to state DisplayUnmute is performed. This transition sets a timer that indicates when the unmute message is removed from the screen. After a LastUnmute event the unmute message is shown. When the timer is finished, there is a delay before a transition to state noDisplay is performed and the unmute message is removed from the screen.

4.3 Stateflow semantics

This section describes the semantics of the notation of Stateflow models. Stateflow executes the model sequentially on a single thread. This leads to sequential steps that form the behaviour. It starts with an event that executes a chart. From this chart other objects in the model are executed one by one.

4.3.1 Executing an event

Events can be generated by Simulink or the Stateflow chart itself. Events generated by Simulink are all events that are not generated by the Stateflow chart. These events awake a chart, execute it and then causes the chart to sleep again, waiting to receive another event. A Stateflow model behaves as a series of sequential steps and a chart execution as a result of a Simulink event has to be finished before the next Simulink event can be received.

Events generated by the Stateflow event itself are the result of a chart execution caused by an event, either generated by Simulink or Stateflow. When these events are generated, the execution of the current event is halted. Now the newly generated Stateflow event executes the chart. When the new event finished its execution, the old event can finish its own execution.

Semantic steps

When an event is broadcast, only the receiver of an event will be executed. The receiver of an event is either a chart or a state. Depending on the status of the receiver, actions are taken:

1. If the receiver of the event is a chart or an active state, then it is executed. For executing a chart see section 4.3.2 and for executing a state see section 4.3.3.

2. If the receiver of the event is an inactive state, nothing happens.

4.3.2 Executing a chart

Charts have three stages of execution. These stages are displayed in table 4.2.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inactive</td>
<td>The chart has no active states.</td>
</tr>
<tr>
<td>Active</td>
<td>The chart has active states and is processing an event.</td>
</tr>
<tr>
<td>Sleeping</td>
<td>The chart has active state, but no event to process.</td>
</tr>
</tbody>
</table>

Table 4.2: Stages of a Stateflow chart.
When a chart starts simulating, it is inactive. The first Simulink event initialises the chart, leaving the chart sleeping. A sleeping chart is ready to receive Simulink events. When a Simulink event is received, the chart becomes active and is executed. After the execution of the chart it becomes sleeping again, waiting to receive the next Simulink event.

Stateflow has the option of generating a first event before the simulation starts. This will initialise the chart before it can receive events during simulation. During this project all models will use this option.

**Initialising a chart**

When a Stateflow chart is inactive, the first event will cause an entry of its child states. If the children are exclusive states the default transition of the chart is executed, else the parallel states are entered in order. In section 4.3.5 the state entries are explained.

**Executing an active chart**

When an active chart receives an event, it will execute its active child states. See section 4.3.3 for an explanation of executing states. If the chart has no child states, it is initialised every time it receives an event.

### 4.3.3 Executing a state

States can be executed by its parent or directly. Every time it is executed the event that caused the execution determines the behaviour. The following steps are used to execute a state:

**Semantic steps**

1. Execute the set of outer flow graphs. Stop the execution of the state if a transition is taken.
2. During actions and valid on-event actions are performed.
3. Execute the set of inner flow graphs. Stop the execution of the state if a transition is taken.
4. Execute the active children in order of entry.

When step 4 is performed the active child states of the current state are executed. If the children are parallel states, the states are executed in the execution order that is shown in the upper right corner.

### 4.3.4 Executing a set of flow graphs

Executing a set of flow graphs is done in two steps. First a valid transition path to a state is explored. Next, if a valid transition path is found, this path is executed. If no valid transition path was found, the execution of the set of flow graphs is terminated.

The execution of a set of flow graphs of a particular type is performed with the following steps:
1. A set of transition segments is ordered.

2. While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination, which can be:

   - **State**: No more transition segments are tested and a transition path is formed by backing up and including the transition segment from each preceding junction until the respective starting transition. This transition path is executed.
   - **Junction with no outgoing transition segments**: Testing stops without any states being exited or entered.
   - **Junction with outgoing transition segments**: Step 1 is repeated with the set of outgoing segments from the junction.

3. After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when all starting transitions have been tested.

 Execute a transition path:

1. The states that are the immediate children of the parent of the transition path are exited. See section 4.3.6 for an explanation of exiting states.

2. The transition action from the final transition segment is executed.

3. The destination state is entered. See section 4.3.5 for an explanation of entering states.

**Ordering the transitions** Sets of transition segments are ordered by one of several methods. Despite the method that is used, the order can be shown at the origin of the transitions in Stateflow.

When a set of transition segments with a common source is checked for a valid transition path, these segments are tested in the fixed order given by Stateflow. When one of the segments is valid the destination is checked for a valid transition path to a state. If the destination finds a valid path or is a state itself, the current segment followed by the transition path is a valid transition path as well. If the destination has no valid transition path, the current segment also is not a part of a valid transition path and the next transition segment is tested for validity. If the destination is a junction and has not outgoing transitions, the search for a transition path is ended.

In figure [4.15](#) an example of the order of checking transition paths is shown. When the outer flow graph of state A is executed, first transition segment one is checked. If segment one is invalid, segment four is checked. If segment one is valid, transitions two and three are checked in order. If segment two or three are valid, a valid transition path is generated from state A to state B or C respectively. If segment two and three are invalid, segment four is checked.

When segment one is invalid and four is valid, segments five and six are checked, just like segments two and three where checked after segment one. If segment four is invalid there is no valid transition path.
4.3. **STATEFLOW SEMANTICS**

The example consists of four possible transition paths that has to be checked. These are the paths from state A to state B, C, D and E respectively.

### 4.3.5 Entering a state

Any inactive state in a Stateflow chart can be entered as a result of a transition. As a result from entering this state multiple other states might be entered too. This is all done by strict rules for entering states.

**Semantic steps**

When a state is entered as a result of a transition, the following steps are performed at this state.

1. If the parent of the state is not active, perform steps 1-4 for the parent.
2. If this is a parallel state, check that all siblings with a higher (i.e., earlier) entry order are active. If not, perform all entry steps for these states first.
3. Mark the state active.
4. Perform any entry actions.
5. Enter children, if needed:
   - If this state has exclusive decomposition, execute the default flow paths for the state.
   - If this state has parallel decomposition, perform entry steps 1-5 for each state according to its entry order.
6. If this is a parallel state, perform all entry actions for the sibling state next in entry order if one exists.
7. If the parent of the transition path is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

These steps are clarified with an example.

**Example**  Figure 4.16 shows an overview of the situation for this example. The root is active which is indicated by the gray background. This root has three parallel child states that are all inactive named A1, A2 and A3. State A2 also has three exclusive child states that are inactive named B1, B2 and B3. The default transition for this state leads to state B1. State B2 has three parallel child states that are inactive named C1, C2 and C3.

![Figure 4.16: Situation for example 2](image)

Now suppose a transition results into the entry of state B2. State A2 is not active and therefore steps one to four have to be executed for this state. A2’s parent is active, therefore step one is skipped. A1 is an inactive parallel sibling with a higher entry order, therefore A1 is entered first. After A1 is entered A2 is activated and its entry actions are performed. Now step one of the entry of state B2 is finished.

State B2 is an exclusive state and therefore step two is not performed. Now B2 is activated and its entry actions are performed. Next its children are entered in order, i.e. first C1 followed by C2 and C3. Step six is not performed because state B2 is an exclusive state.
4.3. STATEFLOW SEMANTICS

Step seven is performed on state B2 resulting in step six and seven being performed at state A2. A2 is a parallel state which has a parallel sibling, A3, with a lower entry order. State A3 is entered. Now state A2 and the transition have the same parent, ending the step sequence initiated by the initial entry of state B2.

The states are entered in the order A1, A2, B2, C1, C2, C3 and finally A3.

4.3.6 Exiting a state

Just like inactive states can be entered, active states can be exited. Exiting a state is always started at the highest level of states that has to be exited and it exits all of its descending states.

Semantic steps

When a state is exited, the following steps has to be performed on the state:

1. If this is a parallel state, make sure that all sibling states that were entered after this state have already been exited. Otherwise, perform all exiting steps on those siblings states.

2. If there are any active children, perform the exit steps on these states in the reverse order they were entered.

3. Perform any exit actions

4. Mark the state as inactive

These steps are clarified with an example.

Example  Figure 4.17 shows the situation of this example. The root is active and has three parallel child states named A1, A2 and A3 that are all active. State A2 has three exclusive child states named B1, B2 and B3 of which state B2 is active.

Now a transition with the root as parent is made, therefore all descendants of the root has to be exited. The children are parallel states and therefore they are exited in opposite to the ordering. State A3 performs its exit steps and is exited first. Next state A2 exits its active child, state B2. When B2 is exited A2 is exited followed by A1.

The states are exited in order A3, B2, A2, A1.
Figure 4.17: Exiting Part A
Chapter 5

Stateflow conversion to mCRL2

5.1 Introduction

In the previous chapter an overview of Stateflow models and its semantics is given. In this chapter the translation from Stateflow to mCRL2 code that is used in this project is discussed. Not all of the Stateflow models can be converted with this technique, but most of the models with the basic Stateflow objects described in the previous chapter are suitable.

5.2 General structure

In order to mimic the behaviour of a Stateflow model, the structure of the mCRL2 model is based on the structure of the Stateflow model execution. First the structure of Statelflow model executions is described, next the structure of the mCRL2 models is explained.

5.2.1 Stateflow structure

Stateflow executes its models according to the semantics given in section 4.3. The steps that have to be performed for a number of situations are described. Figure 5.1 shows the connections between these situations.

![Figure 5.1: General structure of Stateflow execution.](image)

Chart executions may start state executions, state executions may start flow graph executions and flow graph executions may start entries and exits of states. Entering, executing and exiting states procedures are also capable of
calling itself, i.e. the execution of a state may start the execution of a child state.

5.2.2 mCRL2 conversion structure

Because Stateflow executes its models on a single thread, mCRL2 should be able to mimic the behaviour on a single sequential specification. Because this specification will result in a very big process, it is divided into a series of small processes that will call on each other. The structure of these processes is based on the structure of the Stateflow semantics. Figure 5.2 shows the mCRL2 implementation of the Stateflow execution structure.

![Figure 5.2: General structure of mCRL2 conversion.](image)

The Stateflow chart execution is a recursive process that generates one of the possible Simulink events and then executes the chart with that event. This is followed by another execution of the Stateflow chart process.

For every state, its execution steps are implemented as a separate part. In order to execute a state, the corresponding part has to be called. The execution steps include the execution of a set of flow graphs and the parts may call on other state execution, entry and exit processes.

The state entry and exiting steps are also implemented as individual parts. These parts may call on each other like the state execution parts. Not every part has its own process for state entry, execution, and exits. States are divided into sets of sibling states. Every set is represented by one process for state execution, one for state entry and one for state exits, thereby grouping parts together. This way the number of processes is manageable and the processes are not too big. The entry, execution, or exiting of the states is also related to each other. These steps have to be performed at one exclusive state, or all parallel states in every set of siblings.

Stateflow event generation

When a Stateflow event is generated during the execution of the model, the corresponding states are executed with the new event. This event generation is not handled by the chart process.

Data objects

In Stateflow models, the data objects are used during all of the execution procedures. Therefore, they have to be accessible from any mCRL2 process described until now. This is done by storing all values of data objects in a process that executes in parallel of the chart. The chart is now able to read and write values to and from this process. The processes are shown in figure 5.3.
5.3 Data objects

Data objects are implemented on a different process than the Stateflow chart itself. There will be communication actions that allow the main process to read or write the data values.

The different data objects are stored in a list of integers. This means that only integer values can be used for data objects. The list will be of a fixed size and each element holds the value of a single data object. The following code will specify the data process.

\[
\text{sort ListInt} = \text{List}(\text{Int});
\]

\[
\text{act sRead , rRead , cRead : V \# Int;}
\]

\[
\text{sWrite , rWrite , cWrite : V \# Int;}
\]

\[
\text{proc data(d:ListInt)=}
\]

\[
\sum v:V.\text{sRead}(v, d.\text{var2nat}(v)).\text{data}(d)
\]

\[
+ \sum v:V, m: \text{Int}.(\text{var2nat}(v)<(#d)->(\text{rWrite}(v, m).\text{data}(%20\text{writeVar(var2nat(v), m, d, []))}))
\]

\]

First a new sort \text{ListInt} is declared and it denotes a list of Integers. Then the communication actions to read and write values are declared. These actions have two parameters of type \text{V} and type \text{Int} respectively. Type \text{V} is a custom structured type that has all data object names as elements. This structure is not needed, but it is included to be able to use names instead of numbers to indicate the different data objects.

The data process holds a parameter of type \text{ListInt} that will store all the values. There is a choice operator that allows one of the two summation operators to be executed. The first summation over datatype \text{V} allows the main process to read the value of object \(v\), where \(v\) is an element of \(V\). The second summation allows to write value \(m\) to object \(v\), where \(m\) is an integer and \(v\) is an element of \(V\).

In the data process there are two custom function \text{var2nat} and \text{writeVar}. These are functions on the sort \text{V} and are declared in the following code:

\[
\text{sort V = struct}
\]

\[
\{A\}
\]

\[
\{B\}
\]

;
map var2nat : V -> Nat;

eqn var2nat( {A} ) = 0;
var2nat( {B} ) = 1;

Sort \( V \) is a structure with a number of date object names. These names are substituted on location A and B. This list can be as long as needed. Next the function \( \text{var2nat} \) is declared as a function from \( V \) to Natural numbers. Each element of \( V \) is translated to the natural number that indicated on which place the data object \( v \) is stored in the list. The list should always contain a number of elements equal to the highest number that can be generated by this function plus one.

map writeVar :Nat # Int # ListInt # ListInt -> ListInt;

var s,t:ListInt;
n:Nat;
m:Int;
b:Bool;

eqn
( #s==0 )->writeVar(n,m,s,t) = t;
((#s!=0) & & (n> #t))->writeVar(n,m,s,t)
   = writeVar(n,m,tail(s),t<|head(s));
((#s!=0) & & (n=#t))->writeVar(n,m,s,t)
   = writeVar(n,m,tail(s),t<|m);
((#s!=0) & & (n< #t))->writeVar(n,m,s,t) = t++s;

The function \( \text{writeVar} \) is a function that takes a natural number \( n \), an integer \( m \) and two lists \( s \) and \( t \). The function will return the list \( s \) with the integer \( m \) at the \( n \)'th location, instead of the old \( n \)'th value.

5.3.1 State information

The state information keeps track of which states are active and which states are inactive. Each superstate (i.e. states with child states) of the chart has an equally named data object holding the status of its children. All sibling states are numbered to allow the data object named after their parent to identify them. Because state information is implemented as data objects, it will therefore be stored at the list in process \( \text{data} \).

Exclusive states

A superstate with an exclusive decomposition has one active child or no active child. All child states should be numbered, starting at one. Now the value of the state information data object indicates which child state is active. When no child states are active, the data object will be set to zero.
Parallel states
With a superstate with a parallel decomposition either non or all of its children are active. But in the process of entering and exiting the parallel child states, the states are activated/deactivated one by one. Entering a set of parallel states is performed in order, from the first to the last state. Exiting a parallel state is performed in the opposite order. The order of the parallel states is shown in the Stateflow graph in the upper right corner of the state.

The value of the state information data object indicates which is the highest active child state. When a parallel state is active, all of the states with a lower ordering number are also active. When a set of parallel states are entered, the number slowly increases until all states are active, while exiting the set of states would decrease the number to exit the states one by one.

5.4 Events

All events are combined into the structured sort *Events*. The following code declares the sort:

```plaintext
sort events = struct
  E?is E
  | F?is F;
```

Location E and F should be substituted with the event names. This list can be extended with more events. The isEventName is a function with a parameter `e` of type `Events` and that returns a boolean value. This value is true when the element `v` is equal to the corresponding EventName.

Events can execute a process by calling the process `chart` with the corresponding event name.

5.5 Performing actions and action sequences

During the execution of a Stateflow chart, action sequences can be executed. In this report we consider three kinds of actions that can be executed: event broadcasts, variable manipulation and the execution of matlab functions.

5.5.1 Event broadcasts

Event broadcasts are performed by executing the target state of the event with the event name as a parameter. In the mute example, the transition from state `PossibleMute` to `UnMute` has a `LastUnmute` event broadcast as its transition action. The label of the transition is equal to:

```
clk[delay==0]/LastUnmute;
```

The transition action is translated to mCRL2 by starting the chart process with the event as its parameter:

```plaintext
start(LastUnmute).chart(LastUnmute)
```
First an identifier action is performed to indicate the start of an event. Next the chart is executed by that event. When the execution is finished the mCRL2 model will fall back to the original process that initiated this call.

5.5.2 Data manipulation

Data manipulation can be performed by reading all the required data values and process them using the standard operators on numbers available in mCRL2. When an operation used in Stateflow is not available in mCRL2 a new function can be declared and used. Write the result of the calculation to the corresponding data object. The following line of code shows how data object volume is set to the value of data object vlevel:

\[ \text{sum i:Int.rRead(vlevel,i).sWrite(volume,i)} \]

5.5.3 Matlab functions

Create action with parameters equal to the matlab function and perform the action to indicate the function call. The function is not really executed, therefore no functions with results that are used in the Stateflow should be used, or they should be implemented differently. Depending on the results of the functions they can be mimicked by a custom mCRL2 function.

5.5.4 Action sequences

Action sequences consist of multiple actions combined using the sequential operator.

\[ \text{sWrite(volume,0).start(Mute).chart_exec(Mute)} \]

5.6 The Stateflow process

The main process is called stateflow. This process generates Simulink events and executes the chart with those events. An example of the process stateflow is shown below:

\[ \text{proc stateflow= start(Clk).SUO_exec(Clk) . stateflow ;} \]

This process broadcast a Clk event to the active child of chart SUO. The start action indicates the start of an event broadcast in the resulting statespace.

5.7 Executing an active state

As described in section 5.2.2, one state execution process is created for every set of sibling states. This process is named after the parent name of the sibling states followed by "_exec" to indicate it is an state execution process.
5.7. EXECUTING AN ACTIVE STATE

Exclusive state execution processes

The following code show the main structure of the state execution process for exclusive states:

```
proc ParentName_exec(event:Events)=
  sum state:Int.rRead(ParentName,state).( (state == 0)->(intern)
   + (state == 1)->(
     \{A\})
   + (state == 2)->(
     \{B\})
  )
;
```

First the state information about the set of states is read to check the current active state, then the corresponding state is executed. The execution of individual states is explained later in this section. Location \{A\} and \{B\} are replaced with the steps for the execution of a single state.

When the parent is active and has no active child states, the variable state is zero and an intern action is performed to prevent a deadlock. This situation may occur when a Stateflow event is broadcasted by an transition action, when the source is already exited and the destination is not already entered.

Parallel state execution processes

The following code show the main structure of the state execution process for parallel states:

```
proc Parent_exec(event:Events)=
  sum state:Int.rRead(Parent,state).( (state == 0)->(intern)
    <>((state > C)->(
      A.
      B.
    )
  )
);
```

First the state information about the set of states is read to check if the states are all active, then the active states are executed. The execution of individual states is explained later in this section. Location \(A\) and \(B\) are replaced with the steps for the execution of a single state. Location \(C\) is replaced with the number of sibling states minus one. This way the states are only executed when all parallel states are active, i.e. the state information data object points towards the state with the highest number.
When the parent is active and has no active child’s, the variable state is zero and an intern action is performed to prevent a deadlock. This situation may occur when a Stateflow event is broadcasted by an action.

### 5.7.1 Executing a set of flow graphs

The execution of a set of flow graphs cannot be fully covered by nested if-statements because a transition segment that is valid, might not be part of a valid transition path. When this happens, the else-statement should still be executed. For this reason, all possible transition paths are checked one by one. This results in a series of if-statements. A downside of this technique is that there should be a finite number of transition paths. Therefore it is not possible to execute sets of flow graphs with loops. The general structure of the flow graph is shown next:

```plaintext
sum vlevel: Int rRead(vlevel,vl)
  . ( {Transition event and conditions} )->(
    {Execute transition path}
  )<>(
    {Transition event and conditions} )->(
    {Execute transition path}
  )<>(
    {Transition event and conditions} )->(
    {Execute transition path}
  )<>(
    {No valid transition}
  ))
```

Every transition path is written as a if-statement. The transition path first in order is checked first, the transition path second in order is executed in the else-statement of the first if-statement. When no transitions are valid the else statement of the last transition path is executed.

#### {Transition event and conditions}  
The {Transition event and conditions} location consists of a boolean equation that checks the required event and conditions of the transition path.

#### {Execute transition path}  
The {Execute transition path} location performs the execution of the transition path. First the conditional actions of the path are performed, followed by the exit of all child states of the parent of the transition, the execution of transition actions and the entering of the destination state.

#### {No valid transition}  
The {No valid transition} location is used when actions has to be performed in case of no valid transitions.

### 5.7.2 State execution

As described in chapter [chapter number], the execution of a single state requires four steps:

1. Execute the set of outer flow graphs. Stop the execution of the state if a transition is taken.
2. During actions and valid on-event actions are performed.

3. Execute the set of inner flow graphs. Stop the execution of the state if a transition is taken.

4. Execute the active children in order of entry.

First the set of outer flow graphs is executed as described earlier. When no valid transition paths are found, the during and valid on-event actions are performed, followed by the execution of the set of inner flow graphs. When no valid transitions are found, the active children are executed by calling the process that handles the execution of the set of child states. The structure is show below:

\[
(\text{Execute outer flow graphs}) \\
\quad (\text{Perform during and on-event actions}) \\
\quad . \\
\quad (\text{Execute inner flow graphs}) \\
\quad (\text{Execute child states})
\]

All custom elements of this structure are explained earlier this section.

5.7.3 Example

Figure 5.4 shows the state \textit{ActiveAudio} and its descendants. The execution process of state \textit{Mute} is shown below:

```plaintext
proc Mute_exec(event:Events)=
    sum state:Int.rRead(Mute,state).(
        (state == 0)->( 
            intern 
        )
        + (state == 1)->( % State: MuteDelay 
            sum d:Int.rRead(Delay,d).( isClk(event) && (d==0) )->( 
                Mute_exit 
                . Mute_enter([2])
            )
        )
        + (state == 2)->( % State: MuteState 
            intern 
        )
    );
```

;
5.8 Entering a state

As it turns out, the entry sequence of a chart is always the same, from the top down and in case of parts with a parallel decomposition from the first to the last order number. When it comes to parts with an exclusive decomposition there is a choice which of the states has to be entered, either a predefined state or the destination of the first valid default transition path. The predefined state is an exclusive state that exists on the shortest path through the tree of states from the highest level state that is entered to the state that was originally entered. This structural behaviour is used when implementing the entry of states in the mCRL2 specification.

5.8.1 mCRL2 conversion

Every set of sibling states of the Stateflow model is implemented as a process in mCRL2. Each process handles the entry-steps for the states that belong to the set. The process has a list of integers as parameter, indicating which states has to be entered. The general structure of these processes is show below:

```mcrl2
proc State_enter(ln:ListInt)=
    sum n:Nat.rRead(Parent,n).(n != {A})->(
        Parent_enter({A} |> ln)
    )<<(
        (#ln == 0)->(
            {B}
        )<<(
            {C}
        )
    )
```
First a conditional statement checks if the parent state is active. Location \{A\} has to be replaced with the state information number of the parent. If the parent is not active, the entry of the current state is halted and the parent state is entered with the current parameter list following the parent state number.

Next a conditional statement checks if the list is empty. When it is empty, no fixed state has to be entered and the default flow graph is executed. If these states are parallel states, the process calls itself with a zero added to the empty list. The default flow graphs are executed as explained in section 5.7.

If the parent is active and the list is not empty, the state is entered. Location \{C\} is replaced with the action needed for these state entries.

### Entering exclusive states

When entering an exclusive state, the first element of the list indicates which state has to be entered. This state is activated, followed by the entry actions. When the entry actions are performed, the child states are executed with the tail of the list as a parameter. The structure is shown below:

\[
\begin{align*}
  \text{\((head(ln) == 1)\rightarrow(} & \\
  \quad & \text{sWrite(State,1)} \\
  \quad & \text{.\{Perform entry actions\} } \\
  \quad & \text{.\{Enter active child states\}} \\
  \text{)} & \\
  + \text{\((head(ln) == 2)\rightarrow(} & \\
  \quad & \text{sWrite(State,2)} \\
  \quad & \text{.\{Perform entry actions\} } \\
  \quad & \text{.\{Enter active child states\}} \\
  \text{)} & 
\end{align*}
\]

### Entering parallel states

Parallel states are entered in execution order. The structure is shown below:

\[
\begin{align*}
  \text{sWrite(State,1)} & \\
  \text{.\{Perform entry actions\} } \\
  \text{.\{Enter active child states\}} \\
  \quad & \text{.sum new:Nat.rRead(State,new).\((new == 1)\rightarrow(} \\
  \quad & \text{sWrite(State,2)} \\
  \quad & \text{.\{Perform entry actions\} } \\
  \quad & \text{.\{Enter active child states\}} \\
  \quad & \text{)} & \\
\end{align*}
\]

The first state is activated, its entry action performed and its children are entered. In case an event broadcast changed the situation, the next parallel state that has to be entered first checks if the previous state is the highest active state. If this is the case, the second state is activated, its entry action performed and
its children are entered. All following state entries are like the second state entry.

When calling the entry process to enter the children, the state with the number corresponding to the first element of the list receives the tail of the list as a parameter. Other states send an empty list as parameter.

### 5.8.2 Example

The following code shows the entry process of the children of state \textit{Mute} from the mute example in the previous chapter.

```ml
proc Mute_enter(ln:ListInt)=
    sum n:Nat.rRead(ActiveAudio,n).(n != 2)->(
        ActiveAudio_enter(2 |> ln)
    )<>( (#ln == 0)->(
        intern.Mute_enter([1])
    )<>(
        (head(ln) == 1 )->( % State: MuteDelay
            sWrite(Mute,1)
        )
        + (head(ln) == 2 )->( % State: MuteState
            sWrite(Mute,2)
        )
    ));
```

### 5.9 Exiting a state

Compared to the entering of states, the order for exiting states is exactly the other way around. Instead of starting with the first ordered state, we start with the last ordered state and now the children are exited before the state itself is exited. There are no exceptions in this order. If a transition is taken, all child states of the parent of the transition are exited.

#### mCRL2 conversion

Every set of sibling states of the Stateflow model is implemented as a process in mCRL2. Each process handles the exit-steps for the states that belong to the set. The process has no parameters, it will simply exit every active (sub)state belonging to that set. The general structure of these processes is show below:

```ml
proc Parent_exit=
    sum state:Nat.rRead(Parent,state).( (state == 0)->(
        intern
    )
    + (state == 1)->(
        \{A\}
    )
    + (state == 2)->(
```
The first step is to get the state information. After that, conditional statements select the actions belonging to the active state that is exited. \{A\} and \{B\} indicate the location for the exit steps of state one and two respectively. Each state of the part gets its own conditional statement. When the part has a parallel decomposition, location \{C\} is used as well.

**Exclusive states**

In case of an exclusive decomposition location \{A\} and \{B\} has the following code, corresponding to that state:

```plaintext
State_exit
  . {Exit actions of this state}
  . sWrite(Parent,0)
```

First all child states of State are exited. Then the exit actions are performed and finally the state variable of the parent is set to zero to deactivate the state.

**Parallel states**

In case of a parallel decomposition location \{A\} and \{B\} has the following code:

```plaintext
State_exit
  . {Exit actions of this state}

First all child state of State are exited, then the exit actions are performed. Next, at location \{C\}, an conditional statement checks if possible event broadcasts during the action sequences did not change the situation, before lowering the state information variable. After lowering the state information variable the process may call itself to exit other parallel states. The code for location \{C\} is shown below.

```plaintext
  . sum new:Nat.rRead(Parent,new).(new == state)->(
    . sWrite(Parent,state-1)
    (1<state)->(Parent_exit)<>{intern}
  )<>{intern}
```

**5.9.1 Example exit process**

The following code shows the exit process of the children of state Mute from the mute example in the previous chapter.
proc Mute_exit=
  sum state: Nat.rRead(Mute, state).( (state == 0)->( 
    intern 
  ) + (state == 1)->( %State: MuteDelay 
    sWrite(volume, 0)
    . start(Mute).SU0_exec(Mute)
    . sWrite(Mute, 0)
  ) + (state == 2)->( %State: MuteState 
    sWrite(Mute, 0)
  ) )
)

5.10 mCRL2 initialisation

The next code shows the initialisation of a mCRL2 model. It consists of the \textit{initiation} and \textit{data} process in parallel of each other. The data process holds the current values of the data objects and state information variables in a list as a parameter. The read and write actions of the data process are forced to communicate with the \texttt{comm} and \texttt{block} operators. The \texttt{hide} operator hides all read actions to enable the statespace to be reduced.

\begin{verbatim}
init
  hide({cRead},
  block({
    sWrite, rWrite,
    sRead, rRead
  },
  comm({
    sRead | rRead -> cRead,
    sWrite | rWrite -> cWrite
  },
  initiation
  || data([ 
    2, % Value 1
    0 % Value 2
  ])
  )
  )
  ;
\end{verbatim}

The \textit{initiation} process calls the chart entry process to initialise the chart. Next the \textit{initiation} process starts the execution of the chart using process \textit{stateflow}. The following code shows the \textit{initiation} process.

\begin{verbatim}
proc initiation =
\end{verbatim}
5.11 Case study: sound behaviour

The Stateflow model introduced in chapter 4.2 is converted to an mCRL2 model using the conversion methods from this chapter. Some parts of the Stateflow model are not covered by the conversion method. These custom conversions from these parts are explained in this section. The full code resulting from the conversion is shown in chapter C. The second part of this section discusses the statespace of the complete model.

5.11.1 Custom conversions to mCRL2

Receiving remote control inputs

The state ProcessKeys and its descendants are responsible for the reception of inputs from the remote control. The Stateflow model of state ProcessKeys is shown in figure 5.5.

![Stateflow Diagram]

Figure 5.5: State ProcessKeys.

In order to receive these inputs, the first transition from state awaitKeys reads a integer from a matlab function, stores it in data object knr and then apply the Stateflow function generateEvent to it. The function generateEvent reads an integer and broadcasts the corresponding event. The corresponding mCRL2 code of the transition would be:

```ml
SUO_enter([]).stateflow;

5.11. Case study: sound behaviour

The Stateflow model introduced in chapter 4.2 is converted to an mCRL2 model using the conversion methods from this chapter. Some parts of the Stateflow model are not covered by the conversion method. These custom conversions from these parts are explained in this section. The full code resulting from the conversion is shown in chapter C. The second part of this section discusses the statespace of the complete model.

5.11.1 Custom conversions to mCRL2

Receiving remote control inputs

The state ProcessKeys and its descendants are responsible for the reception of inputs from the remote control. The Stateflow model of state ProcessKeys is shown in figure 5.5.

![Stateflow Diagram]

Figure 5.5: State ProcessKeys.

In order to receive these inputs, the first transition from state awaitKeys reads a integer from a matlab function, stores it in data object knr and then apply the Stateflow function generateEvent to it. The function generateEvent reads an integer and broadcasts the corresponding event. The corresponding mCRL2 code of the transition would be:

```ml
ProcessKeys_exit
  . ml_buffer(1)
  . ( intern
    + start(MuteOnOff).SUO_exec(MuteOnOff)
    + start(VolumeUp).SUO_exec(VolumeUp)
    + start(VolumeDown).SUO_exec(VolumeDown)
  )
  . ProcessKeys_enter([1])
```
In this code the conditions for the event are checked first. Next a dummy action represents the Matlab function call, followed by either an intern action or the generation of one of three events. The internal action represents the situation where no inputs from the remote controll are received. The three events that can be generated represents the reception of event MuteOnOff, VolumeUp or VolumeDown.

The Matlab function is mimicked because it reads a random integer from outside the system that is applied to a function. Instead the result from the function generateEvent applied on a random integer is generated.

Timing and delays
This Stateflow model also uses timestamps during the simulation. These timestamps are used to delay the system for a limited time. An example of the use of timestamps can be found in state ProcessKeys. The Stateflow model of state ProcessKeys is shown in figure 5.5.

The transition from state awaitKeys to state Delay sets data object delayEntry to the current timestamp $t$. The transition back from state Delay to state awaitKeys subtracts the stored timestamp from the current timestamp in order to calculate if a time equal to the current value of data object delay has passed since the entry of state Delay.

The mCRL2 model cannot use these timestamps because keeping track of the current time would generate an infinite statespace. Because the Stateflow model is triggered by a clock signal, the exact time that passed is not important, but the clock pulse where the time has passed for the first time is important.

In the mCRL2 model this is implemented by setting data object delay to the amount of time that the system should be delayed. Next the value of delay is decreased by one on every clock pulse. As soon as delay is zero, the time has passed and the system can continue. This technique is also used for the two delays in state OnScreenDisplay.

The time stored in data object delay is in seconds. The Stateflow model is triggered by the clock signal twice per second. Therefore all delay times are multiplied by two in the mCRL2 model.

Constant values for data objects
Some values of data objects in the Stateflow model are never changed. These data objects are not implemented as described in section 5.3. Instead the values of the constant data objects are used directly in the model. This reduces the amount of read actions from the data process, thereby reducing the generated statespace from the model.

5.12 Conclusions
Applying the conversion to the sound behaviour model leads to a state space with 73755 states and 74547 transitions. The linearisation tool is executed using stack datatypes and add delta summands, the state space generation used the compiling jitty rewriter. The generation of the state space takes approximately one minute.
The state space can be significantly reduced by applying state space reduction using one of the equivalence relations. Table 5.1 shows an overview of the reduced state spaces.

<table>
<thead>
<tr>
<th>Minimisation option</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>73755</td>
<td>74547</td>
</tr>
<tr>
<td>Strong bisimulation</td>
<td>58887</td>
<td>59679</td>
</tr>
<tr>
<td>Branching bisimulation</td>
<td>12330</td>
<td>13122</td>
</tr>
<tr>
<td>Trace equivalence</td>
<td>58887</td>
<td>59679</td>
</tr>
<tr>
<td>Weak trace equivalence</td>
<td>12096</td>
<td>12888</td>
</tr>
</tbody>
</table>

Table 5.1: State spaces after reduction.
CHAPTER 5. STATEFLOW CONVERSION TO MCRL2
Chapter 6

Case study: sound behaviour

6.1 Introduction

In the previous chapter the mCRL2 code for example two was created. This chapter will focus on the verification of the model using the state space generated from the mCRL2 code. First the requirements for the system are given. Next the model is verified by those requirements. After the verification of the model, an analysis of the behaviour of the model is made using different values for the different delays.

6.2 System requirements

Before the requirements are introduced, a quick overview of the system is given.

6.2.1 System overview

As described in chapter 4.2.1 the system has three inputs and two outputs. The inputs are implemented as the events VolumeUp, VolumeDown and MuteOnOff. The outputs are implemented as the data objects volume and display and are represented by integer values. Output display can hold three different values which are shown in table 6.1. Output volume can hold any integer value.

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No message is shown.</td>
</tr>
<tr>
<td>1</td>
<td>Mute message is shown.</td>
</tr>
<tr>
<td>2</td>
<td>Unmute message is shown.</td>
</tr>
</tbody>
</table>

Table 6.1: Possible values for data object display.
Predefined values

Besides these inputs and outputs, the requirements use two predefined values, MinVol and MaxVol. These values represent the minimum volume and maximum volume for the system. This way, the requirements can be generic for models with a different amount of volume levels.

States

The system has two states that determine the behaviour of the system, Muted and Unmuted. The system is always considered to be in either one of these states at any time. The state may change during the execution of the system.

The requirements are grouped based on these states. There are global requirements that must hold in both of the states, the unmuted requirements that must hold in the state Unmuted, and the muted requirements that must hold in the Muted state.

6.2.2 Requirements

Tables 6.2, 6.3, and 6.4 show the Global, Unmute and Mute requirements.

| R01 | The initial state of the system is unmuted |
| R02 | volume is never less than MinVol or greater than MaxVol |

Table 6.2: Global requirements

| R03 | The MuteOnOff button will cause a transition to state muted. |
| R04 | The VolumeUp and VolumeDown button will not change the state to muted. |
| R05 | If volume is less than MaxVol, the VolumeUp button increases volume. |
| R06 | If volume is greater than MinVol, the VolumeDown button decreases volume. |
| R10 | When the system becomes unmuted, unmuteMessage is eventually shown, after a finite time the message is removed. |
| R11 | When the system becomes unmuted and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system. |
| R12 | When the system becomes unmuted and the transition is caused by the VolumeUp button, the volume is eventually set to the increased original value of the unmuted system. |
| R13 | When the system becomes unmuted and the transition is caused by the VolumeDown button, the volume is eventually set to the decreased original value of the unmuted system. |

Table 6.3: Unmute requirements
6.3 Verification

The Modal formulas of the requirements in this section are either general for a model with a number of volume levels, or they are specified for a model using three volume levels. When the model uses three volume levels, DataObject MinVol equals zero and MaxVol equals 2. The full requirements used on a model with three volume levels are given in section C.

During the verification of the requirements the requirement itself is stated first. Next the requirement is rewritten in Stateflow terms. Finally the stateflow terms are translated to a modal formula that can be checked on the statespace of the model.

Notation

While explaining the formalisation of the requirements, we will use the Stateflow terms instead of mCRL2 action names. Table 6.5 shows an overview of the different terms. I.e. the Stateflow term “enter Mute” represents the mCRL2 action “cWrite(ActiveAudio, 2)”. The assignment of values to data objects is indicated by the := operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>mCRL2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N := n</td>
<td>cWrite(N, n)</td>
<td>Assign value n to data object N.</td>
</tr>
<tr>
<td>start E</td>
<td>start(E)</td>
<td>Start event E.</td>
</tr>
<tr>
<td>enter S</td>
<td>cWrite(S, n)</td>
<td>Enter state S, assuming its is the nth state.</td>
</tr>
<tr>
<td>exit S</td>
<td>cWrite(S, 0)</td>
<td>Exit state S.</td>
</tr>
</tbody>
</table>

Table 6.5: Action notation of the model formulas

Modal formula patterns

In chapter 3 patterns of modal formulas are created to describe most requirements. During the verification of this model, the modal formulas of the requirements do not fit to patterns. Every requirement is described individually. If an formula is able to be reused, this is indicated during the descriptions of the verification.

6.3.1 Global requirements verification

The global requirements must hold at any time during the execution of the model.
CHAPTER 6. CASE STUDY: SOUND BEHAVIOUR

Requirement 1

The initial state of the system is *unmuted*.

When the simulation starts state *On* should become active. This is verified by the modal formula:

\[
\mu X. (\langle \text{true} \rangle \text{true} \land \neg \text{"enter Mute"} \land \neg \text{"enter On"} X )
\]

The formula consists of a minimal fixed point that require each state to have at least one outgoing transition, none of the outgoing transitions may be *enter Mute* and and all transitions should either perform the action *enter On* or lead to a state that satisfies the formula.

This requirement holds for this model.

Requirement 2

*volume* is never less than *MinVol* or greater than *MaxVol*.

This requirement is broken into two parts. Part *a*, volume never becomes negative, and part *b*, the volume is never bigger than *MaxVol*. The modal formula for part *a* is:

\[
[\text{true*}.'\text{volume} := -\d'] \text{false}
\]

The modal formula for part *b* is:

\[
[\text{true*}.'\text{volume} := [0-2]|(\d\d')'] \text{false}
\]

Both actions in the formulas use UNIX regular expressions. In regular expressions it is possible to specify the occurrence of random characters or digits. We use that property to add a random value after the negative sign for part *a*, or any value but 0, 1 or 2 in part *b*. This assumes that *VolMax* equals 2. If \d in part *a* is substituted with 0, the minus sign turns it into -0, therefor not preventing volume to become zero.

Both formulas consider all paths containing the actions to be false. This requirement holds for this model.

6.3.2 Unmuted requirements verification

The unmuted requirements must hold at any time the system is unmuted.

Requirement 3

The *MuteOnOff* button will cause a transition to state *muted*.

If state *On* is active, the *MuteOnOff* event will cause a transition to state *Mute*. This is represented by the following modal formula:
6.3. VERIFICATION

[ true
. ‘‘enter On"
. (not ‘‘exit On”)*
. ‘‘start MuteOnOff"
]
( μ X.(
	<true> true
and [not ‘‘enter Mute”] X
)
)

First all paths that ends between an entry and an exit of state On are considered, followed by the start of the MuteOnOff event broadcast. All these paths eventually leads to the entry of state Mute. This requirement holds for this model.

Requirement 4

The VolumeUp and VolumeDown button will not change the state to muted.

This requirement is made stronger by creating a formula that checks that no event other than MuteOnOff leads to the entry of state Mute. This is represented by the following modal formula:

[ true
. ‘‘enter AOn"
. (not ‘‘start MuteOnOff”)*
. ‘‘enter Mute"
]false

The formula searches for all traces where state On becomes active followed by the entry of state Mute without a MuteOnOff event being broadcast in the meantime. Any of these traces are considered false. This requirement holds for this model.

Requirement 5

If volume is less than MaxVol, the VolumeUp button increases volume.

Whenever state On is active and volume is less than MaxVol, the VolumeUp event will result in an increase of volume by one. The modal formula used to verify this behaviour is:

( [ true
. ‘‘enter On"
. (not ‘‘exit On”)*
. ‘‘volume := n”
. (not ‘‘exit On”)*
. ‘‘broadcast VolumeUp"
)
CHAPTER 6. CASE STUDY: SOUND BEHAVIOUR

First a path to each state where On is active is created. Then the paths are reduced to the paths where volume is set to \( n \). If the VolumeUp event is performed from these after one of the remaining paths, all paths should end in a state where the volume is eventually set to \( n+1 \).

This formula has to be tested for all \( n \) that are greater than or equal to MinVol and less than MaxVol. This requirement holds in the model.

**Requirement 6**

If volume is greater than MinVol, the VolumeDown button decreases volume.

Whenever state On is active and volume is greater than MinVol, the VolumeUp event will result in a decrease of volume by one. The modal formula used to verify this behaviour is the same as the formula used for requirement 5, except that all possible paths have to end in a state where the volume is eventually set to \( n-1 \).

\[
(\mu X. (\langle \text{true} \rangle \text{true} \\
\quad \text{and} \ [\text{not } \langle \text{volume := } n+1 \rangle ] X \\
\quad ))
\]

This formula has to be tested for all \( n \) that are greater than MinVol and less than or equal to MaxVol. This requirement holds in the model.

**Requirement 10**

When the system becomes unmuted, unmuteMessage is eventually shown, after a finite time the message is removed.

This requirement is split up into two parts. Part a states that display becomes two after a transition from state Mute to state On. Part b states that whenever display becomes two, it will inevitably become either zero or one. The modal formula for part a would be:

\[
[\text{true*} \\
\cdot \langle \text{enter On}\rangle* \\
\cdot (\text{not } \langle \text{exit On}\rangle)* \\
\cdot \langle \text{volume := } n\rangle* \\
\cdot (\text{not } \langle \text{exit On}\rangle)* \\
\cdot \langle \text{broadcast VolumeUp}\rangle]
\]

(\mu X. (\langle \text{true} \rangle \text{true} \\
\quad \text{and} \ [\text{not } \langle \text{volume := } n-1 \rangle ] X \\
\quad ))

This formula has to be tested for all \( n \) that are greater than MinVol and less than or equal to MaxVol. This requirement holds in the model.
This formula considers all paths that ends with the entry of state \textit{On}, but where state \textit{Mute} has been entered before this state. This will exclude the entry of state \textit{On} caused by the initialisation of the system. All these paths should end in a state where the inevitably leads to a transition where display is set to two.

The formula for part b would be:

\[
\begin{align*}
\langle \text{true}\ast, \text{"display} := 2\rangle \\
\langle \text{mu } X. \langle \text{true}\rangle \text{ true} \\
\text{ and [ not } ( \text{"display} := 0 \\
\text{ or } \text{"display} := 1 \\
\text{ ] } X \\
\rangle \\
\rangle
\end{align*}
\]

The formula for part b checks if all paths that ends with display being set to two ends in a state where display will eventually be set to either zero or one.

These formulas are true in the model.

\textbf{Requirement 11}

When the system becomes \textit{unmuted} and the transition is caused by the \textit{MuteOnOff} button, the volume is eventually set to the original value of the unmuted system.

Consider the system where state \textit{On} is active and \textit{volume} is set to \(n\), followed by an transition to state \textit{Mute}. When the system in this situation has a transition to state \textit{On}, the \textit{volume} is eventually set back to \(n\). In order to check this requirement on the model, the following formula pattern is used:

\[
\begin{align*}
\langle \text{true}\ast, \text{"enter On}\rangle \\
\text{ ( not } \langle \text{not } \text{"exit On} \rangle\ast \\
\langle \text{\textit{volume} := n}\rangle \\
\langle \text{ ( not } \langle \text{not } \text{"exit On} \rangle\ast \\
\rangle
\end{align*}
\]
CHAPTER 6. CASE STUDY: SOUND BEHAVIOUR

. "start MuteOnOff"
. ( not "enter Mute" )
. "enter Mute"
. ( not ( "start MuteOnOff"
    or "start VolumeUp"
    or "start VolumeDown"
  )
)
)

This pattern checks for all paths that enter state On, set the volume to n, then exit state On and enter state Mute followed by the broadcast of event MuteOnOff. These paths should lead to a state where the volume is inevitably set to value m. In order to check the requirement, the pattern should be checked for all volume levels where n and m are equal to that volume level. This requirement is true in the model.

Requirement 12

When the system becomes unmuted and the transition is caused by the VolumeUp button, the volume is eventually set to the increased original value of the unmuted system.

Consider the system where state On is active and volume is set to n, followed by an transition to state Mute. When the system in this situation has a transition to state On, the volume is eventually increased by one with a maximum value equal to MaxVol. In order to check this requirement on the model, the formula pattern from requirement eleven is used.

In order to check the requirement, the pattern should be checked for n equals zero to MaxVol minus one and m equals n plus one. This requirement is true in the model.

Requirement 13

When the system becomes unmuted and the transition is caused by the VolumeDown button, the volume is eventually set to the decreased original value of the unmuted system.

Consider the system where state On is active and volume is set to n, followed by an transition to state Mute. When the system in this situation has a transition to state On, the volume is eventually decreased by one with a minimum value equal to MinVol. In order to check this requirement on the model, the formula pattern from requirement eleven is used.

In order to check the requirement, the pattern should be checked for n equals one to MaxVol and m equals n minus one. This requirement is true in the model.
6.3.3 Muted requirements verification

The muted requirements must hold at any time the system is muted.

**Requirement 7**

When the system becomes *muted*, *volume* is eventually set to zero and it is not changed until the system becomes *unmuted*.

This requirement is split into two parts. Part a states that when *Mute* becomes active, *volume* is eventually set to zero. Part b states that when *Mute* becomes active and the *volume* is set to zero, the *volume* is never set to another value than zero. The formula for part a would be:

\[
[\text{true}^* \cdot \text{''enter Mute''} \\
\mu X. (false)
\]

This formula states that all paths that ends with the entry of state Mute will eventually lead to a state where volume is set to zero.

\[
[\text{true}^* \cdot \text{''enter Mute''} \\
\text{not ''volume := 0''}^* \\
\text{''volume := 0''}^* \\
\text{not ''exit Mute''}^* \\
\text{''volume := [1-2]''};]
\]

The formula for part b state that all paths where state *Mute* is entered and *volume* is set to one or two are false. These formulas are true in the model.

**Requirement 8**

When the system becomes *muted*, *muteMessage* is eventually shown and it is not removed until the system becomes *unmuted*.

This requirement is split into two parts. Part a states that when state *Mute* is entered, the display is set to one. Part b states that when state *Mute* is entered and *display* is set to one, the *display* is not set to any other value than one until state *Mute* is exited. Part a is represented by the following modal formula:
The formula for part a checks that after the entry of state \( Mute \), \( display \) is inevitably set to one. The formula for part b is:

\[
\begin{align*}
\text{true*} &. \text{''enter Mute"} \\
&. (\text{not ''display := 1"})* \\
&. \text{''display := 1"} \\
&. (\text{not ''exit Mute"})* \\
&. \text{'display := ['^1'] } \\
\] false
\end{align*}
\]

The formula for part b checks that all paths that enter state \( Mute \), set \( display \) to one and then set it to any other value than one before state \( Mute \) is exited are false.

These formulas are true in the model.

**Requirement 9**

The \( VolumeUp \), \( VolumeDown \) and \( MuteOnOff \) buttons will cause a transition to \emph{unmuted}.

This requirement states that whenever state \( Mute \) is active, the \( MuteOnOff \), \( VolumeUp \) and \( VolumeDown \) events will cause a transition to state \( On \). The modal formula for this requirement would be:

\[
\begin{align*}
\text{true*} &. \text{''enter Mute"} \\
&. (\text{not ''start MuteOnOff"}) \\
&. \text{''start MuteOnOff"} \\
&. \text{''start VolumeUp"} \\
&. \text{''start VolumeDown"} \\
\] \\
( \mu X. ( \begin{align*}
\text{true} &. \text{not ''enter On"} X \\
\end{align*}
) )
\end{align*}
\]

The formula checks if all paths where the first occurrence of a \( MuteOnOff \), \( VolumeUp \) or \( VolumeDown \) event that follows the entry of state \( Mute \), ends in a state that will inevitably lead to the entry of state \( On \).

This formula is true in the model.
6.4 DisplayMin analyses

The delays can be changed to gain an insight of the impact of the delays on the behaviour of the model. During this section the data object DisplayMin is set to three different values to check if the model still behaves according to requirement ten. The delays are set to one, two and the original value three. The model checker returns true for the delays two and three. When the delay is set to one, requirement ten is false.

In order to evaluate the failed requirement, the counterexample returned by the model checker is reflected on the original Stateflow model. The relevant part of the Stateflow model is shown in figure 6.1. Part of the trace is shown below:

01: ‘‘start FirstUnmute”
02: ‘‘exit on”
03: ‘‘FME := 2”
04: ‘‘enter DisplayUnMute”
05: ‘‘enter PossibleDisplay”
06: ‘‘Delay := 4”
07: ‘‘Delay := 3”
08: ‘‘FME := 1”
09: ‘‘Delay := 2”
10: ‘‘FME := 0”
11: ‘‘exit PossibleDisplay”
12: ‘‘enter awaitLast”
13: ‘‘Delay := 1”
14: ‘‘Delay := 0”
15: ‘‘start LastUnmute”
16: ‘‘exit awaitLas”
17: ‘‘LME := 14”
18: ‘‘enter uncertain”

Figure 6.1: State OnScreenDisplay.
CHAPTER 6. CASE STUDY: SOUND BEHAVIOUR

At the start of this trace state MuteOSD.on is active and event FistUnmute is about to be executed. This leads to the transition to state DisplayUnmute and the firstmuteentry delay FME is set to two in lines 2 until 5. Now the state ActiveAudio waits for Delay and OnScreenDisplay waits for FME. FME is set to zero before Delay, leading to a transition to state awaitLast in lines 10 until 12. The following LastUnmute event leads to the transition to state uncertain in lines 15 until 18.

From this point only a transition to state noDisplay is possible, thereby the state display is never entered and data object display is never set to two. If the delay in state ActiveAudio is finished before the delay in state OnScreenDisplay the transition from state PossibleDisplay to display would have been performed. This is the case when DisplayMin is set to the values two and three.

The difference between UnmuteDelayMax and UnmuteDelayMin indicates the corresponding delay time in state ActiveAudio. This difference is two, therefore DisplayMin should be equal or bigger than the difference between UnmuteDelayMax and UnmuteDelayMin to make sure LastUnmute is broadcasted before the DisplayMin delay is finished.

6.5 Evaluating data object values

The sound behaviour model is used to evaluate the impact of delays on the correct behaviour. Instead of converting the model multiple times using different values for the delays, a summation operator or multiple choice operators can be used to automatically set a data object to a random value. The original initiation process looks like:

```
proc initiation=
  . SUO_enter([])
  . stateflow

;)
```

This process can be extended to:

```
proc initiation=
  ( sWrite(UDM,2)
    + sWrite(UDM,4)
    + sWrite(UDM,6)
  )
  . SUO_enter([])
  . stateflow

;)
```

This new initiation process first randomly set constant data object UnmuteDelayMin (UDM) to value two, four or six. Next, the initiation is performed as normal. Model formulas will be checked for all values of this data object.

When a requirement fails on the new model, the first action in the feedback trace would be the write action that indicates which value causes the failure. Next the nature of the failure can be examined.
6.6 Conclusions

The verification of this model did not lead to fairness problems. This is because the mCRL2 models created with the conversion from Stateflow behave sequential. There are two parallel processes in the mCRL2 model of which one only contains actions that communicate with actions in the other process. That process cannot independently execute in a loop.

The second observation about the verification of the generated mCRL2 code concerns the feedback of the model checker. The trace returned by the model checker is long. This is because an event in a Stateflow model triggers a large sequence of actions and hence traces in which a number of events are broadcasted will be a concatenation of all the sequences of actions caused by the events. However, these long traces can be interpreted quickly by looking at the start event actions only. Knowing the broadcasted events shows the path to error situations.

Verifying an average modal formula on the original model takes one minute. After reducing the state space by removing \( \tau \)-actions, that verification takes ten seconds.

When using multiple values for data objects, the size of the state space grows fast. The initiation process used as an example in this chapter (i.e. three different values for one data object) led to a statespace with 234698 states, against the original 73755 states. Using this technique could be fatal for large statespaces, but offers an efficient technique to find wrong behaviour at the delays.
Chapter 7

Case study: TV behaviour

7.1 Introduction

In the previous chapter a case study of the sound behaviour of a TV is verified. This chapter introduces another model that represents the abstract behaviour of an entire TV system. This TV model is larger than the sound behaviour model used in the previous chapters and shows that the conversion can be used on larger real life models.

First a short overview of the Stateflow model is presented, followed by the details of the conversion to a mCRL2 model. Then a number of requirements are specified and verified.

7.2 Stateflow implementation

This section provides a general overview of the Stateflow implementation of the TV model. Only the tree representations of the complete implementation will be shown. Because the verification of this model will be focused on the sound behaviour of the TV, only the relevant parts of the TV are completely explained.

When a state name in the Stateflow model is not unique, the unique name used in the mCRL2 specification is denoted below the name used in the Stateflow specification.

7.2.1 TV environment

Inputs

The inputs of the TV system consist of buttons on the TV itself and on the remote controller. Table 7.1 displays the inputs. Inputs that are not used by the model are not shown in the table. All inputs are implemented in the Stateflow model as events.

Outputs

The output of the TV can be split into three parts, video, audio and the on-screen information display (OSD). The video display is represented by the data objects, ScreenMode and TVMode. The data object ScreenMode indicates if
CHAPTER 7. CASE STUDY: TV BEHAVIOUR

<table>
<thead>
<tr>
<th>Input name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainsOnOff</td>
<td>Turning the TV on and off.</td>
</tr>
<tr>
<td>Standby</td>
<td>When the TV is on, Standby switches between standby and active video.</td>
</tr>
<tr>
<td>Zero . . . Nine</td>
<td>The digits will select a channel or teletext page.</td>
</tr>
<tr>
<td>ChanStepUp</td>
<td>Step up one channel or teletext page.</td>
</tr>
<tr>
<td>ChanStepDown</td>
<td>Step down one channel or teletext page.</td>
</tr>
<tr>
<td>MenuOnOff</td>
<td>Activates/deactivates the menu.</td>
</tr>
<tr>
<td>TxtOnOff</td>
<td>Activates/deactivates teletext.</td>
</tr>
<tr>
<td>PipDsOnOff</td>
<td>Switches between single screen and split screen.</td>
</tr>
<tr>
<td>Display</td>
<td>Show/hide an information display on screen.</td>
</tr>
<tr>
<td>MuteOnOff</td>
<td>Mute/unmute the sound of the TV.</td>
</tr>
<tr>
<td>VolumeUp</td>
<td>Increases/unmutes the volume of the TV.</td>
</tr>
<tr>
<td>VolumeDown</td>
<td>Decreases/unmutes the volume of the TV.</td>
</tr>
</tbody>
</table>

Table 7.1: Inputs of the TV system

the TV screen operates in single screen, dual screen or is turned off. The data object TVMode indicates if the single screen or one of the dual screens shows either video, teletext or the TV menu. Data objects chanr and txtnr indicates the current channel or teletext page that has to be shown.

The audio is represented by one data object volume. Volume is a natural number that indicates the current volume of the TV. There are four different kinds of OSDs. Every OSD is represented by one data object. Table 7.2 shows the data object and the corresponding function of the OSD.

<table>
<thead>
<tr>
<th>Data object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOSDDisplay</td>
<td>Single Controll OSD (SCOSD) shows the current volume every time data object volume changes.</td>
</tr>
<tr>
<td>StatusDisplay</td>
<td>Status OSD shows information about the volume and channel number when button Display is pressed.</td>
</tr>
<tr>
<td>PresetDisplay</td>
<td>Preset OSD shows information about the channel number. Depending on the settings, this OSD is displayed permanently or temporarily after a change of channel.</td>
</tr>
<tr>
<td>MuteDisplay</td>
<td>Mute display shows a mute message when the TV is muted. When the TV is unmuted a unmute message is shown for a short time.</td>
</tr>
</tbody>
</table>

Table 7.2: OSD outputs of the TV system

7.2.2 Top level states

Figure 7.1 shows the top levels of the TV model. The state TV represents the entire TV. The TV is either On or Off. When the TV is on, i.e. state On is active, states InputTransform and TVModes are also active.

State InputTransform processes a number of inputs in order to control the TV. This is performed by generating Statelflow events and changing data objects.

State TVModes represents a sleeping or active TV. When the TV is active,
Figure 7.1: Overview of the TV model.
i.e. state *Active* is active, there are three parts that controls the output of the TV, represented by states *Video*, *Audio* and *OnScreenDisplay*. The subtrees of these states are shown in separate tree representations.

### 7.2.3 Video control

The state *Video* and its descendants are responsible for the video output of the TV. Figure 7.2 shows the subtree.

![Figure 7.2:](image)

### 7.2.4 Audio control

The state *Audio* and its descendants are responsible for the audio output of the TV. Figure 7.3 shows the subtree and figure 7.4 shows state *Audio* and its descendants of the Stateflow model. State *TVSpeakers* is responsible for muting and unmuting the TV and state *Volume* is responsible for the volume level when the TV is not muted.

![Figure 7.3:](image)
7.2.5 OSD control

The state *OSD* and its descendants are responsible for the OSD output of the TV. Figure 7.6 shows the subtree. State *OnScreenDisplay* has four parallel child states that are responsible for the four different OSDs.

*state MuteOSD*

The Stateflow model of state *MuteOSD* is shown in figure 7.5.
Figure 7.6:
7.3 mCRL2 implementation

The mCRL2 code is generated using the conversion from chapter 5. Some parts of the conversion are not explained in that chapter. This section first discusses the custom conversions of this specification. Next some optimisation this statespace are included.

7.3.1 Custom conversions

Delays and timing

The TV model uses timestamps to implement delays in the Stateflow model. This is implemented in mCRL2 as described in chapter 5.11.1. Four time-stamps are used at the four OSDs. The data objects that are used to store the timestamps are \texttt{SCOSDentry}, \texttt{Statusentry}, \texttt{Presetentry} and \texttt{Muteentry}.

Constant values for data objects

Like the mCRL2 code for the sound behaviour model, data objects with constant values are not stored at the data process. This is described in chapter 5.11.1. In the TV model, data objects that are used as outputs and are never read by the model itself are also not stored in the data process. For these data objects, mCRL2 actions are created and executed when a value is written. This way the assignments to these data objects can be detected while model checking.

Besides a small reduction of the statespace, not storing these values in the data process results into less process parameters when simulating the TV model. This

The \texttt{ml\_mod} function

In the TV model the custom Matlab function \texttt{ml\_mod} is used to calculate the modulo of an integer. When the integer is negative, the function results into the devider minus one.

The mCRL2 code of the of the Matlab function is shown below. It requires two integer inputs, the divident and the divisor. The result is an integer value.

\begin{verbatim}
map ml_mod: Int # Int -> Int;
var y,x:Int;

eqn ( x>=y ) -> ml_mod(x,y) = ml_mod(x-y,y) ;
( (x>=0) && (x<y) ) -> ml_mod(x,y) = max(x,0) ;
( x<0 ) -> ml_mod(x,y) = max(y-1,0) ;
\end{verbatim}

7.3.2 Optimising the model

It is very time consuming to generate the complete state space from the current mCRL2 model. In order to reduce the state space, two changes are made to the mCRL2 model. These changes are
Using natural numbers

After linearising the mCRL2 model and before generating the state space, the LPS file can be optimised for state space generation. This requires the summands in the model to be over natural numbers instead of integers. Because all the stored values of the data objects consists of natural numbers, the data process is changed to use naturals instead of integers. This required the type declarations to be changed from Int to Nat.

The code for the custom Matlab function \textit{ml\_mod} is changed to the following code:

\begin{verbatim}
map ml_mod: Int \# Nat -> Nat;
var y,x:Nat;

eqn ( x\geq\text{y} ) -> ml_mod(x,y) = ml_mod(x-y,y) ;
( (x\geq0) \&\& (x<y) ) -> ml_mod(x,y) = \text{max}(x,0) ;
( x<0 ) -> ml_mod(x,y) = \text{max}(y-1,0) ;
\end{verbatim}

Error Using the new mCRL2 code, the generation of the state space leads to an infinite state space. The problem appeared to be in the \textit{ml\_mod} function. The dividend is still an integer because during the calculation integer values are used. Variable \textit{x} is a natural number that is used in the equations to represent the dividend, therefore only allowing the equations to be applied to natural numbers. When a negative value was used as dividend, the function was not rewritten to an integer value. The function call was stored as a natural value. The \textit{ml\_mod} function was called infinite times on the stored function call. This can be solved by changing the variable declarations to the following code:

\begin{verbatim}
var y:Nat;
x:Int;
\end{verbatim}

Broadcast events to subtrees

A Stateflow model broadcasts an event to the entire model. Because most events only trigger behaviour in a limited part of the model, these events can be broadcasted to part of the Stateflow model. This reduces the number of checks for actions and transitions that do not result in actual behaviour of the model. Omitting these checks leads to less actions and to a smaller state space.

7.4 System requirements

The TV model represents the behaviour of the entire TV instead of the partly implemented TV in the sound behaviour model from previous chapter. This also means that there are more requirements available for the TV model. Because the goal is to check the feasibility of this method, not all of these requirements will be verified. The requirements from the sound behaviour model in the previous chapter are extended and verified for this model.
7.4. SYSTEM REQUIREMENTS

7.4.1 System overview

The sound requirements for the TV model are based on five input and two outputs.

The inputs are implemented as the events \textit{VolumeUp}, \textit{VolumeDown}, \textit{MuteOnOff}, \textit{MainOnOff} and \textit{Standby}. The events \textit{VolumeUp}, \textit{VolumeDown} and \textit{MuteOnOff} are the same inputs used in the sound behaviour model. The events \textit{MainOnOff} and \textit{Standby} are used to turn the TV on and off.

The outputs are implemented as the data objects \textit{volume} and \textit{MuteDisplay} and are represented by integer values. Output \textit{volume} is equal to the data object in the sound behaviour model and can hold any integer value. Output \textit{MuteDisplay} is equal to the data object \textit{display} in the sound behaviour model and can hold three different integer values which are shown in table 7.3.

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No message is shown.</td>
</tr>
<tr>
<td>1</td>
<td>Mute message is shown.</td>
</tr>
<tr>
<td>2</td>
<td>Unmute message is shown.</td>
</tr>
</tbody>
</table>

Table 7.3: Possible values for data object display.

Predefined values

Besides these in and outputs, the requirements use two predefined values, \textit{MinVol} and \textit{MaxVol}. These values represent the minimum volume and maximum volume for the system. This way the requirements can be generic for models with a different amount of volume levels. During this verification, \textit{MinVol} is set to zero and \textit{MaxVol} is set to two.

States

The system has three states that determine the behaviour of the system, \textit{Muted}, \textit{Unmuted} and \textit{Disabled}. The system is always considered to be in either one of these states at any time. The state may change during the execution of the system.

The requirements are grouped based on these states. The global requirements must hold in all of the states, the unmuted requirements that must hold in the state \textit{Unmuted} and the muted requirements that must hold in the \textit{Muted} state.

The TV is considered \textit{disabled} when Stateflow state \textit{Active} is not active.

7.4.2 Requirements

Tables 7.4, 7.5, 7.6 shows the Global, Unmuted and Muted requirements.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>When the system leaves the state Disabled, the system becomes unmuted</td>
</tr>
<tr>
<td>R02</td>
<td>\textit{volume} is never less than \textit{MinVol} or greater than \textit{MaxVol}.</td>
</tr>
</tbody>
</table>

Table 7.4: Global requirements
**CHAPTER 7. CASE STUDY: TV BEHAVIOUR**

| R03 | The MuteOnOff button will cause a transition to state muted. |
| R04 | The VolumeUp and VolumeDown button will not change the state to muted. |
| R05 | If volume is less than MaxVol, the VolumeUp button increases volume. |
| R06 | If volume is greater than MinVol, the VolumeDown button decreases volume. |
| R10 | When the system becomes *unmuted* after being *Muted*, unmuteMessage is eventually shown, after a finite time the message is removed. |
| R11 | When the system becomes *unmuted* after being *Muted* and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system. |
| R12 | When the system becomes *unmuted* after being *Muted* and the transition is caused by the VolumeUp button, the volume is eventually set to the increased original value of the unmuted system. |
| R13 | When the system becomes *unmuted* after being *Muted* and the transition is caused by the VolumeDown button, the volume is eventually set to the decreased original value of the unmuted system. |

Table 7.5: Unmute requirements

| R07 | When the system becomes *muted*, volume is eventually set to zero and it is not changed until the system becomes *unmuted*. |
| R08 | When the system becomes *muted*, muteMessage is eventually shown and it is not removed until the system becomes *unmuted*. |
| R09 | The VolumeUp, VolumeDown and MuteOnOff buttons will cause a transition to *unmuted*. |

Table 7.6: Mute requirements

### 7.5 Verification

This section discusses the results of the verification of the TV model. During this section the mCRL2 names for Stateflow states are used because these names are unique for every state. Modal formulas are described using the notation in chapter 6.3.

#### 7.5.1 Global requirements verification

The following requirements must hold at any time.

**Requirement 1**

When the system leaves the state Disabled, the system becomes *unmuted*.

When state *Active* is entered, the TV leaves the state *Disabled*. State SpeakerOn represents the stable situation in the Stateflow model where the
TV is *unmuted*. The modal formula checks if an entry of state *Active* is always followed by an entry of state *SpeakerOn*. The formula is shown below.

\[
[ \text{true*} \\
. \text{"enter Active"} \\
] \\
\text{mu X.} \\
( <\text{true}> \text{true} \\
\text{and ( [not "enter SpeakerOn"] X )} \\
)
\]

This requirement holds for this model.

**Requirement 2**

*volume* is never less than *MinVol* or greater than *MaxVol*.

This requirement is broken into two parts. Part *a*, volume never becomes negative, and part *b*, the volume is never bigger than *MaxVol*. The modal formulas for these parts are equal to the formulas used during the verification of requirement two in the sound behaviour model. This requirement holds for this model.

### 7.5.2 Unmuted requirements verification

The following requirements must hold when the TV is *unmuted*.

**Requirement 3**

The *MuteOnOff* button will cause a transition to state *muted*.

If state *SpeakerOn* is active, the *MuteOnOff* event will cause a transition to state *Mute*. The modal formulas for these parts are equal to the formulas used during the verification of requirement three in the sound behaviour model. This requirement holds for this model.

**Requirement 4**

The VolumeUp and VolumeDown button will not change the state to muted.

This requirement is made stronger by creating a formula that checks that no event other than *MuteOnOff* leads to the entry of state *Mute*. This is represented by the following modal formula:

\[
[ \text{true*} \\
. \text{"enter SpeakerOn"} \\
. (\text{not "start MuteOnOff"})* \\
. \text{"enter Mute"} \\
]\text{false}
\]

The formula searches for all traces where state *SpeakerOn* becomes active followed by the entry of state *Mute* without a MuteOnOff event being broadcast in the meantime. Any of these traces are considered false. This requirement holds for this model.
CHAPTER 7. CASE STUDY: TV BEHAVIOUR

Requirement 5

If volume is less than MaxVol, the VolumeUp button increases volume.

Whenever state SpeakerOn is active and volume is less than MaxVol, the VolumeUp event will result in an increase of volume by one. The modal formula used to verify this behaviour is:

\[
\begin{align*}
[ \text{true}^* & \quad \text{"enter SpeakerOn"} \\
& \quad \text{(not "exit SpeakerOn")}* \\
& \quad \text{"volume := n"} \\
& \quad \text{(not ( "exit SpeakerOn"} \\
& \quad \quad \text{or "volume := x"} \\
& \quad \quad \text{)})} \\
& \quad \text{"start VolumeUp"} \\
& \quad \text{]} \\
& \quad \text{(mu X,} \\
& \quad \quad \text{(true} \\
& \quad \quad \quad \text{and [not "volume := n+1"] X} \\
& \quad \quad \text{)})
\end{align*}
\]

This formula is almost the same as the formula used for the sound behaviour model. The only change is the added or "volume := x" action. This action makes sure that the volume is not changed after it is set to n.

This formula has to be tested for all n that are greater than or equal to MinVol and less than MaxVol. This requirement holds in the model.

Requirement 6

If volume is greater than MinVol, the VolumeDown button decreases volume.

Whenever state SpeakerOn is active and volume is greater than MinVol, the VolumeDown event will result in a decrease of volume by one. The modal formula used to verify this behaviour is the same as the formula used for requirement 5, except that all possible paths have to end in a state where the volume is eventually set to n-1 and the VolumeUp event is replaced with a VolumeDown event.

\[
\begin{align*}
[ \text{true}^* & \quad \text{"enter SpeakerOn"} \\
& \quad \text{(not "exit SpeakerOn")}* \\
& \quad \text{"volume := n"} \\
& \quad \text{(not ( "exit SpeakerOn"} \\
& \quad \quad \text{or "volume := x"} \\
& \quad \quad \text{)})
\end{align*}
\]
This formula has to be tested for all \( n \) that are greater than \( \text{MinVol} \) and less than or equal to \( \text{MaxVol} \). This requirement holds in the model.

**Requirement 10**

When the system becomes **unmuted** after being **Muted**, **unmuteMessage** is eventually shown, after a finite time the message is removed.

This requirement is split up into two parts. Part a states that **display** becomes two after a transition from state **Mute** to state **SpeakerOn**. Part b states that whenever **display** becomes two, it will inevitably become either zero or one. The modal formula for part a would be:

\[
[ \text{true}^*. \ "\text{enter Mute}" \\
. (\text{not} \ "\text{exit Mute}")^* \\
. "\text{start MuteOnOff}" \\
] \\
( \mu X. \\
( <\text{true}> \text{true} \\
\text{and} [\text{not} \ "\text{volume} := n-1"] X \\
) \\
) \\
\]

This formula considers all paths that ends with the entry of state **SpeakerOn** after being in state **Mute**. This will exclude the entry of state **On** caused by the initialisation of the system. All these paths should end in a state the inevitably leads to a transition where display is set to two.

This requirement is not true. When state **MuteOff** (i.e. no mute message is shown) is active and the system changes from state **Mute** to state **SpeakerOn**, **MuteDisplay** is not shown. This situation is caused by a transition from state **MuteOn** to **MuteOff** as a result of a change in data object **app** while the system is muted.

The formula for part b would be:

\[
( \mu X. \\
 [ \text{true}^* \\
. "\text{MuteDisplay} := 2" \\
] \\
) \\
\]
CHAPTER 7. CASE STUDY: TV BEHAVIOUR

The formula for part b checks if all paths that ends with display being set to two, lead to a state where display will eventually be set to either zero or one. This part of the requirements does not hold in the model. When the TV is turned off while display is set to two, display is not set to zero or one. When extending the formula to accept data object screenmode being set to zero or one (i.e. the screen is turned off) as a successful removal of the unmute message, this part of the requirement does hold. The new modal formula is shown below.

\[
\text{<true> true}
\land \left\{ \text{not (}
\begin{array}{l}
\text{"write_MuteDisplay := 0"}
\lor \text{"write_MuteDisplay := 1"}
\end{array}
\right) \times
\begin{array}{l}
\text{"write_ScreenMode := 0"}
\lor \text{"write_ScreenMode := 1"}
\end{array}
\right) \times
\text{X}
\}\]

Requirement 11

When the system becomes unmuted after being Muted and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system.

Consider the system where state SpeakerOn is active and volume is set to \( n \), followed by a transition to state Mute. When the system in this situation has a transition back to state SpeakerOn, the volume is eventually set back to \( n \). In order to check this requirement on the model, the following formula is used:

\[
\text{(}
\begin{array}{l}
\text{[ true*}
\text{. "enter SpeakerOn"}
\text{. ( not "exit SpeakerOn" )*}
\end{array}
\end{array}
\text{. "volume := n"}
7.5. VERIFICATION

This pattern checks for all paths that enter state SpeakerOn, set the volume to n, then exit state SpeakerOn without changing the volume and enter state Mute followed by the broadcast of event MuteOnOff. These paths should lead to a state where the volume is inevitably set to value n. The action "volume := x" represents any action where volume is set to a value.

In order to check the requirement, the formula should be checked for all volume levels where n is set to the possible volume level smaller than MaxVol. This requirement is not true in the model. The feedback from the model checker shows that the data object vlevel, which is used to store the old volume level, is still changed when a VolumeUp or VolumeDown event is generated.

Requirement 12 and 13

When the system becomes unmuted after being Muted and the transition is caused by the VolumeUp button, the volume is eventually set to the increased original value of the unmuted system.

When the system becomes unmuted after being Muted and the transition is caused by the VolumeDown button, the volume is eventually set to the decreased original value of the unmuted system.

Both requirements assume that the TV is being unmuted using the VolumeUp or VolumeDown button. Because requirement 9 is not valid, i.e. the system is not being unmuted by the VolumeUp or VolumeDown button, these requirements cannot be applied to the system.

7.5.3 Muted requirements verification

The muted requirements must hold at any time the system is muted.

Requirement 7

When the system becomes muted, volume is eventually set to zero and it is not changed until the system becomes unmuted.
This requirement is split into two parts. Part a states that when \textit{Mute} becomes active, \textit{volume} is eventually set to zero. Part b states that when \textit{Mute} becomes active and the \textit{volume} is set to zero, the \textit{volume} is never set to another value than zero. The formula for part a would be:

\[
\begin{align*}
&\text{true}^* \\
&\text{"cWrite(TVSpeaker, 1)"} \\
&\text{(not "cWrite(TVSpeaker, 0)")}^* \\
&\text{"start(MuteOnOff)"} \\
&\mu X. \\
&\langle \text{true} \rangle \text{true} \\
&\text{and [not "write_volume(0)"] X} \\
&\end{align*}
\]

This formula states that all paths that end with a \textit{MuteOnOff} event that causes a transition to state \textit{Mute}, will inevitably lead to \textit{volume} being set to zero. This part holds in the TV behaviour model. The formula for part b would be:

\[
\begin{align*}
&\text{true}^* \\
&\text{"cWrite(TVSpeaker, 1)"} \\
&\text{(not "cWrite(TVSpeaker, 0)")}^* \\
&\text{"start(MuteOnOff)"} \\
&\text{(not "cWrite(TVSpeaker, 0)")}^* \\
&\langle \text{"write_volume(1)"} \\
&\text{| "write_volume(2)"} \rangle \\
&\text{false} \\
&\end{align*}
\]

The formula for part b state that all paths where state \textit{Mute} is entered and \textit{volume} is set to one or two are false. These formulas are true in the TV behaviour model.

\textbf{Requirement 8}

When the system becomes \textit{muted}, \textit{muteMessage} is eventually shown and it is not removed until the system becomes \textit{unmuted}.

This requirement is split into two parts. Part a states that when state \textit{Mute} is entered, the display is set to one. Part b states that when state \textit{Mute} is entered and \textit{display} is set to one, the \textit{display} is not set to any other value than one until state \textit{Mute} is exited. Part a is represented by the following modal formula:

\[
\begin{align*}
&\text{true}^* \\
&\text{"cWrite(TVSpeaker, 1)"} \\
&\text{(not "cWrite(TVSpeaker, 0)")}^* \\
&\text{"start(MuteOnOff)"} \\
&\end{align*}
\]
7.5. VERIFICATION

This formula states that all paths that end with a MuteOnOff event that causes a transition to state Mute, will inevitably lead to MuteDisplay being set to one. This requirement holds for this model.

The formula for part b would be:

\[ \text{true}^*. \text{"enter Mute"} \]
\[ \text{(not "exit Mute")*.} \]
\[ \text{( "MuteDisplay := 0"} \]
\[ \text{| "MuteDisplay := 2" } \]
\[\]
\[\text{false}\]

This formula states that all paths to a state where state Mute is active, cannot be followed by MuteDisplay being set to either zero or two. This part does not hold in the TV behaviour model. When the TV is muted, MuteDisplay can be set to zero when app changes its value. When all traces where app is changed are removed from all paths, the formula does hold. The alternative formula is shown below.

\[ \text{true}^*. \]
\[ \text{"enter Mute"} \]
\[ \text{(not ( "exit Mute"} \]
\[ \text{| or "start Change_app "} \]
\[\]
\[\text{)}\]
\[\text{)*} \]
\[ \text{( "MuteDisplay := 0"} \]
\[ \text{| "MuteDisplay := 2" } \]
\[\]
\[\text{false}\]

Requirement 9

The VolumeUp, VolumeDown and MuteOnOff buttons will cause a transition to unmuted.

This requirement states that whenever state Mute is active, the MuteOnOff, VolumeUp and VolumeDown events will cause a transition to state SpeakerOn. The modal formula for this requirement would be:

\[ \text{true}^*. \]
\[ \text{"enter Mute"} \]
\[ \text{(not "exit Mute")*} \]
The formula checks if all paths leading to a state where Mute is active, followed by a E event, ends in a state that will inevitably lead to the entry of state SpeakerOn. This requirement must be checked for E equals events MuteOnOff, VolumeUp and VolumeDown. This requirement only holds for the MuteOnOff event. The VolumeUp and VolumeDown do not have a transition from state Mute to SpeakerOn.

7.6 Conclusions

7.6.1 Final state space

At first the state space could not be generated. After using naturals to store data objects a state space of seventeen million states could be generated. Using the reduced event broadcasts, the resulting state space has close to nine million states. This state space is used for verification.

During the state space generation, confluent τ’s are removed. The commands used for the state space generation are displayed in appendix D together with the mCRL2 code and the formal requirements.

Modal formulas

During the verification of this model, the modal formulas of the sound behaviour model were not always compatible with the formulas for the TV model. Because the inputs and outputs of both systems are equal, reuse of the formulas was expected.

These different results are caused by the difference between the Active state-flow states and the states as defined for the requirements. The requirement states imply that the TV is unmuted right after a MuteOnOff input is detected, while Stateflow allows actions to be performed after an MuteOnOff event but before the actual state is entered.

Safeguard for the necessity modal operator

A necessity operator (i.e. for all paths in a modal formula) may be used for liveness properties. These modal formulas were extended with an extra formula that stated that at least one such path exists. For example, the formula:

\[
[ \text{true}^* \\
  \text{"enter Mute"} \\
  (\text{not "exit Mute"})^* \\
  \text{"start MuteOnOff"}
\]
7.6. CONCLUSIONS

\[
(\mu X. \\
(\langle true\rangle true \\
\quad \text{and} [\text{not } "MuteDisplay := 2"] X \\
\quad )
)\]

was extended with the formula:

\[
<true^* \\
. "enter Mute" \\
. (\text{not } "exit Mute")^* \\
. "start MuteOnOff" \\
> \\
true
\]

This extension does not check if any of the requirements holds, but it checks if there is at least one such path.

When creating the liveness modal formulas, most of the time the programmer assumes there will be at least one such path. If this is not the case, the requirement does not imply any good behaviour. When making an error in the modal formula, the extension will not be accepted and the programmer knows the formulation of the formula is not correct. This extension has prevented a lot of time searching for typos during the execution of the project.

Execution time

The verification of a successful modal formula took about thirty to sixty minutes each. When a modal formula did not hold in the model, a counter example was found in around five minutes. These long execution times caused the verification to slow down significantly. While working with models of this size, it is worth to look at state space reduction techniques.
Chapter 8

Concluding remarks

8.1 Introduction

The main goal of this master project was to check if the mCRL2 modelling
language can be used to verify Stateflow models. While the models from the case
studies are not completely verified, they do show that it is possible to evaluate
Stateflow models. This section looks back to the problems encountered during
this master project and describes possibilities for future work.

8.2 Converting Stateflow models to mCRL2

The first step in verifying Stateflow models is the conversion to mCRL2.

8.2.1 Creating a complete conversion

The conversion from Stateflow to mCRL2 described in this report is not com-
plete. Although the basic elements and semantics are implemented, there are
some parts that have currently no fixed conversion. These parts are:

- History junction objects
- External input data objects
- Early return logic
- Graphical functions
- Truth table functions

8.2.2 Automating the conversion

If Stateflow models have to be verified using mCRL2 on regular basis, automa-
tion of the conversion from Stateflow to mCRL2 is important. Manual conver-
sion easily leads to numerous errors in the code. Most behaviour of objects is
very static and easy to automate, but the action sequences and transition guards
are very dynamic and will be harder to convert automatically. Matlab functions
must be programmed by hand or avoided, because Matlab is too extensive to include in the automated conversion.

During the conversions for the two Stateflow case studies, a script was used to automatically generate the framework for the mCRL2 code of the Stateflow models. The tree of states, including the number of inner and outer transition objects of each state, must be used as input. The script generates the framework of all mCRL2 processes and transitions. From the output of this script, the model-specific details must be included manually. Model-specific details are all action sequences, the Events used trigger transitions, transition sources, transition guards and transition destinations.

The required input of the script, as well as the model specific details are available as clear text in the Matlab model file. From this file, all details needed for the conversion are available.

8.2.3 Evaluating different settings

As shown in case study two, the sound behaviour model, evaluating a model with a constant initialised with a range of numbers can be useful. In the future, including ranges of numbers on a data object during the automatic conversion might be useful. This will increase the state space of the model significantly and is therefore only useful for smaller Stateflow models.

8.3 Verifying the mCRL2 models

8.3.1 Standardised generation of modal formulas

Generating modal formulas to verify a Stateflow model can be very tricky. Between the reception of two Simulink events, a number of steps are performed. These steps are not necessarily performed in the order assumed by the programmer.

The real interesting situations are the ‘stable’ situations where the Simulink events are completely executed and no new Simulink event has started to execute. Some patterns of modal formulas may handle these important situations better than others. A general approach of generating modal formulas for Stateflow models can be very useful. Research on these patterns would be useful.

8.3.2 Modal formula generation tool

The current way of constructing modal formulas is very sensitive for errors. One wrong character in the formula can lead to trivially accepted or rejected formulas for a specific model. A tool that assists with the creation of modal formulas could help. A number of suggestions are described next.

Because all requirements are described in Stateflow terms, a tool that assists in writing modal formulas using those Stateflow terms instead of the crude mCRL2 action names can be very useful. The Matlab file containing the Stateflow model that can be used to generate the mCRL2 code, can also be used to gain all mCRL2 names of states, events and data objects that are used in the formulas. These names can be used to translate the nice modal formulas in Stateflow terms to the crude mCRL2 action names of mCRL2.
Another way to assist with the creation of modal formulas would be a spelling checker that verifies that all used action names are also used in the mCRL2 implementation of the Stateflow model.

8.3.3 Evaluating model checker feedback

When a modal formula is false for a particular model, the feedback of the model checker must be evaluated to reflect the error in the Stateflow model. The feedback is returned in the form of a trace of actions, possibly in a loop. These traces can become very large and hence are difficult to comprehend completely.

A solution to this might be a specific tool that reflects the feedback trace to the Stateflow model. The tool can show the tree representation of the Stateflow model, highlighting the current active states. The data objects and their current value might be shown as well. Next the feedback trace should be simulated by that tool. Because the actions that are performed in the trace indicate the change of states and data object values, the tool can keep track of the corresponding situation in the Stateflow model.

Because there are no choices in the traces, it should be easy to simulate the trace by stepping from action to action or even from broadcasted event to broadcasted event. This way the user does not have to keep track of the corresponding situation himself.

8.4 mCRL2 toolset

Case studies one and two are small models that can easily be generated and verified by the mCRL2 toolbox. The generation and verification can be performed by a user with limited knowledge about the toolbox. After creating the mCRL2 code, no optimisations were needed to obtain short execution times.

The third case study ran into the limits of the mCRL2 tools. During the generation and verification of this model, extended knowledge of the toolbox was required. This makes it more difficult to verify Stateflow models using mCRL2 without extensive learning about the mCRL2 tools.

8.4.1 Warnings

During the master project, a delay was caused by a wrongly typed variable. This variable was implemented as a Natural, while a Integer was required. This variable was used on a function mapping that required an integer as input. As a result of this error, all occurrences of the function using a negative input value are not rewritten to a smaller expression. This caused the state space to explode. Now, a warning is implemented in mCRL2 to indicate that a variable of the wrong kind is used for a rewrite rule.

Besides this warning, it might be useful to optionally generate a warning when any mCRL2 function is not completely rewritten to an instance of the output type of the formula (i.e. no other functions are called in the resulting expressing). Because some models do not need a function to be completely rewritten, this error should not always be generated.
Bibliography


## Appendix A

### mCRL2 operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive numbers</td>
<td>$1, 2, 3, \ldots$</td>
<td>$1, 2, 3, \ldots$</td>
</tr>
<tr>
<td>natural numbers</td>
<td>$0, 1, 2, \ldots$</td>
<td>$0, 1, 2, \ldots$</td>
</tr>
<tr>
<td>integers</td>
<td>$\ldots, -2, -1, 0, 1, 2, \ldots$</td>
<td>$\ldots, -2, -1, 0, 1, 2, \ldots$</td>
</tr>
<tr>
<td>equality</td>
<td>$\approx$</td>
<td>$==\quad$</td>
</tr>
<tr>
<td>inequality</td>
<td>$\neq$</td>
<td>$\neq\quad$</td>
</tr>
<tr>
<td>conditional</td>
<td>$if(...)\quad$</td>
<td>$if(...)\quad$</td>
</tr>
<tr>
<td>conversion</td>
<td>$A2B(\quad)$</td>
<td>$A2B(\quad)$</td>
</tr>
<tr>
<td>less than or equal</td>
<td>$\leq\quad$</td>
<td>$\leq\quad$</td>
</tr>
<tr>
<td>less than</td>
<td>$&lt;\quad$</td>
<td>$&lt;\quad$</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>$\geq\quad$</td>
<td>$\geq\quad$</td>
</tr>
<tr>
<td>greater than</td>
<td>$&gt;\quad$</td>
<td>$&gt;\quad$</td>
</tr>
<tr>
<td>maximum</td>
<td>$max(...)\quad$</td>
<td>$max(...)\quad$</td>
</tr>
<tr>
<td>minimum</td>
<td>$min(...)\quad$</td>
<td>$min(...)\quad$</td>
</tr>
<tr>
<td>absolute value</td>
<td>$abs(\quad)$</td>
<td>$abs(\quad)$</td>
</tr>
<tr>
<td>negation</td>
<td>$-\quad$</td>
<td>$-\quad$</td>
</tr>
<tr>
<td>successor</td>
<td>$succ(\quad)$</td>
<td>$succ(\quad)$</td>
</tr>
<tr>
<td>predecessor</td>
<td>$pred(\quad)$</td>
<td>$pred(\quad)$</td>
</tr>
<tr>
<td>addition</td>
<td>$+\quad$</td>
<td>$+\quad$</td>
</tr>
<tr>
<td>subtractions</td>
<td>$-\quad$</td>
<td>$-\quad$</td>
</tr>
<tr>
<td>multiplication</td>
<td>$\times\quad$</td>
<td>$\times\quad$</td>
</tr>
<tr>
<td>integer div</td>
<td>$\div\quad$</td>
<td>$\div\quad$</td>
</tr>
<tr>
<td>integer mod</td>
<td>$\mod\quad$</td>
<td>$\mod\quad$</td>
</tr>
<tr>
<td>exponentiation</td>
<td>$\exp(\quad)$</td>
<td>$\exp(\quad)$</td>
</tr>
</tbody>
</table>

Table A.1: Operations on numbers
### APPENDIX A. MCRL2 OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>negation</td>
<td>¬</td>
<td>!</td>
</tr>
<tr>
<td>conjunction</td>
<td>∧</td>
<td>&amp;</td>
</tr>
<tr>
<td>disjunction</td>
<td>∨</td>
<td></td>
</tr>
<tr>
<td>implication</td>
<td>⇒</td>
<td>=&gt;</td>
</tr>
<tr>
<td>equality</td>
<td>≈</td>
<td>==</td>
</tr>
<tr>
<td>inequality</td>
<td>≠</td>
<td>!=</td>
</tr>
<tr>
<td>conditional</td>
<td>if(.,.,.)</td>
<td>if(.,.,.)</td>
</tr>
<tr>
<td>universal quantification</td>
<td>∀ : .</td>
<td>forall : .</td>
</tr>
<tr>
<td>existential quantification</td>
<td>∃ : .</td>
<td>exists : .</td>
</tr>
</tbody>
</table>

Table A.2: Operations on booleans

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>construction</td>
<td>[.,.,.,.]</td>
<td>[.,.,.,.]</td>
</tr>
<tr>
<td>element test</td>
<td>∈</td>
<td>in</td>
</tr>
<tr>
<td>length</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>cons</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>snoc</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>concatenation</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>element at position</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>the first element of a list</td>
<td>head(.)</td>
<td>head(.)</td>
</tr>
<tr>
<td>list without its first element</td>
<td>tail(.)</td>
<td>tail(.)</td>
</tr>
<tr>
<td>the last element of a list</td>
<td>rhead(.)</td>
<td>rhead(.)</td>
</tr>
<tr>
<td>list without its last element</td>
<td>rtail(.)</td>
<td>rtail(.)</td>
</tr>
<tr>
<td>equality</td>
<td>≈</td>
<td>==</td>
</tr>
<tr>
<td>inequality</td>
<td>≠</td>
<td>!=</td>
</tr>
<tr>
<td>conditional</td>
<td>if(.,.,.)</td>
<td>if(.,.,.)</td>
</tr>
</tbody>
</table>

Table A.3: Operations on lists
<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>set enumeration</td>
<td>{ \ldots, }</td>
<td>{ \ldots, }</td>
</tr>
<tr>
<td>bag enumeration</td>
<td>{ : \ldots, : }</td>
<td>{ : \ldots, : }</td>
</tr>
<tr>
<td>comprehension</td>
<td>{ _ }</td>
<td>{ _ }</td>
</tr>
<tr>
<td>element test</td>
<td>_ \in _</td>
<td>_ in _</td>
</tr>
<tr>
<td>bag multiplicity</td>
<td>count(_,_)</td>
<td>count(_,_)</td>
</tr>
<tr>
<td>subset/subbag</td>
<td>_ \subseteq _</td>
<td>_ \subseteq _</td>
</tr>
<tr>
<td>proper subset/subbag</td>
<td>_ \subset _</td>
<td>_ \subset _</td>
</tr>
<tr>
<td>union</td>
<td>_ \cup _</td>
<td>_ \cup _</td>
</tr>
<tr>
<td>difference</td>
<td>_ _</td>
<td>_ _</td>
</tr>
<tr>
<td>intersection</td>
<td>_ \cap _</td>
<td>_ \cap _</td>
</tr>
<tr>
<td>set complement</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>convert set to bag</td>
<td>Set2Bag(_)</td>
<td>Set2Bag(_)</td>
</tr>
<tr>
<td>convert bag to set</td>
<td>Bag2Set(_)</td>
<td>Bag2Set(_)</td>
</tr>
<tr>
<td>equality</td>
<td>_ \approx _</td>
<td>_ \approx _</td>
</tr>
<tr>
<td>inequality</td>
<td>_ \neq _</td>
<td>_ \neq _</td>
</tr>
<tr>
<td>conditional</td>
<td>if(_,_,_)</td>
<td>if(_,_,_)</td>
</tr>
</tbody>
</table>

Table A.4: Operations on sets and bags

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>function application</td>
<td>( \ldots, _ )</td>
<td>( \ldots, _ )</td>
</tr>
<tr>
<td>lambda abstraction</td>
<td>( \lambda D_0, \ldots, D_n : \ldots, : D_n )</td>
<td>\lambda D_0, \ldots, D_n : \ldots, : D_n</td>
</tr>
<tr>
<td>equality</td>
<td>_ \approx _</td>
<td>_ \approx _</td>
</tr>
<tr>
<td>inequality</td>
<td>_ \neq _</td>
<td>_ \neq _</td>
</tr>
<tr>
<td>conditional</td>
<td>if(_,_,_)</td>
<td>if(_,_,_)</td>
</tr>
</tbody>
</table>

Table A.5: Lambda abstraction and function application

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor of summation</td>
<td>i, ci(\ldots,_)</td>
<td>ci(\ldots,_)</td>
</tr>
<tr>
<td>recogniser for constructor</td>
<td>i, is-ci(_)</td>
<td>is-ci(_)</td>
</tr>
<tr>
<td>projection(i, j), if declared</td>
<td>prij(_)</td>
<td>prij(_)</td>
</tr>
<tr>
<td>equality</td>
<td>_ \approx _</td>
<td>_ \approx _</td>
</tr>
<tr>
<td>inequality</td>
<td>_ \neq _</td>
<td>_ \neq _</td>
</tr>
<tr>
<td>conditional</td>
<td>if(_,_)</td>
<td>if(_,_)</td>
</tr>
</tbody>
</table>

Table A.6: Operations on structured types
Appendix B

On-screen display

mCRL2 code

\[
\text{sort } \text{state} = \text{struct active?is}_\text{active} \\
| \text{suppressed?is}_\text{suppressed} \\
| \text{inactive?is}_\text{inactive} \\
| \text{removed?is}_\text{removed}
\]

\text{act } \text{keyDisplay, statusTime; } \\
\text{showStatus : Bool; } \\
\text{act } \text{changeChan, presetTime; } \\
\text{showPreset : Bool; } \\
\text{rMode : Bool; } \\
\text{act } \text{sStatus, rStatus, cStatus : Bool; } \\
\text{sPreset, rPreset, cPreset; } \\
\text{rApp, cApp : Bool; } \\
\text{sSC2S, rSC2S; } \\
\text{sSC2P, rSC2P; }
\textbf{proc} \textit{statusOSD} (stStatus : state) =
\begin{align*}
\text{is\_inactive}(stStatus) & \rightarrow \big( \\
& \text{keyDisplay.sStatus(true).rPreset} \\
& \text{.showStatus(true).statusOSD(active)} \\
& + \text{rApp(false).statusOSD(stStatus)} \\
& + \text{rApp(true).statusOSD(removed)} \\
\big) \\
+ \text{is\_active}(stStatus) & \rightarrow \big( \\
& \text{keyDisplay.showStatus(false).sStatus(false)} \\
& \text{.rPreset.statusOSD(inactive)} \\
& + \text{statusTime.showStatus(false).sStatus(false)} \\
& \text{.rPreset.statusOSD(inactive)} \\
& + \text{rSC2S.showStatus(false).sS2SC.sStatus(false)} \\
& \text{.rPreset.statusOSD(inactive)} \\
& + \text{rApp(false).statusOSD(stStatus)} \\
& + \text{rApp(true).showStatus(false).sStatus(false)} \\
& \text{.rPreset.statusOSD(removed)} \\
\big) \\
+ \text{is\_removed}(stStatus) & \rightarrow \big( \\
& \text{keyDisplay.statusOSD(removed)} \\
& + \text{r} \\
& + \text{rApp(true).statusOSD(removed)} \\
\big); \\
\end{align*}
proc \( \text{presetOSD} \) (\( \text{stPreset} : \text{state} \), \( \text{status} : \text{Bool} \), \( \text{app} : \text{Bool} \), \( \text{min} : \text{Bool} \)) =

\( \text{is\_inactive} \) (\( \text{stPreset} \)) → (\( \text{changeChan} \).status → \( \text{presetOSD} \)(suppressed, status, app, min)

\( \diamond \) \( \text{showPreset} \)(true).\( \text{presetOSD} \)(active, status, app, min)

) + ∑ \( b : \text{Bool} \) \( r\text{Status} \)(\( b \)).\( \text{sPreset} \).\( \text{presetOSD} \)(\( \text{stPreset}, b, \text{app}, \text{min} \))

) + \( \text{is\_active} \) (\( \text{stPreset} \)) → (\( r\text{Status} \)(true).\( \text{showPreset} \)(false).\( \text{sPreset} \).\( \text{presetOSD} \)(suppressed, true, app, min)

+ \( r\text{Status} \)(false).\( \text{sPreset} \).\( \text{presetOSD} \)(active, false, app, min)

+ \( r\text{SC2P} \).\( \text{showPreset} \)(false).\( \text{sP2SC} \).\( \text{presetOSD} \)(inactive, status, app, min)

+ \( \text{changeChan} \).\( \text{presetOSD} \)(\( \text{stPreset}, \text{status}, \text{app}, \text{min} \))

+ (\( \text{min} \lor \text{app} \)) → \( \text{presetTime} \).\( \text{showPreset} \)(false)

.\( \text{presetOSD} \)(inactive, status, app, min)

) + \( \text{is\_suppressed} \) (\( \text{stPreset} \)) → (\( r\text{Status} \)(true).\( \text{sPreset} \).\( \text{presetOSD} \)(suppressed, true, app, min)

+ \( r\text{Status} \)(false).\( \text{sPreset} \).\( \text{showPreset} \)(true)

.\( \text{presetOSD} \)(active, false, app, min)

+ \( \text{changeChan} \).\( \text{presetOSD} \)(\( \text{stPreset}, \text{status}, \text{app}, \text{min} \))

+ (\( \text{min} \lor \text{app} \)) → \( \text{presetTime} \).\( \text{presetOSD} \)(inactive, status, app, min)

) + ∑ \( b : \text{Bool} \) \( r\text{Mode} \)(\( b \)).\( \text{presetOSD} \)(\( \text{stPreset}, \text{status}, \text{app}, b \))

+ ∑ \( b : \text{Bool} \) \( r\text{App} \)(\( b \)).\( \text{presetOSD} \)(\( \text{stPreset}, \text{status}, b, \text{min} \));

init \( \tau_I (\nabla_V (\Gamma_C \left( \begin{array}{c}
\text{presetOSD}(\text{active}, \text{false}, \text{false}, \text{false}) \parallel \text{statusOSD}(\text{inactive})
\end{array} \right))) \);

\( C = \{ r\text{Status} \mid r\text{Status} \rightarrow c\text{Status}, r\text{App} \mid r\text{App} \rightarrow c\text{App},
\text{sPreset} \mid r\text{Preset} \rightarrow c\text{Preset} \} \)

\( V = \{ \text{all\_actions} \} / \{ s\text{Status}, r\text{Status}, r\text{App}, s\text{Preset}, r\text{Preset} \} \)

\( I = \emptyset \)
Appendix C

mCRL2 specification of example 2

mCRL2 code

```plaintext
sort Events = struct 

%%%%% Simulink events %%%%%
Clk?isClk 
| VolumeUp?isVolumeUp 
| VolumeDown?isVolumeDown 
| MuteOnOff?isMuteOnOff 

%%%%% Stateflow events %%%%%
| EMute?isEMute 
| FirstUnmute?isFirstUnmute 
| LastUnmute?isLastUnmute 

; 

sort ListInt = List(Int); 

sort V = struct 

%%%%% State information %%%%%
SUO 
| TV 
| ProcessKeys 
| Active 
```
| ActiveAudio  |
| AAOn        |
| Mute        |
| OnScreenDisplay |
| MuteOSD     |
| DisplayUnMute |

%%%%% Data objects %%%%%
| volume |
| vlevel |
| display |

%%%%% Data objects for delays %%%%%
| Delay |
| FME   |
| LME   |

map var2nat : V -> Nat;

eqn var2nat(SUO) = 0;
var2nat(TV) = 1;
var2nat(ProcessKeys) = 2;
var2nat(Active) = 3;
var2nat(ActiveAudio) = 4;
var2nat(AAOn) = 5;
var2nat(Mute) = 6;
var2nat(OnScreenDisplay) = 7;
var2nat(MuteOSD) = 8;
var2nat(DisplayUnMute) = 9;
var2nat(volume) = 10;
var2nat(vlevel) = 11;
var2nat(display) = 12;
var2nat(Delay) = 13;
var2nat(FME) = 14;
var2nat(LME) = 15;

map writeVar : Nat # Int # ListInt # ListInt -> ListInt;

var s,t:ListInt;
n: Nat;
m: Int;
b: Bool;

eqn ( #s==0 )->
writeVar(n,m,s,t) = t;
( (#s!0) && (n > #t) )->
writeVar(n,m,s,t) = writeVar(n,m,tail(s),t\<|head(s));
( (#s!0) && (n==#t) )->
writeVar(n,m,s,t) = writeVar(n,m,tail(s),t\<|m);
\[ ((s \neq 0) \&\& (n < t)) \rightarrow \]
\[ \text{writeVar}(n, m, s, t) = t+s; \]

\[ \text{act } s\text{Read }, r\text{Read }, c\text{Read} : V \text{ # Int}; \]
\[ \text{sWrite }, r\text{Write }, c\text{Write} : V \text{ # Int}; \]

\[ \text{proc data}(d: \text{ListInt}) = \]
\[ \quad \sum v: V. s\text{Read}(v, d.\text{var2nat}(v)).\text{data}(d) \]
\[ + \sum v: V, m: \text{Int}. (\text{var2nat}(v) < (#d)) \rightarrow (\]
\[ \quad \quad r\text{Write}(v, m).\text{data}(\text{writeVar}(\text{var2nat}(v), m, d, [])) \]
\[ ) \]

%%%%%%%%%%%%%%%%%%%
%%%%% Actions %%%%%
%%%%%%%%%%%%%%%%%%%

act intern;
start: Events;
%act test2: Int;
%act test;

%%%%%%%%%%%%%%%%%%%%%
%%%%% Initiation %%%%%
%%%%%%%%%%%%%%%%%%%%%

init hide({cRead, ml_buffer, ml_rgui, intern},
  block({
    sWrite, rWrite,
    sRead, rRead
  }),
  comm({
    sRead | rRead -> cRead,
    sWrite | rWrite -> cWrite
  }),
  initiation
  ||\text{data}([\]
  0 \% SUO
  ,0 \% TV
  ,0 \% ProcessKeys
  ,0 \% Active
  ,0 \% ActiveAudio
  ,0 \% AAOn
  ,0 \% Mute
  ,0 \% OnScreenDisplay
  ,0 \% MuteOSD
  ,0 \% DisplayUnMute
  ,0 \% volume
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

```mcrl2
;0 % vlevel
;0 % display
;0 % Delay
;0 % FME
;0 % LME
]
)
)
)
)
)
}
proc initiation=
SUO_enter([]).stateflow;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Stateflow process %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc stateflow=
  start(Clk).SUO_exec(Clk)
  . sum d:Int.(
    rRead(Delay,d)
    . ( (d>0)->( sWrite(Delay,d-1) )<>(intern) )
  )
  . sum d:Int.(
    rRead(FME,d)
    . ( (d>0)->( sWrite(FME,d-1) )<>(intern) )
  )
  . sum d:Int.(
    rRead(LME,d)
    . ( (d>0)->( sWrite(LME,d-1) )<>(intern) )
  )
  . stateflow;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Execution processes %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc SUO_exec(event:Events)=
  sum state:Int.rRead(SUO,state).( (state == 0)->( intern )
  + (state == 1)->( % State: TV
    TV_exec(event) )
)
proc TV_exec(event:Events)=
sum state:Int.rRead(TV,state).(
   (state == 0)->(
      intern
   )
   + (state > 0)->(
      ProcessKeys_exec(event) % State: ProcessKeys
      . Active_exec(event) % State: Active
   )
);

proc ProcessKeys_exec(event:Events)=
sum state:Int.rRead(ProcessKeys,state).(
   (state == 0)->(
      intern
   )
   + (state == 1)->( % State: awaitKey
      sum d:Int.rRead(Delay,d).(
         ( isClk(event) && d==0 )->(
            ProcessKeys_exit
            . ml_buffer(1)
            . ( intern
            + start(MuteOnOff).SUO_exec(MuteOnOff)
            + start(VolumeUp).SUO_exec(VolumeUp)
            + start(VolumeDown).SUO_exec(VolumeDown)
            )
            . ProcessKeys_enter([1])
         )<(( isClk(event) && ( d > 0 ) )->(
            ProcessKeys_exit
            . ProcessKeys_enter([2])
         )<(
            intern
         ))
   )
   + (state == 2)->( % State: Delay
      sum d:Int.rRead(Delay,d).( isClk(event) && ( d == 0 ))->(
         ProcessKeys_exit
         . ProcessKeys_enter([1])
      )<(
         intern
      )
   )
);

proc Active_exec(event:Events)=
sum state:Int.rRead(Active,state).(
   (state == 0)->(
   )
);
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

intern
)
+ (state > 0)->( 
  ActiveAudio_exec(event) % State: ActiveAudio
  . OnScreenDisplay_exec(event) % State: OnScreenDisplay
)
);

proc ActiveAudio_exec(event:Events)=
  sum state:Int.rRead(ActiveAudio,state).( 
    (state == 0)->( 
      intern
    )
    + (state == 1)->( % State: AAOn
      sum d:Int.rRead(vlevel,d).(
        ( isVolumeUp(event) && ( d<10 ) )->( %MaxVol
          ActiveAudio_exit
          . sWrite(vlevel,d+1).sWrite(Delay,2) %VolDelayMax
          . AAOn_enter([3])
        )<>(( isVolumeDown(event) && (d>0) )->( %MinVol
          ActiveAudio_exit
          . sWrite(vlevel,d-1).sWrite(Delay,2) %VolDelayMax
          . AAOn_enter([3])
        )<>(( isMuteOnOff(event) )->(
          ActiveAudio_exit
          . sWrite(Delay,4) %MuteDelayMax
          . AAOn_enter([2])
        )<>(( AAOn_exec(event) )))
    )
    )
  )
  + (state == 2)->( % State: Mute
    sum v:Int.rRead(vlevel,v).(
      ( isMuteOnOff(event) )->(
        ActiveAudio_exit
        . sWrite(Delay,2) %UnmuteDelayMin
        . AAOn_enter([4])
      )<>(( isVolumeDown(event) && (v>0) )->( %MinVol
        ActiveAudio_exit
        . sWrite(vlevel,v-1).sWrite(Delay,2) %UnmuteDelayMin
        . AAOn_enter([4])
      )<>(( isVolumeUp(event) && (v<10) )->( %MaxVol
        ActiveAudio_exit
        . sWrite(vlevel,v+1).sWrite(Delay,2) %UnmuteDelayMin
        . AAOn_enter([4])
      )<>(( isVolumeUp(event) && (v>=10) )->( %MaxVol
        ActiveAudio_exit
      )))
    )
  )
);
proc AAOn_exec(event:Events)=
  sum state:Int.rRead(AAOn,state).( state == 0)->(
    intern
  ) + (state == 1)->( state: UnMute
    intern
  ) + (state == 2)->( state: PossibleMute
    sum d:Int.rRead(Delay,d).( isClk(event) && (d==0) )->(
      AAOn_exit
      . start(LastUnmute).SUO_exec(LastUnmute)
      . AAOn_enter([1])
    )<>
      intern
  ) + (state == 3)->( state: VolChangeDelay
    sum d:Int.rRead(Delay,d).( isClk(event) && (d==0) )->(
      AAOn_exit
      . AAOn_enter([1])
    )<>
      intern
  ) + (state == 4)->( state: ShortestUnmute
    sum d:Int.rRead(Delay,d).( isClk(event) && (d==0) )->(
      AAOn_exit
      . start(FirstUnmute).SUO_exec(FirstUnmute)
      . sWrite(Delay,4) %UnmuteDelayMax - UnmuteDelayMin
      . AAOn_enter([2])
    )<>
      intern
  );
proc Mute_exec(event:Events)=
sum state:Int.rRead(Mute,state).(
(state == 0)->(  
   intern  
  )
+ (state == 1)->( % State: MuteDelay  
   sum d:Int.rRead(Delay,d).( isClk(event) && (d==0) )->(  
      Mute_exit  
      . Mute_enter([2])  
    )<>(  
      intern  
     )
 )
+ (state == 2)->( % State: MuteState  
   intern  
  ));

proc OnScreenDisplay_exec(event:Events)=
sum state:Int.rRead(OnScreenDisplay,state).(
(state == 0)->(  
   intern  
  )
+ (state == 1)->( % State: MuteOSD  
   MuteOSD_exec(event)  
  ));

proc MuteOSD_exec(event:Events)=
sum state:Int.rRead(MuteOSD,state).(
(state == 0)->(  
   intern  
  )
+ (state == 1)->( % State: noDisplay  
   ( isEMute(event) )->(  
      MuteOSD_exit  
      . MuteOSD_enter([2])  
    )<>(  
      intern  
     )
 )
+ (state == 2)->( % State: On  
   ( isFirstUnmute(event) )->(  
      MuteOSD_exit  
      . sWrite(FME,6) %DisplayMin  
      . MuteOSD_enter([3])  
    )<>(  
      intern  
     )
)
+ (state == 3)->( % State: DisplayUnMute
    ( isEMute(event) )->(
        MuteOSD_exit
        . MuteOSD_enter([2])
    )><(
        DisplayUnMute_exec(event)
    )
)
)

proc DisplayUnMute_exec(event:Events)=
    sum state:Int.rRead(DisplayUnMute,state).( 
        (state == 0)->(
            intern
        )
    )
    + (state == 1)->( % State: PossibleDisplay
        sum f:Int.rRead(FME,f).( isClk(event) && (f==0) )->(
            DisplayUnMute_exit
            . DisplayUnMute_enter([3])
        )><(( isLastUnmute(event) )->(
            DisplayUnMute_exit
            . sWrite(LME,14) %DisplayMax
            . DisplayUnMute_enter([2])
        )><(
            intern
        )
    )
    + (state == 2)->( % State: Display
        sum f:Int.rRead(FME,f).( isClk(event) && (f==0) )->(
            DisplayUnMute_exit
            . DisplayUnMute_enter([4])
        )><(
            intern
        )
    )
    + (state == 3)->( % State: awaitLast
        ( isLastUnmute(event) )->(
            DisplayUnMute_exit
            . sWrite(LME,14) %DisplayMax
            . DisplayUnMute_enter([4])
        )><(
            intern
        )
    )
    + (state == 4)->( % State: uncertain
        sum l:Int.rRead(LME,l).( isClk(event) && (l==0) )->(
            MuteOSD_exit
            . MuteOSD_enter([1])
        )><(
            intern
        )
    )
% Entry processes

act m1_buffer, m1_rcgui : Nat;

proc SUO_enter(ln : ListInt) =
  (#ln == 0) -> (intern.SUO_enter([1])<>
    (head(ln) == 1) -> (
      % State: TV
      sWrite(SUO,1)
      . m1_buffer(0). m1_rcgui(0)
      . TV_enter(tail(ln))
    ));

proc TV_enter(ln : ListInt) =
  sum n : Nat.rRead(SUO,n).(n != 1) -> (SUO_enter(1 |> ln)
    )<>
    ( #ln == 0) -> (intern.TV_enter([1])
    )<>
    (sWrite(TV,1)
      . ProcessKeys_enter(tail(ln))
      . sum new : Nat.rRead(TV,new).(new == 1) -> (sWrite(TV,2)
        . Active_enter(tail(ln))
    )<>
    intern
  ));

proc ProcessKeys_enter(ln : ListInt) =
  sum n : Nat.rRead(TV,n).(n < 1) -> (TV_enter(1 |> ln)
    )<>
    ( #ln == 0) -> (intern.ProcessKeys_enter([1])
    )<>
    (head(ln) == 1) -> ( % State: awaitKey
      sWrite(ProcessKeys,1)
    ) + (head(ln) == 2) -> ( % State: Delay
    )
  )
});
sWrite(ProcessKeys,2)
)
);

proc Active_enter(ln:ListInt)=
  sum n:Nat.rRead(TV,n).(n < 2)->(
    TV_enter(2 |> ln)
  )<>(( #ln == 0)->(
    intern.Active_enter([1])
  )<>(
    sWrite(Active,1)
    . ActiveAudio_enter(tail(ln))
    . sum new:Nat.rRead(Active,new).(new == 1)->(
      sWrite(Active,2)
      . OnScreenDisplay_enter(tail(ln))
    )<>(
      intern
    )
  ));

proc ActiveAudio_enter(ln:ListInt)=
  sum n:Nat.rRead(Active,n).(n < 1)->(
    Active_enter(1 |> ln)
  )<>(( #ln == 0)->(
    intern.ActiveAudio_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: AAOn
      sWrite(ActiveAudio,1)
      . AAOn_enter(tail(ln))
    )
    + (head(ln) == 2 )->( % State: Mute
      sWrite(ActiveAudio,2)
      . Mute_enter(tail(ln))
    )
  ));

proc AAOn_enter(ln:ListInt)=
  sum n:Nat.rRead(ActiveAudio,n).(n != 1)->(
    ActiveAudio_enter(1 |> ln)
  )<>(( #ln == 0)->(
    intern.AAOn_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: UnMute
      sWrite(AAOn,1)
      . sum i:Int.(rRead(vlevel,i).sWrite(volume,i))
    )
    + (head(ln) == 2 )->( % State: PossibleMute
      sWrite(AAOn,2)
    )
    + (head(ln) == 3 )->( % State: VolChangeDelay

  ));
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

```mcrl2
sWrite(AAOn,3)
)
+ (head(ln) == 4 )->( % State: ShortestUnmute
  sWrite(AAOn,4)
  );
)

proc Mute_enter(ln:ListInt)=
  sum n:Nat.rRead(ActiveAudio,n).(n != 2)->(
    ActiveAudio_enter(2 |> ln)
  )<>(( #ln == 0 )->( 
    intern.Mute_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: MuteDelay
      sWrite(Mute,1)
    )
    + (head(ln) == 2 )->( % State: MuteState
      sWrite(Mute,2)
    )
  ));

proc OnScreenDisplay_enter(ln:ListInt)=
  sum n:Nat.rRead(Active,n).(n < 2)->(
    Active_enter(2 |> ln)
  )<>(( #ln == 0 )->( 
    intern.OnScreenDisplay_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: MuteOSD
      sWrite(OnScreenDisplay,1)
      . MuteOSD_enter(tail(ln))
    )
  ));

proc MuteOSD_enter(ln:ListInt)=
  sum n:Nat.rRead(OnScreenDisplay,n).(n != 1)->(
    OnScreenDisplay_enter(1 |> ln)
  )<>(( #ln == 0 )->( 
    intern.MuteOSD_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: noDisplay
      sWrite(MuteOSD,1)
      . sWrite(display,0)
    )
    + (head(ln) == 2 )->( % State: On
      sWrite(MuteOSD,2)
      . sWrite(display,1)
    )
    + (head(ln) == 3 )->( % State: DisplayUnMute
      sWrite(MuteOSD,3)
      . DisplayUnMute_enter(tail(ln))
  ));
```
proc DisplayUnMute_enter(ln:ListInt)=
sum n:Nat.rRead(MuteOSD,n).(n != 3)->(
    MuteOSD_enter(3 |> ln)
)<>( (#ln == 0)->(
    intern.DisplayUnMute_enter([1])
)<>(
    (head(ln) == 1)->( % State: PossibleDisplay
        sWrite(DisplayUnMute,1)
    )
    + (head(ln) == 2 )->( % State: Display
        sWrite(DisplayUnMute,2)
        . sWrite(display,2)
    )
    + (head(ln) == 3 )->( % State: awaitLast
        sWrite(DisplayUnMute,3)
    )
    + (head(ln) == 4 )->( % State: uncertain
        sWrite(DisplayUnMute,4)
    )
    )
));

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Exit processes %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc SUO_exit=
sum state:Nat.rRead(SUO,state). ( (state == 0)->(
    intern
) + (state == 1 )->( %State: TV
    TV_exit
    . sWrite(SUO,0)
    )
);

proc TV_exit=
sum state:Nat.rRead(TV,state). ( (state == 0)->(
    intern
) + (state == 1 )->( %State: ProcessKeys
    ProcessKeys_exit
    )
    + (state == 2 )->( %State: Active
    Active_exit
    )
    ));
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

```ml
). sum new:Nat.rRead(TV,new).(new == state)->(  sWrite(TV,state-1)  . (1<state)->(TV_exit)<>(intern) )<>(  intern  )
);

proc ProcessKeys_exit=
  sum state:Nat.rRead(ProcessKeys,state).(  (state == 0)->(  intern  ) + (state == 1)->( %State: awaitKeyCarl  sWrite(ProcessKeys,0) ) + (state == 2 )->( %State: DelayCarl  sWrite(ProcessKeys,0) ) )
);

proc Active_exit=
  sum state:Nat.rRead(Active,state).(  (state == 0)->(  intern  ) + (state == 1 )->( %State: ActiveAudio  ActiveAudio_exit  ) + (state == 2 )->( %State: OnScreenDisplay  OnScreenDisplay_exit  ) )
  . sum new:Nat.rRead(Active,new).(new == state)->(  sWrite(Active,state-1)  . (1<state)->(Active_exit)<>(intern) )<>(  intern  )
);

proc ActiveAudio_exit=
  sum state:Nat.rRead(ActiveAudio,state).(  (state == 0)->(  intern  ) + (state == 1 )->( %State: AAOn  AAOn_exit  ) + (state == 2 )->( %State: AAOff  AAOff_exit  )  )
```
. sWrite(ActiveAudio,0)
) + (state == 2 )->( %State: Mute
  Mute_exit
  . sWrite(ActiveAudio,0)
)
);

proc AAOn_exit=
  sum state:Nat.rRead(AAOn,state).( (state == 0)->( intern
)
  + (state == 1 )->( %State: UnMute
    sWrite(AAOn,0)
  )
  + (state == 2 )->( %State: PossibleMute
    sWrite(AAOn,0)
  )
  + (state == 3 )->( %State: VolChangeDelay
    sWrite(AAOn,0)
  )
  + (state == 4 )->( %State: ShortestUnmute
    sWrite(AAOn,0)
  )
);

proc Mute_exit=
  sum state:Nat.rRead(Mute,state).( (state == 0)->( intern
)
  + (state == 1 )->( %State: MuteDelay
    sWrite(volume,0)
    . start(EMute)
    . SUO_exec(EMute)
    . sum current:Nat.( rRead(Mute,current)
      . (current == 1)->( sWrite(Mute,0)
        )
    )
  )
  + (state == 2 )->( %State: MuteState
    sWrite(Mute,0)
  )
);

proc OnScreenDisplay_exit=
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

sum state:Nat.rRead(OnScreenDisplay,state).(  
  (state == 0)->(  
    intern  
  )  
  + (state == 1)->( %State: MuteOSD  
      MuteOSD_exit  
      sWrite(OnScreenDisplay,0)  
  )
)
);

proc MuteOSD_exit=  
sum state:Nat.rRead(MuteOSD,state).(  
  (state == 0)->(  
    intern  
  )  
  + (state == 1)->( %State: noDisplay  
      sWrite(MuteOSD,0)  
  )  
  + (state == 2)->( %State: On  
      sWrite(MuteOSD,0)  
  )  
  + (state == 3)->( %State: DisplayUnMute  
      DisplayUnMute_exit  
      sWrite(MuteOSD,0)  
  )
)
);

proc DisplayUnMute_exit=  
sum state:Nat.rRead(DisplayUnMute,state).(  
  (state == 0)->(  
    intern  
  )  
  + (state == 1)->( %State: PossibleDisplay  
      sWrite(DisplayUnMute,0)  
  )  
  + (state == 2)->( %State: Display  
      sWrite(DisplayUnMute,0)  
  )  
  + (state == 3)->( %State: awaitLast  
      sWrite(DisplayUnMute,0)  
  )  
  + (state == 4)->( %State: uncertain  
      sWrite(DisplayUnMute,0)  
  )
)
);}
Modal formulas

Requirement 1

\[ \mu X. ( \langle \text{true} \rangle \text{true} \]
\[ \text{and } [\text{"cWrite(ActiveAudio, 1)"}] \text{false} \]
\[ \text{and } [\text{not } \text{"cWrite(ActiveAudio, 1)"}] X \]
\]

Requirement 2

\[ ( [\text{true*}] \text{. } \text{"cWrite(volume, [-0-2]|(\d\d\d))"} \]
\[ \text{false} \]
\[ ) \text{ } \text{and} ( [\text{true*}] \text{. } \text{"cWrite(volume, -\d+)} \]
\[ \text{false} \]
\]

Requirement 3

\[ [\text{true*}] \text{. } \text{"cWrite(ActiveAudio, 1)"} \]
\[ . (\text{not } \text{"start(MuteOnOff)"})^* \]
\[ . \text{"start(MuteOnOff)"} \]
\[ ) ( \mu X. ( \langle \text{true} \rangle \text{true} \]
\[ \text{and } [\text{not } \text{"cWrite(ActiveAudio, 2)"}] X \]
\]

Requirement 4

\[ [\text{true*}] \text{. } \text{"cWrite(ActiveAudio, 1)"} \]
\[ . (\text{not } \text{"start(MuteOnOff)"})^* \]
\[ . \text{"cWrite(volume, 0)"} \]
\[ . (\text{not } \text{"cWrite(ActiveAudio, 0)"})^* \]
\[ . \text{"start(VolumeUp)"} \]

Requirement 5

\[ ( [\text{true*}] \text{. } \text{"cWrite(ActiveAudio, 1)"} \]
\[ . (\text{not } \text{"cWrite(ActiveAudio, 0)"})^* \]
\[ . \text{"cWrite(volume, 0)"} \]
\[ . (\text{not } \text{"cWrite(ActiveAudio, 0)"})^* \]
\[ . \text{"start(VolumeUp)"} \]
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

( μ X. ( <true> true 
    and [not "cWrite(volume, 1)"] X 
  ) 
) 
) 
and(  
[ true* 
  . "cWrite(ActiveAudio, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "cWrite(volume, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "start(VolumeUp)"
 ]
( μ X.( <true> true 
    and [not "cWrite(volume, 2)"] X 
  ) 
)
) 
) 
and(  
< true* 
  . "cWrite(ActiveAudio, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . ( "cWrite(volume, 0)"|"cWrite(volume, 1)" )
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "start(VolumeUp)"
>true
)

Requirement 6

(  
[ true* 
  . "cWrite(ActiveAudio, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "cWrite(volume, 2)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "start(VolumeDown)"
 ]
( μ X.( <true> true and [not "cWrite(volume, 1)"] X ))
) 
and(  
[ true* 
  . "cWrite(ActiveAudio, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "cWrite(volume, 1)"
  . (not "cWrite(ActiveAudio, 0)"val) 
  . "start(VolumeDown)"
 ]
( μ X.( <true> true and [not "cWrite(volume, 0)"] X ))
) 
and(  
< true* 
  . "cWrite(ActiveAudio, 1)"

}
XXIX

. (not "cWrite(ActiveAudio, 0)")*
. ( "cWrite(volume, 1)" | "cWrite(volume, 2)" )
. (not "cWrite(ActiveAudio, 0)")*
. "start(VolumeDown)"
>true
)

Requirement 7

( [ true*
. "cWrite(ActiveAudio, 2)"
]
( μ X. ( <true> true
    and [not "cWrite(volume, 0)"] X
    )
)
)and(
[ true*
. "cWrite(ActiveAudio, 2)"
. (not "cWrite(volume, 0)")*
. "cWrite(volume, 0)"
. (not "cWrite(ActiveAudio, 0)")*
. "'cWrite\(volume, [1-2]\)$'"
]false
)

Requirement 8

( [ true*."cWrite(ActiveAudio, 2)"]
( μ X. ( <true> true
    and [not "cWrite(display, 1)"] X
    )
)
)and(
[ true*
. "cWrite(ActiveAudio, 2)"
. (not "cWrite(display, 1)")*
. "cWrite(display, 1)"
. (not "cWrite(ActiveAudio, 0)")*
. "'cWrite\(display, [-1]\)$'"
]false
)

Requirement 9

[ true*
. "cWrite(ActiveAudio, 2)"
(}
(not "start(MuteOnOff)")
and (not "start(VolumeUp)")
and (not "start(VolumeDown)")
)*
.
( "start(MuteOnOff)"
| "start(VolumeUp)"
| "start(VolumeDown)"
)
]
( mu X.(  
  <true> true
  and [not "cWrite(ActiveAudio, 1)"] X
  )
  )
)

Requirement 10

(  
  [ true*
    . "cWrite(ActiveAudio, 2)"
    . (not "cWrite(ActiveAudio, 1)")*
    . "cWrite(ActiveAudio, 1)"
  ]  
  ( mu X.(  
    <true> true
    and [not ( "cWrite(display, 0)"
     or "cWrite(display, 1)"
     ) ] X
    )
    )
  )and(
  [ true*
    . "cWrite(display, 2)"
  ]  
  ( mu X.(  
    <true> true
    and [ not ( "cWrite(display, 0)"
     or "cWrite(display, 1)"
     ) ] X
    )
    )
  )and(
  < true*
    . "cWrite(ActiveAudio, 2)"
    . (not "cWrite(ActiveAudio, 1)")*
    . "cWrite(ActiveAudio, 1)"
  >
  true
  )and(
  < true*
    . "cWrite(display, 2)"
  >
true

Requirement 11

( [ true*
   . "cWrite(ActiveAudio, 1)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "cWrite(volume, 0)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "start(MuteOnOff)"
   . ( not "cWrite(ActiveAudio, 2)" )*
   . "cWrite(ActiveAudio, 2)"
   . (not ( "start(MuteOnOff)"
           or "start(VolumeUp)"
           or "start(VolumeDown)"
           )
    )*
   . "start(MuteOnOff)"
 ]
 ( mu X. ( <true>true
       and [ not "cWrite(volume, 0)""] X
       )
    )
   )and(
   < true*
   . "cWrite(ActiveAudio, 1)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "cWrite(volume, 0)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "start(MuteOnOff)"
   . ( not "cWrite(ActiveAudio, 2)" )*
   . "cWrite(ActiveAudio, 2)"
   . (not ( "start(MuteOnOff)"
           or "start(VolumeUp)"
           or "start(VolumeDown)"
           )
    )*
   . "start(MuteOnOff)"
   > true
   )
)

Requirement 12

( [ true*
   . "cWrite(ActiveAudio, 1)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "cWrite(volume, 0)"
   . ( not "cWrite(ActiveAudio, 0)" )*
   . "start(MuteOnOff)"
   . (not ( "start(MuteOnOff)"
           or "start(VolumeUp)"
           or "start(VolumeDown)"
           )
    )*
   . "start(MuteOnOff)"
   )
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

. ( not "cWrite(ActiveAudio, 2)" )*
. "cWrite(ActiveAudio, 2)"
. ( not ( "start(MuteOnOff)"
   or "start(VolumeUp)"
   or "start(VolumeDown)"
  )
  )*
. "start(VolumeUp)"
]
( mu X.( <true>true
   and [not "cWrite(volume, 1)"] X
  )
)
) and( [ true*
 . "cWrite(ActiveAudio, 1)"
 . ( not "cWrite(ActiveAudio, 0)" )*
 . "cWrite(volume, 1)"
 . ( not "cWrite(ActiveAudio, 0)" )*
 . "start(MuteOnOff)"
 . ( not "cWrite(ActiveAudio, 2)" )*
 . "cWrite(ActiveAudio, 2)"
 . ( not ( "start(MuteOnOff)"
   or "start(VolumeUp)"
   or "start(VolumeDown)"
  )
  )*
. "start(VolumeUp)"
]
( mu X.( <true>true
   and [not "cWrite(volume, 2)"] X
  )
)
) and( [ true*
 . "cWrite(ActiveAudio, 1)"
 . ( not "cWrite(ActiveAudio, 0)" )*
 . "cWrite(volume, 2)"
 . ( not "cWrite(ActiveAudio, 0)" )*
 . "start(MuteOnOff)"
 . ( not "cWrite(ActiveAudio, 2)" )*
 . "cWrite(ActiveAudio, 2)"
 . ( not ( "start(MuteOnOff)"
   or "start(VolumeUp)"
   or "start(VolumeDown)"
  )
  )*
. "start(VolumeUp)"
]
( mu X.( <true>true
   and [not "cWrite(volume, 2)"] X
  )
)
XXXIII

) and (  
< true*
. "cWrite(ActiveAudio, 1)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "cWrite(volume, 0)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "start(MuteOnOff)"
. ( not "cWrite(ActiveAudio, 2)" )*
. "cWrite(ActiveAudio, 2)"
. (not ( "start(MuteOnOff)"
     or "start(VolumeUp)"
     or "start(VolumeDown)"
   ) )*
. "start(VolumeUp)"
> true
}

Requirement 13

( [ true*
. "cWrite(ActiveAudio, 1)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "cWrite(volume, 0)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "start(MuteOnOff)"
. ( not "cWrite(ActiveAudio, 2)" )*
. "cWrite(ActiveAudio, 2)"
. (not ( "start(MuteOnOff)"
     or "start(VolumeUp)"
     or "start(VolumeDown)"
   ) )*
. "start(VolumeDown)"
]  
( mu X. ( <true>true
    and [ not "cWrite(volume, 0)" ] X
   )
 )
) and ( [ true*
. "cWrite(ActiveAudio, 1)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "cWrite(volume, 1)"
. ( not "cWrite(ActiveAudio, 0)" )*
. "start(MuteOnOff)"
. ( not "cWrite(ActiveAudio, 2)" )*
. "cWrite(ActiveAudio, 2)"
. (not ( "start(MuteOnOff)"
     or "start(VolumeUp)"
     or "start(VolumeDown)"
   ) )*
APPENDIX C. MCRL2 SPECIFICATION OF EXAMPLE 2

or "start(VolumeDown)"
)
)
"start(VolumeDown)"
]
( mu X.( <true>true
    and [not "cWrite(volume, 0)"] X
  )
)
)

and( [ true*
   . "cWrite(ActiveAudio, 1)"
   . ( not "cWrite(ActiveAudio, 0)" )* 
   . "cWrite(volume, 2)"
   . ( not "cWrite(ActiveAudio, 0)" )* 
   . "start(MuteOnOff)"
   . ( not "cWrite(ActiveAudio, 2)" )* 
   . "cWrite(ActiveAudio, 2)"
   . ( not ( "start(MuteOnOff)"
      or "start(VolumeUp)"
      or "start(VolumeDown)"
    )
  )* 
  . "start(VolumeDown)"
]
( mu X.( <true>true
    and [not "cWrite(volume, 1)"] X
  )
)
)
)

and( < true*
   . "cWrite(ActiveAudio, 1)"
   . ( not "cWrite(ActiveAudio, 0)" )* 
   . "cWrite(volume, 0)"
   . ( not "cWrite(ActiveAudio, 0)" )* 
   . "start(MuteOnOff)"
   . ( not "cWrite(ActiveAudio, 2)" )* 
   . "cWrite(ActiveAudio, 2)"
   . ( not ( "start(MuteOnOff)"
      or "start(VolumeUp)"
      or "start(VolumeDown)"
    )
  )* 
  . "start(VolumeDown)"
> true
)
Appendix D

mCRL2 specification of example 3

mCRL2 code

sort Events = struct

%% Simulink events

  Clk?isClk
  | Zero?isZero
  | One?isOne
  | Two?isTwo
  | Three?isThree
  | Four?isFour
  | Five?isFive
  | Six?isSix
  | Seven?isSeven
  | Eight?isEight
  | Nine?isNine
  | MainsOnOff?isMainsOnOff
  | MuteOnOff?isMuteOnOff
  | MenuOnOff?isMenuOnOff
  | TxtOnOff?isTxtOnOff
  | PipDsOnOff?isPipDsOnOff
  | Standby?isStandby
  | ChanStepDown?isChanStepDown
  | ChanStepUp?isChanStepUp
  | VolumeDown?isVolumeDown
  | VolumeUp?isVolumeUp
  | PreviousChannel?isPreviousChannel
  | Display?isDisplay
  | DisplayBrowser?isDisplayBrowser
  | NextSource?isNextSource
%% Stateflow events %%%
| Digit?isDigit |
| WakeUp?isWakeUp |
| ChanChange?isChanChange |
| EMute?isEMute |
| UnMute?isUnMute |

%%% Implicit events %%%
| Change_app?isChange_app |
| Enter_SCOff?isEnter_SCOff |
| Enter_SCOn?isEnter_SCOn |
| Enter_StatusOn?isEnter_StatusOn |

%%% Data objects %%%
sort ListNat = List(Nat);

sort V = struct
| TVbehavior |
| TV |
| On |
| IT |
| ChannelChange |
| TXTNrChange |
| WakeUpEvent |
| TVModes |
| Active |
| Video |
| SingleScreen |
| DualScreen |
| Main |
| Aux |
| Audio |
| TVSpeaker |
| Volume |
| OSD |
| SCOSD |
| SCOn |
| VolumeBar |
| Show |
| StatusOSD |
| PresetOSD |
| PresetOn |
| MuteOSD |

%%% data objects %%%
| app |
map var2nat : V -> Nat;

eqn % state information
var2nat(TVbehavior) = 0;
var2nat(TV) = 1;
var2nat(On) = 2;
var2nat(IT) = 3;
var2nat(DigitEvent) = 4;
var2nat(ChannelChange) = 5;
var2nat(TXTNrChange) = 6;
var2nat(WakeUpEvent) = 7;
var2nat(TVModes) = 8;
var2nat(Active) = 9;
var2nat(Video) = 10;
var2nat(SingleScreen) = 11;
var2nat(DualScreen) = 12;
var2nat(Main) = 13;
var2nat(Aux) = 14;
var2nat(Audio) = 15;
var2nat(TVSpeaker) = 16;
var2nat(Volume) = 17;
var2nat(OSD) = 18;
var2nat(SCOSD) = 19;
var2nat(SCOn) = 20;
var2nat(VolumeBar) = 21;
var2nat(Show) = 22;
var2nat(StatusOSD) = 23;
var2nat(PresetOSD) = 24;
var2nat(PresetOn) = 25;
var2nat(MuteOSD) = 26;

% data objects
var2nat(app) = 27;
var2nat(vlevel) = 28;
var2nat(txtnr) = 29;
var2nat(SCOSDentry) = 30;
var2nat(Statusentry) = 31;
var2nat(Presetentry) = 32;
var2nat(Muteentry) = 33;
var2nat(digit) = 34;
var2nat(channr) = 35;
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

map writeVar :Nat # Nat # ListNat # ListNat -> ListNat;

var s,t:ListNat;
n: Nat;
m: Nat;
b: Bool;

eqn ( #s==0 ) -> writeVar(n,m,s,t) = t;
((#s!=0) && (n> #t)) -> writeVar(n,m,s,t) = writeVar(n,m,tail(s),t<|head(s));
((#s!=0) && (n==#t)) -> writeVar(n,m,s,t) = writeVar(n,m,tail(s),t<|m);
((#s!=0) && (n< #t)) -> writeVar(n,m,s,t) = t++s;

act sRead ,rRead ,cRead : V # Nat;
sWrite ,rWrite ,cWrite : V # Nat;

proc data(d:ListNat)=
  sum v: V.sRead(v,d.var2nat(v)).data(d)
+ sum v: V,m:Nat.(var2nat(v)<(#d)) ->
  rWrite(v,m).data(writeVar(var2nat(v),m,d,[]))
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Custom Matlab functions %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map ml_mod: Int # Nat -> Nat;

var x:Int;
y: Nat;

eqn ( x<0 ) -> ml_mod(x,y) = 2;
ml_mod(0,y) = 0;
ml_mod(1,y) = 1;
ml_mod(2,y) = 2;
( x>2 ) -> ml_mod(x,y) = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Custom actions %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act intern, intern2, ctau;
  ctau: V # Nat;
act write_volume, write_PresetDisplay, write_SCOSDDisplay, write_StatusDisplay, write_MuteDisplay, write_TVMode, write_ScreenMode,
write_auxnr:Nat;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Initiation %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init
  rename({cRead->ctau,intern->ctau},
   block({
     sWrite,rWrite,
     sRead,rRead
   },
   comm({
     sRead |rRead ->cRead,
     sWrite|rWrite->cWrite
   },
   initiation
   ||data([0 % TVbehavior,
     ,0 % TV,
     ,0 % On,
     ,0 % IT,
     ,0 % DigitEvent,
     ,0 % ChannelChange,
     ,0 % TXTNrChange,
     ,0 % WakeUpEvent,
     ,0 % TVModes,
     ,0 % Active,
     ,0 % Video,
     ,0 % SingleScreen,
     ,0 % DualScreen,
     ,0 % Main,
     ,0 % Aux,
     ,0 % Audio,
     ,0 % TVSpeaker,
     ,0 % Volume,
     ,0 % OSD,
     ,0 % SCOSD,
     ,0 % SCOn,
     ,0 % VolumeBar,
     ,0 % Show,
     ,0 % StatusOSD,
     ,0 % PresetOSD,
     ,0 % PresetOn,
     ,0 % MuteOSD,
     ,0 % app,
     ,0 % vlevel,
     ,0 % txtnr,
     ,0 % SCOSDentry
   ]))}
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

\[
\begin{align*}
&\text{proc initiation=} \\
&\quad \text{TVbehavior enter([1])} \\
&\quad . \text{stateflow} \\
&\quad ; \\
\end{align*}
\]

\[
\begin{align*}
\text{proc stateflow=} \\
&\quad ( \text{sum } i: \text{Nat.}( \text{rRead(SCOSDentry}, i) . (i<0)->(\text{intern})<<((} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
+ start(VolumeDown).TVbehavior_exec(VolumeDown)
+ start(VolumeUp).TVbehavior_exec(VolumeUp)
+ start(Display).TVbehavior_exec(Display)
)
. stateflow
;

act start:Events;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Execution processes %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc TVbehavior_exec(event:Events)=
sum state:Nat.rRead(TVbehavior,state).(
  (state == 0)->(
    intern
  )
+ (state == 1)->( % State: TV
    TV_exec(event)
  )
)
;

proc TV_exec(event:Events)=
sum state:Nat.rRead(TV,state).(
  (state == 0)->(
    intern
  )
+ (state == 1)->( % State: Off
    ( isMainsOnOff(event) )->(
      TV_exit
      . TV_enter([2])
    )<>(
      intern
    )
  )
+ (state == 2)->( % State: On
    ( isMainsOnOff(event) )->(
      TV_exit
      . write_volume(0)
      . TV_enter([1])
    )<>(
      On_exec(event)
    )
  )
)
;

proc On_exec(event:Events)=
sum state:Nat.rRead(On,state).
  (state == 0)->(intern)
++ (state > 0)->(% State: IT
    intern
      . IT_exec(event)
      . TVModes_exec(event)
    )
);

proc IT_exec(event:Events)=
sum state:Nat.rRead(IT,state).
  (state == 0)->(intern)
++ (state > 0)->(DigitEvent_exec(event)
    . ChannelChange_exec(event)
    . TXTNrChange_exec(event)
    . WakeUpEvent_exec(event)
  )
);

proc DigitEvent_exec(event:Events)=
sum state:Nat.rRead(DigitEvent,state).
  (state == 0)->(intern)
++ (state == 1)->(% State: DEInit
    (isZero(event))->(DigitEvent_exit
      . sWrite(digit,0)
      . start(Digit)
      . ChannelChange_exec(Digit)
      . TXTNrChange_exec(Digit)
      . WakeUpEvent_exec(Digit)
      . DigitEvent_enter([1])<>((isOne(event))->(DigitEvent_exit
        . sWrite(digit,1)
        . start(Digit)
        . ChannelChange_exec(Digit)
        . TXTNrChange_exec(Digit)
        . WakeUpEvent_exec(Digit)
        . DigitEvent_enter([1])<>((isTwo(event))->(DigitEvent_exit
          . sWrite(digit,2)
          . start(Digit)
          . ChannelChange_exec(Digit)
          . TXTNrChange_exec(Digit)
          . WakeUpEvent_exec(Digit)
          . DigitEvent_enter([1])<>((isThree(event))->(DigitEvent_exit
            . sWrite(digit,3)
            . start(Digit)
            . ChannelChange_exec(Digit)
            . TXTNrChange_exec(Digit)
            . WakeUpEvent_exec(Digit)
            . DigitEvent_enter([1])<>((isFour(event))->(DigitEvent_exit
              . sWrite(digit,4)
              . start(Digit)
              . ChannelChange_exec(Digit)
              . TXTNrChange_exec(Digit)
              . WakeUpEvent_exec(Digit)
              . DigitEvent_enter([1])<>((isFive(event))->(DigitEvent_exit
                . sWrite(digit,5)
                . start(Digit)
                . ChannelChange_exec(Digit)
                . TXTNrChange_exec(Digit)
                . WakeUpEvent_exec(Digit)
                . DigitEvent_enter([1])<>((isSix(event))->(DigitEvent_exit
                  . sWrite(digit,6)
                  . start(Digit)
                  . ChannelChange_exec(Digit)
                  . TXTNrChange_exec(Digit)
                  . WakeUpEvent_exec(Digit)
                  . DigitEvent_enter([1])<>((isSeven(event))->(DigitEvent_exit
                    . sWrite(digit,7)
                    . start(Digit)
                    . ChannelChange_exec(Digit)
                    . TXTNrChange_exec(Digit)
                    . WakeUpEvent_exec(Digit)
                    . DigitEvent_enter([1])<>((isEight(event))->(DigitEvent_exit
                      . sWrite(digit,8)
                      . start(Digit)
                      . ChannelChange_exec(Digit)
                      . TXTNrChange_exec(Digit)
                      . WakeUpEvent_exec(Digit)
                      . DigitEvent_enter([1])<>((isNine(event))->(DigitEvent_exit
                        . sWrite(digit,9)
                        . start(Digit)
                        . ChannelChange_exec(Digit)
                        . TXTNrChange_exec(Digit)
                        . WakeUpEvent_exec(Digit)
                        . DigitEvent_enter([1])<>((isZero(event))->(DigitEvent_exit
                          . sWrite(digit,0)
                          . start(Digit)
                          . ChannelChange_exec(Digit)
                          . TXTNrChange_exec(Digit)
                          . WakeUpEvent_exec(Digit)
                          . DigitEvent_enter([1])))<>((isOne(event))->(DigitEvent_exit
                            . sWrite(digit,1)
                            . start(Digit)
                            . ChannelChange_exec(Digit)
                            . TXTNrChange_exec(Digit)
                            . WakeUpEvent_exec(Digit)
                            . DigitEvent_enter([1]))<>((isTwo(event))->(DigitEvent_exit
                              . sWrite(digit,2)
                              . start(Digit)
                              . ChannelChange_exec(Digit)
                              . TXTNrChange_exec(Digit)
                              . WakeUpEvent_exec(Digit)
                              . DigitEvent_enter([1]))<>((isThree(event))->(DigitEvent_exit
                                . sWrite(digit,3)
                                . start(Digit)
                                . ChannelChange_exec(Digit)
                                . TXTNrChange_exec(Digit)
                                . WakeUpEvent_exec(Digit)
                                . DigitEvent_enter([1]))<>((isFour(event))->(DigitEvent_exit
                                  . sWrite(digit,4)
                                  . start(Digit)
                                  . ChannelChange_exec(Digit)
                                  . TXTNrChange_exec(Digit)
                                  . WakeUpEvent_exec(Digit)
                                  . DigitEvent_enter([1]))<>((isFive(event))->(DigitEvent_exit
                                    . sWrite(digit,5)
                                    . start(Digit)
                                    . ChannelChange_exec(Digit)
                                    . TXTNrChange_exec(Digit)
                                    . WakeUpEvent_exec(Digit)
                                    . DigitEvent_enter([1]))<>((isSix(event))->(DigitEvent_exit
                                      . sWrite(digit,6)
                                      . start(Digit)
                                      . ChannelChange_exec(Digit)
                                      . TXTNrChange_exec(Digit)
                                      . WakeUpEvent_exec(Digit)
                                      . DigitEvent_enter([1]))<>((isSeven(event))->(DigitEvent_exit
                                        . sWrite(digit,7)
                                        . start(Digit)
                                        . ChannelChange_exec(Digit)
                                        . TXTNrChange_exec(Digit)
                                        . WakeUpEvent_exec(Digit)
                                        . DigitEvent_enter([1]))<>((isEight(event))->(DigitEvent_exit
                                          . sWrite(digit,8)
                                          . start(Digit)
                                          . ChannelChange_exec(Digit)
                                          . TXTNrChange_exec(Digit)
                                          . WakeUpEvent_exec(Digit)
                                          . DigitEvent_enter([1]))<>((isNine(event))->(DigitEvent_exit
                                            . sWrite(digit,9)
                                            . start(Digit)
                                            . ChannelChange_exec(Digit)
                                            . TXTNrChange_exec(Digit)
                                            . WakeUpEvent_exec(Digit)
                                            . DigitEvent_enter([1]))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))}
DigitEvent_exit
  . sWrite(digit,2)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isThree(event) )->(
  DigitEvent_exit
  . sWrite(digit,3)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isFour(event) )->(
  DigitEvent_exit
  . sWrite(digit,4)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isFive(event) )->(
  DigitEvent_exit
  . sWrite(digit,5)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isSix(event) )->(
  DigitEvent_exit
  . sWrite(digit,6)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isSeven(event) )->(
  DigitEvent_exit
  . sWrite(digit,7)
  . start(Digit)
  . ChannelChange_exec(Digit)
  . TXTNrChange_exec(Digit)
  . WakeUpEvent_exec(Digit)
  . DigitEvent_enter([1])
)<>(( isEight(event) )->(
  DigitEvent_exit
  . sWrite(digit,8)
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

```plaintext
.start(Digit)
.ChannelChange_exec(Digit)
.TXTNrChange_exec(Digit)
.WakeUpEvent_exec(Digit)
.DigitEvent_enter([1])
)<>(( isNine(event) )->{
    DigitEvent_exit
    sWrite(digit,9)
    .start(Digit)
    .ChannelChange_exec(Digit)
    .TXTNrChange_exec(Digit)
    .WakeUpEvent_exec(Digit)
    .DigitEvent_enter([1])
}<>((
    intern
    ))))))))})
)
)
)

proc ChannelChange_exec(event:Events)=
sum state:Nat.rRead(ChannelChange,state).(
    (state == 0)->{
        intern
    } + (state == 1)->{ % State: CCinit
        sum x:Nat.( rRead(SingleScreen,x)
            . sum y:Nat.( rRead(Aux,y)
                . sum z:Nat.( rRead(TVModes,z)
                    . ( isChanStepUp(event) &&((x==2)||(y==1)||(z==1)) )->(
                        ChannelChange_exit
                        . sum i:Nat.( rRead(channr,i).sWrite(channr,ml_mod(i+1,3))
                        . start(ChanChange)
                        . PresetOSD_exec(ChanChange)
                        . ChannelChange_enter([1])
                    )<>(( isChanStepDown(event) &&((x==2)||(y==1)||(z==1)) )->(
                        ChannelChange_exit
                        . sum i:Nat.( rRead(channr,i).sWrite(channr,ml_mod(i-1,3))
                        . start(ChanChange)
                        . PresetOSD_exec(ChanChange)
                        . ChannelChange_enter([1])
                    )<>(( isDigit(event) &&((x==2)||(y==1)||(z==1)) )->(
                        ChannelChange_exit
                        . sum i:Nat.( rRead(digit,i).sWrite(channr,i)
                        . start(ChanChange)
                        . PresetOSD_exec(ChanChange)
                        . ChannelChange_enter([1])
                    )<>((
    intern
    ))))))
)}
```

XLIV
proc TXTNrChange_exec(event:Events)="\nsum state:Nat.rRead(TXTNrChange,state).(\n  (state == 0)->(\n    intern\n  )\n  + sum at:Nat. ( rRead(Aux,at). sum st:Nat. ( rRead(SingleScreen,st).\n    (state == 1)->( % State: TNInit\n      ( isChanStepDown(event) )->(\n        TXTNrChange_exit\n        . sum i:Nat.( rRead(txtnr,i).sWrite(txtnr,ml_mod(i-1,3)) )\n        . TXTNrChange_enter([1])\n      )<>(( isChanStepUp(event) )->(\n        TXTNrChange_exit\n        . sum i:Nat.( rRead(txtnr,i).sWrite(txtnr,ml_mod(i+1,3)) )\n        . TXTNrChange_enter([1])\n      )<>(( isDigit(event)&&((at==2)||(st==3)) )->(\n        TXTNrChange_exit\n        . sum tn:Nat.( rRead(digit,tn).sWrite(txtnr,tn) )\n        . TXTNrChange_enter([1])\n      )<>(\n        intern\n      )\n    )\n  )\n)\n);\n"

proc WakeUpEvent_exec(event:Events)="\nsum state:Nat.rRead(WakeUpEvent,state).(\n  (state == 0)->( % State: WEInit\n    sum sl:Nat.( rRead(TVModes,sl).( isChanStepDown(event) && (sl==1) )->(\n      WakeUpEvent_exit\n      . start(WakeUp)\n      . TVModes_exec(WakeUp)\n      . WakeUpEvent_enter([1])\n    )<>(( isChanStepUp(event) && (sl==1) )->(\n      WakeUpEvent_exit\n      . start(WakeUp)\n      . TVModes_exec(WakeUp)\n      . WakeUpEvent_enter([1])\n    )<>(( isDigit(event) && (sl==1) )->(\n      WakeUpEvent_exit\n      . start(WakeUp)\n      . TVModes_exec(WakeUp)\n      . WakeUpEvent_enter([1])\n    ))))\n)\n);"
. TVModes_exec(WakeUp)
  . WakeUpEvent_enter([1])
  )<>(( isVolumeUp(event) && (sl==1) )->(
    WakeUpEvent_exit
    . start(WakeUp)
    . TVModes_exec(WakeUp)
    . WakeUpEvent_enter([1])
  )<>(( isVolumeDown(event) && (sl==1) )->(
    WakeUpEvent_exit
    . start(WakeUp)
    . TVModes_exec(WakeUp)
    . WakeUpEvent_enter([1])
  )<>(
    intern
  )))
)
)
);

proc TVModes_exec(event:Events)=
  sum state:Nat.rRead(TVModes,state).( (state == 0)->(
    intern
  )
+ (state == 1)->( % State: Sleep
    ( isStandby(event) )->(
      TVModes_exit
      . TVModes_enter([2])
    )<>(( isWakeUp(event) )->(
      TVModes_exit
      . TVModes_enter([2])
    )<>(
      intern
    ))
  )
+ (state == 2)->( % State: Active
    ( isStandby(event) )->(
      TVModes_exit
      . write_volume(0)
      . TVModes_enter([1])
    )<>(
      Active_exec(event)
    )
  )
)
);

proc Active_exec(event:Events)=
  sum state:Nat.rRead(Active,state).( (state == 0)->(

intern
)
+ (state > 0)->( 
  intern
  . Video_exec(event)
  . Audio_exec(event)
  . OSD_exec(event)
)
);

proc Video_exec(event:Events)=
sum state:Nat.rRead(Video,state).(
  (state == 0)->(
    intern
  )
  + (state == 1)->( % State: SingleScreen
    SingleScreen_exec(event)
  )
  + (state == 2)->( % State: DualScreen
    DualScreen_exec(event)
  )
)
);

proc SingleScreen_exec(event:Events)=
sum state:Nat.rRead(SingleScreen,state).(
  (state == 0)->(
    intern
  )
  + (state == 1)->( % State: Menu
    ( isTxtOnOff(event) )->( 
      SingleScreen_exit
      . SingleScreen_enter([3])
    )<>(
      ( isMenuOnOff(event) )->(
        SingleScreen_exit
        . SingleScreen_enter([2])
      )<>
      ( isPipDsOnOff(event) )->(
        Video_exit
        . sum i:Nat. ( rRead(channr,i).write_auxnr(i) )
        . Video_enter([2,2,1])
      )<>
    )"
    intern
  )
  )
  + (state == 2)->( % State: SSTV
    ( isTxtOnOff(event) )->(
      SingleScreen_exit
      . SingleScreen_enter([3])
    )<>
    ((( isMenuOnOff(event) )->(
      Video_exit
      . sum i:Nat. ( rRead(channr,i).write_auxnr(i) )
      . Video_enter([2,2,1])
    )<>
      intern
    )))
  )
)
\[\text{SingleScreen\_exit}.
\text{SingleScreen\_enter([1])}
\]\\
\[\text{Video\_exit}
. \text{sum } i: \text{Nat}. ( \text{rRead(chan\_i).write\_auxnr(i)} )
. \text{Video\_enter([2,2,1])}
\]\\\
\[(\text{isPipDsOnOff(event)}) \rightarrow (\]
\[\text{Video\_exit}
. \text{Video\_enter([2,2,2])}
\]\\\
\[\text{write\_TVMode(2)}
.)
\]\\
\[+ (\text{state == 3}) \rightarrow ( \% \text{State: SSTXT} (\]
\[\text{isTxtOnOff(event)}) \rightarrow (\]
\[\text{SingleScreen\_exit}
. \text{SingleScreen\_enter([2])}
\]\\\
\[\text{Video\_exit}
. \text{Video\_enter([2,2,2])}
\]\\\
\[\text{intern}
.)
\])
\]\\
\[\text{proc DualScreen\_exec(event:Events) =}
\text{sum state: Nat. rRead(DualScreen,state). (}
\text{(state == 0) \rightarrow (}
. \text{intern)
\}\]
\\
\[+ (\text{state > 0}) \rightarrow (]
. \text{intern}
. \text{Main\_exec(event)}
. \text{Aux\_exec(event)}
\)
\]
\[\text{proc Main\_exec(event:Events) =}
\text{sum state: Nat. rRead(Main,state). (}
\text{(state == 0) \rightarrow (}
. \text{intern)
\}\]
\\
\[+ (\text{state == 1}) \rightarrow ( \% \text{State: MainTV}
. \text{intern}
\)]
\]
\[\text{proc Aux\_exec(event:Events) =}
\text{sum state: Nat. rRead(Aux,state). (}
\]
(state == 0)->(
    intern
)
+ (state == 1)->( % State: AuxTV
    ( isMenuOnOff(event) )->(
        Video_exit
        . Video_enter([1,1])
    )<>(( isPipDsOnOff(event) )->(
        Video_exit
        . Video_enter([1,2])
    )<>(
        isTxtOnOff(event)
    )<>(
        intern
    )))
+ (state == 2)->( % State: AuxTXT
    ( isTxtOnOff(event) )->(
        Video_exit
        . Video_enter([1,2])
    )<>(( isPipDsOnOff(event) )->(
        Video_exit
        . Video_enter([1,3])
    )<>(
        intern
    )))
)

proc Audio_exec(event:Events)=
    sum state:Nat.rRead(Audio,state).(
        (state == 0)->(
            intern
        )
        + (state > 0)->(
            intern
            . TVSpeaker_exec(event)
            . Volume_exec(event)
        )
    )
;

proc TVSpeaker_exec(event:Events)=
    sum state:Nat.rRead(TVSpeaker,state).(
        (state == 0)->(
            intern
        )
        + (state == 1)->( % State: SpeakerOn
( isMuteOnOff(event) )->(
    TVSpeaker_exit
    . start(EMute)
    . MuteOSD_exec(EMute)
    . TVSpeaker_enter([2])
  )<>(
    sum i:Nat. ( rRead(vlevel,i).write_volume(i) )
  )
+
+ (state == 2)->( % State: Mute
  ( isMuteOnOff(event) )->(
    TVSpeaker_exit
    . start(UnMute)
    . MuteOSD_exec(UnMute)
    . TVSpeaker_enter([1])
  )<>(
    intern
  )
)
)
;

proc Volume_exec(event:Events)=
sum state:Nat.rRead(Volume,state).(
  (state == 0)->(intern)
+ (state == 1)->( % State: VolumeLevel
  sum vl:Nat. ( rRead(vlevel,vl).( isVolumeUp(event) && (vl < 2) ) )->(
    Volume_exit %MaxVol
    . sWrite(vlevel,vl+1)
    . Volume_enter([1])
  )<>(( isVolumeDown(event) && (vl > 0) )->(
    Volume_exit
    . sWrite(vlevel,max(vl-1,0))
    . Volume_enter([1])
  )<>(
    intern
  )))
)
);

proc OSD_exec(event:Events)=
sum state:Nat.rRead(OSD,state).(
  (state == 0)->(intern
+ (state > 0)->(intern

proc SCOSD_exec(event:Events)=
sum state:Nat.rRead(SCOSD,state).(  
(state == 0)->(  
   intern
   )
+ (state == 1)->( % State: SCOff  
      sum ap:Nat.( rRead(app,ap).( isVolumeDown(event) && (ap == 0) )->(  
         SCOSD_exit  
         . SCOSD_enter([2,1,1])  
      )<>(( isVolumeDown(event) && (ap != 0) )->(  
         SCOSD_exit  
         . SCOSD_enter([2,1,2])  
      )<>(( isVolumeUp(event) && (ap == 0) )->(  
         SCOSD_exit  
         . SCOSD_enter([2,1,1])  
      )<>(( isVolumeUp(event) && (ap != 0) )->(  
         SCOSD_exit  
         . SCOSD_enter([2,1,2])  
      )))
   )
+ (state == 2)->( % State: SCOn  
      sum se:Nat.( rRead(SCOSDentry,se).( isClk(event) && (se == 0) )->(  
         SCOSD_exit  
         . SCOSD_enter([1])  
      )<>(( isEnter_StatusOn(event) )->(  
         SCOSD_exit  
         . SCOSD_enter([1])  
      )<>(  
         SCOn_exec(event)  
      )))
   )
   )
  )
);
VolumeBar_exec(event)
)
)
;

proc VolumeBar_exec(event:Events)=
sum state:Nat.rRead(VolumeBar,state).( (state == 0)->(
  intern
)
+ (state == 1)->( % State: Show
  ( isVolumeUp(event) )->(
    VolumeBar_exit
    . VolumeBar_enter([1])
  )<>( ( isVolumeDown(event) )->(
    VolumeBar_exit
    . VolumeBar_enter([1])
  )<>( ( isChange_app(event) )->(
      SCOSD_exit
      . SCOSD_enter([1])
    )<>(
      Show_exec(event)
    )))
)
+ (state == 2)->( % State: Suppress
  sum ap:Nat.( rRead(app,ap). ( isChange_app(event) && (ap == 0) )->(
    VolumeBar_exit
    . VolumeBar_enter([1])
  )<>(
    intern
  )
)
))
);

proc Show_exec(event:Events)=
sum state:Nat.rRead(Show,state).( (state == 0)->(
  intern
)
+ (state == 1)->( % State: nobar
  intern
)
+ (state == 2)->( % State: bar
  intern
)
)
;

proc PresetOSD_exec(event:Events)=


sum state:Nat. rRead(PresetOSD, state).( 
  (state == 0)->( 
    intern 
  )
)+ (state == 1)->( % State: PresetOff 
  sum so:Nat. ( rRead(StatusOSD, so). sum ap:Nat. ( rRead(app, ap) 
    . ( isChanChange(event) && (so != 1) && (ap == 0) && (1 == 0) )->( 
      PresetOSD_exit %minimal 
      . PresetOSD_enter([2,2]) 
    )<>( ( isChanChange(event) && (so != 1) && ((ap != 0) || (1 != 0)) )->( 
      PresetOSD_exit %minimal 
      . PresetOSD_enter([2,3]) 
    )<>( 
      PresetOSD_exit 
      . PresetOSD_enter([2,1]) 
    )<>( 
      intern 
    )))))
)+ (state == 2)->( % State: PresetOn 
  PresetOn_exec(event) 
)
);

proc PresetOn_exec(event: Events)= 
  sum state:Nat. rRead(PresetOn, state).( 
    (state == 0)->( 
      intern 
    )
)+ (state == 1)->( % State: Suppressed 
  sum ap:Nat. ( rRead(app, ap). ( isEnter_SCOff(event) && (ap==0) && (1==0) )->( 
    PresetOn_exit %minimal 
    . PresetOn_enter([2]) 
  )<>( ( isEnter_SCOff(event) && ((ap!=0) || (1!=0)) )->( %minimal 
    PresetOn_exit 
    . PresetOn_enter([3]) 
  )<>( 
    intern 
  ))))
)+ (state == 2)->( % State: PresetFull 
  sum ap:Nat. ( rRead(app, ap). ( isChange_app(event) && (ap != 0) )->( 
    PresetOn_exit 
    . PresetOn_enter([3]) 
  )<>( ( isChanChange(event) )->( 
    PresetOn_exit 
    . PresetOn_enter([2]) 
  )<>( ( isEnter_StatusOn(event) )->( 
    PresetOn_exit 
  )))))
. PresetOn_enter([3])
    )<>(
        intern
    ))))

+ (state == 3)->( % State: PresetNumber
    ( isChanChange(event) )->(
        PresetOn_exit
        . PresetOn_enter([3])
    )<>(sum pe:Nat. rRead(Presetentry,pe).( isClk(event) && (pe == 0) )->(
            PresetOSD_exit
            . PresetOSD_enter([1])
        )<>(
            intern
        ))))
)
)
;

proc StatusOSD_exec(event:Events)=
sum state:Nat.rRead(StatusOSD,state).( (state == 0)->( (state == 1)->( % State: StatusOff
    sum ap:Nat. rRead(app,ap).( isDisplay(event) && (ap==0) )->(
        StatusOSD_exit
        . StatusOSD_enter([2])
    )<>(
        intern
    ))
)
)
+ (state == 2)->( % State: StatusOn
    sum ap:Nat. rRead(app,ap).( isChange_app(event) && (ap!=0) )->(
        StatusOSD_exit
        . StatusOSD_enter([1])
    )<>(sum se:Nat. rRead(Statusentry,se).( isClk(event) && (se==0) )->(
            StatusOSD_exit
            . StatusOSD_enter([1])
        )<>(
            isEnter_SCOn(event) )->(
                StatusOSD_exit
                . StatusOSD_enter([1])
            )<>(
                isDisplay(event) )->(
                    StatusOSD_exit
                    . StatusOSD_enter([1])
                )<>(
                    intern
                ))))}))
)
proc MuteOSD_exec(event:Events)=
sum state:Nat.rRead(MuteOSD,state).(
    (state == 0)->(
        intern
    )
    + (state == 1)->( % State: MuteOff 
        ( isEMute(event) )->(
            MuteOSD_exit
            . MuteOSD_enter([2])
        )<>(
            intern
        )
    )
    + (state == 2)->( % State: MuteOn 
        ( isUnMute(event) )->(
            MuteOSD_exit
            . sWrite(Muteentry,2) %MuteTO
            . MuteOSD_enter([3])
        )<>(
            sum ap:Nat. ( rRead(app,ap).( isChange_app(event) & (ap!=0) ) )->(
                MuteOSD_exit
                . MuteOSD_enter([1])
            )<>(
                intern
            )
        )
    )
    + (state == 3)->( % State: noMute 
        ( isEMute(event) )->(
            MuteOSD_exit
            . MuteOSD_enter([2])
        )<>(
            sum ap:Nat. ( rRead(app,ap).( isChange_app(event) & (ap!=0) ) )->(
                MuteOSD_exit
                . MuteOSD_enter([1])
            )<>(
                sum to:Nat. ( rRead(Muteentry,to).( isClk(event) & (to==0) ) )->(
                    MuteOSD_exit
                    . MuteOSD_enter([1])
                )<>(
                    intern
                )
            )
        )
    )
)


```mcrl2
proc TVbehavior_enter(ln:ListNat)=
 (#ln == 0)->(
   intern.TVbehavior_enter([1]) % {default flow graph}
 )<>(
   (head(ln) == 1 )->( % State: TV
    sWrite(TVbehavior,1)
    . TV_enter(tail(ln))
    )
   )
);

proc TV_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.TV_enter([1])
 )<>(
   (head(ln) == 1 )->( % State: Off
    sWrite(TV,1)
    . write_ScreenMode(0)
   )
  + (head(ln) == 2 )->( % State: On
    sWrite(TV,2)
    . On_enter( tail(ln) )
   )
  ))
);

proc On_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.On_enter([1])
 )<>(
    sWrite(On,1)
    . IT_enter(if(head(ln) ==1,tail(ln),[]))
    . sum new:Nat.(rRead(On,new).(new == 1)->(
     sWrite(On,2)
     . TVModes_enter(if(head(ln) ==2,tail(ln),[]))
    <>(
     intern
    ))
   )
  ))
);

proc IT_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.IT_enter([1])
 )<>(
    sWrite(IT,1)
    . DigitEvent_enter(if(head(ln) ==1,tail(ln),[]))
   ))
;```

---

**APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3**

`proc TVbehavior_enter(ln:ListNat)=
 (#ln == 0)->(
   intern.TVbehavior_enter([1]) % {default flow graph}
 )<>(
   (head(ln) == 1 )->( % State: TV
    sWrite(TVbehavior,1)
    . TV_enter(tail(ln))
   )
   )
);

`proc TV_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.TV_enter([1])
 )<>(
   (head(ln) == 1 )->( % State: Off
    sWrite(TV,1)
    . write_ScreenMode(0)
   )
  + (head(ln) == 2 )->( % State: On
    sWrite(TV,2)
    . On_enter( tail(ln) )
   )
  ))
);

`proc On_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.On_enter([1])
 )<>(
    sWrite(On,1)
    . IT_enter(if(head(ln) ==1,tail(ln),[]))
    . sum new:Nat.(rRead(On,new).(new == 1)->(
     sWrite(On,2)
     . TVModes_enter(if(head(ln) ==2,tail(ln),[]))
    <>(
     intern
    ))
   )
  ))
);

`proc IT_enter(ln:ListNat)=
 ( (#ln == 0)->(
   intern.IT_enter([1])
 )<>(
    sWrite(IT,1)
    . DigitEvent_enter(if(head(ln) ==1,tail(ln),[]))
   ))
;`
. sum new:Nat.(rRead(IT,new).(new == 1)->(
  sWrite(IT,2)
  . ChannelChange_enter(if(head(ln) ==2,tail(ln),[])))<>(
    intern
  )))

. sum new:Nat.(rRead(IT,new).(new == 2)->(
  sWrite(IT,3)
  . TXTNrChange_enter(if(head(ln) ==3,tail(ln),[])))<>(
    intern
  )))

. sum new:Nat.(rRead(IT,new).(new == 3)->(
  sWrite(IT,4)
  . WakeUpEvent_enter(if(head(ln) ==4,tail(ln),[])))<>(
    intern
  )))

proc DigitEvent_enter(ln:ListNat)=
  ( (#ln == 0)->(
    intern.DigitEvent_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: DEInit
      sWrite(DigitEvent,1)
    )
  ))

proc ChannelChange_enter(ln:ListNat)=
  ( (#ln == 0)->(
    intern.ChannelChange_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: CCinit
      sWrite(ChannelChange,1)
    )
  ))

proc TXTNrChange_enter(ln:ListNat)=
  ( (#ln == 0)->(
    intern.TXTNrChange_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: TNInit
      sWrite(TXTNrChange,1)
    )
  ))
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3


proc Video_enter(ln:ListNat) =
  (#ln == 0)->(
    intern.Video_enter([1])<>
  )
+ (head(ln) == 1)->( % State: SingleScreen
    sWrite(Video,1)
    . write_ScreenMode(2)
    . SingleScreen_enter(tail(ln))
  )
+ (head(ln) == 2)->( % State: DualScreen
    sWrite(Video,2)
    . write_ScreenMode(3)
    . DualScreen_enter(tail(ln))
  )
);

proc SingleScreen_enter(ln:ListNat) =
  (#ln == 0)->(
    sWrite(app,0)
    . start(Change_app)
    . OSD_exec(Change_app)
    . SingleScreen_enter([2])<>
  )
+ (head(ln) == 1)->( % State: Menu
    sWrite(SingleScreen,1)
    . sWrite(app,1)
    . start(Change_app)
    . OSD_exec(Change_app)
    . write_TVMode(0)
  )
+ (head(ln) == 2)->( % State: SSTV
    sWrite(SingleScreen,2)
  )
+ (head(ln) == 3)->( % State: SSTXT
    sWrite(SingleScreen,3)
    . sWrite(app,1)
    . start(Change_app)
    . OSD_exec(Change_app)
    . sWrite(txtnr,1)
    . write_TVMode(1)
  )
);

proc DualScreen_enter(ln:ListNat) =
  (#ln == 0)->(  

APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

intern.DualScreen_enter([1])
  )<>
    sWrite(DualScreen,1)
    . Main_enter(if(head(ln) ==1,tail(ln),[[]])
      . sum new:Nat.(rRead(DualScreen,new).(new == 1)->(
        sWrite(DualScreen,2)
        . Aux_enter(if(head(ln) ==2,tail(ln),[]))
      )<>
        intern
      ))
  )

proc Main_enter(ln:ListNat)=
( (#ln == 0)->(
  intern.Main_enter([1])
  )<>
    (head(ln) == 1 )->( % State: MainTV
      sWrite(Main,1)
    )
  ))

proc Aux_enter(ln:ListNat)=
( (#ln == 0)->(
  intern.Aux_enter([1])
  )<>
    (head(ln) == 1 )->( % State: AuxTV
      sWrite(Aux,1)
      . write_TVMode(2)
    )
    + (head(ln) == 2 )->( % State: AuxTXT
      sWrite(Aux,2)
      . write_TVMode(1)
    )
  ))

proc Audio_enter(ln:ListNat)=
( (#ln == 0)->(
  intern.Audio_enter([1])
  )<>
    sWrite(Audio,1)
    . TVSpeaker_enter(if(head(ln) ==1,tail(ln),[]))
      . sum new:Nat.(rRead(Audio,new).(new == 1)->(
        sWrite(Audio,2)
        . Volume_enter(if(head(ln) ==2,tail(ln),[]))
      )<>
        intern
      ))
proc TVSpeaker_enter(ln:ListNat)=
  (#ln == 0)->(
    intern
    . sum vl:Nat. ( 
      rRead(vlevel,vl)
      . write_volume(vl)
    )
    . TVSpeaker_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: SpeakerOn
      sWrite(TVSpeaker,1)
    )
    + (head(ln) == 2 )->( % State: Mute
      sWrite(TVSpeaker,2)
      . write_volume(0)
    )
  )
);

proc Volume_enter(ln:ListNat)=
  (#ln == 0)->(
    intern.Volume_enter([1])
  )<>(
    (head(ln) == 1 )->( % State: VolumeLevel
      sWrite(Volume,1)
    )
  )
);

proc OSD_enter(ln:ListNat)=
  (#ln == 0)->(
    intern.OSD_enter([1])
  )<>(
    sWrite(OSD,1)
    . SCOSD_enter(if(head(ln) ==1,tail(ln),[]))
    . sum new:Nat. (rRead(OSD,new).(new == 1)->(
      sWrite(OSD,2)
      . StatusOSD_enter(if(head(ln) ==2,tail(ln),[]))
    )<>(
      intern
    ))
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

```
sum new:Nat.(rRead(OSD,new).(new == 2)->(
    sWrite(OSD,3)
    . PresetOSD_enter(if(head(ln) ==3,tail(ln),[]))
 )<>(
    intern
))

sum new:Nat.(rRead(OSD,new).(new == 3)->(
    sWrite(OSD,4)
    . MuteOSD_enter(if(head(ln) ==4,tail(ln),[]))
 )<>(
    intern
))
))
}

proc SCOSD_enter(ln:ListNat)=
( (#ln == 0)->(
    intern.SCOSD_enter([1])
 )<>(
    (head(ln) == 1 )->(  % State: SCOff
    sWrite(SCOSD,1)
    . start(Enter_SCOff)
    . StatusOSD_exec(Enter_SCOff)
    . PresetOn_exec(Enter_SCOff)
    . write_SCOSDDisplay(0)
    )
  + (head(ln) == 2 )->(  % State: SCO
    sWrite(SCOSD,2)
    . start(Enter_SCOn)
    . StatusOSD_exec(Enter_SCOn)
    . PresetOn_exec(Enter_SCOff)
    . sWrite(SCOSDentry,2)
    . SCOn_enter(tail(ln))
    )
  )
))
}

proc SCOn_enter(ln:ListNat)=
( (#ln == 0)->(
    intern.SCOn_enter([1])
 )<>(
    (head(ln) == 1 )->(  % State: VolumeBar
    sWrite(SCOn,1)
    . VolumeBar_enter(tail(ln))
    )
  )
))
}
```
proc VolumeBar_enter(ln:ListNat) =
( (#ln == 0)->(
   intern.delta
 )<>(
   (head(ln) == 1 )->( % State: Show
     sWrite(VolumeBar,1)
     . sWrite(SCOSDentry,2)
     . Show_enter(tail(ln))
   )
   + (head(ln) == 2 )->( % State: Suppress
     sWrite(VolumeBar,2)
   )
 ))
;

proc Show_enter(ln:ListNat) =
( (#ln == 0)->(
   (1 == 0)->( %minimal
     intern.Show_enter([2])
   )<>(
     intern.Show_enter([1])
   )
 )<>(
   (head(ln) == 1 )->( % State: nobar
     sWrite(Show,1)
     . write_SCOSDDisplay(1)
   )
   + (head(ln) == 2 )->( % State: bar
     sWrite(Show,2)
     . write_SCOSDDisplay(2)
   )
 ))
;

proc StatusOSD_enter(ln:ListNat) =
( (#ln == 0)->(
   intern.StatusOSD_enter([1])
 )<>(
   (head(ln) == 1 )->( % State: StatusOff
     sWrite(StatusOSD,1)
     . write_StatusDisplay(0)
   )
   + (head(ln) == 2 )->( % State: StatusOn
     sWrite(StatusOSD,2)
     . start(Enter_StatusOn)
     . SCOSD_exec(Enter_StatusOn)
     . PresetOn_exec(Enter_StatusOn)
     . sWrite(Statusentry,2)
     . write_StatusDisplay(1)
   )
))
proc PresetOSD_enter(ln:ListNat) =
  ( (#ln == 0) ->
    intern.PresetOSD_enter([2])
  )<>(
    (head(ln) == 1) ->
      % State: PresetOff
      sWrite(PresetOSD,1)
      . write_PresetDisplay(0)
    ) +
    (head(ln) == 2) ->
      % State: PresetOn
      sWrite(PresetOSD,2)
      . PresetOn_enter(tail(ln))
  )
)

proc PresetOn_enter(ln:ListNat) =
  ( (#ln == 0) ->
    sum st:Nat. ( rRead(StatusOSD,st).(st == 2) ->
      PresetOn_enter([1])
    ) <>
    sum a:Nat. ( rRead(app,a).(( 1==0 )&& ( a==0 )) ->
      %minimal
      PresetOn_enter([2])
    ) <>
    PresetOn_enter([3])
  )
)

proc MuteOSD_enter(ln:ListNat) =
  ( (#ln == 0) ->
    intern.MuteOSD_enter([1])
  ) <>

proc MuteOSD_exit(ln:ListNat) =
  ( (#ln == 0) ->
    intern.MuteOSD_exit([1])
  ) <>

(head(ln) == 1)->( % State: MuteOff
    sWrite(MuteOSD,1)
    . write_MuteDisplay(0)
 )
+ (head(ln) == 2 )->( % State: MuteOn
    sWrite(MuteOSD,2)
    . write_MuteDisplay(1)
 )
+ (head(ln) == 3 )->( % State: noMute
    sWrite(MuteOSD,3)
    . write_MuteDisplay(2)
 )
))
;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Exit processes %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc TVbehavior_exit=
  sum state:Nat.rRead(TVbehavior,state).( (state == 0)->( intern )
    + (state == 1 )->( %State: TV
        TV_exit
        . sWrite(TVbehavior,0)
     )
  )
;

proc TV_exit=
  sum state:Nat.rRead(TV,state).( (state == 0)->( intern )
    + (state == 1 )->( %State: Off
        sWrite(TV,0)
     )
    + (state == 2 )->( %State: On
        On_exit
        . sWrite(TV,0)
     )
  )
;

proc On_exit=
  sum state:Nat.rRead(On,state).( (state == 0)->( intern )
    + (state == 1 )->( %State: Off
        sWrite(On,0)
     )
    + (state == 2 )->( %State: On
        On_exit
        . sWrite(On,0)
     )
  )
;
(state == 0)->(
  intern
 )
+ (state == 1 )->(%State: IT
  IT_exit
 )
+ (state == 2 )->(%State: TVModes
  TVModes_exit
 )
).
.sum new:Nat.rRead(On,new).(new == state)->(
  sWrite(On,max(state-1,0))
  . (1<state)->(On_exit)<>(intern)
 )<>(
  intern
 )
);

proc IT_exit=
  sum state:Nat.rRead(IT,state).
  (state == 0)->(
    intern
  )
+ (state == 1 )->(%State: DigitEvent
    DigitEvent_exit
  )
+ (state == 2 )->(%State: ChannelChange
    ChannelChange_exit
  )
+ (state == 3 )->(%State: TXTNrChange
    TXTNrChange_exit
  )
+ (state == 4 )->(%State: WakeUpEvent
    WakeUpEvent_exit
  )
).
  sum new:Nat.rRead(IT,new).(new == state)->(
    sWrite(IT,max(state-1,0))
    . (1<state)->(IT_exit)<>(intern)
  )<>(
    intern
  )
);

proc DigitEvent_exit=
  sum state:Nat.rRead(DigitEvent,state).
  (state == 0)->(
    intern
  )
}
+ (state == 1 )->( %State: DEInit
    sWrite(DigitEvent,0)
 )
)
)
); 

proc ChannelChange_exit=
  sum state:Nat.rRead(ChannelChange,state).( (state == 0)->(
    intern
  )
  + (state == 1 )->( %State: CCinit
    sWrite(ChannelChange,0)
  )
)
);

proc TXTNrChange_exit=
  sum state:Nat.rRead(TXTNrChange,state).( (state == 0)->(
    intern
  )
  + (state == 1 )->( %State: TNInit
    sWrite(TXTNrChange,0)
  )
)
);

proc WakeUpEvent_exit=
  sum state:Nat.rRead(WakeUpEvent,state).( (state == 0)->(
    intern
  )
  + (state == 1 )->( %State: WEInit
    sWrite(WakeUpEvent,0)
  )
)
);

proc TVModes_exit=
  sum state:Nat.rRead(TVModes,state).( (state == 0)->(
    intern
  )
  + (state == 1 )->( %State: Sleep
    sWrite(TVModes,0)
  )
)
+ (state == 2 )->( %State: Active
   Active_exit
   . sWrite(TV Modes, 0)
   )
);

proc Active_exit=
  sum state:Nat.rRead(Active, state).(
    (state == 0)->(
      intern
    )
    + (state == 1 )->( %State: Video
    Video_exit
    )
    + (state == 2 )->( %State: Audio
    Audio_exit
    )
    + (state == 3 )->( %State: OSD
    OSD_exit
    )
  )
  . sum new:Nat.rRead(Active, new).(new == state)->(
    sWrite(Active, max(state-1, 0))
    . (1<state)->(Active_exit)<>(intern)
  )<>(
    intern
  )
);

proc Video_exit=
  sum state:Nat.rRead(Video, state).(
    (state == 0)->(
      intern
    )
    + (state == 1 )->( %State: SingleScreen
    SingleScreen_exit
    . sWrite(Video, 0)
    )
    + (state == 2 )->( %State: DualScreen
    DualScreen_exit
    . sWrite(Video, 0)
    )
  )
);

proc SingleScreen_exit=
sum state:Nat.rRead(SingleScreen,state).(  
  (state == 0)->(  
    intern  
  )  
)  
+ (state == 1 )->( %State: Menu  
  sWrite(app,0)  
  . start(Change_app)  
  . OSD_exec(Change_app)  
  . sWrite(SingleScreen,0)  
)  
+ (state == 2 )->( %State: SSTV  
  sWrite(SingleScreen,0)  
)  
+ (state == 3 )->( %State: SSTXT  
  sWrite(app,0)  
  . start(Change_app)  
  . OSD_exec(Change_app)  
  . sWrite(SingleScreen,0)  
)  
)  
);  

proc DualScreen_exit=  
  sum state:Nat.rRead(DualScreen,state).(  
    (state == 0)->(  
      intern  
    )  
)  
+ (state == 1 )->( %State: Main  
  Main_exit  
)  
+ (state == 2 )->( %State: Aux  
  Aux_exit  
)  
)  
sum new:Nat.rRead(DualScreen,new).(new == state)->(  
  sWrite(DualScreen,max(state-1,0))  
  . (1<state)->(DualScreen_exit)<>(intern)  
)<>(  
  intern  
)  
);  

proc Main_exit=  
  sum state:Nat.rRead(Main,state).(  
    (state == 0)->(  
      intern  
    )  
)  
+ (state == 1 )->( %State: MainTV  
  Main_exit  
)  
+ (state == 2 )->( %State: Aux  
  Aux_exit  
)  
)  
);
sWrite(Main,0)
)
)
)

proc Aux_exit=
  sum state:Nat.rRead(Aux,state).(
    (state == 0)->(
      intern
    )
    + (state == 1 )->( %State: AuxTV
      sWrite(Aux,0)
    )
    + (state == 2 )->( %State: AuxTXT
      sWrite(Aux,0)
    )
  )
)

proc Audio_exit=
  sum state:Nat.rRead(Audio,state).(
    (state == 0)->(
      intern
    )
    + (state == 1 )->( %State: TVSpeaker
      TVSpeaker_exit
    )
    + (state == 2 )->( %State: Volume
      Volume_exit
    )
  ). sum new:Nat.rRead(Audio,new).(new == state)->(
    sWrite(Audio,max(state-1,0))
    . (1<state)->(Audio_exit)<>(intern)
  )<>(
    intern
  )
)

proc TVSpeaker_exit=
  sum state:Nat.rRead(TVSpeaker,state).(
    (state == 0)->(
      intern
    )
    + (state == 1 )->( %State: SpeakerOn
      sWrite(TVSpeaker,0)
    )
    + (state == 2 )->( %State: Mute
      sWrite(TVSpeaker,0)
    )
  )
)
LXXI

    sum v: Nat. (rRead(vlevel, v).write_volume(v))
    . sWrite(TVSpeaker, 0)
    )
    )
    ;
    
    proc Volume_exit=
    sum state: Nat. rRead(Volume, state). (state == 0)->(
        intern
    )
    + (state == 1)->( %State: VolumeLevel
        sWrite(Volume, 0)
    )
    )
    ;
    
    proc OSD_exit=
    sum state: Nat. rRead(OSD, state). (state == 0)->(
        intern
    )
    + (state == 1)->( %State: SCOSD
        SCOSD_exit
    )
    + (state == 2)->( %State: StatusOSD
        StatusOSD_exit
    )
    + (state == 3)->( %State: PresetOSD
        PresetOSD_exit
    )
    + (state == 4)->( %State: MuteOSD
        MuteOSD_exit
    )
    )
    . sum new: Nat. rRead(OSD, new). (new == state)->(
        sWrite(OSD, max(state-1, 0))
    . (1<state)->( OSD_exit)<>(intern)
    )>(
        intern
    )
    ;
    
    proc SCOSD_exit=
    sum state: Nat. rRead(SCOSD, state). (state == 0)->(
        intern
    )
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

+ (state == 1)->( %State: SCOff
  sWrite(SCOSD,0)
)
+ (state == 2)->( %State: SCOn
  SCOn_exit
  . sWrite(SCOSD,0)
)
;

proc SCOn_exit=
  sum state:Nat.rRead(SCOn,state).(
    (state == 0)->(
      intern
    )
    + (state == 1)->( %State: VolumeBar
      VolumeBar_exit
      . sWrite(SCOn,0)
    )
  )
;

proc VolumeBar_exit=
  sum state:Nat.rRead(VolumeBar,state).(
    (state == 0)->(
      intern
    )
    + (state == 1)->( %State: Show
      Show_exit
      . sWrite(VolumeBar,0)
    )
    + (state == 2)->( %State: Suppress
      sWrite(VolumeBar,0)
    )
  )
;

proc Show_exit=
  sum state:Nat.rRead(Show,state).(
    (state == 0)->(
      intern
    )
    + (state == 1)->( %State: nobar
      sWrite(Show,0)
    )
    + (state == 2)->( %State: bar
      sWrite(Show,0)
    )
  )
;
sWrite(Show,0)
)
;

proc StatusOSD_exit=
  sum state:Nat.rRead(StatusOSD,state).( (state == 0)->(
    intern
  ) + (state == 1 )->( %State: StatusOff
    sWrite(StatusOSD,0)
  ) + (state == 2 )->( %State: StatusOn
    sWrite(StatusOSD,0)
  ) )
;

proc PresetOSD_exit=
  sum state:Nat.rRead(PresetOSD,state).( (state == 0)->(
    intern
  ) + (state == 1 )->( %State: PresetOff
    sWrite(PresetOSD,0)
  ) + (state == 2 )->( %State: PresetOn
    PresetOn_exit
    . sWrite(PresetOSD,0)
  ) )
;

proc PresetOn_exit=
  sum state:Nat.rRead(PresetOn,state).( (state == 0)->(
    intern
  ) + (state == 1 )->( %State: Suppressed
    sWrite(PresetOn,0)
  ) + (state == 2 )->( %State: PresetFull
    sWrite(PresetOn,0)
  ) + (state == 3 )->( %State: PresetNumber
    sWrite(PresetOn,0)
  )
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

)
);

proc MuteOSD_exit=
  sum state:Nat.rRead(MuteOSD,state).(  
    (state == 0)->(  
      intern  
    )  
    + (state == 1 )->( %State: MuteOff  
      sWrite(MuteOSD,0)  
    )  
    + (state == 2 )->( %State: MuteOn  
      sWrite(MuteOSD,0)  
    )  
    + (state == 3 )->( %State: noMute  
      sWrite(MuteOSD,0)  
    )  
  )
);

State space generation script

mcrl22lps -Dvbf0 tv.mcrl2 tv.lps  
lpssuminst -v tv.lps | lpsrewr -v > temp.lps  
lps2lts -vRjittyc -Cctau -r temp.lps tv.aut

Modal formulas

Requirement 1

(  
  < true*  
  . "cWrite(TV Modes, 2)"  
  > true  
) and(  
  [ true*  
    . "cWrite(TV Modes, 2)"  
  ]  
  mu X.  
  ( <true> true  
    and ( [not "cWrite(TV Speaker, 1)"] X )  
  )  
)
Requirement 2

(  
  [ true*  
    .'write_volume\([0-2]\|\d\d+\)\$'  
    ]  
  false  
)and(  
  [ true*  
    .'write_volume\(-\d+\)\$'  
    ]  
  false  
)

Requirement 3

(  
  < true*  
    ."cWrite(TVSpeaker, 1)"  
    .(not "cWrite(TVSpeaker, 0)")*  
    ."start(MuteOnOff)"  
  >  
  true  
)and(  
  [ true*  
    ."cWrite(TVSpeaker, 1)"  
    .(not "cWrite(TVSpeaker, 0)")*  
    ."start(MuteOnOff)"  
  ]  
  ( mu X.  
    ( <true> true  
      and [ not "cWrite(TVSpeaker, 2)"] X  
    )  
  )  
)

Requirement 4

(  
  [ true*  
    ."cWrite(TVSpeaker, 1)"  
    .(not "start(MuteOnOff)" )*  
    ."cWrite(TVSpeaker, 2)"  
  ]  
  false  
)

Requirement 5

(  
  [ true*  

. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")
. "write_volume(0)"
. (not ("cWrite(TVSpeaker, 0)"
    or "write_volume(1)"
    or "write_volume(2)"
)
)*
. "start(VolumeUp)"
]
( μX.
  ( <true> true
    and [not "write_volume(1)"] X
  )
)
)
and(
[ true*
. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")
. "write_volume(1)"
. (not ("cWrite(TVSpeaker, 0)"
    or "write_volume(0)"
    or "write_volume(2)"
)
)*
. "start(VolumeUp)"
]
( μX.
  ( <true> true
    and [not "write_volume(2)"] X
  )
)
)

Requirement 6

(
[ true*
. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")
. "write_volume(2)"
. (not ("cWrite(TVSpeaker, 0)"
    or "write_volume(0)"
    or "write_volume(1)"
)
)*
. "start(VolumeDown)"
]
( μX.
  ( <true> true
}
and [not "write_volume(1)"] X
)
)

)>and(
['true*
. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")*
. "write_volume(1)"
. (not ("cWrite(TVSpeaker, 0)"
   or "write_volume(0)"
   or "write_volume(2)"
)
)*
. "start(VolumeDown)"
]
( mu X.
( <true> true
   and [not "write_volume(0)"] X
   )
)
)

Requirement 7
(
[ 'true*
. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")*
. "start(MuteOnOff)"
]
( mu X.
( <true> true
   and [not "write_volume(0)"] X
   )
)
)>and(
[ 'true*
. "cWrite(TVSpeaker, 1)"
. (not "cWrite(TVSpeaker, 0)")*
. "start(MuteOnOff)"
. (not "cWrite(TVSpeaker, 0)")*
. ( "write_volume(1)"
  | "write_volume(2)"
  )
]
false
)
Requirement 8

\[
( [ \text{true}^* \\
. "cWrite(TVSpeaker, 1)" \\
. (not "cWrite(TVSpeaker, 0)" )^* \\
. "start(MuteOnOff)"
] \\
( \mu X. \\
. <true> true \\
. and [not "write_MuteDisplay(2)"] X 
)
).
)
\]

\text{and}

\[
( [ \text{true}^* \\
. "cWrite(TVSpeaker, 2)" \\
. (not ( "cWrite(TVSpeaker, 0)" \\
. or "start(Change_app)" \\
. )^* \\
. ( "write_MuteDisplay(1)" \\
. | "write_MuteDisplay(3)"
]) \\
false
)
)
\]

Requirement 9

\[
( [ \text{true}^* \\
. "cWrite(TVSpeaker, 2)" \\
. (not "cWrite(TVSpeaker, 0)" )^* \\
. "start(MuteOnOff)"
] \\
( \mu X. \\
. <true> true \\
. and [not "cWrite(TVSpeaker, 1)"] X 
)
).
)
\]

Requirement 10

\[
( [ \text{true}^* \\
. "cWrite(TVSpeaker, 2)" \\
. (not "cWrite(TVSpeaker, 0)" )^* \\n. or "write_MuteDisplay(1)" \\
. | "write_MuteDisplay(3)"
]) \\
false
)
\]
. "start(MuteOnOff)"
] ( mu X.
  ( <true> true
   and [not "cWrite(MuteOSD, 3)"] X )
 )
)and(
  [ true*
   . "write_MuteDisplay(3)"
 ] ( mu X.
   ( <true> true
     and [ not ("write_MuteDisplay(1)"
                 or "write_MuteDisplay(2)"
                 or "write_ScreenMode(0)"
                 or "write_ScreenMode(1)"
                 ) X
     )
   )
 )
)

Requirement 11
(**)

( [ true*
  . "cWrite(TVSpeaker, 1)"
  . ( not "cWrite(TVSpeaker, 0)" )*
  . "write_volume(0)"
  . ( not ("cWrite(TVSpeaker, 0)"
            or "write_volume(1)"
            or "write_volume(2)"
            )*
    )
  . "start(MuteOnOff)"
  . ( not "cWrite(TVSpeaker, 2)" )*
  . "cWrite(TVSpeaker, 2)"
  . (not ( "start(MuteOnOff)"
            or "start(VolumeUp)"
            or "start(VolumeDown)"
            or "write_volume(1)"
            or "write_volume(2)"
            or "cWrite(TVSpeaker, 0)"
          )
    )*)
APPENDIX D. MCRL2 SPECIFICATION OF EXAMPLE 3

```mcrl2
[ true
  . "start(MuteOnOff)"
] ( mu X.
  ( <true> true
    and [not "write_volume(0)"] X )
)
)

[ true*
  . "cWrite(TVSpeaker, 1)"
  . ( not "cWrite(TVSpeaker, 0)" )*
  . "write_volume(1)"
  . ( not (
      "cWrite(TVSpeaker, 0)"
      or "write_volume(0)"
      or "write_volume(2)"
    )*
  )
  . "start(MuteOnOff)"
  . ( not "cWrite(TVSpeaker, 2)" )*
  . "cWrite(TVSpeaker, 2)"
  . (not ( "start(MuteOnOff)"
         or "start(VolumeUp)"
         or "start(VolumeDown)"
         or "write_volume(0)"
         or "write_volume(2)"
         or "cWrite(TVSpeaker, 0)"
      )
  )*
  . "start(MuteOnOff)"
] ( mu X.
  ( <true> true
    and [not "write_volume(1)"] X )
)
)

[ true*
  . "cWrite(TVSpeaker, 1)"
  . ( not "cWrite(TVSpeaker, 0)" )*
  . "write_volume(2)"
  . ( not (
      "cWrite(TVSpeaker, 0)"
      or "write_volume(1)"
      or "write_volume(0)"
    )*
  )
  . "start(MuteOnOff)"
  . ( not "cWrite(TVSpeaker, 2)" )*
  . "cWrite(TVSpeaker, 2)"
```
(not ( "start(MuteOnOff)"
    or "start(VolumeUp)"
    or "start(VolumeDown)"
    or "write_volume(1)"
    or "write_volume(0)"
    or "cWrite(TVSpeaker, 0)"
  )
)

"start(MuteOnOff)"
]

( mu X.
  ( <true> true
    and [not "write_volume(2)"] X
  )
)

)and

(< true*
  . "cWrite(TVSpeaker, 1)"
  . ( not "cWrite(TVSpeaker, 0)" )*
  . "write_volume(0)"
  . ( not ( "cWrite(TVSpeaker, 0)"
    or "write_volume(1)"
    or "write_volume(2)"
  )*
    . "start(MuteOnOff)"
    . ( not "cWrite(TVSpeaker, 2)" )*
    . "cWrite(TVSpeaker, 2)"
    . (not ( "start(MuteOnOff)"
      or "start(VolumeUp)"
      or "start(VolumeDown)"
      or "write_volume(1)"
      or "write_volume(2)"
    )
  )
  . "start(MuteOnOff)"
  . ( not "cWrite(TVSpeaker, 2)" )*
  . "cWrite(TVSpeaker, 2)"
)

Requirement 12

NA
Requirement 13

NA