Analysing the Control Software of the
Compact Muon Solenoid Experiment
at the Large Hadron Collider

Yi-Ling Hwong\textsuperscript{1,*}, Vincent J.J. Kusters\textsuperscript{1,2}, and Tim A.C. Willemse\textsuperscript{2}

\textsuperscript{1} CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
\textsuperscript{2} Department of Mathematics and Computer Science, Eindhoven University of Technology P.O. Box 513, 5600 MB Eindhoven, The Netherlands

\textbf{Abstract.} The control software of the CERN Compact Muon Solenoid experiment contains over 30,000 finite state machines. These state machines are organised hierarchically: commands are sent down the hierarchy and state changes are sent upwards. The sheer size of the system makes it virtually impossible to fully understand the details of its behaviour at the macro level. This is fuelled by unclarities that already exist at the micro level. We have solved the latter problem by formally describing the finite state machines in the mCRL2 process algebra. The translation has been implemented using the \textit{ASF+SDF meta-environment}, and its correctness was assessed by means of simulations and visualisations of individual finite state machines and through formal verification of subsystems of the control software. Based on the formalised semantics of the finite state machines, we have developed dedicated tooling for checking properties that can be verified on finite state machines in isolation.

\section{Introduction}

The Large Hadron Collider (LHC) experiment at the European Organization for Nuclear Research (CERN) is built in a tunnel 27 kilometres in circumference and is designed to yield head-on collisions of two proton (ion) beams of 7 TeV each. The Compact Muon Solenoid (CMS) experiment is one of the four big experiments of the LHC. It is a general purpose detector to study the wide range of particles and phenomena produced in the high-energy collisions in the LHC. The CMS experiment is made up of 7 subdetectors, with each of them designed to stop, track or measure different particles emerging from the proton collisions. Early 2010, it achieved its first successful 7 TeV collision, breaking its previous world record, setting a new one.

The control, configuration, readout and monitoring of hardware devices and the detector status, in particular various kinds of environment variables such as temperature, humidity, high voltage, and low voltage, are carried out by the Detector Control System

\* This work has been supported in part by a Marie Curie Initial Training Network Fellowship of the European Community’s Seventh framework program under contract number (PITN-GA-2008-211801-ACEOLE).
The control software of the CMS detector is implemented with the Siemens commercial Supervision, Control And Data Acquisition (SCADA) package PVSS-II and the CERN Joint Controls Project (JCOP) framework [9]. The architecture of the control software for all four big LHC experiments is based on the SMI++ framework [5, 6]. Under the SMI++ framework, the real world is viewed as a collection of objects behaving as finite state machines (FSMs). These FSMs are described using the State Manager Language (SML). A characteristic of the used architecture is the regularity and relatively low complexity of the individual FSMs and device drivers that together constitute the control software; the main source of complexity is in the cooperation of these FSMs. Cooperation is strictly hierarchical, consisting of several layers; see Figure 1 for a schematic overview. The FSMs are organised in a tree structure where every node has one parent and zero or more children, except for the top node, which has no parent. Nodes communicate by sending commands to their children and state updates to their parents, so commands are refined and propagated down the hierarchy and status updates are sent upwards. Hardware devices are typically found only at the bottom-most layer.

The FSM system in the CMS experiment contains over 30,000 nodes. On average, each FSM contains 5 logical states. Based on our early experiments with some subsystems, we believe that $10^{30,000}$ states is a very conservative estimate of the size of the state space for the full control system. The sheer size of the system significantly contributes to its complexity. Complicating factors in understanding the behaviour of the system are the diversity in the development philosophies in subgroups responsible for controlling their own subdetectors, and the huge amount of parameters to be monitored. In view of this complexity, it is currently impossible to trace the root cause of problems when unexpected behaviours manifest themselves. A single badly designed FSM may be sufficient to lead to a livelock, resulting in non-responsive hardware devices, potentially ruining expensive and difficult experiments. Considering the scientific importance of these experiments, this justifies the use of rigorous methods for understanding and analysing the system.

Our contributions are twofold. First, we have formalised SML by mapping its language constructs onto constructs in the process algebraic language mCRL2 [7]. Second, based on our understanding of the semantics of SML, we have identified properties that
can be verified for FSMs in isolation, and for which we have developed dedicated veri-
ification tooling.

Using the ASF+SDF meta-environment [12], we have developed a prototype trans-
lation implementing our mapping of SML to mCRL2. This allowed us to quickly as-
ssess the correctness of the translation through simulation and visualisation of FSMs
in isolation, and by means of formal verification of small subsystems of the control
software, using the mCRL2 toolset. The feedback obtained by the verification and sim-
ulation enabled us to further improve the transformation. The use of the ASF+SDF
meta-environment allowed us to repeat this cycle in quick successions, and, at the same
time, maintain a formal description of the translation. Although the ASF+SDF Meta
Environment development was discontinued in 2010, we chose it over similar prod-
ucts as ATL because we were already familiar with it and because its syntax-driven,
functional approach results in very clear translation rules.

Our dedicated verification tools allow the developers at CERN to quickly perform
behavioural sanity checks on their design, and use the feedback of the tools to further
improve on their designs in case of any problems. Results using these tools so far are
favourable: with only a fraction of the total number of FSMs inspected so far, several
problems have surfaced and have been fixed.

Outline We give a cursory overview of the core of the SML language in Section 2. The
mCRL2 semantics of this core are then explained in Section 3, and we briefly elaborate
on the methodology we used for obtaining this semantics. Our dedicated verification
tools for SML, together with some of the results obtained so far, are described in further
detail in Section 4. We summarise our findings and suggestions in Section 5.

2 The State Manager Language

The finite state machines used in the CMS experiment are described in the State Man-
ger Language (SML) [5, 6]. We present the syntax and the suggested meaning of the
core of the language using snapshots of a running example; we revisit this example in
our formalisation in Section 3. Note that in reality, SML is larger than presented here,
but the control system is made up largely of FSMs employing these core constructs
only.

Listing 1 shows part of the definition of a class in SML. Conceptually, this is the
same kind of class known from object-oriented programming: the class is defined once,
but can be instantiated many times. An instantiation is referred to as a Finite State
Machine. A class consists of one or more state clauses; Listing 1 only shows the state
clause for the OFF state. Intuitively, a state clause describes how the FSM should behave
when it is in a particular state. Every state clause consists of a list of when clauses and
a list of action clauses, either of which may be empty.

A when clause has two parts: a guard which is a Boolean expression over the states
of the children of the FSM and a referer which describes what should happen if the
guard evaluates to true. The base form of a guard is $P$ in state $S$, where $S$ is the
name of a state (or a set of state names) and $P$ is a child pattern. A child pattern consists
of two parts: the first part is either ANY or ALL and the second part is the name of a class
or the literal $\text{FwCHILDREN}$. The intended meaning is straightforward:
class: $FWPART\_TOP\$RPC\_Chamber\_CLASS
state: OFF
  when ( ( $ANY\$FwCHILDREN in_state ERROR ) or
    ( $ANY\$FwCHILDREN in_state TRIPPED ) ) move_to ERROR
  when ( $ANY\$RPC\_HV in_state {RAMPING\_UP,
    RAMPING\_DOWN} ) move_to RAMPING
  when ( ( $ALL\$RPC\_LV in_state ON ) and
    ( $ALL\$RPC\_HV in_state STANDBY ) ) move_to STANDBY
  when ( ( $ALL\$RPC\_HV in_state ON ) and
    ( $ALL\$RPC\_LV in_state ON ) ) move_to ON
  when ( ( $ALL\$FwCHILDREN in_state ON ) and
    ( $ALL\$RPC\_T in_state OK ) ) move_to ON
action: STANDBY
  do STANDBY $ALL\$RPC\_HV
  do ON $ALL\$RPC\_LV
action: OFF
  do OFF $ALL\$FwCHILDREN
action: ON
  do ON $ALL\$FwCHILDREN

Listing 1: Part of the definition of the Chamber class in SML.

$ALL\$FwCHILDREN in_state ON
means “all children are in the ON state”, and:

$ANY\$RPC\_HV in_state {RAMPING\_UP, RAMPING\_DOWN}
evaluates to true if “some child of class RPC\_HV is either in state RAMPING\_UP or state RAMPING\_DOWN”.

A referer is either of the form move_to S, indicating that the finite state machine changes its state to S, or of the form do A, indicating that the action with name A should be executed next. If the guards of more than one when clause evaluate to true, the topmost enabled referer is executed. Whenever the FSM moves to a new state, it executes the when clauses, starting from the top when clause, to see if it should stay in this state (all guards are false) or if it should go to another state (some guard is true). It is therefore possible that a single move_to referer or statement (see below) triggers a series of state changes.

An action clause consists of a name and a list of statements. When an FSM receives a command while in a state S, it looks inside the state clause of state S for an action clause with the same name as the command and if such an action clause exists, it executes its statement list. If no such action exists, the command is ignored. For example, if the Chamber finite state machine from Listing 1 is in state OFF and it receives an ON command, it will execute the last action clause.
The most commonly used statement is \texttt{do C P}, which means that the command C is sent to all children which match the child pattern P. After a command is sent, the child is marked \textit{busy}. When a child sends its new state back, this \textit{busy} flag is removed. The \texttt{do} statement is non-blocking, \textit{i.e.}, it does not wait for the children to respond with their new state. The child pattern always starts with \texttt{ALL} in this context. SML also provides \texttt{if} and \texttt{move_to} statements, as we illustrated in Listing 2.

\begin{verbatim}
action: STANDBY
  do STANDBY $ALL$RPC_HV
  do ON $ALL$RPC_LV
  if ( $ALL$RPC_LV in_state ON ) then
    do ON $ALL$RPC_HV
    do ON $ALL$RPC_LV
    if ( $ALL$RPC_HV in_state ON ) then
      do ON $ALL$RPC_HV
      move_to ON
    endif
  else
    do STANDBY $ALL$RPC_LV
    do STANDBY $ALL$RPC_HV
    do STANDBY $ALL$FwCHILDREN
  endif
\end{verbatim}

\textbf{Listing 2:} An example of a more complex \textit{action} clause.

The \texttt{move_to S} statement immediately stops execution of the \textit{action clause} and causes the FSM to move to the S state. The \texttt{if G then S1 else S2 endif} statement blocks as long as there is a child, referred to in G, that has a busy flag. If the guard G evaluates to true, then S1 is executed and otherwise S2 is executed. The else clause is optional.

3 A Formal Semantics for SML

We use the process algebra mCRL2 [7] to formalise the semantics of programs written in SML. The formal translation of SML into mCRL2 can be found in the appendices.

Our choice for mCRL2 is motivated largely by the expressive power of the language, its rich data language rooted in the theory of Abstract Data Types, its available tool support, and our understanding of the advantages and disadvantages of mCRL2. Before we address the translation of SML to mCRL2, we briefly describe the mCRL2 language.

3.1 A Brief Overview of mCRL2

The mCRL2 language consists of two distinct parts: a \textit{data language} for describing the data transformations and data types, and a \textit{process language} for specifying system behaviours. For a comprehensive language tutorial, we refer to \url{http://mcrl2.org}.
The data language, which is rooted in the theory of *abstract data types*, includes built-in definitions for many of the commonly used data types, such as Booleans, Integers, Natural numbers, etc., and allows users to specify their own data sorts. In addition, container sorts, such as lists, sets and bags are available.

The process specification language of mCRL2 consists of only a small number of basic operators and primitives. The language is inspired by process algebras such as ACP [1], and has both an axiomatic and an operational semantics.

A set of (parameterised) actions are used to model atomic, observable events. Processes are constructed compositionally: the non-deterministic choice between processes \( p \) and \( q \) is denoted \( p+q \); their sequential composition is denoted \( p \cdot q \), and their parallel composition is denoted \( p \parallel q \). In addition, there are facilities to enforce communication between different actions and abstracting from actions.

The main feature of the process language is that processes can depend on data. For instance, \( b \rightarrow p<>q \) denotes a conditional choice between processes \( p \) and \( q \): if \( b \) evaluates to *true*, it behaves as process \( p \), and otherwise as process \( q \). In a similar vein, \( \sum d:D.p(d) \) describes a (possibly infinite) choice between processes \( p \) with different values for variable \( d \).

### 3.2 From SML to mCRL2

We next present our formalisation of SML in mCRL2. Every SML class is converted to an mCRL2 process definition; the behaviour of an FSM is then described by the behaviour of a process instance. Each FSM maintains a state and a pointer to the code it is currently executing. In addition, an FSM is embedded in a global tree-like configuration that identifies its parent, and its children. In order to faithfully describe the behaviour of an FSM, we therefore equip each mCRL2 process definition for a class \( X \) with this information as follows:

\[
\text{proc } X\text{.CLASS}(\text{self}: \text{Id}, \text{parent: Id, s: State, chs: Children, phase: Phase, aArgs: ActPhaseArgs})
\]

Parameter \( \text{self} \) represents a unique identifier for a process instance, and \( \text{parent} \) is the identifier of \( \text{self}'s \) parent in the tree. Parameter \( s \) is used to keep track of the state of the FSM. The state information of \( \text{self}'s \) children is stored in \( \text{chs} \) of sort \( \text{Children} \), which is a list of sort \( \text{Child} \), a structured sort:

\[
\text{Children} = \text{List(Child)};
\text{Child} = \text{struct child(id:Id, state:State, ptype:PType, busy:Bool)};
\]

The above structured sort \( \text{Child} \) can be thought of as a named tuple; \( \text{id} \) represents the unique identifier of a child, \( \text{state} \) is the state that this child sent to \( X \) in its last state-update message, \( \text{ptype} \) maintains the FSM class of this child, and \( \text{busy} \) is the flag that indicates that the child is still processing the last command \( X \) sent to it. This flag is set after sending a message to the child, and reset when it responds with its new state. Whenever \( X \) receives a state-update message from one of its children, the \( \text{chs} \) structure is updated accordingly. This structure is used to evaluate the *when clauses* and to determine to which processes commands have to be sent.
The phase parameter has value `WhenPhase` if the FSM is executing the `when clauses` and `ActionPhase` otherwise; `Phase` is a simple structured sort containing these two values. The phases will be explained in detail in the following section. Finally, `aArgs` is a structure that contains information we only need in the `action phase`. It is defined as follows:

\[
\text{ActPhaseArgs} = \text{struct actArgs}(\text{cq: CommandQueue, nrf: IdList, pc: Int, rsc: Bool})
\]

We forego a discussion of the `nrf` and `rsc` parameters, which are solely used during an initialisation phase. The command queue `cq` contains messages that are to be sent to an FSM’s children. Specifically, when executing a `do C P` statement, we add a pair with the child’s id and the command `C` to `cq`, for every child matching the child pattern `P`. The command queue is subsequently emptied by sending the messages stored in `cq`.

---

**Fig. 2.** Overview of the `when phase` and the `action phase`.

**Phases** During the `when phase`, a process executes `when clauses` until it reaches a state in which none of the guards evaluate to true. It then moves to the `action phase`. In the `action phase`, a process can receive a command from its parent or a state-update message from one of its children. This process is illustrated in Figure 2. After handling the command or message, it returns to the `when phase`.

Translating the `when phase` turns out to be rather straightforward: for each state a process term consisting of nested if-then-else statements is introduced, formalised by
mCRL2 expressions of the form $b \rightarrow p<>q$ (if $b$, then act as process $p$, otherwise as $q$). Each if-clause represents exactly one when clause. The else-clause of the last when clause sends a state-update message (represented by the mCRL2 action $\text{send\_state}$) with the current state to the parent of this FSM and moves to the action phase. An example is given in Translation 1.

| SML                                                                 | mCRL2                                                                 |
|----------------------------------------------------------------------|------------------------------------------------------------------------|
| state: OFF                                                          | $\text{instate\_OFF}(s) \&\& \text{isWhenPhase}(\text{phase}) \rightarrow (\text{translation\_of\_G1} \rightarrow \text{move\_state}(\text{self,S1}).\text{X\_CLASS}(\text{self,}\text{parent,S1,chs,phase,aArgs}) <> \text{translation\_of\_Gn} \rightarrow \text{move\_state}(\text{self,Sn}).\text{X\_CLASS}(\text{self,}\text{parent,Sn,chs,phase,aArgs}) <> \text{send\_state}(\text{self,}\text{parent,s}).\text{move\_phase}(\text{self,ActionPhase}).\text{X\_CLASS}(\text{self,}\text{parent,s,chs,ActionPhase,reset(aArgs)}))$ |

**Translation 1:** Simplified translation of the when clauses of a state OFF. Note that $p.q$ describes the process $p$ that, upon successful termination, continues to behave as process $q$.

The move\_state action indicates that the process changes its state. The send\_state action communicates with the receive\_state action to a comm\_state action, representing the communication of the new state to the parent. Note that the state is sent only if none of the guards are true. Upon sending the new state, the process changes to the action phase, signalled by a move\_phase action.

Modelling the action phase is more involved as we need to add some terms for initialisation and sending messages. We will focus on the translation of the action clauses and the code which handles state-update messages.

SML allows for an arbitrary number of statements and an arbitrary number of (nested) if-statements in every action clause. We uniquely identify the translation of every statement with an integer label. After executing a statement, the pc(aArgs) program counter is set to the label of the statement which should be executed next. There are two special cases here:

- Label 0, the clause selector. When entering the action phase, the program counter is set to 0. Upon receiving a command, the clause selector sets the program counter to the label of the first statement of the action clause that should handle the command.
- Label -1, end of action. After executing an action, the program counter is set to -1, signalling that the command queue must be emptied and the process must change to the when phase.
An example is given in Translation 2. The receive_command action models the reception of a command that was sent by the FSM's parent. Such a command is ignored if no action clause handles it. In the example, observe that both after ignoring a command and after completing the execution of the STANDBY action handler, the program counter is set to -1. A process term not shown here then empties the command queue by issuing a sequence of send_command actions, and subsequently returns to the when phase. Note that these send_command actions and receive_command actions are meant to synchronise, resulting in a comm_command action. This is enforced at a higher level in the specification.

Translation 2: Simplified translation of the action clauses of a state OFF:

Since a do statement is asynchronous, the children can send their state-update at any time during the action phase. This is dealt with as follows. Suppose a state-update message is received. If this precedes the reception of a command in this action phase,
we simply process the state-update and move to the *when phase*. If we are in the middle of executing an *action clause*, we process the state-update, but do not move to the *when clause*.

### 3.3 Validating the Formalisation of SML

The challenge in formalising SML is in correctly interpreting its language constructs. We combined two strategies for assessing and improving the correctness of our semantics: informal discussions with the development team of the language and applying formal analysis techniques on sample FSMs taken from the control software.

The discussions with the SML development team were used to solidify our initial understanding of SML and its main constructs. Based on these discussions, we manually translated several FSMs into mCRL2, and validated the resulting processes manually using the available simulation and visualisation tools of mCRL2. This revealed a few minor issues with our understanding of the semantics of SML, alongside many issues that could be traced back to sloppiness in applying the translation from SML to mCRL2 manually.

In response to the latter problem, we eliminated the need for manually translating FSMs to mCRL2. To this end, we utilised the ASF+SDF meta-environment (see [12, 10]) to rapidly prototype an automatic translator that, ultimately, came to implement the translation scheme we described in the previous section. The *Syntax Definition Formalism* (SDF) was used to describe the syntax of both SML and mCRL2, whereas the *Algebraic Specification Formalism* (ASF) was used to express the term rewrite rules that are needed to do the actual translation. Apart from the gains in speed and the consistency in applying the transformations that were brought about by the automation, the automation also served the purpose of formalising the semantics of SML.

The final details of our semantics were tested by analysing relatively well-understood subsystems of the control software in mCRL2. We briefly discuss our findings using a partly simplified subsystem, colloquially known as the *Wheel*, see Figure 3. The Wheel subsystem is a component of the Resistive Plate Chamber (RPC) subdetector of the CMS experiment. It belongs to the barrel region of the RPC subdetector. Each Wheel subsystem contains 12 sectors, each sector is equipped with 4 muon stations which are made of Drift Tube chambers. We forego a detailed formal discussion of this subsystem (for details, we refer to [11]), but only address our analysis of this subsystem using formal analyses techniques, and the impact this had on our understanding of the semantics and the transformation. It is important to keep in mind that the analysis was conducted primarily to assess the quality of our translation, the correctness of the subsystem being only secondary.

The mCRL2 specification of the *Wheel* subsystem was obtained by combining the mCRL2 processes obtained by running our prototype implementation on each involved FSM. Generating the state space of the *Wheel* subsystem takes roughly one minute using the symbolic state space generation tools offered by the LTSmin tools [4]. This toolset can be integrated in the mCRL2 toolset. For the discussed configuration, the state space is still of modest proportions, measuring slightly less than 5 million states and 24 million transitions. Varying the amount of children of class *Sector* causes a dramatic growth of the state space. Using 3 instead of 2 children of class *Sector* yields
roughly 800 million states; using 4 children of class Sector, leads to 120 billion states, and requires half a day.

Apart from repeating the simulations and visualisations, at this stage we also applied model checking to systematically probe the translation. Together with the development team of the Wheel subsystem, a few basic requirements were formalised in the first-order modal $\mu$-calculus [8], see Table 1. The first-order modal $\mu$-calculus is the default requirement specification language in the mCRL2 toolset.

The studied subsystem was considered to satisfy all stated properties. While smoothing out details in the translation of SML to mCRL2, the deadlock-freedom property was violated every now and then, indicating issues with our interpretation of SML. These were mostly concerned with the semantics of the blocking and non-blocking constructs of SML, and the complex constructs used to model the message passing between FSMs and their children.

The absence of intermediate states in the when phase was violated only once in our verification efforts. A more detailed scrutiny of the run revealed a problem in our translation, which was subsequently fixed.

The third requirement, stating the inevitability of a state change by a child once such a state change has been commissioned, failed to hold. The violation is caused by the overriding of commands by subsequent commands that are issued immediately. Discussions with the development teams revealed that the violations are real, i.e., they are within the range of real behaviour, suggesting that our formalisation was adequate. The property was modified to ignore the spurious runs, resulting in the following property:

\[
\text{nu } X. \ [\text{true}]X \land \land \text{[comm}_\text{command}(i,i,c,c)](\text{mu } Y. \ <\text{true}>true \land \land \text{[comm}_\text{command}(i,i,c,c'))(\text{mu } Y. \\
\text{[!}(\text{comm}_\text{state}(i,c,i,c2s(c))) \land \land \text{exists } c':\text{Command. comm}_\text{command}(i,i',c')])Y)
\]

The final requirement also failed to hold. The violation is similar spirited to the violation of the third requirement, and, again found to comply to reality. The weakened
Table 1. Basic requirements for the Wheel subsystem; \textit{i:Id} denotes an identifier of an FSM; \textit{i_c:Id} denotes a child of FSM \textit{i}; \textit{c:Command} denotes a command; \textit{c2s(c)} denotes the state with the homonymous command name, \textit{e.g.}, \textit{c2s(ON)} = \textit{ON}.

1. Absence of deadlock:
   \[
   \text{nu } X. \ [\text{true}]X \ && \ <\text{true}>\text{true}
   \]

2. Absence of intermediate states in the \textit{when phase}:
   \[
   \text{nu } X. \ [\text{true}]X \ &\& \\
   [\exists s:\text{State}. \text{move_state}(i,s)](\text{nu } Y. \\
   [(!\text{move_phase}(i,\text{ActionPhase}))Y \\
   & \& [\exists s:\text{State}. \text{move_state}(i,s)]\text{false})
   \]

3. Responsiveness:
   \[
   \text{nu } X. \ [\text{true}]X \ &\& \\
   [\text{comm}\_\text{command}(i, i_c, c)](\mu Y. \\
   <\text{true}>\text{true} \ &\& [!\text{comm}\_\text{state}(i_c, i, c2s(c))]Y)
   \]

4. Progress:
   \[
   \text{nu } X. \ [\text{true}]X \ &\& \\
   \mu Y. <\exists s:\text{State}. \text{move_state}(i,s)>\text{true} \\
   || \\
   (<\text{true}>\text{true} \ &\& [\text{true}]Y)
   \]

requirement that was subsequently agreed upon expresses the attainability of some state change:

\[
\text{nu } X. \ [\text{true}]X \ &\& \\
\mu Y. <\exists s:\text{State}. \text{move_state}(i,s)>\text{true} \ || \ <\text{true}>Y
\]

Neither visual inspection of the state space using 2D and 3D visualisation tools, nor simulation using the mCRL2 simulators revealed any further incongruences in our final formalisation of SML, sketched in the previous section.

4 Dedicated Tooling for Verification

Some desired properties, such as the absence of loops within the \textit{when phase}, can be checked by analysing an FSM in isolation, using the transformation to mCRL2. However, the verifications using the modal \(\mu\)-calculus currently require too much overhead to serve as a basis for lightweight tooling that can be integrated in the SML development environment.

In an attempt to improve on this situation, we explored the possibilities of using \textit{Bounded Model Checking (BMC)} \cite{3, 2}. The basic idea of BMC is to check for a counterexample in bounded runs. If no bugs are found using the current bound, then the bound is increased until either a bug is found, the problem becomes intractable, or some pre-determined upper bound is reached upon which the verification is complete.
The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods. SAT procedures do not necessarily suffer from the space explosion problem, and a modern SAT solver can handle formulas with hundreds of thousands of variables or more, see e.g. [2].

We have applied BMC techniques for the detection of move_to loops and the detection of unreachable states and trap states. As an example of a move_to loop, consider the excerpt of the ECALfw_CoolingDee FSM class in Listing 3, which our tool found to contain issues. If an instance of ECALfw_CoolingDee has one child in state ERROR and one in state NO_CONNECTION, it will loop indefinitely between these two states. Once this happens, an entire subsystem may enter a livelock and become unresponsive.

```
state: ERROR
  when ( $ANY$FwCHILDREN in_state NO_CONNECTION ) move_to NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK

state: NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK
  when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
```

Listing 3: An excerpt from the ECALfw_CoolingDee FSM that exhibits a loop within the when phase.

We first convert this problem into a graph problem as follows. Let $F$ be an FSM and $M$ be a Kripke structure. A state in $M$ corresponds to the combined state of $F$ and its children, e.g., if $F$ is in state $ON$ and has two children which are in state $OFF$, then the corresponding state in $M$ is $(ON, OFF, OFF)$. There is a transition between two states $s_1$ and $s_2$ in $M$ if and only if $s_1$ can do a move_to action to $s_2$ in $F$. Moreover, every state in $M$ is an initial state. It thus suffices to inspect $M$ instead of $F$, as stated by the following lemma:

**Lemma 1.** $F$ contains a loop of move_to actions if and only if $M$ contains a loop.

We next translate the problem of detecting a loop in $M$ into a SAT problem. First, we consider executions of length $k$; afterwards, we show that we can statically choose $k$ such that we can find every loop.

Let the predicate $\text{in\_state}$ be defined as follows: $\text{in\_state}(s, p, i)$ holds if and only if the process with identifier $p$ is in state $s$ after $i$ steps. We assign the identifier zero to the FSM under consideration and the numbers $1, 2, 3, \ldots$ to its children. The resulting formula will have three components: the state constraints, the transition relation and the loop condition.

Using the state constraints, we ensure the FSM to always be in exactly one state. Moreover, the states of the children should not change during the execution of the when phase, per the semantics in the previous section. This is straightforwardly expressed as a boolean formula on the $\text{in\_state}$ predicate.

Next, we encode the transition relation: the relation between $\text{in\_state}(s, 0, i)$ and $\text{in\_state}(s', 0, i + 1)$ for every $i$. In other words, the move_to steps the parent process is
allowed to take. This involves converting the when clauses for each state of the parent FSM, taking care the semantics as outlined in the previous section is reflected. The last ingredient is the loop condition: if \textit{in\_state}(s, 0, 0) holds, then \textit{in\_state}(s, 0, i) must hold for some \(i > 1\), indicating that the parent returned to the state in which it started.

The final SAT formula is obtained by taking the conjunction of the state constraints, the transition relation and the loop condition. It is not hard to see that if this formula is satisfiable, then there is a loop in \(M\) and hence in \(F\). It is more difficult to show that if there is a loop, then the formula is satisfiable. Let \(n\) be the total number of states of the FSM and let \(n_t\) be the total number of states of each child class \(t\). We then have the following result:

\textbf{Theorem 1.} All possible loops in \(F\) can be found by considering paths of length at most \(n\) in an FSM configuration \(F\) having \(n_t\) children for each child class \(t\).

\textit{Proof (sketch).} Since \(F\) only has \(n\) states, the longest possible loop also contains \(n\) states. Since every state in \(M\) is an initial state, every possible loop can by found by doing \(n\) steps from an initial state.

It remains to show that all loops can be found by considering a configuration with \(n_t\) children for each child class \(t\). This follows from the fact that SML guards are restricted to check for any or all children in a particular state. \(\Box\)

A second desirable behavioural property of an FSM is that all states should remain reachable during the execution of an FSM. While we can again easily encode this property into the modal \(\mu\)-calculus, we use a more direct approach to detect violations of this property by constructing a graph that captures all potential state changes. For this, we determine whether there is a configuration of children such that \(F\) can execute a \textit{move\_to} action from a state \(s\) to a state \(s'\). Doing so for all pairs \((s, s')\) of states of \(F\) yields a graph encoding all possible state changes of \(F\).

Computing the strongly connected components (SCCs) of the thusly obtained graph gives sufficient information to pinpoint violations to the reachability property: the presence of more than a single SCC means that one cannot move back and forth these SCCs (by definition of an SCC), and, therefore, their states. Note that this is an underapproximation of all errors that can potentially exist, as the actual reachability dynamically depends on the configuration of the children of an FSM. Still, as the state change graph of the \textit{Endcap} FSM class in Figure 4 illustrates, issues can be found in production FSMs: the \texttt{OFF} state can never be reached from any of the other states. Using the graphs generated by our tools, such issues are quickly explained and located.

\textbf{Results} The results using our dedicated tools for performing these behavioural sanity checks on isolated FSMs are very satisfactory: of the several hundreds of FSM classes contained in the control system, we so far analysed 40 FSM classes and found 6 to contain issues. In 4 of these, we found logical errors that could give rise to livelocks in the system due to the presence of loops in the when phase; an example thereof is given in Listing 3. Somewhat unexpectedly, all loops were found to involve two states. Note that the size of the average FSM class (in general more than 100 lines of SML code, and at least two children) means that even short loops such as the ones identified so far remain unnoticed and are hard to pinpoint. The remaining two FSM classes were
found to violate the required reachability of states, see e.g. Figure 4. The speed at which the errors can be found (generally requiring less than a second) means that the sanity checks could easily be incorporated in the design cycle of the FSMs.

5 Conclusion

We discussed and studied the State Machine Language (SML) that is currently used for programming the control software of the CMS experiment running at the Large Hadron Collider. To fully understand the language, we formalised it using the process algebraic language mCRL2. The quality of our formalisation was assessed using a combination of simulation and visualisation of the behaviour of FSMs in isolation and formally verifying small subsystems using model checking. To facilitate, among others, the assessment, the translation of SML to mCRL2 was implemented using the ASF+SDF meta-environment. Based on our understanding of the semantics of SML, we have built dedicated tools for performing sanity checks on isolated FSMs. Using these tools we found several issues in the control system. These tools have been well-received by the engineers at CERN, and are considered for inclusion in the development environment.

Our formalisation of SML opens up the possibility of verifying realistically large subsystems of the control system; clearly, it will be one of the most challenging verification problems currently available. In our analysis of the Wheel subsystem, we have only used a modest set of tools for manipulating the state space; symmetry reduction, partial order reduction, parallel exploration techniques, abstractions and abstract interpretation were not considered at this point. It remains to be investigated how such techniques fare on this problem.

Acknowledgments. We thank Giel Oerlemans, Dennis Schunselaar and Frank Staals from the Eindhoven University of Technology for their contribution to a first draft of the
ASF+SDF translation. We also thank Frank Glege and Robert Gomez-Reino Garrido from the CERN CMS DAQ group for their support and advice, and Clara Gaspar for discussions on SML. Jaco van de Pol is thanked for his help with the LTSmin toolset.

References

1. J.C.M. Baeten, T. Basten, and M.A. Reniers. Process Algebra: Equational Theories of Communicating Processes, volume 50 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2010.
2. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. Advances in Computers, 58:118–149, 2003.
3. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Proc. of TACAS, volume 1579 of LNCS, pages 193–207. Springer, 1999.
4. Stefan Blom, Jaco van de Pol, and Michael Weber 0002. LTSmin: Distributed and symbolic reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, CAV, volume 6174 of LNCS, pages 354–359. Springer, 2010.
5. B. Franek and C. Gaspar. SMI++ object-oriented framework for designing and implementing distributed control systems. IEEE Transactions on Nuclear Science, 52(4):891–895, 2005.
6. C. Gaspar and B. Franek. SMI++—Object-oriented framework for designing control systems for HEP experiments. Computer physics communications, 110(1–3):87–90, 1998.
7. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. v. Weerdenburg. Analysis of distributed systems with mCRL2. In Process Algebra for Parallel and Distributed Processing, pages 99–128. Chapman Hall, 2009.
8. J.F. Groote and T.A.C. Willemse. Model-checking processes with data. Science of Computer Programming, 56(3):251–273, 2005.
9. O. Holme, M. González-Berges, P. Golonka, and S. Schmeling. The JCOP Framework. Technical Report CERN-OPEN-2005-027, CERN, Geneva, Sep 2005.
10. P. Klint. A meta-environment for generating programming environments. ACM Trans. Softw. Eng. Methodol., 2(2):176–201, 1993.
11. P. Paolucci and G. Polese. The detector control systems for the cms resistive plate chamber, 2008. CERN-CMS-NOTE-2008-036, see http://cdsweb.cern.ch/record/1167905.
12. M. Van den Brand, A. Van Deursen, J. Heering, H.A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, Proc. of Compiler Construction, volume 2027 of LNCS, pages 365–370. Springer, 2001.
A ASF and SDF files

A.1 midtools.sdf

```plaintext
1 module midtools
5 imports
 basic/Whitespace
 basic/Comments
 basic/Booleans
 basic/Integers
10 exports
 sorts
 MId
 MIds
15 lexical restrictions
 MId : [a-zA-Z0-9_] -> MId
 lexical syntax
20 context-free syntax
25Concatenate two MIds.
 concat(MId, MId) -> MId
25 Return whether an MId is in a list of MIds.
 contains(MId, MId*) -> Boolean
30 Given a list, remove all duplicates. If the list contains two identical elements,
 removeDuplicates(MId*) -> MId*
35 Remove all occurrences of an element from a list.
 remove(MId, MId*) -> MId*
40 Compute the set intersection of two lists. The resulting list has no duplicates.
 intersect(MId*, MId*) -> MId*
45 Returns true iff the list is empty.
 empty(MId*) -> Boolean
50 Returns the length of a list of MIds.
 length(MId*) -> Integer
0 hiddens
50 variables
 "$mid"[0-9]* -> MId
 "$mid+"[0-9]* -> MId+
 "$mid*"[0-9]* -> MId*
 "$i" -> Integer
55 lexical variables
 "#midHead"[0-9]* -> [a-zA-Z_]
 "#midTailChar"[0-9]* -> ([a-zA-Z0-9_] )
 "#midTail"[0-9]* -> ([a-zA-Z0-9_] )
```

A.2 midtools.asf

```plaintext
1 equations
5 [concat-1]
 concat(mid(#midHead1 #midTail1), mid(#midHead2 #midTail2)) = mid(#midHead1 #midTail1 #midHead2 #midTail2)
10 [contains-empty]
 contains($mid, ) = false
15 [contains-match]
 contains($mid, $mid $mid*) = true
20 [contains-nomatch]
 $mid1 != $mid2 => contains($mid1, $mid2 $mid*) = contains($mid1, $mid2)
```
\[\text{removeDuplicates}() = 20\]
\[\text{contains}(\text{mid}, \text{mid}^*) = \text{true} \implies \text{removeDuplicates}(\text{mid}, \text{mid}^*) = \text{removeDuplicates}(\text{mid}^*)\]
\[\text{removeDuplicates}(\text{mid}, \text{mid}^*) = \text{mid} \text{ removeDuplicates}(\text{mid}^*)\]
\[\text{remove}() = \text{true}\]
\[\text{contains}(\text{mid}, \text{mid}^*) = \text{false} \implies \text{remove}(\text{mid}, \text{mid}^*) = \text{remove}(\text{mid}, \text{mid}^*)\]
\[\text{remove}(\text{mid1}, \text{mid2}, \text{mid}^*) = \text{remove}(\text{mid2}, \text{mid}^*)\]
\[\text{intersects}(\text{mid}, \text{mid}^*, \text{mid}^*) = \text{false} \implies \text{intersect}(\text{mid}, \text{mid}^*, \text{mid}^*) = \text{intersect}(\text{mid}^*, \text{mid}^*)\]
\[\text{empty}() = \text{true}\]
\[\text{empty}(\text{mid}^*) = \text{false}\]
\[\text{length}(\text{mid}^*) = 0\]
\[\text{length}(\text{mid}^*) = 1 + 1\]

A.3 cfsm.sdf

%% Authors:
%% Vincent Kusters
%% Dennis Schumelaar

module cfsm
imports
  basic/Comments
  basic/Whitespace
  midtools
exports
  context-free start-symbols
  FSMSpecification
sorts
  Identifier
  FSMSpecification
  FSMClass
  FSMStateClause
  FSMWhenClause
  FSMReferer
  FSMActionClause
  FSMStatement
  FSMParameter
  FSMExpression
  FSMChildrenSpec
  FSMChildrenAny

lexical syntax

\[ 
'|' \text{ or} '-' \text{ or} '[a-zA-Z0-9]+ \text{ or} \text{ LAYOUT} \\
'\text{ASSOCIATED}' \text{ or} '[a-zA-Z0-9]+ \text{ or} \text{ LAYOUT} \\
(\text{a-zA-DU-x}) \rightarrow \text{ Identifier} 
\]

context-free syntax

\[ 
\text{FSMStateNameSpec} \rightarrow \text{Identifier} 
\]

%% Author: Vincent Kusters.

module mcrlt

imports


A.4 mcrlt.sdf
lexical restrictions
MId -~/ a-zA-Z0-9_\-\砾\\"\\']

context-free start-symbols

lexical syntax
We disallow comments starting with "%%", since this leads to ambiguity with ASF+SDF comments.

context-free syntax
Keywords (B1)
"sort" -> MId (reject)
"map" -> MId (reject)
"var" -> MId (reject)
"eqn" -> MId (reject)
"act" -> MId (reject)
"proc" -> MId (reject)
"init" -> MId (reject)
"delta" -> MId (reject)
"tau" -> MId (reject)
"sum" -> MId (reject)
"block" -> MId (reject)
%% Sort expressions and sort declarations (B4)
'Bool' | 'Pos' | 'Nat' | 'Int' | 'Real' -> SortExpr
'List' ('SortExpr ') -> SortExpr
'Set' ('SortExpr ') -> SortExpr
'MId : SortExpr ' -> SortSpec
Domain -> SortExpr
{SortExpr #}+ -> Domain
'sort' SortDecl+ -> SortSpec
MIds ';' -> SortDecl
MIds '=' SortExpr ';' -> SortDecl
MIds '=' 'struct' {ConstrDecl |'}+ ';' -> SortDecl

%% Difference with the specification in the reader:
%% the "MId part is obligatory.
MId ('(' ProjDecls ')')? ('?' MId) -> ConstrDecl
(MId ':') Domain -> ProjDecl
(ProjDecl ',')+ -> ProjDecl

%% Declarations of constructors and mappings (B5)
MId ':' SortExpr -> IdDecl
MIds ':' SortExpr ';'? -> IdsDecl
('cons' | 'map') OpDecl+ -> OpSpec
IdsDecl ';' -> OpDecl

%% Declaration of equations (B6)
'eqn' EqnDecl+ -> EqnSpec
'var' IdsDecl+ 'eqn' EqnDecl+ -> EqnSpec
DataExpr '=' DataExpr ';' -> EqnDecl
DataExpr '+' DataExpr '=' DataExpr ';' -> EqnDecl

%% Data expressions (B7)
MId | Integer | 'true' | 'false' | '()' | '{'} -> DataExpr
'[' DataExprs ']' -> DataExpr
'{' DataExprs '}' -> DataExpr
'{' BagEnumElts '}' -> DataExpr
'{' IdDecl '|' DataExpr '}' -> DataExpr
'(' DataExpr ')' -> DataExpr
{'!' | '#'} DataExpr -> DataExpr
('forall' | 'exists') IdDecl '.' DataExpr -> DataExpr
DataExpr 'whr' DataExprs 'end' -> DataExpr
{DataExpr ','}+ -> DataExprs
DataExpr ':' DataExpr -> BagEnumElt
{BagEnumElt ','}+ -> BagEnumElts

%% Communication and renaming (B8)
'MId '(' ')')? -> MId
('(' MId ','+ ')' -> MIdSet
'MId ('->' MId')? -> ComExpr
(')' (ComExpr ','')+ ')' -> ComExprSet
MId ->' MId -> RedExpr
(')' (RedExpr ',')+ ')' -> RedExprSet
%% Process expressions (B9)
Mid -> ProcExpr
Mid "(" DataExprs ")" -> ProcExpr
"delta" -> ProcExpr
"tau" -> ProcExpr

%% Sum was moved to the context-free priorities section.
("block" | "allow" | "hide") "(" MAIdSet "," ProcExpr ")" -> ProcExpr
"rename" "(" RenExprSet "," ProcExpr ")" -> ProcExpr
"comm" "(" CommExprSet "," ProcExpr ")" -> ProcExpr

"/" ProcExpr "/" -> ProcExpr

ProcExpr with binary operators was moved to the context-free priorities section.

%% Action declaration (B10)
MIds ("":" Domain") ";" -> ActDecl
"act" ActDecl+ -> ActSpec

%% Process and initial state declaration (B11)
Mid "=" ProcExpr ";" -> ProcDecl
Mid "(" (DefDecl "," )")" "=" ProcExpr ";" -> ProcDecl
"proc" ProcDecl+ -> ProcSpec
"init" ProcExpr ";" -> Init

%% Syntax of an mCRL2 specification (B12)
SortDecl* OpSpec* EqnSpec* ActSpec* ProcSpec* Init* -> MCRL2Spec

\section{Module for converting the CERN Finite State Machines into mCRL2 code.}

\module cfsm2mcrl2
\imports cfsm
\imports script
\imports genericclauses
\imports midtools
\imports basic\Integers
\imports basic\BoolCon
\exports context-free start-symbols
\exports context-free syntax

\function cfsm2mcrl2(FSMClass+)
cfsm2mcrl2(FSMClass+) -> ProcSpec+

\function cfsm2mcrl2bm(FSMClass+)
cfsm2mcrl2bm(FSMClass+) -> ProcSpec+

\function generateSorts(FSMClass+)
generateSorts(FSMClass+) -> ProcSpec

\section{A.5 cfsm2mcrl2.sdf}

module cfsm2mcrl2
imports cfsm
imports script
imports genericclauses
imports midtools
imports basic\Integers
imports basic\BoolCon
exports context-free start-symbols
exports context-free syntax

function cfsm2mcrl2(FSMClass+)
function cfsm2mcrl2bm(FSMClass+)
function generateSorts(FSMClass+)

22
Function to generate the list of process names from a list of process specifications.

\[
\text{fsmGenerateSorts(FSMClass)} \rightarrow \text{SortDecl+}
\]

Function to convert a number of FSM classes into process definitions.

\[
\text{fsmClasses2Mcrl2Procs(FSMClass+, BoolCon)} \rightarrow \text{ProcSpec+}
\]

Conversion functions for a list of states and a single state in the FSM.

\[
\text{convertStates(FSMStateClause*, ProcName, BoolCon, MId*)} \rightarrow \text{ProcExpr}
\]

Conversion functions for the parts in each state. These functions are grouped by phase. First the functions needed in the when-phase. Functions that are required in both phases are listed in with the when-phase.

The conversion of the when-clauses requires a third parameter: the name of the state we are currently converting.

\[
\text{convertWhenClauses(FSMWhenClause*, ProcName, MId, ActionClauseTuple*)} \rightarrow \text{ProcExpr}
\]

Conversion functions for the action clauses. The conversion of the when-clauses requires a third parameter: the name of the state we are currently converting.

\[
\text{convertReferer(FSMReferer, ProcName, MId, ActionClauseTuple*)} \rightarrow \text{ProcExpr}
\]

Conversion functions for the statements inside action clauses.

\[
\text{convertStatements(FSMStatement*, ProcName, PC, PC, PC)} \rightarrow \text{ProcExpr}
\]

Conversion functions for the generation of bottom monitors.

\[
\text{isAnyState(MId*)} \rightarrow \text{DataExpr}
\]

\[
\text{createObedientCommandAcceptor(FSMActionClause*, ProcName, MId*)} \rightarrow \text{ProcExpr}
\]

For the when clauses we need to add a clause describing that we are in a certain state.

\[
\text{isStateCheck(MId)} \rightarrow \text{DataExpr}
\]

\[
\text{isCommandCheck(MId)} \rightarrow \text{DataExpr}
\]

\[
\text{isCommandCheck(MId, MId)} \rightarrow \text{DataExpr}
\]

Convert a name of a state into an StateName as we will use in Mcrl2 (OFF => S_OFF).

\[
\text{toMcrlStateName(MId)} \rightarrow \text{MId}
\]

Helpers for the translation of if statements.

\[
\text{insertIfBlockingWaiter(ProcName, PC)} \rightarrow \text{ProcExpr}
\]

Convert a name of an action into an CommandName as we will use in Mcrl2 (OFF => C_OFF).

\[
\text{toMcrlCmdName(MId)} \rightarrow \text{MId}
\]

Helpers for the generation of bottom monitors.

\[
\text{isAnyState(MId)} \rightarrow \text{DataExpr}
\]

\[
\text{createObedientCommandAcceptor(FSMActionClause*, ProcName, MId*)} \rightarrow \text{ProcExpr}
\]

Convert a name of a state into an StateName as we will use in Mcrl2 (OFF => S_OFF).

\[
\text{toMcrlStateName(MId)} \rightarrow \text{MId}
\]

Convert a name of an action into an CommandName as we will use in Mcrl2 (OFF => C_OFF).

\[
\text{toMcrlCmdName(MId)} \rightarrow \text{MId}
\]
These functions are required to generate the sort declarations from the FSM classes.

First functions to create a declaration that is used in a struct from the name of the class/state/action. So from some class myClass it generates:

\[
\text{Class } \text{myClass}'.
\]

Similarly for states and actions:

\[
\text{convertClassNamesToTypeConstrDecl(MId*) -> ConstrDecl | |
\]

The following functions are traversal functions that simply gather all definitions:

\[
\text{collectClasses(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMClass,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAnySpecific,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAllSpecific,MId*) -> MId* |
\]

\[
\text{collectStates(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause+, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause*, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateNameSpec, MId*) -> MId* |
\]

\[
\text{collectCommands(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectCommands(FSMActionClause,MId*) -> MId* |
\]

Function to add something to a set such that we do not introduce duplicates:

\[
\text{addToSet(MId,MId*) -> MId* |
\]

These functions are required to generate the sort declarations from the FSM classes.

First functions to create a declaration that is used in a struct from the name of the class/state/action. So from some class myClass it generates:

\[
\text{Class } \text{myClass}'.
\]

Similarly for states and actions:

\[
\text{convertClassNamesToTypeConstrDecl(MId*) -> ConstrDecl | |
\]

The following functions are traversal functions that simply gather all definitions:

\[
\text{collectClasses(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMClass,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAnySpecific,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAllSpecific,MId*) -> MId* |
\]

\[
\text{collectStates(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause+, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause*, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateNameSpec, MId*) -> MId* |
\]

\[
\text{collectCommands(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectCommands(FSMActionClause,MId*) -> MId* |
\]

Function to add something to a set such that we do not introduce duplicates:

\[
\text{addToSet(MId,MId*) -> MId* |
\]

These functions are required to generate the sort declarations from the FSM classes.

First functions to create a declaration that is used in a struct from the name of the class/state/action. So from some class myClass it generates:

\[
\text{Class } \text{myClass}'.
\]

Similarly for states and actions:

\[
\text{convertClassNamesToTypeConstrDecl(MId*) -> ConstrDecl | |
\]

The following functions are traversal functions that simply gather all definitions:

\[
\text{collectClasses(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMClass,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAnySpecific,MId*) -> MId* |
\]

\[
\text{collectClasses(FSMChildrenAllSpecific,MId*) -> MId* |
\]

\[
\text{collectStates(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause+, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause*, MId*) -> MId* |
\]

\[
\text{collectStates(FSMWhenClause, MId*) -> MId* |
\]

\[
\text{collectStates(FSMStateNameSpec, MId*) -> MId* |
\]

\[
\text{collectCommands(FSMSpecification,MId*) -> MId* |
\]

\[
\text{collectCommands(FSMActionClause,MId*) -> MId* |
\]

Function to add something to a set such that we do not introduce duplicates:

\[
\text{addToSet(MId,MId*) -> MId* |
\]
A.6  cfsm2mcrl2.asf

equations

convertor function to convert a FSM class to an MCRL2 specification, with the added property that the result will be a bottom monitor. That is, it has no children and whenever it would normally check the state of children, it will instead check randomStateChanges.

convert a number of FSM classes into processes.

In our main function we define the process instance and process declaration.

Function for converting a list of states. Each state translation translates to a Process Expression, so this means a list should be translated using the alternative (+) operator


%% Function to convert a single state.
(convertState-nobm)
\langle \text{ActionClauseTuple*}, \text{procExpr} \rangle \rightarrow \text{gatherComponents\text{fromActionClauses}(\text{fromActionClause*}, \text{procName}, \text{fromCurrentState}, 1)}

\text{convertState}(\text{state: fromCurrentState fromWhenClause* fromActionClause*}, \text{procName}, \text{false}, \text{null}) =

\{
  \text{// BEGIN STATE}
  \text{convertWhenClauses(fromWhenClause*}, \text{procName}, \text{fromCurrentState}, \text{actionClauseTuple*}) +
  \text{// BEGIN ACTION CLAUSES}
  \text{These are the rules:}
  \text{pc(aArgs) == 0} \Rightarrow \text{no command received yet}
  \text{pc(aArgs) > 0} \Rightarrow \text{command received, executing action clause}
  \text{pc(aArgs) == -1} \Rightarrow \text{action clause executed, but still must send commands}
  \text{Since the FSM language allows for an arbitrary amount of statements and}
  \text{an arbitrary amount of nested if-statements, we cannot simply do a}
  \text{sequential translation. It is for this reason that we use a label to}
  \text{identify the translation of every statement. After executing a}
  \text{statement, a program counter is set to the label of the statement which}
  \text{should be executed next. There are two special cases here:}
  \text{Label 0, the clause selector. In the action phase, we always first}
  \text{have pc == 0. When we receive a command, the clause selector}
  \text{determines the label of the first statement of the action clause}
  \text{that should handle the command. The program count is then set to}
  \text{this label.}
  \text{Label -1, end of action. After executing an action, the program}
  \text{counter is set to -1 to signify that we should now empty the}
  \text{sendqueue and move to the when phase.}

\text{Examples can be found in the translation function of the if-statement.}

\text{// BEGIN INITIALIZATION CHECK}
\text{((isStateCheck(fromCurrentState)) \&\& ((isActPhase(phase)) \&\& ((initialized(chs)))) \&\& (pc(aArgs) == 0) \&\& (cq(aArgs) == \[])) \rightarrow start_initialization(self)};
\text{procName(self, parent, s, chs, phase,}
\text{actArgs(\[]), children_to_ids(chs), 0, rsc(aArgs))} \rightarrow

\text{// END INITIALIZATION CHECK}

\text{// BEGIN CLAUSE SELECTOR}
\text{((initialized(chs)) ->
  ((isStateCheck(fromCurrentState)) \&\& (isActPhase(phase)) \&\& (cq(aArgs) == \[])) \&\& (pc(aArgs) == 0) ->
  sum c:Command.(
  rc(parent, self, c).
  constructClauseSelectors(fromActionClauseTuple*, \text{procName})
  )
  )
  )

\text{// END CLAUSE SELECTOR}

\text{combineActionClauseComponents(fromActionClauseTuple*, \text{procName}, \text{fromCurrentState})}
\}

\text{// END ACTION CLAUSES}

\text{// BEGIN CLASSES}
\text{\}

\text{// END CLASSES}

\text{// BEGIN STATE}
\text{\}

\text{// END STATE}

%% Function to convert a single state (bottom monitor variant).
(convertState-bm)
\langle \text{ActionClauseTuple*}, \text{procExpr} \rangle \rightarrow \text{inAnyState(collectStates(fromWhenClause*, ))}

\text{convertState}(\text{state: fromCurrentState fromWhenClause* fromActionClause*}, \text{procName}, \text{true}, \text{null}) =

\{
  \text{// BEGIN STATE}
  \text{convertWhenClauses(fromWhenClause*}, \text{procName}, \text{fromCurrentState, ActionClauseTuple*}) +

% BEGIN ACTION CLAUSES
(isStateCheck($fsmCurrentState)) ->
sum c: Command.
140
rc(parent, self, c).
141
$procExpr
142
% END ACTION CLAUSES

% END STATE

%%%%%%%%%%%%%%%%%% When Clauses Stuff

(convertWhenClauses-empty)
$mcrl2CurrentState := toMcrlStateName($fsmCurrentState)
150
===>
convertWhenClauses(, $procName, $fsmCurrentState, $actionClauseTuple*) =
155
156
% BEGIN WHEN FALLTHROUGH
(((isStateCheck($fsmCurrentState)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
160
move_phase(self, ActionPhase).
$procName(self, parent, s, chs, ActionPhase, reset(aArgs))
% END WHEN FALLTHROUGH

[convertReferer-moveto]
$mcrl2NewState := toMcrlStateName($fsmNewState)
170
===>
convertReferer(move_to $fsmNewState, $procName, $mcrl2CurrentState, $actionClauseTuple*) =
175
176
% BEGIN WHEN FALLTHROUGH
((isStateCheck($fsmCurrentState)) && (isWhenPhase(phase)) ->
ss(self, parent, s).
180
move_phase(self, ActionPhase).
$procName(self, parent, s, chs, ActionPhase, reset(aArgs))
% END WHEN FALLTHROUGH

[convertReferer-do]
<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := getActionClauseTupleForActionName($actionClauseTuple*, $fsmActionName)
185
===>
convertReferer(do $fsmActionName, $procName, $mcrl2CurrentState, $actionClauseTuple*) =
190
move_phase(self, ActionPhase).
$mcrlActionEffect

%% Note: the empty list is not allowed here since there must always be an corresponding action. If not, the FSM is inconsistent.

(getActionClauseTupleForActionName-many-match)
<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := $actionClauseTuple
195
===>
g.getActionClauseTupleForActionName($actionClauseTuple $actionClauseTuple*, $fsmActionName1) =
g.getActionClauseTupleForActionName($actionClauseTuple*, $fsmActionName1)

%% When we have multiple elements in our list of when clauses we translate into
%% a form of 'c -> a.X <> b' in which 'b' is the translation of the remaining
%% when clauses

(gatherComponentsFromActionClauses-many)
$mcrl2CurrentState := toMcrlStateName($fsmCurrentState)
200
===>
gatherComponentsFromActionClauses(, $procName, $fsmCurrentState, $pc) =
gatheringComponentsFromActionClauses\(\text{many}\)

\[
\begin{align*}
\text{start}_\text{pc} & := \text{avail}_\text{pc}, \\
\text{avail}_\text{pc} & := \text{avail}_\text{pc} + 1, \\
\text{mcrl}_\text{Command} & := \text{toMcrlCmdName}(\text{fsmActionName}), \\
\langle \text{procExpr}, \text{avail}_\text{pc} \rangle & := \text{convertStatement}(\text{fsmStatement*}, \text{procName}, \text{start}_\text{pc}, -1, \text{avail}_\text{pc} + 1), \\
\langle \text{sectionClauseTuple}, \text{avail}_\text{pc} \rangle & := \text{gatherComponentsFromActionClauses}(\text{fsmActionClause*}, \text{procName}, \text{fsmCurrentState}, \text{avail}_\text{pc} + 3)
\end{align*}
\]

\[
\begin{align*}
\text{gatherComponentsFromActionClauses}(\text{action: fsmActionName fsmStatement* fsmActionClause*}, \text{procName}, \text{fsmCurrentState}, \text{avail}_\text{pc}) & = \\
\langle \langle \text{fsmActionName}, \\
\text{start}_\text{pc}, \\
\langle \langle \text{isStateCheck}(\text{fsmCurrentState}) \&\& \langle \text{isActPhase}(\text{phase}) \&\& (\text{cq}(\text{aArgs}) == [])\rangle\rangle, \\
\langle \langle \text{procExpr} \rangle, \\
\text{sectionClauseTuple}, \text{procName}, \text{fsmCurrentState}, \text{avail}_\text{pc} \rangle \rangle\rangle
\end{align*}
\]

\[
\begin{align*}
\text{gatherComponentsFromActionClauses}(\text{action: fsmActionName fsmStatement* fsmActionClause*}, \text{procName}, \text{fsmCurrentState}, \text{avail}_\text{pc}) & = \\
\langle \langle \text{fsmActionName}, \\
\text{start}_\text{pc}, \\
\langle \langle \text{isStateCheck}(\text{fsmCurrentState}) \&\& \langle \text{isActPhase}(\text{phase}) \&\& (\text{cq}(\text{aArgs}) == [])\rangle\rangle, \\
\langle \langle \text{procExpr} \rangle, \\
\text{sectionClauseTuple}, \text{procName}, \text{fsmCurrentState}, \text{avail}_\text{pc} \rangle \rangle\rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{empty}\)} &= \delta \\
\text{combineActionClauseComponents\(\text{many}\)} &= (\text{fsmActionName}, \text{start}_\text{pc}, \text{mcrlActionCondition}, \text{mcrlActionEffect}) \Rightarrow \text{sectionClauseTuple} \\
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{combineActionClauseComponents\(\text{many}\)} &= (\langle \text{sectionClauseTuple} \rangle, \text{procName}, \text{fsmCurrentState} = \{ \\
\text{beginAction} \Rightarrow \text{sectionClauseTuple} \rangle, \\
\text{endAction} \Rightarrow \text{sectionClauseTuple} \rangle
\end{align*}
\]
% Some FSMs in the bottommost layer might have actions which do not contain any statements.
convertStatements($fsmStatement, $procName, $start_pc, $jump_pc, $avail_pc) =

% BEGIN STATEMENT NOP
  (pc(aArgs) == $start_pc) ->
    noop_statement(self).
% END STATEMENT NOP

% The final statement in a block should jump to the next block (indicated by $jump_pc).
convertStatements($fsmStatement, $procName, $start_pc, $jump_pc, $avail_pc) =
convertStatements($fsmStatement, $procName, $start_pc, $jump_pc, $avail_pc)

% BEGIN STATEMENT DO
  (pc(aArgs) == $start_pc) ->
    queue_messages(self).
% END STATEMENT DO

% BEGIN STATEMENT MOVE_TO
  (pc(aArgs) == $start_pc) ->
    ss(self, parent, $mcrl2NewState).
% END STATEMENT MOVE_TO

% BEGIN STATEMENT IF-THEN-ENDIF
  (pc(aArgs) == $start_pc1) ->
    (busy_children(chs) != []) ->
      insertIfBlockingWaiter($procName, $start_pc1)
    <>
    (convertExpr($fsmExpr)) ->
      enter_then_clause(self).
% END STATEMENT IF-THEN-ENDIF
Suppose we have the following FSM statements (pseudocode):

Queue STANDBY to c1 (-> 10, since 10 is the first available label)
Queue ON command to c1 (-> 12; note that 11 is reserved for the statement after the if statement)
Queue OFF command to c1 (-> 13)
Queue STANDBY command to c3 (-> 15; 15 is used by the then-block)
Queue OFF command to c3 (-> 16; unreachable due to the move_to on the previous line)
Queue ON command to c2 (-> 14; note that 14 is taken by the first statement of the else clause)
Queue ON command to c2 (-> 15)
Queue STANDBY command to c3 (-> 16; end of this block, so jump to the statement after the if statement)
Queue OFF command to c1 (-> 17; 17 is used by the then-block)
Queue OFF command to c2 (-> 18; special case: after a move_to we leave the action phase)
Queue ON command to c4 (-> 19; last statement in the list, so we jump to jump_pc)

% BEGIN THEN
$procExpr1
% END THEN
% END STATEMENT IF-THEN-ELSE-ENDIF

---

We will now give a simplified translation of these statements. For every statement, we give the label of the statement at the beginning of the line, and the label of the next statement at the end of the line, along with some explanations.

We assume that:
- start_pc1 = 5
- jump_pc = 6
- avail_pc = 10

The simplified translation follows:

5. queue STANDBY to c1 (-> 10, since 10 is the first available label)
10. IF there is a busy child
	ELSE IF b
	THEN enter_then_clause(self) (-> 12; note that 11 is reserved for the statement after the if statement)
	ELSE enter_else_clause(self) (-> 13)
12. queue ON command to c1 (-> 14; note that 13 is taken by the first statement of the else clause)
14. queue ON command to c2 (-> 15)
15. queue STANDBY command to c3 (-> 16; end of this block, so jump to the statement after the if statement)
16. move_to ERROR state (-> 17; special case: after a move_to, we leave the action phase)
17. queue OFF command to c3 (-> 18; unreachable due to the move_to on the previous line)
11. send ON command to c4 (-> 6; last statement in the list, so we jump to jump_pc)
$procExpr_1$

$procExpr_2$

$procExpr_3$

$avail_pc_4$

### Expressions

**Conversion of expressions.** We have two compound types: and-expressions and or-expressions:

- **[convertExpr-and]**
  
  ```
  convertExpr($fsmExpr_0 \text{ and } fsmExpr_1) = \text{convertExpr($fsmExpr_0)} \text{ && convertExpr($fsmExpr_1)}
  ```

- **[convertExpr-or]**
  
  ```
  convertExpr($fsmExpr_0 \text{ or } fsmExpr_1) = \text{convertExpr($fsmExpr_0)} \text{ || convertExpr($fsmExpr_1)}
  ```

- **And the expressions which check if certain children are in some specific state.** These depend on the specified children (either all children, any child, all children of a certain type, or any child of a certain type).

- **[convertExpr-allchildren]**
  
  ```
  convertExpr($fsmChildrenAllFwChildren \text{ in}_state fsmStateNameSpec) = \text{all_in_state(convertChildrenSpec($fsmChildrenAllFwChildren), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-anychildren]**
  
  ```
  convertExpr($fsmChildrenAnyFwChildren \text{ in}_state fsmStateNameSpec) = \text{any_in_state(convertChildrenSpec($fsmChildrenAnyFwChildren), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-allinstatespecific]**
  
  ```
  convertExpr($fsmChildrenAllSpecific \text{ in}_state fsmStateNameSpec) = \text{all_in_state(convertChildrenSpec($fsmChildrenAllSpecific), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-anyinstatespecific]**
  
  ```
  convertExpr($fsmChildrenAnySpecific \text{ in}_state fsmStateNameSpec) = \text{any_in_state(convertChildrenSpec($fsmChildrenAnySpecific), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-notallchildren]**
  
  ```
  convertExpr(!($fsmChildrenAllFwChildren \text{ in}_state fsmStateNameSpec)) = \text{!(all_in_state(convertChildrenSpec($fsmChildrenAllFwChildren), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-notallinstatespecific]**
  
  ```
  convertExpr(!($fsmChildrenAllSpecific \text{ in}_state fsmStateNameSpec)) = \text{!(all_in_state(convertChildrenSpec($fsmChildrenAllSpecific), convertStateNameSpec($fsmStateNameSpec))}
  ```

- **[convertExpr-bracket]**
  
  ```
  convertExpr($fsmExpr) = (convertExpr($fsmExpr))
  ```

- **Expressions can have brackets, simply leave them as they are and translate the expression inside them.**

- **[convertExpr-brackets]**
  
  ```
  convertExpr(($fsmExpr)) = (convertExpr($fsmExpr))
  ```

- **Apply the filter on the childrenlist.**
\[ \text{convertChildrenSpec}(\text{ALL} \text{ $fsmClassName}) = \text{filter}_\text{children}(\text{chs}, \text{concat}(\text{$fsmClassName$, \_CLASS})) \]

\[ \text{convertChildrenSpec}(\text{ANY} \text{ $fsmClassName}) = \text{filter}_\text{children}(\text{chs}, \text{concat}(\text{$fsmClassName$, \_CLASS})) \]

\[ \text{convertChildrenSpec}(\text{all}) = \text{convertChildrenSpec}(\text{ALL} \text{ $fsmClassName}) = \text{chs} \]

\[ \text{convertChildrenSpec}(\text{any}) = \text{convertChildrenSpec}(\text{ANY} \text{ $fsmClassName}) = \text{chs} \]

\[ \text{Conversion of the StateNameSpec in the FSMs into a list of states. A StateNameSpec is either a simple state name:} \]

\[ \text{convertStateNameSpec}(\text{single}) = \text{toMcrlStateName}(\text{$fsmStateName$}) \]

\[ \text{Or a set of state names in the form \{'name1', 'name2', ...\'. We can then distinguish two cases: We have exactly one statename or multiple statenames:} \]

\[ \text{convertStateNameSpec}(\text{multiple}) = \text{toMcrlStateName}(\text{$fsmStateNames$}) \]

\[ \text{inAnyState}(\text{empty}) = \text{false} \]

\[ \text{inAnyState}(\text{one}) = \text{isStateCheck}(\text{$mid$}, \text{s1}) \]

\[ \text{inAnyState}(\text{many}) = \text{isStateCheck}(\text{$mid$, $mid+$}) \]

\[ \text{create check if the currentState is state id, i.e. convert idStateCheck(idState) into: isMyState(s).} \]

\[ \text{mid}(\text{idHead, idTail}) = \text{toMcrlIsFunction}(\text{toMcrlStateName}(\text{id1})) \]

\[ \text{isStateCheck}(\text{id1}) = \text{mid}(\text{idHead, idTail})(\text{id1}) \]

\[ \text{create sort declaration stuff} \]

\[ \text{fsmGenerateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]

\[ \text{generateSorts}(\text{fsmClass}) = \]

\[ \text{convertClassNamesToTypeConstrDecl}(\text{collectClasses}(\text{fsmClass})) \]

\[ \text{convertStatesToStateConstrDecl}(\text{collectStates}(\text{fsmClass})) \]

\[ \text{convertCommandsToCmdConstrDecl}(\text{collectCommands}(\text{fsmClass})) \]

\[ \text{create Sort declaration stuff} \]
convertClassNamesToTypeConstrDecl($mid $mid+) = convertClassNamesToTypeConstrDecl($mid) | convertClassNamesToTypeConstrDecl($mid+)

%% Traversal functions to collect the classnames, action-names and statenames.

650 [collect-class-definition]
collectClasses(class: $FWPART_$TOP$ $fsmClassName $fsmState+, $mid*) = addToSet($fsmClassName, $mid*)

655 [collect-class-exprall]
collectClasses($ALL$fromClassName, $mid*) = addToSet(concat($fromClassName, _CLASS), $mid*)

660 [collect-action]
collectCommands(action: $fsmActionName $fsmStatement+, $mid*) = addToSet($fsmActionName, $mid*)

665 [collect-state]
collectStates(state: $fsmStateName $fsmWhenClause+ $fsmActionClause*, $mid*) = addToSet($fsmStateName, $mid*)

670 [collect-state-when]
collectStates(where ( $fsmExpr ) move_to $fsmStateName, $mid*) = addToSet($fsmStateName, $mid*)

675 [collect-state-statenamespec]
$mid*1 := collectStates({ $fsmStateNameSpecs* }, $mid*)
===>
collectStates( { $fsmStateName, $fsmStateNameSpecs*} , $mid*) = addToSet($fsmStateName, $mid*1)

680 [collect-state-statenamespec-1-element]
collectStates( $fsmStateName, $mid*) = addToSet($fsmStateName, $mid*)

%%%% Constructing a PType for an mcrl2 specification.

685 [mcrl2GetPTypes-1]
mcrl2GetPTypes($procSpec+) = PType = struct mcrl2PTypesFromProcSpecs($procSpec+);

690 [mcrl2PTypesFromProcSpecs-one]
mcrl2PTypesFromProcSpecs(proc $procDecl+) = mcrl2PTypesFromProcDecls($procDecl+)

695 [mcrl2PTypesFromProcSpecs-many]
mcrl2PTypesFromProcSpecs(proc $procDecl+$procSpec+) = mcrl2PTypesFromProcDecls($procDecl+) | mcrl2PTypesFromProcSpecs($procSpec+)

700 [mcrl2PTypesFromProcDecls-one-1]
mcrl2PTypesFromProcDecls($mid = $procExpr;) = convertClassNamesToTypeConstrDecl($mid)

705 [mcrl2PTypesFromProcDecls-many-1]
mcrl2PTypesFromProcDecls($mid = $procExpr; $procDecl+)= convertClassNamesToTypeConstrDecl($mid) | mcrl2PTypesFromProcDecls($procDecl+)

710 [mcrl2PTypesFromProcDecls-one-2]
mcrl2PTypesFromProcDecls($mid ( $idsDecls ) = $procExpr;) = convertClassNamesToTypeConstrDecl($mid) | mcrl2PTypesFromProcDecls($procDecl+)

715 [mcrl2PTypesFromProcDecls-many-2]
mcrl2PTypesFromProcDecls($mid ( $idsDecls ) = $procExpr; $procDecl+) = convertClassNamesToTypeConstrDecl($mid) | mcrl2PTypesFromProcDecls($procDecl+)

[addToSet-empty]
addToSet($mid,) = $mid

[addToSet-multisame]
addToSet($mid,$mid $mid*) = $mid $mid*

[addToSet-multidiff]
$mid != $mid1
===>
addToSet($mid,$mid1 $mid*) = $mid1 addToSet($mid, $mid*)

A.7  genericclauses.sdf

module genericclauses
  imports basic/Comments
  imports mcrlt
  exports context-free syntax

  % Insert the generic code that sends the commands to the children.
  insertGenericClauses(MId) -> ProcExpr

10
\textbf{A.8} genericclauses.asf

\begin{verbatim}

\texttt{\{insertGeneric\}}

\texttt{\{insertGenericClauses($\texttt{fsmClassName}$) =}

\texttt{\% BEGIN GENERIC CLAUSES (shared by all states)}

\texttt{\% Whenever we are not sending a command to the children, a child may}

\texttt{\% spontaneously change its state due to a hardware event and send its}

\texttt{\% state upward. Such state-change messages are called notifications.}

\texttt{\% Notifications can occur in the following cases:}

\texttt{\% (1) After initialization, while in the action phase:}

\texttt{\% (1.a) We have not received a command yet in this action phase.}

\texttt{\% (1.b) We are executing an action, or we finished executing an action but still have to send}

\texttt{\% some commands.}

\texttt{\% (2) During initialization.}

\texttt{\% Note that this implies that we never receive notifications during the}

\texttt{\% execution of the when phase (i) and we never receive notifications}

\texttt{\% directly after we finish sending the last command after executing an}

\texttt{\% action (ii).}

\texttt{\% The rationale behind this is as follows:}

\texttt{\% (i) The execution of the when clauses is a noninteractive process: the}

\texttt{\% system decides what the new state is, based only on local information.}

\texttt{\% (ii) After sending the last command, the model moves immediately into}

\texttt{\% the when phase. We should therefore not accept notifications at}

\texttt{\% this point.}

\texttt{\% (1.a) We have initialized and we have not yet received a command in this}

\texttt{\% action phase. We now accept notifications:}

\texttt{\% sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&}

\texttt{\% (pc(aArgs) == 0) && (initialized(chs))) ->}

\texttt{\% rs(id, self, s1).

\texttt{\% move_phase(self, WhenPhase).

\texttt{\% $\texttt{fsmClassName}(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), WhenPhase, reset(aArgs)))}}

\texttt{\% (1.b) We are in the middle of executing an action, or we finished}

\texttt{\% executing and still have to send some commands. We accept}

\texttt{\% notifications, but we don't move to the when phase, since we still}

\texttt{\% must execute one or more statements.}

\texttt{\% sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&}

\texttt{\% (pc(aArgs) > 0) ||

\texttt{\% ((pc(aArgs) == -1) && (cq(aArgs) != [ ]))) ->

\texttt{\% rs(id, self, s1).

\texttt{\% $\texttt{fsmClassName}(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), phase, aArgs))}}

\texttt{\% Clause to send commands added by actions in the initialization phase.}

\texttt{\% Note that we keep track of the children which have not yet responded.}

\texttt{\% We use the nrf list.}

\texttt{\% ((isActPhase(phase)) && (cq(aArgs) != [ ])) && (!initialized(chs))) ->}

\texttt{\% sc(self, id(head(cq(aArgs))), command(head(cq(aArgs))))).

\texttt{\% $\texttt{fsmClassName}(self, parent, s, update_busy(id(head(cq(aArgs))), true, chs), phase, actArg balances atcq(aArgs)), (id(head(cq(aArgs))))->(nrf(aArgs)), pc(aArgs), rsc(aArgs))}}

\texttt{\% Clause to send commands added by actions after the initialization phase.}

\texttt{\% Note that we don't keep track of the children which have not yet}

\texttt{\% responded. Recepients are only marked busy and not added to the nrf list.}

\texttt{\% ((isActPhase(phase)) && (cq(aArgs) != [ ])) && initialized(chs) ->}

\texttt{\% sc(self, id(head(cq(aArgs))), command(head(cq(aArgs))))).

\texttt{\% $\texttt{fsmClassName}(self, parent, s, update_busy(id(head(cq(aArgs))), true, chs), phase, actArg balances atcq(aArgs)), ([] pc(aArgs), rsc(aArgs))}}

\end{verbatim}
% (2) Clause to receive the new states from the children during
% initialization.

Note that some children may spontaneously change state and send a
notification. This may cause us to receive more than one message from a
child while we wait for all children to respond. If a child sends two
state messages, we will only consider the last state when we process the
when clauses.

\[
\text{sum id:Id.(sum s1:State.((isActPhase(phase)) && (cq(aArgs) == []) && (nrf(aArgs) != []) && (is_child(id, chs)) &&
! (initialized(chs))))} \rightarrow
\text{rs(id, self, s1).}
\]

\[
((initialized(update_state(id,s1,chs))) \rightarrow
\text{end_initialization(self).}
\]

when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
when ( $ANY$FwCHILDREN in_state RAMPING ) move_to RAMPING
when ( $ALL$FwCHILDREN in_state STANDBY ) move_to STANDBY
when ( $ALL$FwCHILDREN in_state OFF ) move_to OFF

\[
\text{state: OFF !color: FwStateOKNotPhysics}
\]

\[
\text{when ( ( $ALL$FwCHILDREN not_in_state OFF ) and}
\text{ ( $ANY$FwCHILDREN in_state STANDBY or $ANY$FwCHILDREN in_state ON or $ANY$FwCHILDREN in_state ERROR ))} \rightarrow
\text{move_phase(self, WhenPhase).}$
\]

\[
\text{B Wheel subsystem}
\]
( $\text{ALL$FwCHILDREN not\_in\_state ERROR} ) \text{ move\_to RAMPING}

\text{when ( ( $\text{ANY$FwCHILDREN in\_state OFF} ) and ( $\text{ALL$FwCHILDREN not\_in\_state ERROR} ) ) move\_to OFF}

\text{when ( $\text{BALL$FwCHILDREN in\_state ON} ) move\_to ON}

55
\text{when ( ( $\text{ANY$FwCHILDREN in\_state STANDBY} ) and ( $\text{ALL$FwCHILDREN not\_in\_state ERROR} ) ) move\_to STANDBY}

\text{action ON \text{ visible: 1}}
\text{do ON $\text{ALL$FwCHILDREN}}

\text{60}
\text{action STANDBY \text{ visible: 1}}
\text{do STANDBY $\text{ALL$FwCHILDREN}}

\text{action OFF \text{ visible: 1}}
\text{do OFF $\text{ALL$FwCHILDREN}}

\text{state: RAMPING \text{ color: FullStateAttention1}}
\text{when ( $\text{ANY$FwCHILDREN in\_state ERROR} ) move\_to ERROR}
\text{when ( $\text{ALL$FwCHILDREN in\_state ON} ) move\_to ON}
\text{when ( $\text{ALL$FwCHILDREN in\_state STANDBY} ) move\_to STANDBY}
\text{when ( ( $\text{ALL$FwCHILDREN not\_in\_state RAMPING} ) and ( $\text{ANY$FwCHILDREN in\_state OFF} ) ) move\_to OFF}
\text{when ( ( $\text{ALL$FwCHILDREN not\_in\_state RAMPING} ) and ( $\text{ANY$FwCHILDREN in\_state STANDBY} ) ) move\_to STANDBY}

\text{C Wheel translation}

1 \text{ sort}\n\text{Phase = struct WhenPhase \& ActionPhase | ActionPhase ? is\text{ActionPhase};}
\text{ActPhaseArgs = struct act Args (cq: CommandQueue, arf: IdList, pt: Int, res: Bool); Id = Nat;}
\text{IdList = List(\text{Id}); Child = struct child(id:Id, state:State, ptype:PType, busy:Bool); Children = List(\text{Child}); ChildCommand = struct childcommand(id:Id, command:Command); CommandQueue = List(\text{ChildCommand}); PType = struct RPC\_Wheel\_CLASS ? is\text{RPC\_Wheel\_CLASS}; State = struct S\_FSM\_UNINITIALIZED ? is\text{S\_FSM\_UNINITIALIZED | S\_OFF ? is\text{S\_OFF | S\_ERROR ? is\text{S\_ERROR | S\_RAMPING ? is\text{S\_RAMPING | S\_STANDBY ? is\text{S\_STANDBY | S\_ON ? is\text{S\_ON}; Command = struct C\_STANDBY ? is\text{C\_STANDBY | C\_OFF ? is\text{C\_OFF | C\_ON ? is\text{C\_ON;}

15 \text{ act}\n\text{rc,sc,cc: Id \# Id \# Command; rs,ss,cs: Id \# Id \# State; move\_state: Id \# State; move\_phase: Id \# Phase; ignored\_command: Id \# Command; queue\_messages: Id; enter\_then\_clause: Id; enter\_else\_clause: Id; skip\_then\_clause: Id; start\_initialization: Id; end\_initialization: Id; noop\_statement: Id;}

15 \text{ map}\n\text{in\_state: Child \# State -> Bool; in\_any\_of\_states: Child \# List(\text{State}) -> Bool; any\_in\_state: Children \# List(\text{State}) -> Bool; all\_in\_state: Children \# List(\text{State}) -> Bool; is\_child: Id \# Children -> Bool; filter\_children: Children \# PType -> Children; filter\_children\_accu: Children \# PType -> Children; send\_command: Command \# Children -> CommandQueue; update\_state: Id \# State \# Children -> Children; update\_busy\_all: Bool \# Children -> Children; remove: Id \# IdList -> IdList; initialized: Children -> Bool; children\_to\_ids: Children -> IdList; busy\_children: Children -> Children; update\_pt: ActPhaseArgs \# Int -> ActPhaseArgs; reset: ActPhaseArgs -> ActPhaseArgs; var cq: CommandQueue;}
eqn

in_state(child(id,s,t,b),s1) = s == s1;

in_any_of_states(ch,[]) = false;

in_any_of_states(ch,s|>sl) = in_state(ch,s) || in_any_of_states(ch,sl);

all_in_state([], sl) = true;

all_in_state(ch|> sl, sl) = in_any_of_states(ch,sl) && all_in_state(ch,sl);

is_child(id, []) = false;

is_child(id, child(id1,s,t,b) |> chs) = id == id1 || is_child(id, chs);

filter_children(chs, t) = filter_children_accu(chs, t, []);

filter_children_accu([], t, chs_accu) = chs_accu;

filter_children_accu(child(id,s,t1,b) |> chs, t, chs_accu) = 
if(t==t1,
    filter_children_accu(chs, t, child(id,s,t,b) |> chs_accu),
    filter_children_accu(chs, t, chs_accu));

send_command(cmd, []) = [];

send_command(cmd, child(id,s,t,b) |> chs) =
    childcommand(id,cmd) |> send_command(cmd,chs);

update_state(id, s, []) = [];

update_state(id, s, child(id1,s1,t,b) |> chs) = 
if(id==id1,
    child(id1,s,t,b) |> chs,
    child(id1,s1,t,b) |> update_state(id,s,chs));

update_busy(id, b, []) = [];

update_busy(id, b, child(id,s,t,b) |> chs) =
if(id==id,
    child(id,s,t,b) |> chs,
    child(id,s,t,b) |> update_busy(id,b,chs));

update_busy_all(b, []) = [];

update_busy_all(b, child(id,s,t,b) |> chs) = child(id,s,t,b) |> update_busy_all(b, chs);

remove(id, []) = [];

remove(id, id) =
if (id == id,
    id,
    id |> remove(id, ids));

initialized(chs) = !any_in_state(chs, [S_FSM_UNINITIALIZED]);

children_to_ids([]) = [];

children_to_ids(child(id,s,t,b) |> chs) = id |> children_to_ids(chs);

busy_children([]) = [];

busy_children(child(id,s,t,true) |> chs) = child(id,s,t,true) |> busy_children(chs);

busy_children(child(id,s,t,false) |> chs) = busy_children(chs);

update_pc(actArgs(cq, ids, pc, b), pc1) = actArgs(cq, ids, pc1, b);

reset(actArgs(cq, ids, pc, b)) = actArgs([], [], 0, b);

proc RPC_Wheel_CLASS(self: Id, parent: Id, s: State, chs: Children, phase: Phase, aArgs: ActPhaseArgs) =

% BEGIN STATE
% ------------------
% BEGIN WHEN CLAUSES
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
    any_in_state(chs, [S_ERROR])) ->
move_state(self, S_ERROR).
RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN
(()
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
(any_in_state(chs, [S_RAMPING]))) ->
move_state(self, S_RAMPING).
RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN
(()
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
(all_in_state(chs, [S_STANDBY]))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
(()
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
((!(any_in_state(chs, [S_OFF]))) && (any_in_state(chs, [S_STANDBY])))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
(()
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_OFF(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
(pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
start_initialization(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
sq(c:Command. (rc(parent, self, c). isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
(isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
(isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))) +
% END CLAUSE SELECTOR
% BEGIN ACTION
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
run c:Command. (rc(parent, self, c). isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
(isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
(isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
ss(self, parent, s).
ignore_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))) +
% END ACTION
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
start_initialization(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
run c:Command. (rc(parent, self, c). isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
(isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
(isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
ss(self, parent, s).
ignore_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))) +
% END CLAUSE SELECTOR
% BEGIN ACTION
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
run c:Command. (rc(parent, self, c). isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
(isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
(isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
ss(self, parent, s).
ignore_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))) +
% END ACTION
((
    % BEGIN ACTION
    (((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) -> 
      
      % BEGIN STATEMENT DO
      ((pc(aArgs) == 2) -> 
        queue_messages(self).
        (RPC_Wheel_CLASS(self, parent, s, chs, phase,
          actArgs(send_command(C_OFF, chs), [], -1, rsc(aArgs))))
      )
      % END STATEMENT DO
    )
    % END ACTION
  ) +

((
    % BEGIN ACTION
    (((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) -> 
      
      % BEGIN STATEMENT DO
      ((pc(aArgs) == 3) ->
        queue_messages(self).
        (RPC_Wheel_CLASS(self, parent, s, chs, phase,
          actArgs(send_command(C_ON, chs), [], -1, rsc(aArgs))))
      )
      % END STATEMENT DO
    )
    % END ACTION
  )
)

% END ACTION CLAUSES
% ------------------
% END STATE
% =========

% BEGIN STATE
% ------------------
% BEGIN WHEN CLAUSES
% % BEGIN WHEN

((isS_STANDBY(s)) && (isWhenPhase(phase)) && 
  any_in_state(chs, [S_ERROR])) -> move_state(self, S_ERROR).

RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN

((isS_STANDBY(s)) && (isWhenPhase(phase)) && 
  any_in_state(chs, [S_RAMPING])) -> move_state(self, S_RAMPING).

RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN

((isS_STANDBY(s)) && (isWhenPhase(phase)) && 
  all_in_state(chs, [S_ON])) -> move_state(self, S_ON).

RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>
% END WHEN

((isS_STANDBY(s)) && (isWhenPhase(phase)) && 
  any_in_state(chs, [S_OFF])) -> move_state(self, S_OFF).

RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>
% END WHEN

% BEGIN WHEN FALLTHROUGH

((isS_STANDBY(s)) && (isWhenPhase(phase)) -> ss(self, parent, s).
  move_phase(self, ActionPhase).
  RPC_Wheel_CLASS(self, parent, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH

% END WHEN CLAUSES
% ------------------

39
\% BEGIN ACTION CLAUSES
\% BEGIN INITIALIZATION CHECK
((isS_STANDBY(s)) && (isActPhase(phase)) && (!(initialized(chs))) && (pc(aArgs) == 0) && (nrf(aArgs) == \[])) ->
  start_initialization(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
\% END INITIALIZATION CHECK
\% BEGIN CLAUSE SELECTOR

((initialized(chs)) ->
  (isS_STANDBY(s) && isActPhase(phase) && (cq(aArgs) == \[]) && (pc(aArgs) == 0)) ->
  sum c:Command.
    rc(parent, self, c).
    (isC_ON(c) ->
      RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
      (isC_OFF(c) ->
        RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
        (isC_STANDBY(c) ->
          RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
          ignored_command(self, c).
          RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))
) +

\% END CLAUSE SELECTOR

(\% BEGIN ACTION

((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == \[])) ->

(\% BEGIN STATEMENT DO

((pc(aArgs) == 1) ->
  queue_messages(self).
  RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs(send_command(C_ON, chs), [], -1, rsc(aArgs))))) +

\% END STATEMENT DO

) +

\% END ACTION

((\% BEGIN ACTION

((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == \[])) ->

(\% BEGIN STATEMENT DO

((pc(aArgs) == 2) ->
  queue_messages(self).
  RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs(send_command(C_OFF, chs), [], -1, rsc(aArgs))))) +

\% END STATEMENT DO

) +

\% END ACTION

((\% BEGIN ACTION

((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == \[])) ->

(\% BEGIN STATEMENT DO

((pc(aArgs) == 3) ->
  queue_messages(self).
  RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs(send_command(C_STANDBY, chs), [], -1, rsc(aArgs))))) +

\% END STATEMENT DO

) +

\% END ACTION

(delta)))
)
\% END ACTION CLAUSES

\% END STATE

\% ===========

\% BEGIN STATE

\% BEGIN WHEN CLAUSES

\% BEGIN WHEN

(\% BEGIN STATEMENT DO

(delta)))
)
\% END ACTION CLAUSES

\% END STATE

\% ===========
(\(\text{isS\_ON(s)}\) && (\(\text{isWhenPhase(phase)}\) && (\(\text{any\_in\_state(chs, [S\_ERROR])}\))) -> move\_state(self, S\_ERROR).

RPC\_Wheel\_CLASS(self, parent, S\_ERROR, chs, phase, aArgs) <>

\(\text{--- END WHEN ---}\)

(\(\text{isS\_ON(s)}\) && (\(\text{isWhenPhase(phase)}\) && (\(\text{any\_in\_state(chs, [S\_RAMPING])}\))) -> move\_state(self, S\_RAMPING).

RPC\_Wheel\_CLASS(self, parent, S\_RAMPING, chs, phase, aArgs) <>

\(\text{--- END WHEN ---}\)

(\(\text{isS\_ON(s)}\) && (\(\text{isWhenPhase(phase)}\) && (\(\text{any\_in\_state(chs, [S\_OFF])}\))) -> move\_state(self, S\_OFF).

RPC\_Wheel\_CLASS(self, parent, S\_OFF, chs, phase, aArgs) <>

\(\text{--- END WHEN ---}\)

(\(\text{isS\_ON(s)}\) && (\(\text{isWhenPhase(phase)}\) && (\(\text{any\_in\_state(chs, [S\_STANDBY])}\))) -> move\_state(self, S\_STANDBY).

RPC\_Wheel\_CLASS(self, parent, S\_STANDBY, chs, phase, aArgs) <>

\(\text{--- END WHEN ---}\)

\% BEGIN WHEN FALLTHROUGH

(\(\text{isS\_ON(s)}\) && (\(\text{isActPhase(phase)}\) && (!\(\text{initialized(chs)}\)) && (pc(aArgs) == 0) && (nrf(aArgs) == [])) -> start\_initialization(self).

RPC\_Wheel\_CLASS(self, parent, s, chs, phase, actArgs([], children\_to\_ids(chs), 0, rsc(aArgs))) <>

\(\text{--- END INITIALIZATION CHECK ---}\)

\% BEGIN CLAUSE SELECTOR

\% BEGIN STATEMENT DO

((pc(aArgs) == 1) -> queue\_messages(self).

RPC\_Wheel\_CLASS(self, parent, s, chs, phase, update\_pc(aArgs, 1)) <>

\% END STATEMENT DO

\% END CLAUSE SELECTOR

\% BEGIN ACTION CLAUSES

\% BEGIN INITIALIZATION CHECK

((\(\text{isS\_ON(s)}\) && (\(\text{isActPhase(phase)}\) && (cq(aArgs) == [])) && (pc(aArgs) == 0)) -> run c:\text{Command}.

* (pc(aArgs) == 0) && (\(\text{isC\_STANDBY(c)}\) -> RPC\_Wheel\_CLASS(self, parent, s, chs, phase, update\_pc(aArgs, 1)) <>

\% END INITIALIZATION CHECK

\% BEGIN CLAUSE SELECTOR

((\(\text{isS\_ON(s)}\) && (\(\text{isActPhase(phase)}\) && (cq(aArgs) == [])) && (pc(aArgs) == 0)) -> run c:\text{Command}.

* (pc(aArgs) == 0) && (\(\text{isC\_STANDBY(c)}\) -> RPC\_Wheel\_CLASS(self, parent, s, chs, phase, update\_pc(aArgs, 1)) <>

\% END CLAUSE SELECTOR

\% BEGIN ACTION

\% BEGIN STATEMENT DO

queue\_messages(self).

RPC\_Wheel\_CLASS(self, parent, s, chs, phase, actArgs(send\_command(C\_STANDBY, chs), [], -1, rsc(aArgs))) <>

\% END STATEMENT DO

\% END ACTION

\% BEGIN ACTION

\% BEGIN STATEMENT DO

queue\_messages(self).

RPC\_Wheel\_CLASS(self, parent, s, chs, phase, actArgs(send\_command(C\_STANDBY, chs), [], -1, rsc(aArgs))) <>

\% END STATEMENT DO

\% END ACTION
(RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs(send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

% END STATEMENT DO
)) +

% END ACTION

(delta)))))
})

% END ACTION CLAUSES

% END STATE

% =========

% BEGIN STATE

% BEGIN WHEN CLAUSES

% BEGIN INITIALIZATION CHECK
((isS_ERROR(s)) && (isActPhase(phase)) && (!(initialized(chs))) && (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->

start_initialization(self).

)) +

% END WHEN CLAUSES

% BEGIN ACTION CLAUSES

% BEGIN INITIALIZATION CHECK
((isS_ERROR(s)) && (isActPhase(phase)) && (!(initialized(chs))) && (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->

start_initialization(self).

)
RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs([]), children_to_ids(chs), 0, rsc(aArgs)) <>

% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->

(((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) && (pc(aArgs) == 0)) -> sum c:Command. (rc(parent, self, c). (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (ss(self, parent, s). ignored_command(self, c). RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))

% END CLAUSE SELECTOR
% BEGIN ACTION
((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

% BEGIN STATEMENT DO
((pc(aArgs) == 1) -> queue_messages(self). RPC_Wheel_CLASS(self, parent, s, chs, phase, actArgs(send_command(C_ON, chs), [], -1, rsc(aArgs))))

% END STATEMENT DO
% END ACTION

% BEGIN STATE
((isS_RAMPING(s)) && (isActPhase(phase)) && (any_in_state(chs, [S_ERROR]))) -> move_state(self, S_ERROR). RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>

% END STATE
% =========

% BEGIN WHEN CLAUSES
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) && (all_in_state(chs, [S_ON]))) -> move_state(self, S_ON). RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>

% END WHEN
% END CLAUSE SELECTORS
BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
(all_in_state(chs, [S_STANDBY]))) ->
moveset(self, S_STANDBY).

RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>

END WHEN

BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
((!(any_in_state(chs, [S_RAMPING]))) && (any_in_state(chs, [S_OFF])))) ->
moveset(self, S_OFF).

RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>

END WHEN

BEGIN WHEN FALLTHROUGH
(((isS_RAMPING(s)) && (isActPhase(phase))) ->
ss(self, parent, s).

move_phase(self, ActionPhase).

RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))

END WHEN FALLTHROUGH

BEGIN INITIALIZATION CHECK
((isS_RAMPING(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
(pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
start_initialization(self).

RPC_Wheel_CLASS(self, parent, s, chs, phase,
actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>

END INITIALIZATION CHECK

BEGIN CLAUSE SELECTOR
((initialized(chs)) ->

(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
sum c:Command.
rc(parent, self, c).
(isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <>
(isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <>
(isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <>
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1))))))))

END CLAUSE SELECTOR

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_STANDBY, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 3) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 4) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 5) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 6) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 7) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION

BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->

BEGIN STATEMENT DO
((pc(aArgs) == 8) ->
queue_message(self).
RPC_Wheel_CLASS(self, parent, s, chs, phase, send_command(C_OFF, chs), [], -1, rsc(aArgs)))))

END STATEMENT DO

END ACTION
% END STATE DO
% END ACTION
}
% END ACTION CLAUSES
% -----------------
% END STATE
% =========
)
% BEGIN GENERIC CLAUSES (shared by all states)

sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
(pc(aArgs) == 0) && (initialized(chs))) ->
rs(id, self, s1).
RPC_Wheel_CLASS(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), phase, aArgs)) +

sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
((pc(aArgs) > 0) ||
((pc(aArgs) == -1) && (cq(aArgs) != [])))) ->
rs(id, self, s1).
RPC_Wheel_CLASS(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), phase, aArgs)) +

((isActPhase(phase)) && (cq(aArgs) != []) && (initialized(chs))) ->
sc(self, id(head(cq(aArgs))), command(head(cq(aArgs))))).
RPC_Wheel_CLASS(self, parent, s, update_busy(id(head(cq(aArgs))), true, chs), phase, aArgs)) +

((isActPhase(phase)) && (cq(aArgs) != []) && (!((initialized(chs))))) ->
sc(self, id(head(cq(aArgs))), command(head(cq(aArgs)))).
RPC_Wheel_CLASS(self, parent, s, update_busy(id(head(cq(aArgs))), true, chs), phase, aArgs)) +

sum id:Id.(sum s1:State.(((isActPhase(phase)) && (cq(aArgs) == []) && (nrf(aArgs) != []) && (is_child(id, chs)) &&
(!(initialized(chs)))) ->
rs(id, self, s1).
end_initialization(self).

RPC_Wheel_CLASS(self, parent, s, update_state(id, s1, chs), phase, aArgs)).

RPC_Wheel_CLASS(self, parent, s, update_state(id, s1, chs), phase, aArgs)) +

((isActPhase(phase)) && (cq(aArgs) == []) && (initialized(chs)) && (pc(aArgs) == -1)) ->
mov_phase(self, phase).
RPC_Wheel_CLASS(self, parent, s, chs, phase, aArgs)) +

% END GENERIC CLAUSES

};

init
allow({cs, cc, move_state, move_phase, ignored_command,
queue_messages, enter_then_clause, enter_else_clause,
skip_then_clause, start_initialization, end_initialization,
noop_statement),
comm({rs|ls -> cs, rc|sc -> cc},
RPC_Ahead_CLASS(1, S_OFF, [],
    ActionPhase, actArgs([], [], 0, false))};