Formalising the State Machine Modelling Tool (SMMT)

MASTER THESIS

Goirle
December 1, 2023

J.E.P.M. van Laarhoven

TU/e Student ID: 1230803

`info@jordivanlaarhoven.nl`

`j.e.p.m.v.laarhoven@student.tue.nl`

`jordi.vanlaarhoven@cpp.canon`

Supervisors	Version
Dr. Ir. T.A.C. Willemse[1]	
Dr. Ir. L.C.M. van Gool[2]	1.0.2
Ir. O. Bunte[1]	

1 Eindhoven University of Technology, Eindhoven, The Netherlands
2 Canon Production Printing, Venlo, The Netherlands

Abstract

Model-Driven (Software) Engineering (MDSE) is gaining popularity in industry. More and more companies acknowledge the benefits of using MDSE to develop their software components. A company that exploits model-driven software engineering is Canon Production Printing (Venlo, The Netherlands). At Canon Production Printing, the State Machine Modelling Tool (SMMT) was developed to enable the modelling of software components using state machines. In this graduation report, we present SMMT and formally define a subset of the SMMT language. A translation from SMMT specifications to mCRL2 specifications is defined to allow for model checking on the SMMT specifications. We show that this translation can correctly generate mCRL2 specification for the existing SMMT specifications at Canon Production Printing. Furthermore, we show how the mCRL2 toolset can be used to prove the correctness of SMMT specifications.

Contents

Chapter 1

Introduction

Model-Driven (Software) Engineering (MDSE) is gaining popularity in industry. More and more companies acknowledge the benefits of using MDSE to develop their software components. One advantage of MDSE is that it raises the level of abstraction. This simplifies the development of complex components, which increases the productivity of the software engineers. Another major benefit of using MDSE is that it opens the door to model checking. Using the created models, model checkers can be used to analyse whether certain properties are satisfied by a model, properties that may not be verifiable using test cases. Both advantages result in a more cost-effective approach.

A company that exploits model-driven software engineering is Canon Production Printing (Venlo, The Netherlands). One of the modelling languages that has been developed over the years is the Open Interaction Language (OIL). OIL is a modelling language that is intended to express the behaviour of control-software components, which is not limited to printers. The semantics of the Open Interaction Language has been formally defined, which allows a translation from specifications in OIL to mCRL2 model specifications.

The mCRL2 model checker is developed by the Eindhoven University of Technology and consist of both a language [1] and a powerful toolset [2]. The mCRL2 language is used to describe the behaviour of processes based on the algebra of communicating processes [3] that is extended with data. Among others, the tool enables the generation and reduction of the state space of an mCRL2 specification. Using the toolset, one can verify whether properties are satisfied for the given specification.

This graduation project concerns the State Machine Modelling Tool (SMMT). SMMT is a tool and modelling language developed at Canon Production Printing to model the behaviour of software components using state machines. Executable C++ and C# code can be generated from the SMMT specifications. Unlike OIL, SMMT has not been developed with a focus on creating a formally defined language to allow for model checking. Instead, SMMT was developed to closely resemble the state machines of the existing Boost Statechart C++ library [4, 5] but with some of the additional expressive power of OIL.

In the remainder of this chapter, we discuss related research (Section 1.1) and introduce the research question that serves as a basis for this graduation project (Section 1.2).

1.1 Related Work

In this section, we discuss a selection of domain specific programming languages of which the semantics have been formalized and that are translated to the mCRL2 language or the language of other model checkers for automatic property verification. We discuss research that has been performed on the analysis, translation and verification of languages based on state machines.

1.1.1 Open Interaction Language (OIL)

A language that is closely related to SMMT is the Open Interaction Language (OIL) by L. van Gool. OIL and SMMT are both domain specific modelling languages developed within Canon Production Printing. Using OIL, the behaviour of control-software components can be specified, analysed, and visualized. In "Formal verification of OIL component specifications using mCRL2", O. Bunte et al. [6] defined the formal semantics of OIL specifications. A translation to mCRL2 specifications with the same behaviour as the OIL specification has been defined, based on the formal operational semantics. Using the mCRL2 toolset [2], properties can be expressed using the first-order modal mu-calculus and automatically verified on a given mCRL2 specification. OIL specifications are originally based on XML. A domain specific language has been designed and implemented by J. Denkers et al. [7] which, together with some additional syntactic sugar to OIL, creates a more user-friendly way to specify components using OIL. This DSL has been implemented using the Spoofax language workbench [8]. Using both the XML and DSL syntax of OIL, C++ code can be automatically generated, the correctness of which has partially been verified [6]. An experimental code generator has been implemented by M. Frenken [9] and has been optimized by T. Buskens [10].

1.1.2 Dezyne

Another modelling language that has been formalized is the Dezyne language by the company Verum [11]. The Dezyne toolset allows engineers to specify and design software systems. Furthermore, the toolset allows for formal verification. In "Formalising the Dezyne Modelling Language", R. Beusekom et al. [12] formalised the Dezyne language by encoding the language in the mCRL2 process algebra [1]. In this paper, Beusekom et al. discuss the grammar of the Dezyne language using the Extended Backus-Naur Form (EBNF), which is a notation to denote the grammar of a language, standardized in ISO/IEC 14977 [13]. For each of the constructs in the Dezyne language a transformation to the mCRL2 language is introduced and a sketch of the formalisation of the execution semantics is presented. Using the Dezyne language executable code can be generated.

1.1.3 Coco

Another platform that aims at simplifying the development of software components is the Coco platform by Cocotec [14]. Using their Coco language, the behaviour of systems can be modelled. The tool uses the internally developed model checker FDR4 [15] which can automatically verify whether certain properties are satisfied in the model. The tool offers a graphical interface for debugging the models. Using UML sequence diagrams, traces can be visualized to show how a property is violated. From the Coco models, executable C++ code can be generated.

1.1.4 Event-B

In "Formal verification and validation of run-to-completion style state charts using Event-B", K. Morris et al. [16] present how they formally verified and validated state charts using Event-B and Rodin. Morris et al. discuss a new notation for state chart modelling, on which the Event-B toolset is used for theorem proving. The Event-B tooling is supported by the Rodin platform, that is used for refinement and mathematical proofs. A scenario-based approach is introduced, based on the ProB model checker. Certain scenarios can be recorded and replayed to check whether states changed since the original run of the scenario. The Rodin theorem prover automatically verifies whether invariant properties are satisfied on the models. They also provide a complementary way to verify the model using the LTL model checker. This procedure allows to check for expected reactions to environmental triggers.

1.1.5 CERN

Another organization that adopted the MDSE approach is CERN. For their Large Hadron Collider (LHC), state machines are used to generate executable code. In "Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider", Y. Ling Hwong et al. [17] formalised the semantics of the state machines by converting the state machines to the process algebra language of mCRL2. Using the translation from state machines to the mCRL2 process algebra language and the formal semantics, Ling Hwong et al. were able to develop tooling for checking whether certain properties hold on each state machine in isolation. Using their translation to mCRL2, various bugs were found in the models, which have been solved since.

1.1.6 Cordis SUITE

Various papers have been published on formalising models based on UML models. As SMMT is based on state machines, it is therefore interesting to investigate papers that discuss translation of UML models to mCRL2. An example of such a language is the Cordis SUITE. In "Formal Verification of an Industrial UML-like Model using mCRL2", A. Stramaglia and J. Keiren [18] present how they formalised the semantics of the Cordis SUITE, which is a low-code developing platform for machine-control applications that is based on UML models. A translation from the Cordis models to the process algebra language of mCRL2 has been defined, based on the semantics of the Cordis language. In the paper, Stramaglia and Keiren show how powerful the usage of model checkers like mCRL2 is, through an experiment (Pneumatic Cylinder) in which bugs were found in the generated mCRL2 model and its implementation.

1.1.7 Executable UML Specifications

In "Towards model checking executable UML specifications in mCRL2", H. Hansen et al. [19] present their translation for translating executable UML specifications to mCRL2 specifications. In the earlier mentioned research projects, properties on the generated mCRL2 models were directly performed using the first order modal mu-calculus using the mCRL2 toolset. In the paper by Hansen et al., tooling is presented that allows the software engineer to inspect error traces using UML sequence diagrams, instead of a labelled transition system. Using their toolset, the software engineer can express the expected behaviour as a UML state machine. In this state machine, the engineer can add "error" actions for behaviour that is not expected to happen. In this way, checking whether some safety property is satisfied in a model boils down to a reachability check for such "error" actions in mCRL2. Unfortunately, their system only allows the verification of safety properties but not liveness properties. Liveness properties would require the use of temporal logics.

1.1.8 Internet of Things

In "The AbU Language: IoT Distributed Programming Made Easy", M. Pasqua et al. [20] introduce a domain specific language that can be used to control IoT devices. The language is built on the Attribute-based memory Updates communication model which is an extension of the ECA rules-based system. ECA systems use Event-Condition-Action rules for controlling the IoT device. The AbU language enhances the ECA rules with an interaction mechanism where communication is similar to broadcasting. However, not all communication is sent via a central processing infrastructure. Instead, decentralized edge servers are used.

The AbU language is a domain specific language in which programs can be developed consisting of a list of IoT devices, a list of type declarations and a collection of ECA rules. A device declaration covers the name and description of the device, as well as all details about the resources of the device such as in- and output resources, an invariant that must be maintained and references to a set of ECA rules. Type declarations specify several custom types that can be used in the device specification. The ECA rules specify the behaviour of the devices. The language is event driven, when an event occurs for which a rule has been defined and the condition of that rule is satisfied, the action that is defined by the ECA rule is performed.

In the paper, M. Pasqua et al. first introduce the syntax of the AbU language. After which the formal operational semantics of the AbU language is presented. Experiments are discussed which show the effectiveness of the AbU language. Unfortunately, there is no support for model checking or other verification tools. Hence, it is not possible to verify whether certain properties hold in a AbU model/specification. In the future work section, the authors show interest in implementing a verification approach based on temporal logic like CTL or mu-calculus to allow for verification.

1.2 Problem Statement

In this section we present the research question that will serve as a basis for this master's graduation project at the Eindhoven University of Technology. In this project, we perform research on the State Machine Modelling Tool (SMMT).

SMMT is a tool that can be used to model the behavior of complex software components using state machines. As mentioned before, SMMT has been developed with a focus to closely resemble the existing Boost C++ library. There exists no formal definition of SMMT. As the available documentation of SMMT does not cover all constructs of the language, it remains unclear for the engineers at Canon Production Printing how certain behavior can be modelled using SMMT. As a consequence, engineers may model behavior of software components in SMMT of which the expected behaviour does not correspond to the actual behaviour.

A testing library has been developed to determine the correctness of the existing SMMT specifications. The testing library requires the engineer to specify the behavior that they want to test on an SMMT specification as a separate SMMT specification. The test SMMT specification sends events to the SMMT specification and defines the correctness based on the responses received from the SMMT specification. According to the engineers that are familiar with the testing library, this approach to determine the correctness of an SMMT specification is time-consuming and error prone. Furthermore, there are limitations on what properties can be checked on an SMMT specification using the testing library. A formal definition of SMMT would allow us to define a translation from SMMT specifications to mCRL2 specifications. Using such an mCRL2 specification, the engineer can verify whether certain properties hold on the specification. Furthermore, there exist various restrictions that must be satisfied by each SMMT specification that are not checked by the implementation of SMMT in MPS. The translation from SMMT specifications to mCRL2 specifications enables us to automatically verify whether these restrictions as defined in the generated mCRL2 specification hold for each SMMT specification.

The aim of this graduation project is to answer the following research question:

"To what extent can we formalise and prove the correctness of SMMT specifications using the mCRL2 model checker?"

To formalise and prove the correctness of SMMT specifications, we first performed an analysis of SMMT. In this analysis we discuss how the engineers at Canon Production Printing experience SMMT and discuss the insights we gained from the 26 SMMT specifications that have been developed at Canon Production Printing at the time of writing. The results of the analysis show the relevance of formally defining SMMT and defining a translation from SMMT specifications to mCRL2 specifications to allow for model checking. Furthermore, we have analysed the relevance and frequency of the constructs of the SMMT language to determine the order in which the constructs of SMMT were formalised.

To formally define SMMT, we defined the syntax, static semantics and operational semantics of SMMT. A mathematical model has been defined that is used to represent an SMMT specification. The static semantics has been defined as a number of restrictions that must be satisfied on this mathematical model. Finally, the operational semantics of SMMT has been defined as a labelled transition system. Unfortunately, we have not able to formally define the complete set of constructs of the language of SMMT due to time constraints.

We have defined a translation that translates SMMT specifications to mCRL2 specifications. The mCRL2 specification that is generated from an SMMT specification closely correlates to the formal definition of SMMT. The generated mCRL2 contains a representation of the mathematical model of the SMMT specification. Using this representation of the mathematical model, we have defined mapping and equations that correspond to the validation checks and the functions that are used to define the operational semantics. The process specification of the mCRL2 specification correlates to the definition of the operational semantics.

Various experiments have been performed to prove the correctness of the translation that we defined. We defined a translation to generate an mCRL2 specification from the generated C++ code that is generated by SMMT. Using the mCRL2 toolset, we have verified whether the behavior of the two mCRL2 specifications define the same behavior. These experiments have been performed on the 26 SMMT specifications that exist at the time of writing. We have shown that the translation is correct for 25 out of the 26 SMMT specifications and that the correctness could not be determined for 1 SMMT specification. Furthermore, several issues in the implementation of SMMT in MPS were found when performing these experiments. We have shown how properties can be checked on the generated mCRL2 specifications and shown several properties that we expect must hold for all SMMT specifications.

Outline This thesis is structured as follows. We first informally introduce SMMT in Chapter 2. Next, we discuss our analysis of SMMT in Chapter 3. In Chapter 4 we introduce several definitions that are used to formally define SMMT. We formally define a subset of the constructs of the SMMT language in Chapter 5. In Chapter 6, we introduce the mCRL2 model checker and toolset and we discuss the approach for defining the translation from SMMT to mCRL2 specifications. In Chapter 7 we discuss the experiments that we have performed to verify the correctness of the translation and to verify the correctness of the existing SMMT specifications. We discuss the results and answer the research question in Chapter 8. Finally, we discuss opportunities for future research in Chapter 9.

Chapter 2

State Machine Modelling Tool

The State Machine Modelling Tool (SMMT) is a modelling tool and language developed by Canon Production Printing that allows its user to model behavior of software components using state machines. Furthermore, SMMT enables the generation of C# and C++ code from these SMMT models. In this section we informally introduce the State Machine Modelling Tool. We discuss the history and the goal of SMMT in Section 2.1. In Section 2.2, we introduce the constructs of the SMMT language. To get some intuition on how the generated code of an SMMT specification behaves, we discuss how the events are handled during execution in Section 2.3.

2.1 Introduction to SMMT

At Canon Production Printing the Model-Driven Software development methodology was adopted a couple of decades ago. Using modelling software, models were designed consisting of state machines that modelled the expected behavior of software components. The engineers used these models to manually develop executable code that behaves according to the model. This workflow occasionally led to problems, as there was no coupling between the designed models and the code that was developed using these model. As a consequence, this process occasionally led to inconsistencies between the behavior as modelled by the state machine and the behavior of the developed executable code.

At first, the Boost C++ State Chart library was used to implement state machines. While this library enabled the modelling of simple state machines, it remained difficult to model the behavior of complex components. To simplify this workflow, two languages were developed. The first language that was developed at Canon Production Printing to simplify the modelling of software components using state machines is the Open Interaction Language (OIL). OIL was developed with a focus on creating a language with high expressive power that is formally well-defined. An advantage of having a formally well-defined language is that this enables model checking. This enables the engineer to check whether properties hold on their OIL specifications using a model checker. The engineers at Canon Production Printing developed the SCM C++ library that, in contrast to OIL, was not designed with a focus on creating a formally well-defined language. Instead, the aim of the SCM library was to create a library that closely resembles the existing state machines from the Boost C++ library but with some of the additional expressive power of OIL. To simplify the modelling of software components using the SCM C++ library, the State Machine Modelling Tool (SMMT) was developed using JetBrains MPS [21]. Using the State Machine Modelling Tool, state machines can be modelled both textually and graphically. Furthermore, the tool enables the generation of executable C++ and C# code from the models. The generated C++ and C# code makes use of the SCM library to implement the behavior of the state machines in that language.

2.2 SMMT Constructs

In this section, we introduce the constructs of the SMMT language. In SMMT, each specification represents a single state machine. An SMMT specification consists of five sections, as shown in figure 2.1, these are: a *name*, a *namespace*, a collection of *OnEvents*, a collection of *DoEvents* and a *region*.

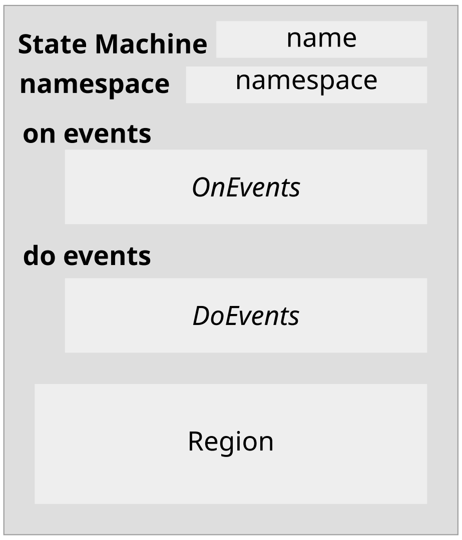
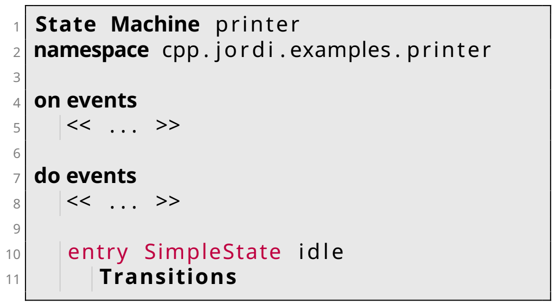


Figure 2.1: Overview of an SMMT Specification

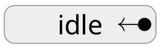
SMMT has been implemented in JetBrains MPS [21] in which two representations have been developed that can be used by the engineer to model the behavior of a component as a state machine: a textual and a graphical representation. In all our examples we show both representations. In the graphical representation of SMMT, the section of the specification consisting of the name, namespace and lists of *OnEvents* and *DoEvents* is also shown. However, in our examples, we omit these details in the graphical representation to avoid repeating information.

In the remainder of this section, we introduce each section of an SMMT specification. To explain the constructs of the SMMT language, we gradually build an SMMT specification modelling the behavior of a simplistic printer. In Figure 2.2, an SMMT specification is given consisting of a single entry state called `idle`. The specification does not have any events or transitions.



```
1  State Machine printer
2  namespace cpp.jordi.examples.printer
3
4  on events
5     << ... >>
6
7  do events
8     << ... >>
9
10    entry SimpleState idle
11       Transitions
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.2: An SMMT specification representing a printer that is always idle

In the representations of an SMMT specification, ≪ ... ≫ denotes an empty field in which a component can be inserted. For example, this enables us to define *OnEvents* and *DoEvents* in lines 5 and 8 for the SMMT specification in Figure 2.2a respectively.

2.2.1 Name and Namespace

As shown in Figure 2.1, each SMMT specification has a name and a namespace. The namespace is the fully qualified name of the SMMT specification,. That is, an SMMT specification can be identified by its namespace. The name is of type `String`, the namespace consists of one or more `Strings` separated by dots. Here, a `String` is defined as a value that corresponds to the regular expression `[a-z]([a-z0-9_])*`.

In the running example (Figure 2.2) the name and namespace of the SMMT specification are defined on lines 1 and 2 of the textual representation respectively. That is, the name of the SMMT specification in the running example is `printer` and the namespace is `cpp.jordi.examples.printer`. Note that, the name and namespace do not depend on each other. That is, the name of the SMMT specification is not required to be contained in the namespace of the SMMT specification.

2.2.2 Events

SMMT distinguishes two types of events, namely *OnEvents* and *DoEvents*. *OnEvents* are events that are produced externally and may cause transitions of the SMMT specification to fire that are defined for that *OnEvent*. On the other hand, *DoEvents* are actions produced by the SMMT specification that serve as a response to processed *OnEvent* actions. Both *OnEvents* and *DoEvents* can be parametrized, where each parameter is of boolean type or of a custom specified type. In both the graphical and textual representation of SMMT, the type of a parameter is only shown if the parameter is of custom type. Hence, if the type of a parameter is not shown when the parameter of an *OnEvent* or a *DoEvent* is declared, then this parameter is of boolean type.

As shown in Figure 2.1, each SMMT specification has a section in which the *OnEvents* are defined that can be processed by the state machine of that SMMT specification. Similarly, the SMMT specification contains a section in which the *DoEvents* are defined that can be produced by the state machine of the SMMT specification.

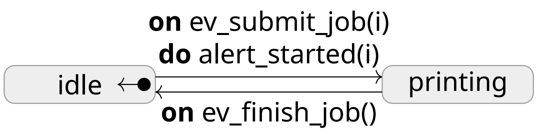
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
  ev_submit_job(i : int)
  ev_finish_job(<< ... >>)

do events
  alert_started(i : int)

  entry SimpleState idle
    Transitions
      on ev_submit_job(i)
        do alert_started(i)
        go printing
  SimpleState printing
    Transitions
      on ev_finish_job()
        go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.3: An SMMT specification of a printer that is idle or printing

We expand our running example with a state called `printing` and add *OnEvents* `ev_submit_job(i : int)` and `ev_finish_job()` that signal that the printer must start printing job `i` and that the printer must finish the job respectively. Furthermore, we add *DoEvent* `alert_started(i : int)` that alerts the user that the printer has started printing job `i`. *OnEvent* `ev_submit_job(i : int)` and *DoEvent* `alert_started(i : int)` have a parameter of custom type `int` called `i` which indicates the page of the job that must be printed. The extended running example is shown in Figure 2.3.

2.2.3 Region

The *region* of an SMMT specification is hierarchically structured and consists of one or more states. Each state of the SMMT specification consists of several outgoing transitions, entry handlers, exit handlers and can contain a nested region. In the remainder of this section, we discuss the types of states, the transitions, entry- and exit handlers informally.

2.2.3.1 State Types

The SMMT language supports four different types of states: *SimpleStates*, *CompositeStates*, *ParallelStates* and *JointStates*. The *execution state* of an SMMT specification consists of a subset of the states of the SMMT specification. We say that a state is an *active* state if, and only if, the state is an element of the execution state. We introduce the notion of an *entry* state. An *entry* state is a state that becomes active when the parent of that state is initiated. Each state, except for states of type *JointState*, can be an *entry state*. In the graphical representation of an SMMT specification, an entry state is indicated by the entry symbol (⇠●).

To distinguish between the type of states in the SMMT specifications that are graphically depicted in the remainder of this report, we added a red prefix S, C, P or J in front of the name of each state. These red prefixes indicate the type of the state: *SimpleState*, *CompositeState*, *ParallelState* and *JointState* respectively. In the remainder of this section, we introduce each of the four different state types.

SimpleState States of type *SimpleState* represent states that have no children. In the running example (Figure 2.3), both states `idle` and `printing` are of type *SimpleState*.

CompositeState The *CompositeState* type allows for children that further specify the behavior of the *CompositeState*. A *CompositeState* has one or more children of which exactly one is an entry state. Furthermore, if a *CompositeState* s is active, then exactly one child of *CompositeState* s must be active.

In our running example, a *CompositeState* can be used to further specify the behavior of the `printing` state. For example, we might need to apply color correction on the submitted job prior to printing the job. In Figure 2.4 we extend our running example of Figure 2.3 with states `color_correction` and `printing_job`. Furthermore, we add a transition from state `color_correction` to state `printing_job` for *OnEvent* `ev_print_job()` to ensure that the printer can only start printing the job whenever *OnEvent* `ev_print_job()` has been processed. In this example, state `printing` is a *CompositeState* and states `idle`, `color_correction` and `printing_job` are of type *SimpleState*.

ParallelState States of type *ParallelState* have one or more children that further specify the behavior of the *ParallelState*. In contrast to *CompositeStates*, all children of a *ParallelState* are active as long as the *ParallelState* itself is active. Furthermore, all children of a *ParallelState* are entry states, except for the children of the *ParallelState* that are of type *JointState*.

In the running example of Figure 2.4, a printer is shown that first applies color correction to a job after which the job gets printed. Using a *ParallelState* we can model that the printer applies color correction and scales the job in parallel, after which the job gets printed. In Figure 2.5, we extend the running example with *ParallelState* `preparing_job` that allows the color correction and scaling of the job to be done simultaneously.

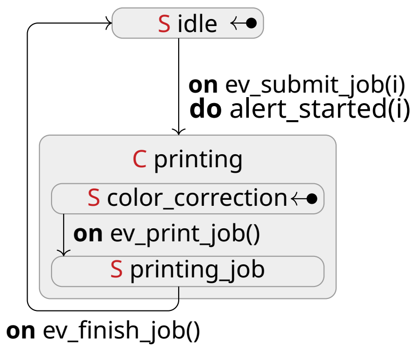
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_submit_job(i : int)
    ev_finish_job(<< ... >>)
    ev_print_job(<< ... >>)

do events
    alert_started(i : int)

    entry SimpleState idle
        Transitions
            on ev_submit_job(i)
                do alert_started(i)
                go printing
    CompositeState printing
        Transitions
        entry SimpleState color_correction
            Transitions
                on ev_print_job()
                    go printing_job
        SimpleState printing_job
            Transitions
                on ev_finish_job()
                    go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.4: An SMMT specification consisting of a printer that applies color correction before printing a job

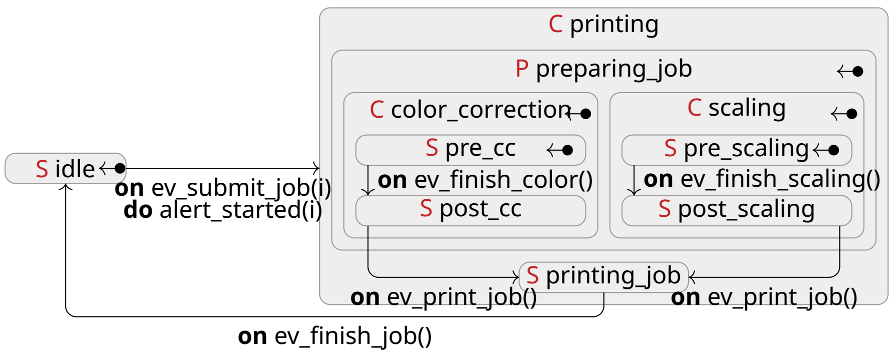
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_submit_job(i : int)
    ev_finish_job(<< ... >>)
    ev_finish_color(<< ... >>)
    ev_finish_scaling(<< ... >>)
    ev_print_job(<< ... >>)

do events
    alert_started(i : int)

    entry SimpleState idle
        Transitions
            on ev_submit_job(i)
                do alert_started(i)
                go printing
    CompositeState printing
        Transitions
        entry ParallelState preparing_job
            Transitions
            entry CompositeState color_correction
                Transitions
                entry SimpleState pre_cc
                    Transitions
                        on ev_finish_color()
                            go post_cc
                SimpleState post_cc
                    Transitions
                        on ev_print_job()
                            go printing_job
            entry CompositeState scaling
                Transitions
                entry SimpleState pre_scaling
                    Transitions
                        on ev_finish_scaling()
                            go post_scaling
                SimpleState post_scaling
                    Transitions
                        on ev_print_job()
                            go printing_job
        SimpleState printing_job
            Transitions
                on ev_finish_job()
                    go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.5: An SMMT specification consisting of a printer that applies color correction and/or scales the job before printing the print job

JointState States of type *JointState* are used to join the behavior of one or more nested states of a *ParallelState*. A *JointState* refers to several states but does not have any children itself and can only occur inside a *ParallelState*. Furthermore, a *JointState* may only refer to nested states or *JointStates* of the *ParallelState* in which the *JointState* occurs that can be active at the same point in time. A *JointState* is active when all states that the *JointState* refers to are active.

Consider the example in Figure 2.5. When the printer reaches the `preparing_job` state, the printer starts to scale the document and apply color correction. However, in this example, as soon as either the color correction or scaling is completed, we are allowed to start printing the job when *OnEvent* `ev_print_job()` is processed. Suppose we want that both preparation steps are completed before the printer starts printing the job. To achieve this, a *JointState* can be used that refers to both states `post_cc` and `post_scaling` with an outgoing transition defined for *OnEvent* `ev_print_job()`. This only allows the printer to start printing if both the color correction process and scaling has been completed. In Figure 2.6, we extend the running example of Figure 2.5 with *JointState* `joint_scaling_cc` that ensures that the printer only starts printing after both preparation steps are completed.

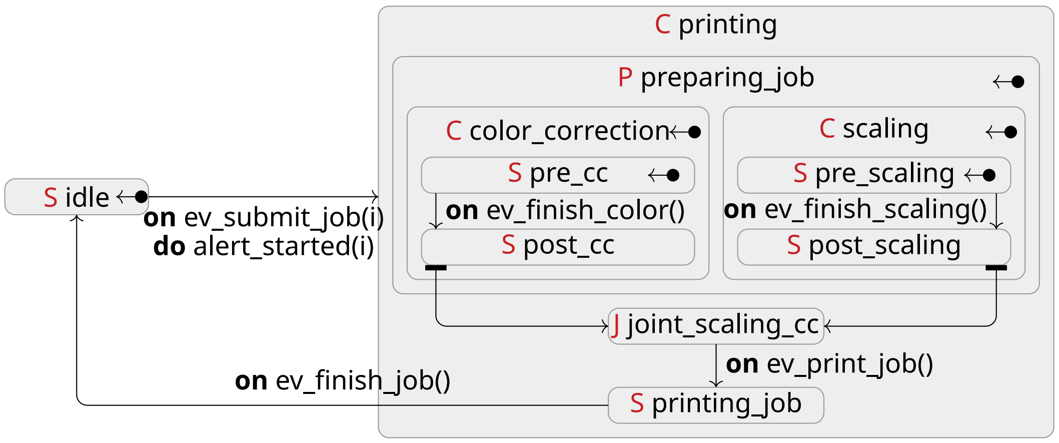
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_submit_job(i : int)
    ev_finish_job(<< ... >>)
    ev_finish_color(<< ... >>)
    ev_finish_scaling(<< ... >>)
    ev_print_job(<< ... >>)

do events
    alert_started(i : int)

    entry SimpleState idle
        Transitions
            on ev_submit_job(i)
                do alert_started(i)
                go printing
    CompositeState printing
        Transitions
        entry ParallelState preparing_job
            Transitions
            entry CompositeState color_correction
                Transitions
                entry SimpleState pre_cc
                    Transitions
                        on ev_finish_color()
                            go post_cc
                SimpleState post_cc
                    Transitions
            entry CompositeState scaling
                Transitions
                entry SimpleState pre_scaling
                    Transitions
                        on ev_finish_scaling()
                            go post_scaling
                SimpleState post_scaling
                    Transitions
            JointState joint_scaling_cc
                Transitions
                    on ev_print_job()
                        go printing_job
                Joins post_cc
                      post_scaling
        SimpleState printing_job
            Transitions
                on ev_finish_job()
                    go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.6: An SMMT specification consisting of a printer that applies color correction and scales the job before printing the job

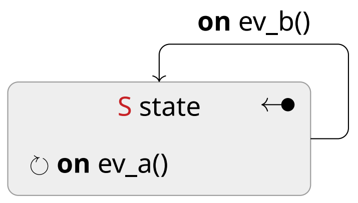
2.2.3.2 Transitions

Zero or more outgoing transitions can be defined for each state of an SMMT specification. Each outgoing transition consists of an *OnEvent* for which the transition is defined and an optional target state. In case no target state is specified, the model interprets the transition as an internal transition, as indicated by the `internal` keyword. Furthermore, a guard and a list of *BehavioralActions* may be defined for each outgoing transition. A *BehavioralAction* is either a *DoEvent* that is produced as a response to the processed *OnEvent*, a *SelfPost* action to repost the *OnEvent* that was processed or a *Forward* action to allow the *OnEvent* to be handled by the ancestors of the state for which the *Forward* action is defined.

The difference between the representations of internal and external transitions are shown in Figure 2.7. In this example, the transition defined on lines 13 and 14 of the textual representation is an internal transition. The transition defined on lines 15 and 16 is an external transition. In the graphical representation of the SMMT specification, internal transitions are denoted using the self-loop icon ↻ and are shown within the state for which the transition is defined. External transitions are shown with an arrow that exits and enters the state. When an external transition fires, the exit- and entry-handlers of the state are triggered if the transition is fired. Internal transitions do not trigger the exit- and entry-handlers of the state when the internal transition fires.

```
 1  State Machine printer
 2  namespace cpp.jordi.examples.printer
 3
 4  on events
 5     ev_a(<< ... >>)
 6     ev_b(<< ... >>)
 7
 8  do events
 9     << ... >>
10
11     entry SimpleState state
12        Transitions
13           on ev_a()
14              go (internal)
15           on ev_b()
16              go state
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.7: Representations of Internal and External Transitions

An outgoing transition of a state s that is defined for an *OnEvent* e fires if, and only if, state s is active when *OnEvent* e is processed and the guard of this transition evaluates to true. Furthermore, a transition from state s cannot fire if there exists a nested state of state s that has a transition that can fire that is defined for *OnEvent* e. When a transition fires, all *BehavioralActions* that are defined for this transition are produced. In Chapter 5 we formally define which transitions fire when an *OnEvent* is processed.

The guard of a transition is a boolean expression that may consist of the standard logical connectives (and, or and not), boolean literals (true and false) and references to the parameters of the *OnEvent* for which the transition is defined. SMMT allows references to custom typed parameters of the *OnEvent* for which the transition is defined in the guard. However, the generated C++ and C# code that is generated from the SMMT specification requires that the reference is of boolean type. No operators have been implemented in SMMT to compare the values of custom typed parameters. Hence, we cannot compare custom typed parameters and therefore references to custom type variables may cause errors in the generated C++ and C# code.

2.2.3.3 Entry- and Exit-Handlers

For each state of an SMMT specification, zero or more entry- and exit-handlers can be defined. Entry- and exit-handlers model behavior that is performed when states are entered and exited respectively. In the following paragraphs we introduce the entry and the exit handlers and give an example of both.

Entry Handlers An entry handler specifies the actions that need to be performed when the state for which the entry handler is defined is entered. SMMT consists of two types of entry handlers: conditional entry handlers and otherwise entry handlers. Conditional entry handlers are entry handlers that are defined for an *OnEvent* e and state s that are triggered if, and only if, state s is entered by a transition defined for *OnEvent* e. An otherwise entry handlers that is defined for a state s fires if, and only if, state s is entered and no conditional entry handlers are triggered when state s is entered.

Both type of entry handlers consist of zero or more *DoEvent* and *Forward BehavioralActions* that are produced when the entry handler is triggered. Each entry handler can have a target state to which a transition is instantiated when the handler is triggered. Additionally, conditional entry handlers can consist of a *Forward BehavioralAction*. Finally, a guard can be defined for a conditional entry handler that restricts when the entry handler is triggered.

In Figure 2.8 an SMMT specification is shown that contains entry handlers. State `state_b` contains both a conditional and otherwise entry handler. When state `state_a` is active and *OnEvent* `ev_a()` is processed, *DoEvent* `re_a()` is produced on entry of state `state_b`. Furthermore, if state `state_a` is active and *OnEvent* `ev_b()` is processed then *DoEvent* `re_b()` is produced on entry of state `state_b`, as there are no conditional entry handlers that are triggered by *OnEvent* `ev_b()`.

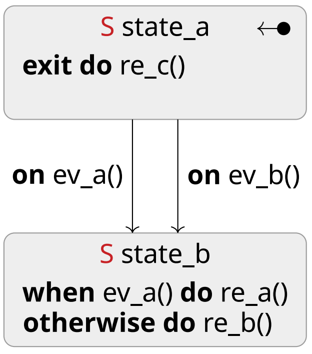
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_a(<< ... >>)
    ev_b(<< ... >>)

do events
    re_a(<< ... >>)
    re_b(<< ... >>)
    re_c(<< ... >>)

    entry SimpleState state_a
        Exit Handlers
            exit do re_c()
        Transitions
            on ev_a()
                go state_b
            on ev_b()
                go state_b
    SimpleState state_b
        Entry Handlers
            when ev_a()
                do re_a()
            otherwise
                do re_b()
        Transitions
```

(a) Textual Representation



(b) Graphical Representation

Figure 2.8: An SMMT Specification consisting of Entry and Exit Handlers

Exit Handlers An exit handler specifies the actions that are performed when the state for which the exit handler is defined is exited. Each exit handler consists of zero or more *DoEvents* and *SelfPost BehavioralActions* that are produced when the exit handler is triggered. In contrast to entry handlers, exit handlers cannot be defined for a specific *OnEvent*, nor can they have guards, *Forward BehavioralActions* or a target state.

State `state_a` of the SMMT specification shown in Figure 2.8 contains an exit handler. When state `state_a` is active and either *OnEvent* `ev_a()` or `ev_b()` is processed, *DoEvent* `re_c()` is produced when the state is exited.

2.3 Execution

From the models that are designed using the State Machine Modelling Tool, executable C++ and C# code can be generated. The generated code makes use of the SCM library. In SMMT, the transitions of an SMMT specification are triggered when *OnEvents* are processed. *OnEvents* are handled asynchronously using an event queue. A First-In-First-Out (FIFO) policy is used to dispatch the *OnEvents* in order of arrival. When an *OnEvent* occurs, the *OnEvent* is added to the end of the event queue. As soon as the first *OnEvent* is added to the queue, the *OnEvent* is processed by the SMMT specification.

When an *OnEvent* is processed, the replies are temporarily queued. Replies are the *BehavioralActions* that are produced in response to the received *OnEvents*. If the *OnEvent* cannot be handled by the SMMT specification, an internal software exception is thrown. Whenever an internal software exception is thrown, the execution of the SMMT specification is terminated. For all transitions that are fired due to the processed *OnEvent*, the respective exit handlers are triggered. Next, all queued replies are handled and the transition to the target states take place. For all states that were entered, the entry handlers are triggered and the entered states are initiated. This procedure repeats as long as there are *OnEvents* in the queue.

The execution state after an *OnEvent* has been processed is obtained by removing all states that cannot be active simultaneously with the target state of the transitions that fire. Next, all target states and ancestors of each target state of the transitions that fire are added to the execution state. Finally, the obtained execution state is initiated.

Chapter 3

Analysis of SMMT

In this section we discuss our analysis on the State Machine Modelling Tool. We spoke with several engineers at Canon Production Printing that have used SMMT to model software components, as well as the engineer that developed the State Chart Modelling (SCM) library. In Section 3.1 we discuss the observations and experiences of the engineers at Canon Production Printing regarding the State Machine Modelling Tool. At Canon Production Printing, at the time of writing, the behavior of 26 software components have been modelled using SMMT. We discuss our analysis of the existing SMMT specifications in Section 3.2.

3.1 Observations and Experiences of Engineers

As mentioned before, we spoke with several engineers at Canon Production Printing that have used the State Machine Modelling Tool to model the behavior of software components. In this section we discuss the observations and experiences of the engineers at Canon Production Printing with SMMT that were shared by these engineers through interviews. We first discuss the insights that we gained through these meetings that are related to the language of the State Machine Modelling Tool. Next, we discuss the observations and experiences of the engineers regarding the implementation of SMMT in JetBrains MPS and the usage thereof. Finally, we briefly discuss the testing library of the SCM library that can be used to test the generated SCM code.

3.1.1 Language of the State Machine Modelling Tool

The engineers that used the State Machine Modelling Tool unanimously agree that the tool greatly helps with the development of software components. Prior to the introduction of SMMT, the engineers were required to model the behavior of the software components directly using the State Chart Modelling (SCM) library or using the Boost C++ library [4, 5]. The State Chart Modelling library is a very powerful library that could directly be used by the engineers to model the behavior of software components. The absence of restrictions on the coding patterns and techniques that should be used by these engineers leads to a lack of uniformity between the developed executable code of different projects. The State Machine Modelling Tool ensures that the coding style and patterns of the code that are generated for each project are uniform. The State Machine Modelling Tool restricts the patterns that can be used to model the behavior of a software component. As a consequence, the executable code that is generated by the code generator of SMMT is uniform.

Most of the constructs of the SMMT language are deemed intuitive by the engineers that used the State Machine Modelling Tool. However, most of the engineers that were interviewed were unfamiliar with constructs like the *SelfPost* and *Forward BehavioralActions*.

The documentation only contains an explanation for the most frequently used constructs of the SMMT language. Constructs like the *SelfPost* and *Forward BehavioralActions* that are not understood by most engineers are either not explained in the documentation or are briefly discussed without any details.

In SMMT specifications with parallel behavior, there are usually multiple *JointStates* that perform the same behavior when executed but refer to different states. The number of *JointStates* grows quickly with the number of nested states of each *ParallelState*. As these *ParallelStates* tend to clutter the graphical representation of the SMMT specification, the engineers would prefer if *JointStates* that perform the same actions when executed could be merged using a join clause. This join clause would consist of the disjunction of the joined states of each *JointState*.

3.1.2 JetBrains MPS

The State Machine Modelling Tool has been implemented in the JetBrains MPS language workbench. JetBrains MPS is a language workbench that uses projectional editing. MPS projects the abstract syntax tree of an SMMT specification into a textual or graphical representation of the SMMT specification that can be edited by the engineer. The modifications that are made in the projections of the SMMT specification are direct modifications in the abstract syntax tree of that specification. In this section, we discuss the experiences and observations of the engineers regarding the usage of SMMT in JetBrains MPS.

Learning Curve In contrast to a textual editor, the engineer is not able to freely type code in the projections of the SMMT specification. Instead, engineers need to insert the constructs of SMMT through one of the projections. As most software engineers are familiar with textual editors, most engineers require some time to get familiar with projectional editing.

Without prior knowledge on how the engineer is supposed to work with MPS, editing an SMMT specification could be problematic, as inserting or editing constructs of an SMMT specification may require several key-binds to be used. For example, to add a transition from a state s to state t that fires for *OnEvent* e and produces *DoEvent* d we would need to perform the following steps:

1. Hover over the state s, select the transition icon and draw a transition to state t;
2. Select the transition and hit key-bind `ctrl+space` to select *OnEvent* e;
3. Select the transition and hit key-bind `alt+enter` to open the intentions menu;
4. Select "add action" and hit key-bind `ctrl+space` to select *DoEvent* d.

As shown by these four steps to add a single transition, the engineer must know various key-binds to edit an SMMT specification. This is one of the reasons why engineers that are inexperienced with MPS experience a steep learning curve when editing or creating SMMT specifications in MPS. As the interfaces of MPS are intuitive and the number of different key-binds that are used is limited, the engineers tend to familiarise themselves with MPS within several hours.

Project Setup and Integration To set up a new SMMT project, the engineers are required to go through a series of steps that are explained in the documentation of SMMT. There is no option inside MPS to directly set up a new SMMT MPS project. The process to set up a new project requires the copying and editing of various properties and build files. Due to the lack of an easy and quick way to set up a new SMMT project in MPS, engineers tend to avoid the use of the State Machine Modelling Tool when modelling simple software components, as the overhead of setting up a project would exceed the additional time it takes to manually implement the code using the SCM library.

The integration of the generated executable C++ and C# code is not documented in the documentation of the State Machine Modelling Tool. Neither does the documentation contain details on how the engineer is supposed to include the SCM library in the file, about how the state machine of the SMMT specification should be initiated or how the custom types and reply class should be defined.

Projections of the SMMT Specification The engineers unanimously agree that the graphical representation of SMMT is a great addition to the tool, especially when modelling the behavior of simple software component. However, the graphical representation is not powerful enough to get a clear overview of more complicated SMMT specifications.

The engineers that are familiar with the State Machine Modelling Tool observe that the behavior that is modelled by an SMMT specification is not always understood correctly by engineers that are less familiar with SMMT, especially for SMMT specifications that include parallel behavior. The engineers would like a projection in SMMT that would allow the engineers to model the parallel behavior in a less cluttered and more comprehensible manner. Such a projection would be beneficial for the engineers to understand SMMT specifications that include parallel behavior. Developing such a representation is outside the scope of this graduation project. Alternatively, a simulation tool to simulate the behavior of the SMMT specification would help the engineers to understand the behavior of an SMMT specification.

The engineers observe that modifications to the projections of SMMT specifications are processed slowly. For example, when moving a state of an SMMT specification it may take a couple of seconds before the modification is saved and rendered. Furthermore, some modifications may break the SMMT specifications. The engineers observed that SMMT specifications broke when dragging states and transitions in the graphical representation of the SMMT specification. Furthermore, undoing and redoing actions may cause the SMMT specification to break.

Due to the problems that arise when editing an SMMT specification in JetBrains MPS, engineers generally avoid SMMT when they model a simple software component that can easily be modelled directly using the SCM library. Furthermore, rather than directly using the State Machine Modelling Tool to model the behavior of software components, the state machine is in practice first modelled on paper or on a whiteboard after which it is digitalised using software like Draw.io, PlantUML or Enterprise Architect. When all feedback on the digitalised version of the state machine has been processed, the state machine is modelled using the State Machine Modelling Tool. A graphical overview of the modelling process that is used in practice is shown in Figure 3.1.

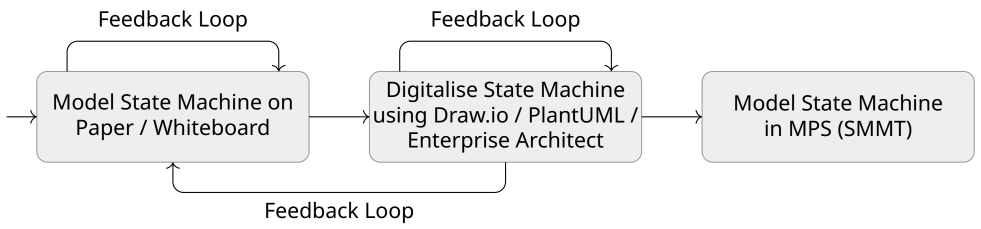

Feedback Loop
Feedback Loop
Model State Machine on Paper / Whiteboard
Digitalise State Machine using Draw.io / PlantUML / Enterprise Architect
Model State Machine in MPS (SMMT)
Feedback Loop

Figure 3.1: Graphical Overview of the Modelling Process

Optimally, the behavior of the software component would directly be modelled in an SMMT MPS project as this would be less time-consuming and less error-prone. In the current workflow, the state machine is modelled two times, both using external drawing software and in an SMMT MPS project. The external drawing software is considered to be more flexible than the graphical representation of MPS. Hence, the state machine is only modelled in an SMMT MPS project after the engineers have finalised the design of the state machine.

3.1.3 SCM Test Library

The SCM library that is used by SMMT to generate executable C++ and C# code has a testing library that can be used to perform tests on the generated code. Using the SCM Test library, the engineers model the behavior of the state machine of the SMMT specification that they want to test as a separate state machine. To test if the behavior of the state machine that is tested is correct, the output channel of the test state machine and the input channel of the main state machine are coupled and vice versa. The test state machine sends *OnEvents* to the main state machine and listens to the *DoEvents* that are produced by the main state machine. In this way, the test state machine can verify whether the correct responses are received based on the produced *OnEvents*.

The engineers that were interviewed and were familiar with the SCM Testing library mentioned that the library is powerful to test whether the modelled behavior in the SMMT specification corresponds to what they expect. However, the engineers mention that there are limitations on the tests that can be run with the testing library.

Unfortunately, the SMMT tooling does not allow the engineers to generate tests automatically using the SCM test library. Therefore, the engineers must create these tests manually, which is a time-consuming process in which errors are easily made.

3.2 Analysis of Existing Models

At Canon Production Printing the State Machine Modelling Tool has been used to model the behavior of 26 software components. Table 3.2 shows an analysis on the structure of the SMMT specifications. The table shows for each construct how frequently it occurs in each SMMT specification. The names of the models have been replaced by the letters A to Z.

From the analysis of the structure of the existing SMMT specifications, we see that only 8 out of the 26 existing SMMT specifications include parallelism. That is, only specifications J, K, L, N, T, U, V and Z contain states of type *ParallelState*. As *JointStates* can only occur as a nested state of a *ParallelState*, there are also only 8 out of the 26 existing SMMT specifications that contain states of type *JointState*.

An important insight is that the *Forward* and *SelfPost BehavioralActions* do not occur in any of the 26 existing SMMT specifications. As mentioned before, the users of the SMMT tool that were interviewed were unfamiliar with the *Forward* and *SelfPost BehavioralActions*, which might explain why they are not used in the existing SMMT specifications.

The entry- and exit handlers only occur in the SMMT specifications that contain *ParallelStates*, except for SMMT specification Y. Out of the 26 existing SMMT specifications, there are only 3 specifications that contain conditional entry handlers. Furthermore, there exist only two SMMT specifications that contain exit handlers. The unfamiliarity of the software engineers with the entry- and exit handlers may explain why they are rarely used in practice.

3.3 Conclusion

The lack of documentation on the setup, the integration and the constructs of the State Machine Modelling Tool combined with the steep learning curve of JetBrains MPS has been found problematic for engineers that are unfamiliar with SMMT and MPS. However, engineers tend to get familiar with MPS rather quickly, therefore we leave this out of the scope for this graduation project. The lack of documentation on the language of SMMT is a more severe problem. The engineers that are responsible for maintaining SMMT should improve and extend the documentation that is currently available. The formal definition that is defined in Chapter 5 could be used as a basis to improve and extend the documentation of each of the constructs of SMMT.

SMMT Specification	SimpleStates	CompositeStates	ParallelStates	JointStates	Total States	OnEvents	DoEvents	BooleanParameters	CustomTypeParameters	Transitions	DoEventActions	ForwardActions	SelfPostActions	Guards	Conditional Entry Handlers	Otherwise Entry Handlers	Exit Handlers
A	48	8	0	0	56	51	80	2	35	129	149	0	0	8	0	0	0
B	16	2	0	0	18	22	17	2	0	28	24	0	0	2	0	0	0
C	17	3	0	0	20	21	18	1	0	56	45	0	0	0	0	0	0
D	20	4	0	0	24	21	21	1	0	67	50	0	0	0	0	0	0
E	14	2	0	0	16	19	18	1	6	30	33	0	0	2	0	0	0
F	13	2	0	0	15	18	18	1	6	27	28	0	0	2	0	0	0
G	9	1	0	0	10	12	9	1	0	18	13	0	0	0	0	0	0
H	9	1	0	0	10	11	8	1	0	18	11	0	0	0	0	0	0
I	6	0	0	0	6	6	4	1	0	7	6	0	0	0	0	0	0
J	32	7	1	5	45	38	38	8	2	67	63	0	0	18	0	8	0
K	26	6	1	14	47	35	35	3	2	59	63	0	0	8	0	14	0
L	49	10	1	23	83	55	41	8	8	133	100	0	0	16	0	9	0
M	12	3	0	0	15	16	13	4	0	31	30	0	0	8	0	0	0
N	14	3	1	13	31	8	12	0	0	15	19	0	0	0	3	18	0
O	14	1	0	0	15	20	15	2	0	25	22	0	0	2	0	0	0
P	12	0	0	0	12	10	11	0	0	14	14	0	0	0	0	0	0
Q	4	0	0	0	4	5	4	4	0	20	6	0	0	6	0	0	0
R	2	0	0	0	2	4	1	1	0	8	4	0	0	0	0	0	0
S	9	1	0	0	10	8	5	3	0	32	19	0	0	10	0	0	0
T	17	9	2	3	31	15	8	3	0	34	12	0	0	2	0	6	1
U	27	13	1	9	50	22	14	7	0	80	24	0	0	20	0	18	0
V	22	6	1	9	38	13	15	2	16	40	16	0	0	6	2	19	2
W	30	1	0	0	31	24	39	0	0	72	68	0	0	0	0	0	0
X	5	0	0	0	5	4	4	1	0	18	5	0	0	2	0	0	0
Y	5	1	0	0	6	7	7	1	17	13	12	0	0	0	1	0	0
Z	14	3	1	13	31	8	8	0	0	15	12	0	0	0	0	18	0
$\sum$	446	87	9	89	631	473	463	58	92	1056	848	0	0	112	6	110	3

Figure 3.2: Structural Analysis on the existing SMMT Specifications

Based on the analysis of the existing SMMT specifications, we define the following order in which the constructs of SMMT are formalised: *SimpleStates*, *CompositeStates*, Transitions (without guards and *BehavioralActions*), *ParallelStates*, *JointStates*, Transitions (with guards and *DoEvents*), Otherwise Entry Handlers, Conditional Entry Handlers and Exit Handlers. Due to a lack of time, the *SelfPost BehavioralActions* and *Forward BehavioralActions* are not supported by the translation that we have defined.

The projections of the SMMT Specifications that exist in the State Machine Modelling Tool are sufficient for visualizing simple SMMT specifications but are not powerful enough to visualize the parallel behavior of SMMT specifications. A visualization that could visualize the parallel behavior of an SMMT specification would help the engineers to understand the SMMT specification. Alternatively, a simulation tool could help as well in understanding an SMMT specification. Using the mCRL2 language and toolset, LTSs can be generated that visualise the behavior of an SMMT specification. Furthermore, the toolset contains a tool that can be used to simulate the behavior of the mCRL2 specification. Unfortunately, this does not allow the engineer to simulate the SMMT specification in MPS directly. Having a simulation tool within MPS that would simulate the SMMT specification using for example the graphical representation of the SMMT specification would be more beneficial for the engineers. Nevertheless, the views that can be generated using the mCRL2 toolset and the simulation tool of the mCRL2 toolset can be useful for the engineers to debug and analyse the behavior of the SMMT specification that they modelled.

The SCM library that is used by the generated executable code consists of a testing library that can be used to perform tests on the generated code. However, the test cases that can be defined using this testing library are limited. This shows why expanding the State Machine Modelling Tool with a translation to mCRL2 specifications is beneficial, as the mCRL2 toolset allows us to verify properties on the mCRL2 specification that cannot be verified using the testing library of the SCM library. Furthermore, as the tests that are defined using the SCM testing library cannot be generated from SMMT and thus must be defined manually, testing is a time-consuming and error-prone task. Using an automated translation to mCRL2 specification, these properties can automatically be tested on the mCRL2 specification. A disadvantage of performing tests via mCRL2 specifications is that this requires the engineers to know how properties can be expressed as a modal mu-calculus formula.

Chapter 4

Mathematical Preliminaries

In Chapter 5 we formally define the SMMT language. To define the syntax and semantics of SMMT specifications, we need to introduce some definitions. We first define the notation used for lists and define the *unique existential quantifier*. We define a *child relation* and define the notion of a *subregion*. Finally, we define *labelled transition systems* (LTSs) that are used to define the semantics of an SMMT specification.

First we define some notation that is used in the remainder of this report to reason about lists. The length of a list $\mathbb{L}$ is denoted by $|\mathbb{L}|$. We write $\mathbb{L}[i]$ to denote the value in list $\mathbb{L}$ at index i, where $0 \leq i < |\mathbb{L}|$. The first element in a list has index 0. We write $x \in \mathbb{L}$ if there exists an index i, $0 \leq i < |\mathbb{L}|$, such that $\mathbb{L}[i] = x$.

We define the *unique existential quantifier* to express whether there exists exactly one element in the domain that satisfies a predicate.

Definition 1 (Unique Existential Quantifier)**.** Let X be a set of variables and let $P(x)$ be a predicate over a variable $x \in X$. The *unique existential quantifier*, denoted by $\exists!$, is defined as follows:

$$(\exists!_{x \,\in\, X} : P(x)) \Leftrightarrow (\exists_{x \,\in\, X} : (P(x) \land (\neg\exists_{x' \,\in\, X} : P(x') \land x \neq x')))$$

We define a *child relation* that defines how the states in a hierarchically structured setting are related. This relation allows us to reason about the *children*, *parent*, *descendants* and *ancestors* of a state in a set of states.

Definition 2 (Child Relation)**.** We define a *child relation* $\sqsubset \subseteq S \times S$ as a relation over some set of states S. For any $s, s' \in S$, we say that s is a *child* of s' and s' is a *parent* of s if, and only if, $s \sqsubset s'$. Furthermore, we define the *descendant relation* $\sqsubset^+$ as the transitive closure of child relation $\sqsubset$. We say that s is a *descendant* of s' and s' is an *ancestor* of s if, and only if, $s \sqsubset^+ s'$. We define $\sqsubset^*$ as the reflexive transitive closure of child relation $\sqsubset$. Constraints 1 and 2 must hold on child relation $\sqsubset$:

Constraint 1: A state has at most 1 parent, formally expressed as:

$$\forall_{s,\, s',\, s'' \,\in\, S} : (s \sqsubset s' \land s \sqsubset s'') \Rightarrow (s' = s'')$$

Constraint 2: For any state $s \in S$ it holds that s cannot be an ancestor of itself, that is:

$$\forall_{s \,\in\, S} : s \not\sqsubset^+ s$$

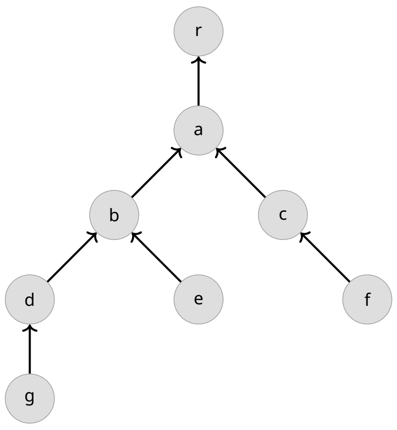


Figure 4.1: Child Relation $\sqsubset$

Example 1 (Child Relation)**.** Let $S = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathtt{f}, \mathsf{g}, \mathtt{r}\}$ be a set of states and let

$$\sqsubset = \{(\mathsf{a}, \mathtt{r}), (\mathsf{b}, \mathsf{a}), (\mathsf{c}, \mathsf{a}), (\mathsf{d}, \mathsf{b}), (\mathsf{e}, \mathsf{b}), (\mathtt{f}, \mathsf{c}), (\mathsf{g}, \mathsf{d})\}$$

be the child relation defined over S. Child relation $\sqsubset$ has been visualized in Figure 4.1. We have that state b has children d and e and descendants d, e and g. Furthermore, state b has parent a and ancestors a and r.

Using the definition of a child relation over some set of states S, we can define the notion of a *subregion* of a state in $s \in S$. We define a *subregion* of a state $s \in S$ as a set consisting of state s and all descendants of s.

Definition 3 (Subregion)**.** Let $\sqsubset$ be a child relation over some set of states S. The *subregion* of a state $s' \in S$ is defined as the set consisting of state s' and all descendants of s'. We define the *subregion* of a state $s' \in S$ using function $\mathcal{SR}(S, s')$, which is defined as follows:

$$\mathcal{SR}(S, s') = \{s \in S \mid s \sqsubset^* s'\}$$

Example 2 (Subregion)**.** Let S be the set of states and $\sqsubset$ be the child relation as defined in Example 1. We have that:

$$\begin{aligned}
\mathcal{SR}(S, \mathtt{r}) &= S & \mathcal{SR}(S, \mathsf{d}) &= \{\mathsf{d}, \mathsf{g}\} \\
\mathcal{SR}(S, \mathsf{a}) &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathtt{f}, \mathsf{g}\} & \mathcal{SR}(S, \mathsf{e}) &= \{\mathsf{e}\} \\
\mathcal{SR}(S, \mathsf{b}) &= \{\mathsf{b}, \mathsf{d}, \mathsf{e}, \mathsf{g}\} & \mathcal{SR}(S, \mathtt{f}) &= \{\mathtt{f}\} \\
\mathcal{SR}(S, \mathsf{c}) &= \{\mathsf{c}, \mathtt{f}\} & \mathcal{SR}(S, \mathsf{g}) &= \{\mathsf{g}\}
\end{aligned}$$

To formally define the behavior of an SMMT specification, we define labelled transition systems (LTSs).

Definition 4 (Labelled Transition System (LTS))**.** A *Labelled Transition System (LTS)* is a tuple $(ST, L, \rightarrow, s_0)$, where ST is the set of states, L is the set of actions, $\rightarrow \subseteq ST \times L \times ST$ is the set of transitions and $s_0 \in ST$ is the initial state.

In this paper we use the commonly used notation $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$.

Chapter 5

Formal Definition of SMMT

In this chapter we formally define a subset of the language of the State Machine Modelling Tool. This subset consists of *SimpleStates*, *CompositeStates*, *ParallelStates* and transitions without guards or *BehavioralActions*. Due to time constraints we have not been able to formally define the syntax and semantics of the complete set of constructs of SMMT.

The formal definition of SMMT that is presented in this chapter is derived from the discussions with the engineers at Canon Production Printing and the analysis of the code generator, the generated code and the SCM library. Furthermore, the developer of the SCM library mentioned several restrictions that are not implemented in SMMT, which should be satisfied by all SMMT specifications. These restrictions have been included in the formal definition of SMMT.

We create a mathematical model of the syntax of an SMMT specification in Section 5.1 considering only states of type *SimpleState* and *CompositeState* and transitions without guards and *BehavioralActions*. Using the mathematical model of the syntax of SMMT, we specify the behavior of an SMMT specification by defining the operational semantics. The operational semantics is defined in Section 5.2. In Section 5.3, we extend the syntax and semantics of SMMT specifications with states of type *ParallelState*.

In the remainder of this report we denote variables using lowercase letters (e.g., a, b, c), functions using calligraphic, capital letters (e.g., $\mathcal{A}, \mathcal{B}, \mathcal{C}$), sets using capital letters (e.g., A, B, C) and lists using blackboard, capital letters (e.g., $\mathbb{A}, \mathbb{B}, \mathbb{C}$). We write $\mathcal{L}(X)$ to denote sets of lists of type X. Furthermore, we write $\emptyset$ and $[\,]$ to denote the empty set and list respectively.

To explain the definitions presented in this chapter, we use a running example as shown in Figure 5.1. The SMMT specification shown in Figure 5.1 corresponds to the SMMT specification as shown in Figure 2.4 without *BehavioralActions* and parameters.

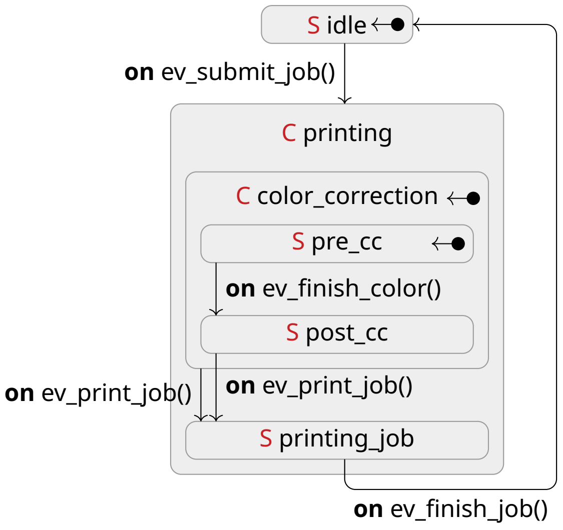
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_print_job(<< ... >>)
    ev_finish_job(<< ... >>)
    ev_finish_color(<< ... >>)
    ev_submit_job(<< ... >>)

do events
    << ... >>

    entry SimpleState idle
        Transitions
            on ev_submit_job()
                go printing
    CompositeState printing
      Transitions
        entry CompositeState color_correction
            Transitions
                on ev_print_job()
                    go printing_job
            entry SimpleState pre_cc
                Transitions
                    on ev_finish_color()
                        go post_cc
            SimpleState post_cc
                Transitions
                    on ev_print_job()
                        go printing_job
        SimpleState printing_job
            Transitions
                on ev_finish_job()
                    go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 5.1: An SMMT specification consisting of a printer that applies color correction before printing a job

5.1 Abstract Syntax of an SMMT Specification

In this section we present the mathematical model of the abstract syntax of an SMMT specification. This mathematical model consists only of states of type *SimpleState*, states of type *CompositeState* and transitions without guards or *BehavioralActions*. The mathematical model of an SMMT specification as given in Definition 5 adheres closely to the implementation of SMMT in Jetbrains MPS.

Definition 5 (SMMT Specification)**.** The abstract syntax of an SMMT specification can mathematically be defined using a tuple $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ where:

- E is the set of *OnEvents* of the SMMT specification.
- S_S and S_C are the set of *SimpleStates* and the set of *CompositeStates* of the SMMT specification respectively. Sets S_S and S_C are disjoint, that is: $S_S \cap S_C = \emptyset$.
- $ES \subseteq S_S \cup S_C$ is the set of entry states of the SMMT specification.
- $\sqsubset \subseteq S \times S$ is a *child relation* (Definition 2), where $S = S_S \cup S_C$.
- $\mathcal{T} : S \to \mathcal{L}(E \times S)$ associates each state $s \in S$ with the list of outgoing transitions of state s, where $S = S_S \cup S_C$. Each outgoing transition $\langle e, s' \rangle \in \mathcal{T}(s)$ of a state $s \in S$ consist of target state $s' \in S$ and an *OnEvent* $e \in E$.

The implementation of SMMT uses the `sequence` type of MPS [22] to define the collection of states, *OnEvents* and outgoing transitions for each state. SMMT requires that all states and all *OnEvents* of an SMMT specification are unique. Furthermore, the ordering of the states and *OnEvents* does not affect the operational semantics of an SMMT specification. Hence, we represent the collection of states and *OnEvents* using sets in the mathematical model of an SMMT specification (Definition 5). As SMMT does allow each state to have multiple outgoing transitions that are defined for the same *OnEvent* and target state, we cannot represent the collection of outgoing transitions of each state using a set. Hence, we represent the outgoing transitions of each state using a list in the mathematical model of an SMMT specification (Definition 5).

Example 3 (SMMT Specification)**.** Using the mathematical model of Definition 5, the SMMT specification of the running example in Figure 5.1 can be expressed as a tuple
$\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$, where:

- $E = \{\mathtt{ev\_print\_job()}, \mathtt{ev\_finish\_job()}, \mathtt{ev\_finish\_color()}, \mathtt{ev\_submit\_job()}\}$
- $S_S = \{\mathtt{idle}, \mathtt{pre\_cc}, \mathtt{post\_cc}, \mathtt{printing\_job}\}$
- $S_C = \{\mathtt{printing}, \mathtt{color\_correction}\}$
- $ES = \{\mathtt{idle}, \mathtt{pre\_cc}, \mathtt{color\_correction}\}$
- $\sqsubset = \{(\mathtt{color\_correction}, \mathtt{printing}), (\mathtt{printing\_job}, \mathtt{printing}),$
  $(\mathtt{pre\_cc}, \mathtt{color\_correction}), (\mathtt{post\_cc}, \mathtt{color\_correction})\}$
- $\mathcal{T}(\mathtt{idle}) = [\langle \mathtt{ev\_submit\_job()}, \mathtt{printing} \rangle]$
- $\mathcal{T}(\mathtt{printing}) = [\,]$
- $\mathcal{T}(\mathtt{color\_correction}) = [\langle \mathtt{ev\_print\_job()}, \mathtt{printing\_job} \rangle]$
- $\mathcal{T}(\mathtt{pre\_cc}) = [\langle \mathtt{ev\_finish\_color()}, \mathtt{post\_cc} \rangle]$
- $\mathcal{T}(\mathtt{post\_cc}) = [\langle \mathtt{ev\_print\_job()}, \mathtt{printing\_job} \rangle]$

- $\mathcal{T}(\texttt{printing\_job}) = [\langle \texttt{ev\_finish\_job()}, \texttt{idle} \rangle]$

Descendant relation $\sqsubset^+$ is the transitive closure of child relation $\sqsubset$. Hence, for SMMT specification $\texttt{M}$, the descendant relation is defined as follows:

- $\sqsubset^+ = \{(\texttt{color\_correction}, \texttt{printing}), (\texttt{pre\_cc}, \texttt{printing}),$
  $(\texttt{post\_cc}, \texttt{printing}), (\texttt{printing\_job}, \texttt{printing}),$
  $(\texttt{pre\_cc}, \texttt{color\_correction}), (\texttt{post\_cc}, \texttt{color\_correction})\}$

To reason about the set of states of an SMMT specification $\texttt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$, we introduce function $\mathcal{S}(\texttt{M})$ that returns the set consisting of all states of SMMT specification $\texttt{M}$.

Definition 6 (Set of States)**.** Let $\texttt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The set of all states of an SMMT specification $\texttt{M}$, denoted by $\mathcal{S}(\texttt{M})$, is defined as follows:

$$\mathcal{S}(\texttt{M}) = S_S \cup S_C$$

In the remainder of this section, we discuss the restrictions on the set of states $\mathcal{S}(\texttt{M})$, the set of entry states ES, child relation $\sqsubset$ and on the transition relation $\mathcal{T}$. Furthermore, we define the *entry child* and the *entry descendant* relation that are used to define the operational semantics of an SMMT specification in Section 5.2.

Restriction on the set of states $\mathcal{S}(\texttt{M})$

An SMMT specification that has no states cannot have any transitions either. As mentioned in Chapter 2, the execution of an SMMT specification is terminated if an *OnEvent* is processed for which no transition is defined from some state. Therefore, the execution of an SMMT specification without states will terminate when any *OnEvent* is processed. Hence, it is not relevant to define the semantics of an SMMT specification without any states. We require the set of states of an SMMT specification to be non-empty, as defined by Restriction 1.

Restriction 1 (Non-Empty Set of States)**.** Let $M = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. We require that the set of states of SMMT specification $\texttt{M}$ is non-empty, that is we require that the following restriction holds:

$$\mathcal{S}(\texttt{M}) \neq \emptyset$$

Restrictions on the set of entry states ES

As mentioned in Chapter 2, the region of an SMMT specification puts restrictions on the entry set of an SMMT specification. Namely, the set of entry states must contain exactly one state that has no parent and it must contain exactly one child for each *CompositeState* in the set of entry states.

To define the restrictions on the set of entry states ES of an SMMT specification, we introduce the notion of a *root state* as a state that has no parent. In Definition 7, we define the set of *root states* $\mathcal{R}(\texttt{M}) \subseteq \mathcal{S}(\texttt{M})$ of an SMMT specification $\texttt{M}$.

Definition 7 (Root State)**.** Let $\texttt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The set of *root states* $\mathcal{R}(\texttt{M}) \subseteq \mathcal{S}(\texttt{M})$ of SMMT specification $\texttt{M}$ is defined as follows:

$$\mathcal{R}(\texttt{M}) = \{s \in \mathcal{S}(\texttt{M}) \mid \neg\exists_{s' \in \mathcal{S}(\texttt{M})} : s \sqsubset s'\}$$

Example 4 (Root State)**.** Let $\texttt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.1). For SMMT specification $\texttt{M}$ we have that:

$$\mathcal{R}(\texttt{M}) = \{\texttt{idle}, \texttt{printing}\}$$

Using the definition of a root state, we can formally define the restrictions on the set of entry states ES of an SMMT specification as defined in Restrictions 2 and 3.

Restriction 2 (Exactly One Entry Root State)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. We require that there exists exactly one state in the set of entry states ES of SMMT specification $\mathtt{M}$ that is a root state. Hence, the following condition must hold:

$$\exists!_{s \,\in\, ES} : s \in \mathcal{R}(\mathtt{M})$$

Restriction 3 (A *CompositeState* has exactly one entry child)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. For each *CompositeState* $s' \in S_C$ of SMMT specification $\mathtt{M}$, exactly one child $s \in \mathcal{S}(\mathtt{M})$ of *CompositeState* s' must be contained in the set of entry states ES. Hence, the following condition must hold:

$$\forall_{s' \,\in\, S_C} : (\exists!_{s \,\in\, ES} : s \sqsubset s')$$

Restrictions on the child relation $\sqsubset$

As mentioned in Chapter 2, the different types of states put restrictions on the child relation $\sqsubset$. That is, a *SimpleState* has no children and a *CompositeState* has at least one child. By Restriction 3 it follows that each *CompositeState* has at least one child. We require that Restriction 4 must hold on each SMMT specification.

Restriction 4 (A *SimpleState* has no children)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. For all *SimpleStates* $s' \in S_S$, there exists no state $s \in \mathcal{S}(\mathtt{M})$ such that state s is a child of state s', formally expressed as:

$$\forall_{s' \,\in\, S_S} : (\neg\exists_{s \,\in\, \mathcal{S}(\mathtt{M})} : s \sqsubset s')$$

Restriction on the transition relation $\mathcal{T}$

We only consider SMMT specifications for which all states have at most 1 transition defined for each *OnEvent* $e \in E$ as defined by Restriction 5. That is, this restriction ensures that the behavior of the SMMT specification is deterministic.

Restriction 5 (Deterministic Transitions)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. For each state $s \in \mathcal{S}(\mathtt{M})$ there may only be at most one transition $t \in \mathcal{T}(s)$ for each *OnEvent* $e \in E$, formally expressed as:

$$\forall_{s \,\in\, \mathcal{S}(\mathtt{M})} : \Big(\forall_{e \,\in\, E} : \Big|\big\{0 \le i < |\mathcal{T}(s)| \bigm| \exists_{s' \,\in\, \mathcal{S}(\mathtt{M})} : \langle e, s' \rangle = \mathcal{T}(s)[i]\big\}\Big| \le 1\Big)$$

Hence, by Restriction 5 all outgoing transitions that are defined for each state $s \in \mathcal{S}(\mathtt{M})$ are unique. The order of the transitions per state does not affect the operational semantics of an SMMT specification. Therefore, we can interpret the list of transitions per state as a set.

Entry Child Relation $\sqsubset_{ES}$ and Entry Descendant Relation $\sqsubset^+_{ES}$

We introduce the notion of an *entry child*. Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. A state $s \in \mathcal{S}(\mathtt{M})$ is an *entry child* of state $s' \in \mathcal{S}(\mathtt{M})$ if, and only if, state s is a child of state s' and state s is an entry state. We define the entry child function $\mathcal{EC}(\mathtt{M}, s')$ that returns the set of entry children of a state $s' \in \mathcal{S}(\mathtt{M})$.

By Restriction 3 we have that there always exists exactly one entry child for each *CompositeState* $s' \in \mathcal{S}(\mathtt{M})$. Hence, function $\mathcal{EC}(\mathtt{M}, s')$ returns a set consisting of exactly one state for each *CompositeState* $s' \in S_C$. Furthermore, for each *SimpleState* $s' \in S_S$, the function returns the empty set as a *SimpleState* has no children by Restriction 4.

Definition 8 (Entry Child Function)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. For any state $s' \in \mathcal{S}(\mathtt{M})$, the set of *entry children* $\mathcal{EC}(\mathtt{M}, s')$ is defined as follows:

$$\mathcal{EC}(\mathtt{M}, s') = \{s \in ES \mid s \sqsubset s'\}$$

Example 5 (Entry Child Function)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.1. For all *CompositeStates* $s' \in S_C$, the set of entry children $\mathcal{EC}(\mathtt{M}, s')$ is defined as follows:

$$\begin{aligned} \mathcal{EC}(\mathtt{M}, \mathtt{printing}) &= \{\mathtt{color\_correction}\} \\ \mathcal{EC}(\mathtt{M}, \mathtt{color\_correction}) &= \{\mathtt{pre\_cc}\} \end{aligned}$$

To simplify future definitions, we define an *entry child relation* $\sqsubset_{ES}$ in Definition 9 based on the entry child function $\mathcal{EC}(\mathtt{M}, s')$.

Definition 9 (Entry Child Relation $\sqsubset_{ES}$)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. We define *entry child relation* $\sqsubset_{ES}$ such that $s \sqsubset_{ES} s'$ if, and only if, state $s \in \mathcal{S}(\mathtt{M})$ is the entry child of state $s' \in \mathcal{S}(\mathtt{M})$, that is:

$$\forall_{s,s' \in \mathcal{S}(\mathtt{M})} : s \sqsubset_{ES} s' \Leftrightarrow s \in \mathcal{EC}(\mathtt{M}, s')$$

Note that, the entry child relation $\sqsubset_{ES}$ can be defined as the intersection of the child relation $\sqsubset$ and the Cartesian product of the set of entry states ES and the set of *CompositeStates* S_C.

Lemma 1 (Entry Child Relation $\sqsubset_{ES}$)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The entry child relation $\sqsubset_{ES}$ as defined in Definition 9 is equivalent to $\sqsubset \cap (ES \times S_C)$, that is:

$$\sqsubset_{ES} \equiv \sqsubset \cap (ES \times S_C)$$

The proof of Lemma 1 can be found in Appendix A.

Example 6 (Entry Child Relation)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.1). In Example 5 we defined the entry child function for the running example. Hence, the entry child relation $\sqsubset_{ES}$ is defined as follows:

$$\sqsubset_{ES} = \{(\mathtt{color\_correction}, \mathtt{printing}), (\mathtt{pre\_cc}, \mathtt{color\_correction})\}$$

Note that this is indeed equivalent to $\sqsubset \cap (ES \times S_C)$, since we have that:

$$\begin{aligned} \sqsubset &= \{(\mathtt{color\_correction}, \mathtt{printing}), (\mathtt{printing\_job}, \mathtt{printing}), \\ &\quad (\mathtt{pre\_cc}, \mathtt{color\_correction}), (\mathtt{post\_cc}, \mathtt{color\_correction})\} \\ (ES \times S_C) &= \{(\mathtt{idle}, \mathtt{printing}), (\mathtt{idle}, \mathtt{color\_correction}), \\ &\quad (\mathtt{color\_correction}, \mathtt{printing}), (\mathtt{color\_correction}, \mathtt{color\_correction}), \\ &\quad (\mathtt{pre\_cc}, \mathtt{printing}), (\mathtt{pre\_cc}, \mathtt{color\_correction})\} \end{aligned}$$

Using the entry child relation, we define the *entry descendant relation*. A state $s \in \mathcal{S}(\mathtt{M})$ is an *entry descendant* of state $s' \in \mathcal{S}(\mathtt{M})$ if, and only if, state s is a descendant of state s', state s is an entry state and all states that are both an ancestor of state s and a descendant of state s' are entry states. We define the *entry descendant relation* in Definition 10.

Definition 10 (Entry Descendant Relation $\sqsubset^+_{ES}$)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. We define the *entry descendant relation* $\sqsubset^+_{ES}$ as the transitive closure of the entry child relation $\sqsubset_{ES}$.

Example 7 (Entry Descendant Relation)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.1). For the running example, the entry child relation $\sqsubset_{ES}$ has been defined in Example 6. Hence, the entry descendant relation $\sqsubset^+_{ES}$ of the running example is defined as follows:

$$\sqsubset^+_{ES} = \{(\mathtt{color\_correction}, \mathtt{printing}), (\mathtt{pre\_cc}, \mathtt{printing}), (\mathtt{pre\_cc}, \mathtt{color\_correction})\}$$

5.2 Semantics of an SMMT Specification

In this section we define the operational semantics of an SMMT specification consisting of *SimpleStates*, *CompositeStates* and transitions without guards or *BehavioralActions*. We first introduce the notion of an *execution state*. An *execution state* describes a set of states in which the SMMT specification can be when the specification is executed. We say that a state is *active* if, and only if, the state occurs in the execution state of an SMMT specification.

Set of Execution States $\mathcal{EXS}(\mathtt{M})$

When executing an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$, we have that exactly one of the root states of SMMT specification M must be in the execution state at any point in time. Furthermore, for each *CompositeState* $s' \in S_C$ in execution state $EX \subseteq \mathcal{S}(\mathtt{M})$, there must be exactly one child $s \in \mathcal{S}(\mathtt{M})$ of s' in EX. Finally, if a state $t \in \mathcal{S}(\mathtt{M})$ is contained in execution state EX, then all ancestors $t' \in \mathcal{S}(\mathtt{M})$ of state t must be contained in execution state EX. We define the set of all execution states $\mathcal{EXS}(\mathtt{M})$ of SMMT specification M in Definition 11.

Definition 11 (Set of Execution States)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The set of *execution states* of SMMT specification M, $\mathcal{EXS}(\mathtt{M})$, is defined as follows:

$$\begin{aligned}\mathcal{EXS}(\mathtt{M}) = \{EX \subseteq \mathcal{S}(\mathtt{M}) \mid\ & (\exists!_{r \,\in\, EX} : r \in \mathcal{R}(\mathtt{M})) \\ & \wedge (\forall_{s' \,\in\, (S_C \,\cap\, EX)} : (\exists!_{s \,\in\, \mathcal{S}(\mathtt{M})} : s \sqsubset s' \wedge s \in EX)) \\ & \wedge (\forall_{t \,\in\, EX} : (\forall_{t' \,\in\, \mathcal{S}(\mathtt{M})} : t \sqsubset^+ t' \Rightarrow t' \in EX))\}\end{aligned}$$

Example 8 (Set of Execution States)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.1). The set of execution states $\mathcal{EXS}(\mathtt{M})$ of the running example is defined as follows:

$$\begin{aligned}\mathcal{EXS}(\mathtt{M}) = \{&\{\mathtt{idle}\}, \{\mathtt{printing}, \mathtt{color\_correction}, \mathtt{pre\_cc}\}, \\ &\{\mathtt{printing}, \mathtt{color\_correction}, \mathtt{post\_cc}\}, \{\mathtt{printing}, \mathtt{printing\_job}\}\}\end{aligned}$$

By Definition 11 it follows that the set of execution state $\mathcal{EXS}(\mathtt{M})$ contains exactly one execution state for each *SimpleState* $s \in S_S$. Furthermore, each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ consists of exactly one *SimpleState* $s \in S_S$ and all ancestors of s.

Lemma 2 (Execution State)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The set of execution states $\mathcal{EXS}(\mathtt{M})$ consists of exactly one execution state for each *SimpleState* $s \in S_S$ that contains state s and all ancestors of state s. Hence, the set of execution states can be derived as follows:

$$\mathcal{EXS}(\mathtt{M}) \equiv \bigcup_{s \,\in\, S_S} \{\{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^* s'\}\}$$

The proof of Lemma 2 can be found in Appendix A.

Initial Execution State $\mathcal{I}(\mathtt{M})$

The initial execution state of an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ consists of all entry states $s \in ES$ of which all ancestors are contained in the set of entry states ES. We define the *initial execution state* $\mathcal{I}(\mathtt{M})$ of SMMT specification M in Definition 12.

Definition 12 (Initial Execution State)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The *initial execution state* $\mathcal{I}(\mathtt{M})$ of SMMT specification M is defined as:

$$\mathcal{I}(\mathtt{M}) = \{s \in ES \mid \forall_{s' \,\in\, \mathcal{S}(\mathtt{M})} : s \sqsubset^+ s' \Rightarrow s' \in ES\}$$

Example 9 (Initial Execution States). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.1). The initial execution state $\mathcal{I}(\mathtt{M}) \in \mathcal{EXS}(\mathtt{M})$ of SMMT specification M is defined as follows:

$$\mathcal{I}(\mathtt{M}) = \{\mathtt{idle}\}$$

Lemma 3 (Initial Execution State is an Execution State). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The initial execution state $\mathcal{I}(\mathtt{M})$ is an execution state, that is:

$$\mathcal{I}(\mathtt{M}) \in \mathcal{EXS}(\mathtt{M})$$

The proof of Lemma 3 can be found in Appendix A.

Set of Prioritised Transitions $\mathcal{PT}(\mathtt{M}, EX)$

When an *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, there could exist multiple states $s \in EX$ that have a transition defined for *OnEvent* e. A transition can only fire from the deepest nested state $s \in EX$ that has a transition defined for *OnEvent* $e \in E$. By Lemma 2, it follows that all states in execution state EX are related by descendant relation $\sqsubset^+$. Hence, there exists at most one deepest nested state $s \in EX$ that has a transition defined for *OnEvent* e. We refer to the transition that is defined for this state s and *OnEvent* e as the *prioritised transition* for *OnEvent* e in execution state EX.

For an execution state $EX \in \mathcal{EXS}(\mathtt{M})$ we define the set of prioritised transitions over execution state EX as the union of all prioritised transitions of each state $s \in EX$. We define the set of *prioritised transitions* over an execution state in Definition 13.

Definition 13 (Prioritised Transitions). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. For each execution state $EX \in \mathcal{EXS}(\mathtt{M})$, the set of *prioritised transitions* $\mathcal{PT}(\mathtt{M}, EX)$ is defined as follows:

$$\mathcal{PT}(\mathtt{M}, EX) = \bigcup_{s \,\in\, EX} \{\langle e, s' \rangle \in \mathcal{T}(s) \mid \neg\exists_{x \,\in\, EX} : (x \sqsubset^+ s \wedge (\exists_{x' \,\in\, \mathcal{S}(\mathtt{M})} : \langle e, x' \rangle \in \mathcal{T}(x)))\}$$

Furthermore, we define $\mathcal{PT}_e(\mathtt{M}, EX)$ as the set consisting of the target states of prioritised transitions in $\mathcal{PT}(\mathtt{M}, EX)$ that are defined for *OnEvent* $e \in E$. For an SMMT specification M in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, $\mathcal{PT}_e(\mathtt{M}, EX)$ is defined as follows:

$$\mathcal{PT}_e(\mathtt{M}, EX) = \{s' \in \mathcal{S}(\mathtt{M}) \mid \langle e, s' \rangle \in \mathcal{PT}(\mathtt{M}, EX)\}$$

By Restriction 5 it follows that set $\mathcal{PT}_e(\mathtt{M}, EX)$ contains at most one state for each *OnEvent* $e \in E$ and execution state $EX \in \mathcal{EXS}(\mathtt{M})$.

Example 10 (Prioritised Transitions). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.1). The set of prioritised transitions of execution state $EX = \{\mathtt{printing}, \mathtt{color\_correction}\}$ is defined as follows:

$$\mathcal{PT}(\mathtt{M}, EX) = \{\langle \mathtt{ev\_finish\_color()}, \mathtt{post\_cc} \rangle, \langle \mathtt{ev\_start\_printing()}, \mathtt{printing\_job} \rangle\}$$

Furthermore, we have that:

$$\mathcal{PT}_{\mathtt{ev\_finish\_color()}}(\mathtt{M}, EX) = \{\mathtt{post\_cc}\}$$
$$\mathcal{PT}_{\mathtt{ev\_start\_printing()}}(\mathtt{M}, EX) = \{\mathtt{printing\_job}\}$$

Enabled *OnEvents*

Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. When an *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, the transitions in $\mathcal{PT}(\mathtt{M}, EX)$ that are defined for *OnEvent* e fire. We say that an *OnEvent* is *enabled* if, and only if, a transition is defined for *OnEvent* e in the set of prioritised transitions $\mathcal{PT}(\mathtt{M}, EX)$. That is, *OnEvent* $e \in E$ is enabled in execution state $EX \in \mathcal{EXS}(\mathtt{M})$ if the set of prioritised transitions that are defined for *OnEvent* e is non-empty.

Definition 14 (Enabled *OnEvent*)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M})$ be the execution state of SMMT specification $\mathtt{M}$. Function $\mathcal{E}(\mathtt{M}, EX, e)$ determines whether *OnEvent* $e \in E$ of SMMT specification $\mathtt{M}$ is *enabled* in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, which is defined as follows:

$$\mathcal{E}(\mathtt{M}, EX, e) = (\mathcal{PT}_e(\mathtt{M}, EX) \neq \emptyset)$$

Example 11 (Enabled *OnEvent*)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.1) in execution state $EX = \{$`printing`, `color_correction`, `pre_cc`$\}$. In execution state EX, only *OnEvents* `ev_finish_color()` and `ev_print_job()` are enabled.

Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M})$ be the execution state of SMMT specification $\mathtt{M}$. By Restriction 5 we have that each state in execution state EX has at most one transition defined for each *OnEvent* $e \in E$. Furthermore, by Lemma 2 and the definition of an enabled *OnEvent* (Definition 14) it follows that there exists only one deepest nested state $s \in EX$ that has a transition defined for enabled *OnEvent* e. Hence, the set of prioritised transitions of SMMT specification $\mathtt{M}$ in execution state EX contains exactly one prioritised transition for each enabled *OnEvent* $e \in E$.

Lemma 4 (Exactly One Prioritised Transition for an enabled *OnEvent*)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The set of prioritised transitions $\mathcal{PT}_e(\mathtt{M}, EX)$ contains exactly one target state if *OnEvent* $e \in E$ is enabled in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, that is:

$$\forall_{EX \,\in\, \mathcal{EXS}(\mathtt{M})} : (\forall_{e \,\in\, E} : (\mathcal{E}(\mathtt{M}, EX, e) \Rightarrow (|\mathcal{PT}_e(\mathtt{M}, EX)| = 1)))$$

The proof of Lemma 4 can be found in Appendix A.

Execution State Update $\mathcal{ESU}(\mathtt{M})$

When an enabled *OnEvent* is processed by an SMMT specification, the execution state is updated to the set consisting of the target states of the transition that fires and all ancestors and entry descendants of this target state. We define the execution state update function $\mathcal{ESU}(\mathtt{M}, EX, e)$ that defines the execution state after an enabled *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$.

Definition 15 (Execution State Update)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The execution state update function $\mathcal{ESU}(\mathtt{M}, EX, e)$ defines the execution state after an enabled *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, which is defines as follows:

$$\mathcal{ESU}(\mathtt{M}, EX, e) = \{s \in \mathcal{S}(\mathtt{M}) \mid \exists_{s' \,\in\, \mathcal{PT}_e(\mathtt{M}, EX)} : s' \sqsubset^+ s \vee s \sqsubset^*_{ES} s'\}$$

where $\sqsubset^*_{ES}$ is the reflexive transitive closure of entry child relation $\sqsubset_{ES}$.

Example 12 (Execution State Update)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.1). When enabled *OnEvent* `ev_print_job()` is processed in execution state $EX = \{\mathtt{printing}, \mathtt{color\_correction}, \mathtt{post\_cc}\}$ then the execution state updated as defined by the execution state update function $\mathcal{ESU}(\mathtt{M}, EX, e)$:

$$\mathcal{ESU}(\mathtt{M}, EX, \mathtt{ev\_start\_printing()}) = \{\mathtt{printing}, \mathtt{printing\_job}\}$$

The set of active states after *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, $\mathcal{ESU}(\mathtt{M}, EX, e)$, is an execution state if *OnEvent* e is an enabled *OnEvent*.

Lemma 5 (Execution State Update returns an Execution State)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. For each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ and each enabled *OnEvent* $e \in E$, $\mathcal{ESU}(\mathtt{M}, EX, e)$ is an execution state, that is:

$$\forall_{EX \in \mathcal{EXS}(\mathtt{M})} : (\forall_{e \in E} : (\mathcal{E}(\mathtt{M}, EX, e) \Rightarrow (\mathcal{ESU}(\mathtt{M}, EX, e) \in \mathcal{EXS}(\mathtt{M}))))$$

The proof of Lemma 5 can be found in Appendix A.

Operational Semantics

When an *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M})$ that is not enabled, an internal software exception is thrown and the execution of the SMMT specification is terminated. To denote that the execution of the SMMT specification has terminated, we introduce a *failure state* F that is reached if the execution is terminated. Failure state F has a self-loop with action `FAIL` to indicate that a failure occurred during execution.

The *operational semantics* of an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ is defined in Definition 16.

Definition 16 (Operational Semantics)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The *operational semantics* of SMMT specification M is given by LTS $\mathtt{L} = \langle ST, L, \rightarrow, s_0\rangle$, where:

- $ST = \mathcal{EXS}(\mathtt{M}) \cup \{\mathsf{F}\}$
- $L = E \cup \{\mathtt{FAIL}\}$
- $\rightarrow \subseteq ST \times L \times ST$ such that for all $EX, EX' \in \mathcal{EXS}(\mathtt{M})$ and $e \in E$ we have that:
  - $EX \xrightarrow{e} EX'$ if, and only if, $\mathcal{E}(\mathtt{M}, EX, e) \wedge EX' = \mathcal{ESU}(\mathtt{M}, EX, e)$
  - $EX \xrightarrow{e} \mathsf{F}$ if, and only if, $\neg\mathcal{E}(\mathtt{M}, EX, e)$
  - $\mathsf{F} \xrightarrow{\mathtt{FAIL}} \mathsf{F}$
- $s_0 = \mathcal{I}(\mathtt{M})$

Example 13 (Operational Semantics)**.** Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example of Figure 5.1 as given in Example 3. The operational semantics of SMMT specification `M` is denoted using the LTS that is shown in Figure 5.2.

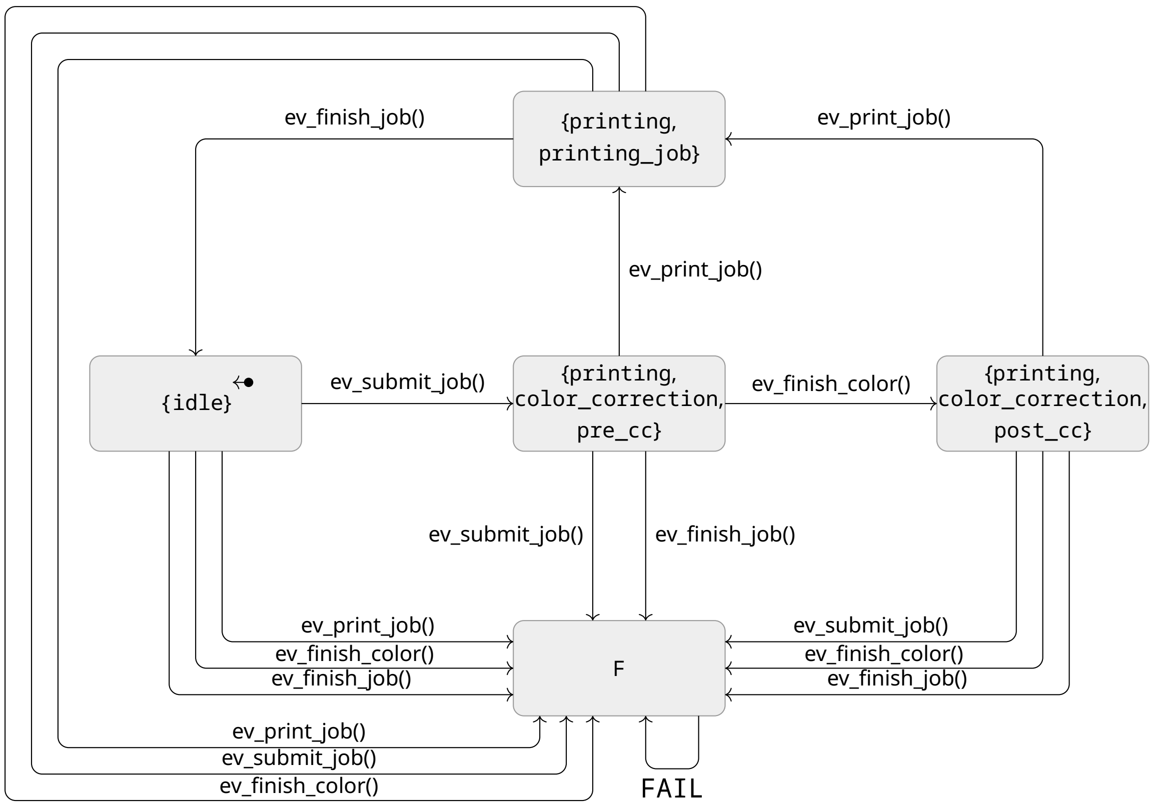


Figure 5.2: Operational Semantics denoted using an LTS for the Running Example (Figure 5.1)

5.3 Parallel States

In this section we extend our formal definition of SMMT as presented in Sections 5.1 and 5.2 with states of type *ParallelState*. We expand the example in Figure 2.5 with an `error` state in Figure 5.3. The example in Figure 5.3 is used as a running example to explain the definitions that are presented in this section.

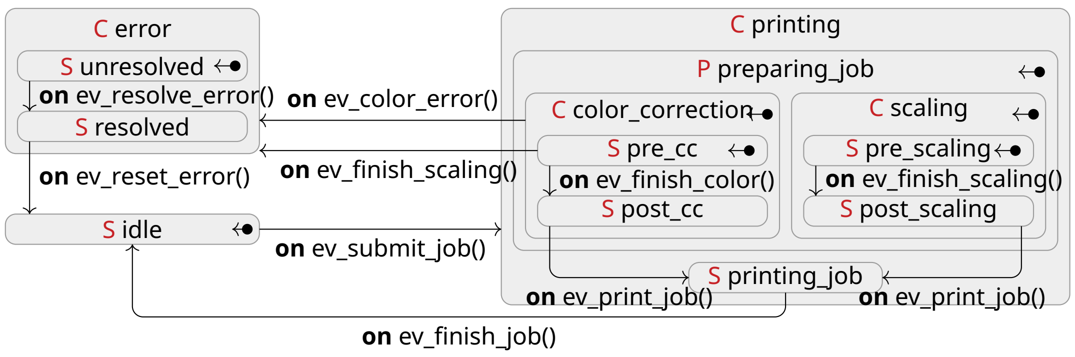
```
State Machine printer
namespace cpp.jordi.examples.printer

on events
    ev_print_job(<< ... >>)
    ev_finish_job(<< ... >>)
    ev_finish_color(<< ... >>)
    ev_finish_scaling(<< ... >>)
    ev_resolve_error(<< ... >>)
    ev_reset_error(<< ... >>)
    ev_color_error(<< ... >>)
    ev_submit_job(<< ... >>)

do events
    << ... >>

    entry SimpleState idle
        Transitions
            on ev_submit_job()
                go printing
    CompositeState error
        Transitions
        entry SimpleState unresolved
            Transitions
                on ev_resolve_error()
                    go resolved
        SimpleState resolved
            Transitions
                on ev_reset_error()
                    go idle
    CompositeState printing
        Transitions
        entry ParallelState preparing_job
            Transitions
            entry CompositeState color_correction
                Transitions
                    on ev_color_error()
                        go error
                entry SimpleState pre_cc
                    Transitions
                        on ev_finish_color()
                            go post_cc
                        on ev_finish_scaling()
                            go error
                SimpleState post_cc
                    Transitions
                        on ev_print_job()
                            go printing_job
            entry CompositeState scaling
                Transitions
                entry SimpleState pre_scaling
                    Transitions
                        on ev_finish_scaling()
                            go post_scaling
                SimpleState post_scaling
                    Transitions
                        on ev_print_job()
                            go printing_job
        SimpleState printing_job
            Transitions
                on ev_finish_job()
                    go idle
```

(a) Textual Representation



(b) Graphical Representation

Figure 5.3: An SMMT specification consisting of a printer that applies color correction and/or scales the job before printing the print job during which errors may occur

In Section 5.3.1 we discuss the syntax of SMMT specifications that include states of type *SimpleStates*, *CompositeStates* and *ParallelStates* and transitions without guards or *BehavioralActions*. We define the semantics of SMMT specifications with *ParallelStates* in Section 5.3.2.

5.3.1 Abstract Syntax of an SMMT Specification

In this section we extend the mathematical model that we defined in Definition 5 with states of type *ParallelStates*. We redefine the definition of our mathematical model of an SMMT specification.

Definition 17 (SMMT Specification with *ParallelStates*). The abstract syntax of an SMMT specification can be mathematically defined using a tuple $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ where:

- E is the set of *OnEvents* of the SMMT specification.
- S_S, S_C and S_P are the sets of *SimpleStates*, the set of *CompositeStates* and the set of *ParallelStates* of the SMMT specification respectively. Sets S_S, S_C and S_P are pairwise disjoint, that is: $S_S \cap S_C = S_S \cap S_P = S_C \cap S_P = \emptyset$.
- $ES \subseteq S_S \cup S_C \cup S_P$ is the set of entry states of the SMMT specification.
- $\sqsubset \subseteq S \times S$ is a *child relation* (Definition 2), where $S = S_S \cup S_C \cup S_P$.
- $\mathcal{T} : S \to \mathcal{L}(E \times S)$ associates each state $s \in S$ with the list of outgoing transitions of state s, where $S = S_S \cup S_C \cup S_P$. Each outgoing transition $\langle e, s'\rangle \in \mathcal{T}(s)$ of a state $s \in S$ consist of target state $s' \in S$ and an *OnEvent* $e \in E$.

To reason about the set of all states of an SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$, we introduce function $\mathcal{S}(\mathtt{M}')$ that returns the set consisting of all states of SMMT specification $\mathtt{M}'$.

Definition 18 (Set of States (SMMT Specifications with *ParallelStates*)). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The set of all states of SMMT specification $\mathtt{M}'$, denoted by $\mathcal{S}(\mathtt{M}')$, is defined as follows:

$$\mathcal{S}(\mathtt{M}') = S_S \cup S_C \cup S_P$$

We define the restrictions that are put on SMMT specifications by states of type *ParallelState* in addition to all restrictions that were discussed in Chapter 5.1. Next, we extend the entry child and entry descendant relation to allow for states of type *ParallelState*.

Restrictions on SMMT Specifications

In Section 5.1, we discussed the restrictions on an SMMT specification without *ParallelStates*. These restrictions (Restrictions 1 to 5) must hold for SMMT specifications with *ParallelStates* as well. States of types *ParallelStates* put further restrictions on the set of entry states ES and the child relation $\sqsubset$. We require that Restrictions 6 and 7 hold for all SMMT specifications with *ParallelStates*.

As discussed in Chapter 2, all children of a *ParallelState* are entry states. To adhere closely to the implementation of SMMT in MPS, we therefore require that for each SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ it holds that all children of a *ParallelState* are in the set of entry states as defined by Restriction 6.

Restriction 6 (All children of a *ParallelState* are entry states). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. We require that all children of a *ParallelState* are contained in the set of entry states ES. That is, we require that:

$$\forall_{s' \in S_P} : (\forall_{s \in \mathcal{S}(\mathtt{M}')} : s \sqsubset s' \Rightarrow s \in ES)$$

ParallelStates were introduced to model behavior that is meant to be performed in parallel. If a *ParallelState* $s \in S_P$ is active, then all children of state s are active. Hence, this allows that the subregion (Definition 3) of each child of state s can handle an *OnEvent* in parallel. A

ParallelState with only one child and no *ParallelState* descendants cannot model parallel behavior, as such a *ParallelState* has no descendant with two active children if the *ParallelState* is active. To ensure that the *ParallelStates* are only used to model parallel behavior, we restrict that each *ParallelState* has at least two children. This restriction is defined by Restriction 7.

Restriction 7 (A *ParallelState* has at least two children)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. We require that all *ParallelStates* have at least two children. That is, we require that:

$$\forall_{s'' \in S_P} : (\exists_{s,s' \in \mathcal{S}(\mathtt{M}')} : s \neq s' \wedge s \sqsubset s'' \wedge s' \sqsubset s'')$$

Entry Child Relation $\sqsubset_{ES}$ and Entry Descendant Relation $\sqsubset_{ES}^{+}$

A state $s \in \mathcal{S}(\mathtt{M}')$ is an *entry child* of state $s' \in \mathcal{S}(\mathtt{M}')$ if, and only if, state s is a child of state s' and state s is an entry state. Hence, the definition of the set of entry children of a state $s' \in \mathcal{S}(\mathtt{M}')$, denoted by $\mathcal{EC}(\mathtt{M}', s')$, corresponds to Definition 8 where all occurrences of $\mathtt{M}$ are replaced by $\mathtt{M}'$. By Restriction 6 we have that all children $s \in \mathcal{S}(\mathtt{M}')$ of a *ParallelState* $s' \in S_P$ are entry states. Hence, all children $s \in \mathcal{S}(\mathtt{M}')$ of *ParallelState* $s' \in S_P$ are entry children of *ParallelState* s'. Therefore, the entry child and entry descendant relations correspond to Definitions 9 and 10 where each occurrence of $\mathtt{M}$ is replaced by $\mathtt{M}'$.

Note that, the entry child relation $\sqsubset_{ES}$ can be defined as the intersection of the child relation $\sqsubset$ and the Cartesian product of the set of entry states ES and the union of the set of *CompositeStates* S_C and the set of *ParallelStates* S_P.

Lemma 6 (Entry Child Relation $\sqsubset_{ES}$ (SMMT Specifications with *ParallelStates*))**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The entry child relation $\sqsubset_{ES}$ as defined in Definition 9, where all occurrences of $\mathtt{M}$ are replaced by $\mathtt{M}'$, is equivalent to $\sqsubset \cap (ES \times (S_C \cup S_P))$, that is:

$$\sqsubset_{ES} \equiv \sqsubset \cap (ES \times (S_C \cup S_P))$$

The proof of Lemma 6 can be found in Appendix A.

5.3.2 Semantics of an SMMT Specification

In Section 5.2 we discussed the semantics of an SMMT specification without *ParallelStates*. In this section we extend the semantics for SMMT specifications that include *ParallelStates* as presented in Section 5.3.1. For each definition in Section 5.2, we discuss the modifications that are required for each definition to be compatible with SMMT specifications with *ParallelStates*.

Set of Execution States $\mathcal{EXS}(\mathtt{M}')$

In Definition 11 we defined the set of execution states for an SMMT specification without *ParallelStates*. By Definition 11, we have that each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ of an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$:

- Contains exactly one root state, that is: $\exists!_{r \in EX} : r \in \mathcal{R}(\mathtt{M})$
- Contains exactly one child $s \in \mathcal{S}(\mathtt{M})$ for each *CompositeState* $s \in S_C$ in execution state EX, that is:

$$\forall_{s' \in (S_C \cap EX)} : (\exists!_{s \in \mathcal{S}(\mathtt{M})} : s \sqsubset s' \wedge s \in EX)$$

- Contains all ancestors $u' \in \mathcal{S}(\mathtt{M})$ of each state $u \in EX$, that is:

$$\forall_{u \in EX} : (\forall_{u' \in \mathcal{S}(\mathtt{M})} : u \sqsubset^{+} u' \Rightarrow u' \in EX)$$

The three restrictions that are defined for an execution state of an SMMT specification without *ParallelState* must also be satisfied by an execution state of an SMMT specification with *ParallelStates*. As mentioned in Chapter 2, all children $s \in \mathcal{S}(\texttt{M}')$ of a *ParallelState* $s' \in S_P$ must be in the execution state if, and only if, s' is in the execution state. We extend the definition of the set of execution states to include this restriction.

Definition 19 (Set of Execution States (SMMT Specifications with *ParallelStates*))**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The set of execution states $\mathcal{EXS}(\texttt{M}')$ of SMMT specification $\texttt{M}'$ is defined as follows:

$$\begin{aligned}\mathcal{EXS}(\texttt{M}') = \{EX \subseteq \mathcal{S}(\texttt{M}') \mid\ & (\exists!_{r \in EX} : r \in \mathcal{R}(\texttt{M}')) \\ & \wedge (\forall_{s' \in (S_C \cap EX)} : (\exists!_{s \in \mathcal{S}(\texttt{M}')} : s \sqsubset s' \wedge s \in EX)) \\ & \wedge (\forall_{t' \in (S_P \cap EX)} : (\forall_{t \in \mathcal{S}(\texttt{M}')} : t \sqsubset t' \Rightarrow t \in EX)) \\ & \wedge (\forall_{u \in EX} : (\forall_{u' \in \mathcal{S}(\texttt{M}')} : u \sqsubset^+ u' \Rightarrow u' \in EX))\}\end{aligned}$$

Example 14 (Set of Execution States (SMMT Specifications with *ParallelStates*))**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.3). The set of execution states $\mathcal{EXS}(\texttt{M}')$ of the running example is defined as follows:

$$\begin{aligned}\mathcal{EXS}(\texttt{M}') = \{&\{\texttt{idle}\}, \{\texttt{error}, \texttt{unresolved}\}, \{\texttt{error}, \texttt{resolved}\}, \\ &\{\texttt{printing}, \texttt{preparing\_job}, \texttt{color\_correction}, \texttt{pre\_cc}, \texttt{scaling}, \texttt{pre\_scaling}\}, \\ &\{\texttt{printing}, \texttt{preparing\_job}, \texttt{color\_correction}, \texttt{pre\_cc}, \texttt{scaling}, \texttt{post\_scaling}\}, \\ &\{\texttt{printing}, \texttt{preparing\_job}, \texttt{color\_correction}, \texttt{post\_cc}, \texttt{scaling}, \texttt{pre\_scaling}\}, \\ &\{\texttt{printing}, \texttt{preparing\_job}, \texttt{color\_correction}, \texttt{post\_cc}, \texttt{scaling}, \texttt{post\_scaling}\}, \\ &\{\texttt{printing}, \texttt{printing\_job}\}\}\end{aligned}$$

Initial Execution State $\mathcal{I}(\texttt{M}')$

When defining the semantics of an SMMT specification without *ParallelStates*, we defined the initial execution state of an SMMT specification $\texttt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ as the set consisting of all entry states $s \in ES$ of which all ancestors are contained in the set of entry states.

Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\texttt{M}')$ be the execution state of SMMT specification $\texttt{M}'$. By Restriction 6, all children $s \in \mathcal{S}(\texttt{M}')$ of an active *ParallelState* $s' \in (S_P \cap EX)$ must be contained in execution state EX. By restriction 6, it follows that all children of a *ParallelState* are entry states. Therefore, the initial execution state of an SMMT specification with *ParallelStates* can be defined as the set consisting of all entry states $s \in ES$ of which all ancestors are contained in the set of entry states. Hence, the definition of the initial execution state $\mathcal{I}(\texttt{M}')$ corresponds to Definition 12, where all occurrences of $\texttt{M}$ are replaced by $\texttt{M}'$.

Example 15 (Initial Execution States (SMMT Specifications with *ParallelStates*))**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.3). The initial execution state $\mathcal{I}(\texttt{M}') \in \mathcal{EXS}(\texttt{M}')$ of SMMT specification $\texttt{M}'$ is defined as follows:

$$\mathcal{I}(\texttt{M}') = \{\texttt{idle}\}$$

Lemma 7 (Initial Execution State is an Execution State (SMMT Specifications with *Parallel-States*))**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The initial execution state $\mathcal{I}(\texttt{M}')$ is an execution state, that is:

$$\mathcal{I}(\texttt{M}') \in \mathcal{EXS}(\texttt{M}')$$

The proof of Lemma 7 can be found in Appendix A. All children $s \in \mathcal{S}(\texttt{M}')$ of a *ParallelState* $s' \in S_P$ are active when *ParallelState* s' is active. Therefore, an execution state can contain more than one *SimpleState*. Hence, Lemma 2 does not hold for SMMT specification with *ParallelStates*.

Set of Prioritised Transitions $\mathcal{PT}(\mathtt{M}', EX)$

Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification without *ParallelStates* as defined in Definition 5. When defining the set of prioritised transitions of SMMT specification $\mathtt{M}$, we defined that a transition from state $s \in \mathcal{S}(\mathtt{M})$ cannot fire if there exists a descendant $s' \in \mathcal{S}(\mathtt{M})$ of state s for which a transition is defined for the same *OnEvent*.

Likewise, a transition from a state $s \in \mathcal{S}(\mathtt{M}')$ of an SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ does not fire if there exists a descendant $s' \in \mathcal{S}(\mathtt{M}')$ of s that has a transition defined for the same *OnEvent* $e \in E$. The set of prioritised transitions corresponds to Definition 13, where each occurrence of $\mathtt{M}$ is replaced by $\mathtt{M}'$. Similarly, the set of target states $\mathcal{PT}_e(\mathtt{M}', EX)$ corresponds to the definition of $\mathcal{PT}_e(\mathtt{M}', EX)$ (Definition 13) where each occurrence of $\mathtt{M}$ is replaced by $\mathtt{M}'$.

In contrast to SMMT specifications without *ParallelStates*, an SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ can fire more than one transition when an *OnEvent* $e \in E$ is processed. With *ParallelStates*, the set of prioritised transitions $\mathcal{PT}(\mathtt{M}', EX)$ can have multiple transitions that are defined for *OnEvent* $e \in E$ if one or more *ParallelStates* $s \in S_P$ are contained in execution state EX. If a *ParallelState* $s \in S_P$ is contained in execution state EX, then all children of s must be contained in execution state EX. The set of prioritised transitions can contain an outgoing transition for the subregion of each child of a *ParallelState* $s \in S_P$ if s occurs in execution state EX. Let $s', s'' \in \mathcal{S}(\mathtt{M}')$ be two distinct children of state s. As each state can have at most one parent, it follows that a state in the subregion of s' is not related to any state in the subregion of s'' by descendant relation $\sqsubset^+$. Hence, an outgoing transition for the same *OnEvent* $e \in E$ could be contained in the set of prioritised transitions for a state in the subregion of s' and for a state in the subregion of s''. Thus, the set of prioritised transitions $\mathcal{PT}(\mathtt{M}', EX)$ can contain multiple transitions for each *OnEvent* $e \in E$.

Example 16 (Prioritised Transitions (SMMT Specifications with *ParallelStates*)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3). The set of prioritised transitions of execution state $EX = \{\mathtt{printing}, \mathtt{preparing\_job}, \mathtt{color\_correction}, \mathtt{pre\_cc}, \mathtt{scaling}, \mathtt{pre\_scaling}\}$ is defined as follows:

$$\begin{aligned}\mathcal{PT}(\mathtt{M}', EX) = \{&\langle \mathtt{ev\_finish\_color()}, \mathtt{post\_cc}\rangle,\\ &\langle \mathtt{ev\_finish\_scaling()}, \mathtt{post\_scaling}\rangle,\\ &\langle \mathtt{ev\_finish\_scaling()}, \mathtt{error}\rangle,\\ &\langle \mathtt{ev\_color\_error()}, \mathtt{error}\rangle\}\end{aligned}$$

Furthermore, we have that:

$$\begin{aligned}\mathcal{PT}_{\mathtt{ev\_finish\_color()}}(\mathtt{M}', EX) &= \{\mathtt{post\_cc}\}\\ \mathcal{PT}_{\mathtt{ev\_finish\_scaling()}}(\mathtt{M}', EX) &= \{\mathtt{post\_scaling}, \mathtt{error}\}\\ \mathcal{PT}_{\mathtt{ev\_color\_error()}}(\mathtt{M}', EX) &= \{\mathtt{error}\}\end{aligned}$$

Conflicting States

To define when an *OnEvent* is enabled for SMMT specifications with *ParallelStates*, we first introduce the notion of conflicting states.

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. In case multiple transitions fire when an *OnEvent* $e \in E$ is processed, it must be the case that all target states of the transitions that fire are allowed to be active at the same time. If this property is violated, the set consisting of all active states after handling the transitions is not an execution state.

For example, if *OnEvent* `ev_finish_scaling()` is processed in execution state $EX = \{\mathtt{printing}, \mathtt{preparing\_job}, \mathtt{color\_correction}, \mathtt{pre\_cc}, \mathtt{scaling}, \mathtt{pre\_scaling}\}$ of the running example (Figure 5.3), states `post_scaling` and `error` would become active. However, as states `post_scaling` and `error` are in the subregion of distinct root states, there does not exist an execution state in which both states post_scaling and error are contained.

We define the notion of *conflicting* states. Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. We say that two states $s, s' \in \mathcal{S}(\texttt{M}')$ are *conflicting* if, and only if, there exists no execution state $EX \in \mathcal{EXS}(\texttt{M}')$ such that states s and s' are contained in EX.

Definition 20 (Conflicting States)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. Function $\mathcal{CS}(\texttt{M}', s, s')$ determines whether states $s, s' \in \mathcal{S}(\texttt{M}')$ are *conflicting*, which is defined as follows:

$$\mathcal{CS}(\texttt{M}', s, s') = \neg\exists_{EX \in \mathcal{EXS}(\texttt{M}')} : \{s, s'\} \subseteq EX$$

Example 17 (Conflicting States)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3). As each execution state can only have one root state and all ancestors of each state in the execution state must be contained in the execution state, it follows that two states that are a descendant of distinct root states are conflicting. For example, states error and post_scaling are conflicting as they are in the subregion of distinct root states.

By the definition of an execution state (Definition 28), each execution state contains exactly one child of each *CompositeState* that is contained in the execution state. Hence, all states that are contained in subregions of distinct children of a *CompositeState* are conflicting. For example, states pre_cc and post_cc are conflicting as they are contained in the subregion of distinct children of *CompositeState* color_correction.

Examples of states that are not conflicting are states pre_cc and post_scaling, and states error and unresolved.

For each execution state $EX \in \mathcal{EXS}(\texttt{M}')$ it should hold that the target states of all prioritised transitions that fire for an *OnEvent* $e \in E$ are not conflicting with each other. We define function $\mathcal{CT}(\texttt{M}', EX, e)$ that determines whether the targets of the transitions that fire if *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\texttt{M}')$ are conflicting.

Definition 21 (Conflicting Target States)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. Function $\mathcal{CT}(\texttt{M}', EX, e)$ determines whether there exist states $s, s' \in \mathcal{PT}_e(\texttt{M}', EX)$ that are conflicting. Function $\mathcal{CT}(\texttt{M}', EX, e)$ is defined as follows:

$$\mathcal{CT}(\texttt{M}', EX, e) = \exists_{s,s' \in \mathcal{PT}_e(\texttt{M}',EX)} : \mathcal{CS}(\texttt{M}', s, s')$$

Example 18 (Conflicting Target States)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3). The set of target states of the transitions that fire when *OnEvent* ev_finish_scaling() is processed in execution state $EX = \{\texttt{printing}, \texttt{preparing\_job}, \texttt{color\_correction}, \texttt{pre\_cc}, \texttt{scaling}, \texttt{pre\_scaling}\}$ is defined by $\mathcal{PT}_e(\texttt{M}', EX)$.

$$\mathcal{PT}_{\texttt{ev\_finish\_scaling()}}(\texttt{M}', EX) = \{\texttt{post\_scaling}, \texttt{error}\}$$

As states post_scaling and error are conflicting, we therefore have that:

$$\mathcal{CT}(\texttt{M}', EX, \texttt{ev\_finish\_scaling()}) = \texttt{true}$$

By Definition 21 we have that the target states of the transitions that fire when an *OnEvent* is processed are conflicting if, and only if, there exist two target states that are conflicting. Hence, it follows that these target states are conflicting if, and only if, there exist no execution state that contains all target states.

Lemma 8 (Conflicting States)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification, $EX \in \mathcal{EXS}(\texttt{M}')$ be an execution state and $e \in E$ be an *OnEvent*. The target states in $\mathcal{PT}_e(\texttt{M}', EX)$ conflict if, and only if, there does not exist an execution state $EX' \in \mathcal{EXS}(\texttt{M}')$ such that $\mathcal{PT}_e(\texttt{M}', EX) \subseteq EX'$. Hence, it follows that:

$$\mathcal{CT}(\texttt{M}', EX, e) \equiv \neg\exists_{EX' \in \mathcal{EXS}(\texttt{M}')} : \mathcal{PT}_e(\texttt{M}', EX) \subseteq EX'$$

The proof of Lemma 8 can be found in Appendix A.

Enabled OnEvents

In Section 5.2 we defined an enabled *OnEvent* $e \in E$ of an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ in execution state $EX \in \mathcal{EXS}(\mathtt{M})$ as an *OnEvent* for which a prioritised transition is defined. That is, for which the set of target states of the prioritised transitions is non-empty. States of type *ParallelState* put two additional constraints on when an *OnEvent* is enabled.

To simplify the definition of an enabled *OnEvent* we first define the transition existence function $\mathcal{TE}(\mathtt{M}', X, e)$ that determines whether a transition is defined for *OnEvent* $e \in E$ for some state $s \in X$ where $X \subseteq \mathcal{S}(\mathtt{M}')$ of SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$.

Definition 22 (Transition Existence)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. Function $\mathcal{TE}(\mathtt{M}', X, e)$ defines whether a transition is defined for *OnEvent* $e \in E$ for some state $s \in X$ where $X \subseteq \mathcal{S}(\mathtt{M}')$, which is defined as follows:

$$\mathcal{TE}(\mathtt{M}', X, e) = \exists_{s \in X} : (\exists_{s' \in \mathcal{S}(\mathtt{M}')} : \langle e, s'\rangle \in \mathcal{T}(s))$$

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. An *OnEvent* $e \in E$ is *enabled* in execution state EX if, and only if:

- The set of prioritised transitions in execution state EX contains a prioritised transition that is defined for *OnEvent* e. Hence, the set of target states of the prioritised transitions that are defined for *OnEvent* e in execution state EX, $\mathcal{PT}_e(\mathtt{M}', EX)$, is non-empty, that is:
$$\mathcal{PT}_e(\mathtt{M}', EX) \neq \emptyset$$
- The target states of the prioritised transitions that are defined for *OnEvent* e in execution state EX are not conflicting, that is:
$$\neg\mathcal{CT}(\mathtt{M}', EX, e)$$
- For all active *ParallelStates* $s' \in S_P$ in execution state EX it holds that there exists a child $s \in \mathcal{S}(\mathtt{M}')$ of *ParallelState* s' for which a transition is defined for *OnEvent* e in the subregion of a child s'. Furthermore, for all children $s'' \in \mathcal{S}(\mathtt{M}')$ of *ParallelState* s' it must hold that a transition can fire for *OnEvent* e or that no transition is defined for *OnEvent* e, that is, either one of the following should hold:
  - There exists an active state in the subregion of child s'' for which a transition is defined for *OnEvent* e, that is:
  $$\mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s'') \cap EX, e)$$
  - There exist no state in the subregion of child s'' for which a transition is defined for *OnEvent* e, that is:
  $$\neg\mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s''), e)$$

  That is, the following should hold:
  $$\forall_{s' \in (S_P \cap EX)} : ((\exists_{s \in EX} : s \sqsubset s' \wedge \mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s), e)) \wedge$$
  $$(\forall_{s'' \in EX} : (s'' \sqsubset s' \Rightarrow (\mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s'') \cap EX, e) \vee \neg\mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s''), e)))))$$

We define function $\mathcal{E}(\mathtt{M}', EX, e)$ that determines whether an *OnEvent* $e \in E$ is *enabled* in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$.

Definition 23 (Enabled *OnEvent*). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M}')$ be the execution state of SMMT specification $\mathtt{M}'$. Function $\mathcal{E}(\mathtt{M}', EX, e)$ determines whether *OnEvent* $e \in E$ of SMMT specification $\mathtt{M}'$ is *enabled* in execution state EX, which is defined as follows:

$$\begin{aligned}
&\mathcal{E}(\mathtt{M}', EX, e)\\
&= (\mathcal{PT}_e(\mathtt{M}', EX) \neq \emptyset) \wedge \neg \mathcal{CT}(\mathtt{M}', EX, e)\\
&\quad \wedge (\forall_{s' \in (S_P \cap EX)} : (\exists_{s \in EX} : s \sqsubset s' \wedge \mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s), e))\\
&\qquad \wedge (\forall_{s'' \in EX} : (s'' \sqsubset s' \Rightarrow (\mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s'') \cap EX, e) \vee \neg \mathcal{TE}(\mathtt{M}', \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s''), e)))))
\end{aligned}$$

Example 19 (Enabled *OnEvent*). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.3). Let $EX = \{$`printing`, `preparing_job`, `color_correction`, `pre_cc`, `scaling`, `pre_scaling`$\}$ be an execution state of SMMT specification $\mathtt{M}'$. In execution state EX, *OnEvents* `ev_finished_color()` and `ev_color_error()` are enabled *OnEvents*. *OnEvent* `ev_finish_scaling()` is not enabled as the target states of the transitions that would fire are conflicting. All other *OnEvents* are not enabled as there are no transitions defined for the *OnEvent* from some state in execution state EX.

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. An internal software exception is thrown and the execution is terminated if an *OnEvent* $e \in E$ of SMMT specification $\mathtt{M}'$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$ that is not enabled, that is, $\neg\mathcal{E}(\mathtt{M}', EX, e)$.

Execution State Update $\mathcal{ESU}(\mathtt{M}', EX, e)$

In Definition 15 we defined how the execution state is updated when a transition fires in an SMMT specification without *ParallelStates*. In this section we define the execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ for SMMT specifications with *ParallelStates*.

Each execution state of an SMMT specification $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T} \rangle$ can be expressed as a single *SimpleState* and all ancestors of this state, which follows from Lemma 2. Therefore, the execution state update function has been defined as the set consisting of the target state $s' \in \mathcal{S}(\mathtt{M})$ of the transition that fires due to the occurrence of *OnEvent* e, as well as the ancestors and entry descendants of s' (Definition 15).

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification with *ParallelStates*. We cannot define the execution state update $\mathcal{ESU}(\mathtt{M}', EX, e)$ as the set consisting of all target states $s' \in \mathcal{PT}_e(\mathtt{M}', EX)$ and the ancestors and entry descendants of each target state s'. This approach would result in a set of active states X that may contain *CompositeStates* for which multiple children are contained in X and *ParallelStates* for which not all children are contained in X.

For example, consider the example in Figure 5.4. Assume that both transitions that are shown fire when *OnEvent* `a()` is processed. According to Definition 15, the execution state would consist of states `c1`, `s1` and `s2` after *OnEvent* `a()` is processed. However as state `c1` is a *CompositeState*, we have that states `s1` and `s2` are conflicting. Hence, this shows that we cannot define the execution state update as defined in Definition 15.

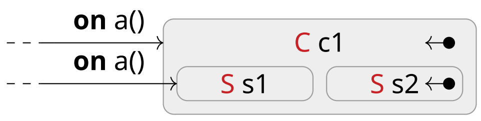


Figure 5.4: Example SMMT Specification

The execution state of an SMMT specification with *ParallelStates* after an enabled *OnEvent* $e \in E$ is processed in an execution state $EX \in \mathcal{EXS}(\mathtt{M}')$ is determined in three steps:

1. First, all states of the execution state that conflict with the target state of any transition that fires are removed from the execution state.

2. Next, the target states of the transitions that fire and the ancestors of these target states are added to the execution state.
3. Finally, the obtained execution is initiated to ensure that:
   - The execution state contains exactly one child of each *CompositeState* that is contained in the execution state.
   - The execution state contains all children of each *ParallelState* that is contained in the execution state.

In the remainder of this section we discuss these three steps in more detail and define the execution state update function that determines the execution state after an *OnEvent* $e \in E$ is processed in an execution state $EX \in \mathcal{EXS}(\mathtt{M}')$.

When an enabled *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$, the target states of all transitions that fire and the ancestors of these target states are added to the set of active states. We define the set of *entered targets* $\mathcal{ET}(\mathtt{M}', EX, e)$ to represent these states. That is, $\mathcal{ET}(\mathtt{M}', EX, e)$ consists of all states in $\mathcal{PT}_e(\mathtt{M}', EX)$ and all ancestors thereof.

Definition 24 (Entered Targets)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. The set of *entered targets* $\mathcal{ET}(\mathtt{M}', EX, e)$ consists of all target states and ancestors of the target states of the transitions that fire when *OnEvent* e is processed in execution state EX. The set of entered targets $\mathcal{ET}(\mathtt{M}', EX, e)$ is defined as follows:

$$\mathcal{ET}(\mathtt{M}', EX, e) = \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \in \mathcal{PT}_e(\mathtt{M}', EX)} : s' \in \mathcal{SR}(\mathcal{S}(\mathtt{M}'), s)\}$$

Example 20 (Entered Targets)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3). Let $EX = \{\mathtt{printing}, \mathtt{preparing\_job}, \mathtt{color\_correction}, \mathtt{pre\_cc}, \mathtt{scaling}, \mathtt{pre\_scaling}\}$ be an execution state of SMMT specification $\mathtt{M}'$. The set of entered targets when *OnEvent* `ev_color_error()` is processed in execution state EX is defined as follows:

$$\mathcal{ET}(\mathtt{M}', EX, \mathtt{ev\_color\_error()}) = \{\mathtt{error}\}$$

Before the entered targets are added to the execution state, all states $s \in EX$ that conflict with an entered target are removed from execution state EX. We define the set of *exited states* $\mathcal{XS}(\mathtt{M}', EX, e)$ as the set of all states $s \in EX$ that conflict with an entered target.

Definition 25 (Exited States)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. The set of *exited states* $\mathcal{XS}(\mathtt{M}', EX, e)$ consists of all states in execution state EX that conflict with an entered target. The set of exited states $\mathcal{EX}(\mathtt{M}', EX, e)$ is defined as follows:

$$\mathcal{EX}(\mathtt{M}', EX, e) = \{s \in EX \mid \exists_{s' \in \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', s, s')\}$$

Example 21 (Exited States)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3). Let $EX = \{\mathtt{printing}, \mathtt{preparing\_job}, \mathtt{color\_correction}, \mathtt{pre\_cc}, \mathtt{scaling}, \mathtt{pre\_scaling}\}$ be an execution state of SMMT specification $\mathtt{M}'$. The set of *exited states* when *OnEvent* `ev_color_error()` is processed is defined as follows:

$$\mathcal{XS}(\mathtt{M}', EX, \mathtt{ev\_color\_error()}) = EX$$

The set of entered targets $\mathcal{ET}(\mathtt{M}', EX, e)$ consists only of the target states and the ancestors of each target state of the prioritised transitions that fire when *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. Let X be the set of active states that is obtained after the exited states have been removed from EX and the entered targets have been added to EX. The obtained set of active states X may contain *CompositeStates* for which no child is contained in X. For example, consider the example shown in Figure 5.5. *CompositeState* `state_b` is the target state of the prioritised transitions that fires when *On-Event* ev_a is processed in execution state $\{\texttt{state\_a}\}$. We have that:

$$\begin{aligned} X &= (EX \setminus \mathcal{XS}(\texttt{M}', EX, e)) \cup \mathcal{ET}(\texttt{M}', EX, e) \\ &= (\{\texttt{state\_a}\} \setminus \{\texttt{state\_a}\}) \cup \{\texttt{state\_b}\} \\ &= \{\texttt{state\_b}\} \end{aligned}$$

Hence, no child of *CompositeState* `state_b` is contained in set X.

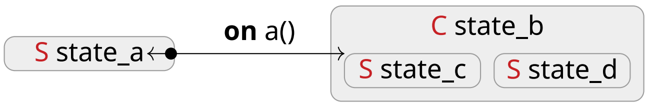


Figure 5.5: Example Missing Children

Furthermore, X may contain *ParallelStates* for which not all children are contained in X. For example, not all children $s \in \mathcal{S}(\texttt{M}')$ of a *ParallelState* $s' \in (S_P \cap X)$ are contained in X if *ParallelState* s' is contained in the set of entered targets and is not contained in execution state EX. We introduce the notion of a *missing child*.

Definition 26 (Missing Child)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. Let $X \subseteq \mathcal{S}(\texttt{M}')$ be a subset of the set of states $\mathcal{S}(\texttt{M}')$. A state $s \in \mathcal{S}(\texttt{M}')$ is a *missing child* of X if one of the following holds:

- There exists a *CompositeState* $s' \in X$ such that s is the entry child of s' and X does not contain any child of s', that is:

$$\exists_{s' \in (S_C \cap X)} : s \sqsubset_{ES} s' \wedge \neg(\exists_{s'' \in X} : s'' \sqsubset s')$$

- There exists a *ParallelState* $s'' \in X$ such that s is an entry child of s'' and s is not contained in X.

$$\exists_{s'' \in (S_P \cap X)} : s \sqsubset_{ES} s'' \wedge s \notin X$$

We define function $\mathcal{MC}(\texttt{M}', X)$ that returns the set of missing children of a set of states $X \subseteq \mathcal{S}(\texttt{M}')$, which is defined as follows:

$$\begin{aligned} \mathcal{MC}(\texttt{M}', X) = \{s \in \mathcal{S}(\texttt{M}') \mid &(\exists_{s' \in (S_C \cap X)} : s \sqsubset_{ES} s' \;\wedge\; \neg(\exists_{s'' \in X} : s'' \sqsubset s')) \\ &\vee\; (\exists_{s'' \in (S_P \cap X)} : s \sqsubset_{ES} s'' \;\wedge\; s \notin X)\} \end{aligned}$$

Furthermore, we define the set of *missing descendants*, denoted by $\mathcal{MD}(\texttt{M}', X)$ that returns the set of missing children of a set of states $X \subseteq \mathcal{S}(\texttt{M}')$ and all states that are an entry descendant of one of these missing children. That is:

$$\mathcal{MD}(\texttt{M}'X) = \{s \in \mathcal{S}(\texttt{M}') \mid \exists_{s' \in \mathcal{MC}(\texttt{M}', X)} : s \sqsubset^*_{ES} s'\}$$

We are required to initiate the set of active states X to ensure that all missing descendants are added to X. We define *initiation* function $\mathcal{INIT}(\texttt{M}', X)$ that initiates the set of active states X by adding all missing descendants of each state in X.

Definition 27 (Initiation Function)**.** Let $\texttt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification and let X be a subset of the set of states $\mathcal{S}(\texttt{M}')$. The *initiation* function $\mathcal{INIT}(\texttt{M}', X)$ returns the set consisting of all states in X and all missing descendants $s \in \mathcal{MD}(\texttt{M}', X)$. The initiation function is defined as follows:

$$\mathcal{INIT}(\texttt{M}', X) = X \cup \mathcal{MD}(\texttt{M}', X)$$

Example 22 (Initiation Function)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Figure 5.3) and let $X = \{\mathtt{error}\}$. Initiation function $\mathcal{INIT}(\mathtt{M}', X)$ adds all missing descendants $s \in \mathcal{MD}(\mathtt{M}', X)$ to X, we have that:

$$\mathcal{INIT}(\mathtt{M}', \{\mathtt{error}\}) = \{\mathtt{error}, \mathtt{unresolved}\}$$

We define the execution state update function using the set of entered targets $\mathcal{ET}(\mathtt{M}, EX, e)$ (Definition 24), the set of exited states $\mathcal{XS}(\mathtt{M}, EX, e)$ (Definition 25) and the initiation function $\mathcal{INIT}(\mathtt{M}, X)$ (Definition 27). The execution state after an enabled *OnEvent* $e \in E$ is processed is obtained by first removing the exited states from the execution state after which the entered targets are added to the execution state. Finally, we initiate the obtained set of active states to obtain the execution state EX' after *OnEvent* e has been processed.

Definition 28 (Execution State Update (SMMT Specifications with *ParallelStates*)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. The execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ defines the execution state after enabled *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$, which is defines as follows:

$$\mathcal{ESU}(\mathtt{M}', EX, e) = \mathcal{INIT}\Big(\mathtt{M}', (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{ET}(\mathtt{M}', EX, e)\Big)$$

Example 23 (Execution State Update (SMMT Specifications with *ParallelStates*)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification of the running example (Example 5.3). Let $EX = \{\mathtt{printing}, \mathtt{preparing\_job}, \mathtt{color\_correction}, \mathtt{pre\_cc}, \mathtt{scaling}, \mathtt{pre\_scaling}\}$ be the execution state of SMMT specification $\mathtt{M}'$. Function $\mathcal{ESU}(\mathtt{M}', EX, \mathtt{ev\_color\_error()})$ defines the execution state after *OnEvent* `ev_color_error()` is processed in execution state EX, which is defined as follows:

$$\begin{aligned}
&\mathcal{ESU}(\mathtt{M}', EX, \mathtt{ev\_color\_error()})\\
&\quad= \mathcal{INIT}\Big(\mathtt{M}', (EX \setminus \mathcal{XS}(\mathtt{M}', EX, \mathtt{ev\_color\_error()})) \cup \mathcal{ET}(\mathtt{M}', EX, \mathtt{ev\_color\_error()})\Big)\\
&\quad= \mathcal{INIT}\Big(\mathtt{M}', (EX \setminus EX) \cup \{\mathtt{error}\}\Big)\\
&\quad= \mathcal{INIT}(\mathtt{M}', \{\mathtt{error}\})\\
&\quad= \{\mathtt{error}, \mathtt{unresolved}\}
\end{aligned}$$

The execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ returns a set of states that must be active after *OnEvent* $e \in E$ is processed. The set of states that is returned by execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ is an execution state if *OnEvent* $e \in E$ is enabled in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$.

Lemma 9 (Execution State Update returns an Execution State (SMMT Specifications with *ParallelStates*))**.** For all SMMT specifications $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$, for each execution state $EX \in \mathcal{EXS}(\mathtt{M}')$ and for each enabled *OnEvent* $e \in E$, $\mathcal{ESU}(\mathtt{M}', EX, e)$ is an execution state, that is:

$$\mathcal{E}(\mathtt{M}', EX, e) \Rightarrow \mathcal{ESU}(\mathtt{M}', EX, e) \in \mathcal{EXS}(\mathtt{M}')$$

The proof of Lemma 9 can be found in Appendix A.

Operational Semantics

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. When an *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$ that is not enabled, an internal software exception is thrown and the execution of the SMMT specification is terminated. To denote that the execution of the SMMT specification has terminated, we introduce a *failure state* F that is reached if the execution is terminated. Failure state F contains a self-loop with action `FAIL` to indicate that a failure occurred during execution.

The execution state after an enabled *OnEvent* $e \in E$ is processed is defined by execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ (Definition 28). We define the *operational semantics* of an SMMT specification $\mathtt{M}'$ in Definition 29.

Definition 29 (Operational Semantics with *ParallelStates*)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. The *operational semantics* of SMMT specification $\mathtt{M}'$ is given by LTS $\mathtt{L} = \langle ST, L, \rightarrow, s_0 \rangle$, where:

- $ST = \mathcal{EXS}(\mathtt{M}') \cup \{\mathsf{F}\}$
- $L = E \cup \{\mathtt{FAIL}\}$
- $\rightarrow \subseteq ST \times L \times ST$ such that for all $EX, EX' \in \mathcal{EXS}(\mathtt{M}')$ and $e \in E$ we have that:
  - $EX \xrightarrow{e} EX'$ if, and only if, $\mathcal{E}(\mathtt{M}', EX, e) \wedge EX' = \mathcal{ESU}(\mathtt{M}', EX, e)$
  - $EX \xrightarrow{e} \mathsf{F}$ if, and only if, $\neg\mathcal{E}(\mathtt{M}', EX, e)$
  - $\mathsf{F} \xrightarrow{\mathtt{FAIL}} \mathsf{F}$
- $s_0 = \mathcal{I}(\mathtt{M}')$

Example 24 (Operational Semantics)**.** Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification of the running example (Figure 5.3). The operational semantics of SMMT specification $\mathtt{M}'$ is denoted using the LTS that is shown in Figure 5.6. In Figure 5.6, only the enabled *OnEvents* are shown that can be processed in each execution state. Therefore, for any execution state shown in Figure 5.6, there should be a transition to failure state F for each *OnEvent* for which no transition is shown in Figure 5.6.

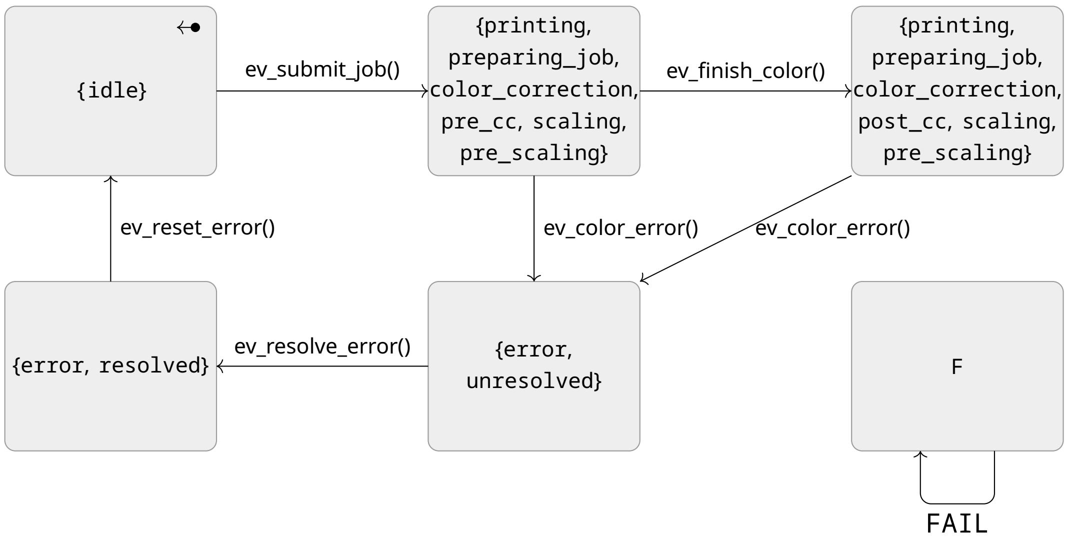

{idle}
ev_submit_job()
{printing, preparing_job, color_correction, pre_cc, scaling, pre_scaling}
ev_finish_color()
{printing, preparing_job, color_correction, post_cc, scaling, pre_scaling}
ev_reset_error()
ev_color_error()
ev_color_error()
{error, resolved}
ev_resolve_error()
{error, unresolved}
F
FAIL

Figure 5.6: Operational Semantics denoted using an LTS for the Running Example (Figure 5.3)

Chapter 6

Translating SMMT to mCRL2

In this chapter we discuss how we used the mCRL2 model checker to enhance the State Machine Modelling Tool with model checking functionality. First, we introduce the mCRL2 model checker in Section 6.1. The mCRL2 model checker is used to, among others, automatically verify properties on mCRL2 specifications. We discuss how each SMMT specification is translated to an mCRL2 specification in Section 6.2.

6.1 The mCRL2 Model Checker

In this graduation project, we used the mCRL2 model checker to enhance the State Machine Modelling Tool with model checking functionality. The mCRL2 language [1] is a formal specification language that extends the algebra of communicating processes (ACP [3]) with data. The mCRL2 language has an associated toolset [2] that is developed at the Eindhoven University of Technology.

An mCRL2 specification consists of a data specification and a process specification. In the data specification the types (sorts), constructors (defining the elements of each sort), mappings (the operations defined on sorts) and corresponding equations (the rules defining each mapping) are defined that can be used in the process specification. The process specification consist of the actions that are used in the model to denote the behavior of the modelled component. Furthermore, the process specification consist of several processes that model the behavior of the component. Using the mCRL2 toolset, all process equations can automatically be rewritten to *Linear Process Equations (LPEs)*. The structure of a Linear Process Equation, as defined in [2], is given by Definition 30.

Definition 30 (Linear Process Equation)**.** A Linear Process Equation (LPE) is of the following shape:

$$P(\bar{d}:D) = \sum_{i \in I} \text{sum } \bar{e_i} : E_i \; . \; c_i(\bar{d}, \bar{e_i}) \to a_i(\bar{f_i}(\bar{d}, \bar{e_i})) \cdot P(\bar{g_i}(\bar{d}, \bar{e_i}))$$

where P is the name of the process, I is some index set, $\bar{d}$ and $\bar{e_i}$ are vectors of variables, D and E_i are data types, c_i is a boolean expression, a_i is an action, $\bar{f_i}$ is a vector of expressions that gives values for arguments of a_i, $\bar{g_i}$ is a vector of expressions that represents the next state. Expressions $c_i, \bar{f_i}$ and $\bar{g_i}$ may depend on the variables in $\bar{d}$ and $\bar{e_i}$. We use the notation $\sum_{i \in I} p_i$ as the shorthand notation for $p_1 + p_2 + \ldots + p_n$, assuming that $I = \{1, 2, \ldots, n\}$. If I is an empty set, then $\sum_{i \in I} p_i = \delta$, where δ denotes a deadlock. Furthermore, we use the notation sum $\bar{e_i} : E_i \; . \; \phi$ to declare variables that may be used in the remainder of the process equation: ϕ.

Using the mCRL2 toolset, we can automatically convert an LPS to an LTS using the `lps2lts` tool. Furthermore, the tools `lps2pbes` and `lts2pbes` allow us to verify whether a property is satisfied for a model given the LTS and LPS respectively. These properties are defined as a mu-calculus formula. An overview of the tools for generating LTSs and to verify whether certain properties hold on a model is shown in Figure 6.1.

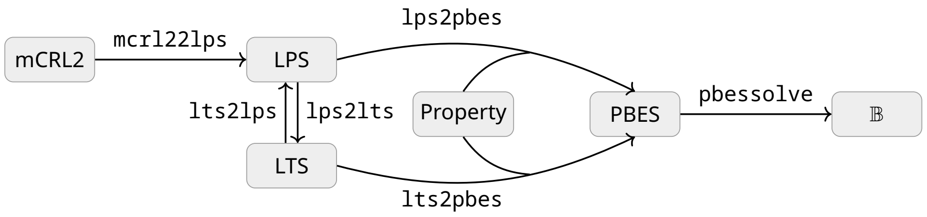


Figure 6.1: Generating LPSs and LTSs from mCRL2 specifications and checking properties defines as mu-calculus formulae using the mCRL2 toolset. The nodes show the representations used by the mCRL2 toolset, the edges are labelled with the tool that performs the transformation.

The modal mu-calculus as defined in [1] is an algebra for defining properties over an LTS that is used by the mCRL2 toolset. In this graduation project, we only consider a subset of the mu-calculus.

Definition 31 (Modal Mu-Calculus Formulae)**.** Let $\mathtt{L} = \langle ST, L, \rightarrow, s_0\rangle$ be a Labelled Transition System (LTS). The syntax of a modal mu-calculus formula ϕ is given by the following grammar:

$$\phi \coloneqq \texttt{false} \mid \texttt{true} \mid \neg\phi \mid \phi\vee\phi \mid \phi\wedge\phi \mid \phi\Rightarrow\phi \mid \langle a\rangle\phi \mid [a]\phi \mid \forall_{\bar{d}:D}.\phi \mid \exists_{\bar{d}:D}.\phi \mid \mu X.\phi \mid \nu X.\phi \mid X$$

where $a \in L$ is an action, $\bar{d}$ is a vector of data variables and D is the data type and X is a fixpoint variable. Let $L' \subseteq L$ be a set of actions and L'^{*} be a sequence of the actions of set L'. We use the following short-hand notations as defined in [1]:

$$\langle L'\rangle\phi = \bigvee_{a\,\in\,L'}\langle a\rangle\phi \qquad [L']\phi = \bigwedge_{a\,\in\,L'}[a]\phi$$
$$\langle L'^{*}\rangle\phi = \mu Z.(\langle L'\rangle Z\vee\phi) \qquad [L'^{*}]\phi = \nu Z.([L']Z\wedge\phi)$$

We write `true` instead of L' to denote the complete set of actions L according to the syntax of the μ-calculus in mCRL2.

To define the subset of states of an LTS in which a property is satisfied, we define how a modal mu-calculus formula is evaluated.

Definition 32 (Evaluation of a Modal Mu-Calculus Formula)**.** Let $\langle ST, L, \rightarrow, s_0\rangle$ be a Labelled Transition System (LTS) and $v : X \rightarrow 2^{ST}$ be a valuation where X is the set of fixpoint variables. The semantics $[\![\phi]\!]^v$ of an modal mu-calculus formula ϕ is defined as follows:

$$
\begin{aligned}
[\![\texttt{false}]\!]^v &= \emptyset \\
[\![\texttt{true}]\!]^v &= ST \\
[\![\neg\phi]\!]^v &= ST \setminus [\![\phi]\!] \\
[\![\phi_1 \wedge \phi_2]\!]^v &= [\![\phi_1]\!]^v \cap [\![\phi_2]\!]^v \\
[\![\phi_1 \vee \phi_2]\!]^v &= [\![\phi_1]\!]^v \cup [\![\phi_2]\!]^v \\
[\![\phi_1 \Rightarrow \phi_2]\!]^v &= (ST \setminus [\![\phi_1]\!]^v) \cup [\![\phi_2]\!]^v \\
[\![\langle a\rangle\phi]\!]^v &= \{s \in ST \mid \exists_{s' \,\in\, ST} : s \xrightarrow{a} s' \wedge s' \in [\![\phi]\!]^v\} \\
[\![[a]\phi]\!]^v &= \{s \in ST \mid \forall_{s' \,\in\, ST} : s \xrightarrow{a} s' \Rightarrow s' \in [\![\phi]\!]^v\} \\
[\![\forall_{\bar{d}:D}.\phi]\!]^v &= \bigcap_{e \,\in\, \mathbb{D}} [\![\phi]\!]^{v[\bar{d}:=e]} \\
[\![\exists_{\bar{d}:D}.\phi]\!]^v &= \bigcup_{e \,\in\, \mathbb{D}} [\![\phi]\!]^{v[\bar{d}:=e]} \\
[\![\mu X.\phi]\!]^v &= \bigcap_{T \,\subseteq\, ST} \{T \mid T = [\![\phi]\!]^{v[X:=T]}\} \\
[\![\nu X.\phi]\!]^v &= \bigcup_{T \,\subseteq\, ST} \{T \mid T = [\![\phi]\!]^{v[X:=T]}\} \\
[\![X]\!]^v &= v(X)
\end{aligned}
$$

where $\mathbb{D}$ denotes the set containing all values that correspond to data type D.

A state $s \in ST$ of a Labelled Transition System $(ST, L, \rightarrow, s_0)$ satisfies a modal mu-calculus formula ϕ if, and only if, $s \in [\![\phi]\!]^v$.

6.2 Translation

Using the formal definition of the SMMT language, a translation from SMMT specifications to mCRL2 specifications has been defined. As discussed before, each mCRL2 specification consists of a data and process specification that can be transformed into a Linear Process Specification (LPS). In this section we define translation `SMMT2MCRL2` that translates an SMMT specification into an mCRL2 specification. The mCRL2 specification that is generated by the `SMMT2MCRL2` translation consists of four sections:

- A representation of the mathematical model of an SMMT specification in mCRL2.
- Validation checks on the representation of the mathematical model of an SMMT specification that correspond to the restrictions discussed in Chapter 5 and the constraints of the child relation (Definition 2).
- Mappings and equations that correspond to the definitions that are used in Chapter 5 to define the operational semantics of an SMMT specification.
- The mCRL2 process specification defining the operational semantics as defined in Definition 29.

In the remainder of this section we discuss each of the sections of the mCRL2 specification in more detail. Throughout these sections, various mappings called α are defined to map the elements of an SMMT specification into their respective counterparts in the mCRL2 specification. Note that these α mappings are not mappings that are included in the mCRL2 specification.

6.2.1 Representation of the mathematical model of an SMMT Specification

The first section of the mCRL2 specification consist of a representation of the mathematical model (Definition 17) of an SMMT specification. An SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ is translated to an instance of structured sort Spec. Structured sort Spec consist of a single constructor `sm` that represents an SMMT specification in mCRL2. Figure 6.2 presents structured sort Spec in the mCRL2 language.

```
sort Spec = struct sm(
    OnEvents : List(OnEvent),
    States : LState,
    EntryStates : List(State),
    Children : State → List(State),
    Descendants : State → List(State),
    EntryDescendants : State → List(State),
    Transitions: State → List(Transition)
);
```

Figure 6.2: Structured Sort Spec

The efficiency of an mCRL2 specification depends, among others, on the number of computations that are performed when the tools of the mCRL2 toolset are applied on the mCRL2 specification. Hence, we want to avoid recomputing the same computation multiple times to increase the efficiency of the mCRL2 specification. Therefore, we add a mapping representing the descendant relation and a mapping representing the entry descendant relation. While it is possible to determine these relations from the child relation and set of entry states, pre-computing these mappings results in less computations when using the tools of the mCRL2 toolset on the mCRL2 specification. Therefore, adding the descendant and entry descendant relations as a pre-computed mapping to the mCRL2 specification results in a more efficient mCRL2 specification.

Note that an instance of type Spec uses lists to store the data instead of sets. While mCRL2 does support sets, an efficiency gain can be achieved when using lists. Furthermore, lists are more convenient when defining the equations as we can iterate over list instances. Since the ordering of the sets in the SMMT specification are irrelevant, it follows that the sets can be converted into lists without loss of information. However, we must ensure that the elements of each list remain unique.

In the remainder of this subsection, we introduce how an SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ is translated into an instance of type Spec. First, we discuss how the states and *OnEvents* of an SMMT specification are translated in the mCRL2 specification. Next, we discuss the mappings that are defined to represent the children, descendant, entry descendant and transition relation. Finally, we discuss how an SMMT specification $\mathtt{M}'$ is translated to an instance of type Spec.

6.2.1.1 *OnEvents* E and States $\mathcal{S}(\mathtt{M}')$

To represent the *OnEvents* and states of an SMMT specification $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ in the mCRL2 specification, we define structured sorts `OnEvent` and `State` that represent the *OnEvents* and states of the SMMT specification respectively.

We define mapping $\alpha_E(e)$ that for each *OnEvent* $e \in E$ returns the corresponding element in structured sort `OnEvent`. Furthermore, we define mapping $\alpha_E(E)$ that maps a set of *OnEvents* $E' \subseteq E$ into a list of elements of structured sort `OnEvent`. That is, set E' is translated into an instance of type `List(OnEvent)` that contains exactly one element $\alpha_E(e)$ of type `OnEvent` for each *OnEvent* $e \in E'$. Similarly, we define mapping functions $\alpha_S(s)$ and $\alpha_S(S)$ to map the states from the SMMT specification into the respective counterparts in the mCRL2 specification, where $S \subseteq \mathcal{S}(\mathtt{M}')$.

Let $E' = \{e_1, e_2, \ldots, e_n\} \subseteq E$ and $S = \{s_1, s_2, \ldots, s_n\} \subseteq \mathcal{S}(\mathtt{M}')$. Structured sorts OnEvent and State are defined as follows in the mCRL2 specification:

$$\begin{aligned}\mathtt{OnEvent} &= \mathtt{struct}\ \ \alpha_E(\mathsf{e}_1) \mid \alpha_E(\mathsf{e}_2) \mid \ \ldots\ \mid \alpha_E(\mathsf{e}_n);\\ \mathtt{State} &= \mathtt{struct}\ \ \alpha_S(\mathsf{s}_1) \mid \alpha_S(\mathsf{s}_2) \mid \ \ldots\ \mid \alpha_S(\mathsf{s}_n);\end{aligned}$$

Example 25 (Structured Sorts OnEvent and State). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.3. Structured sorts OnEvent and State in the mCRL2 specification that is generated for SMMT specification $\mathtt{M}'$ are defined as shown in Figure 6.3.

```
sort OnEvent = struct ev_print_job | ev_finish_job | ev_finish_color | ev_finish_scaling
                 | ev_resolve_error | ev_reset_error | ev_color_error | ev_submit_job;
     State = struct st_idle | st_error | st_unresolved | st_resolved | st_printing | st_preparing_job
                | st_color_correction | st_pre_cc | st_post_cc | st_scaling | st_pre_scaling
                | st_post_scaling | st_printing_job;
```

Figure 6.3: Structured Sorts OnEvent and State

We define a structured sort LState to maintain the partition of the set of states of SMMT specification $\mathtt{M}'$. An instance of sort LState consists of a list of elements of type State for the set of *SimpleStates*, the set of *CompositeStates* and the set of *ParallelStates*. In the mCRL2 specification, structured sort LState is defined as presented in Figure 6.4.

```
sort LState = struct states(ss : List(State), cs : List(State), ps : List(State));
```

Figure 6.4: Structured Sort LState

We define mapping function $\alpha_{LS}(\mathtt{M}')$ that translates the set of states S_S, S_C and S_P of an SMMT specification $\mathtt{M}'$ into an instance of type LState. Mapping $\alpha_{LS}(\mathtt{M}')$ is defined as follows:

$$\alpha_{LS}(\mathtt{M}') = \mathtt{states}(\alpha_S(S_S), \alpha_S(S_C), \alpha_S(S_P))$$

Example 26 (Set of States). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.3. The states of SMMT specification $\mathtt{M}'$ can be denoted using the instance of type LState that is shown in Figure 6.5.

```
states([st_idle, st_unresolved, st_resolved, st_pre_cc, st_post_cc, st_pre_scaling, st_post_scaling,
    st_printing_job], [st_error, st_printing, st_color_correction, st_scaling], [st_preparing_job])
```

Figure 6.5: Instance of type LState representing the states of SMMT specification $\mathtt{M}'$

Let ls be an instance of type LState. We define projection functions ss_s(ls), ss_c(ls) and ss_p(ls) that return the list of *SimpleStates*, list of *CompositeStates* and list of *Parallel-State* of LState instance ls respectively. The projection functions of type LState are shown in Figure 6.6.

6.2.1.2 Child Relation $\sqsubset$, Descendant Relation $\sqsubset^+$ and Entry Descendant Relation $\sqsubset^+_{ES}$

We represent the child relation of an SMMT specification $\mathtt{M}'$ in mCRL2 using mapping child_relation. Mapping child_relation maps an instance s of type State to a list of State instances consisting of all children of s. For each state $s' \in \mathcal{S}(\mathtt{M}')$ of SMMT specification $\mathtt{M}'$, we add the following equation to the mCRL2 specification:

$$\mathtt{child\_relation}(\alpha_S(s')) = \alpha_S(\{s \in \mathcal{S}(\mathtt{M}') \mid s \sqsubset s'\})$$

```
map ss_s : LState → List(State);
    ss_c : LState → List(State);
    ss_p : LState → List(State);
var ss, cs, ps : List(State);
eqn ss_s(states(ss, cs, ps)) = ss;
    ss_c(states(ss, cs, ps)) = cs;
    ss_p(states(ss, cs, ps)) = ps;
```

Figure 6.6: Projection functions for instances of type `LState`

Example 27 (Child Relation `child_relation`). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.3. Figure 6.7 shows the child relation mapping `child_relation` and corresponding equations for SMMT specification $\mathtt{M}'$.

```
map child_relation : State → List(State);
eqn child_relation(st_idle) = [];
    child_relation(st_error) = [st_unresolved, st_resolved];
    child_relation(st_unresolved) = [];
    child_relation(st_resolved) = [];
    child_relation(st_printing) = [st_preparing_job, st_printing_job];
    child_relation(st_preparing_job) = [st_color_correction, st_scaling];
    child_relation(st_color_correction) = [st_pre_cc, st_post_cc];
    child_relation(st_pre_cc) = [];
    child_relation(st_post_cc) = [];
    child_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
    child_relation(st_pre_scaling) = [];
    child_relation(st_post_scaling) = [];
    child_relation(st_printing_job) = [];
```

Figure 6.7: Child Relation Mapping `child_relation`

Similarly, we add a mapping to define the descendant relation and entry descendant relation of SMMT specification $\mathtt{M}'$, `desc_relation` and `entry_desc_relation` respectively. Both mappings map each instance s of type `State` to an instance of type `List(State)` representing the descendants and entry descendants respectively. For each state $s' \in \mathcal{S}(\mathtt{M}')$ of SMMT specification $\mathtt{M}'$, we add the following equation to the mCRL2 specification:

$$\begin{aligned}\texttt{desc\_relation}(\alpha_S(s')) &= \alpha_S(\{s \in \mathcal{S}(\mathtt{M}') \mid s \sqsubset^+ s'\})\\ \texttt{entry\_desc\_relation}(\alpha_S(s')) &= \alpha_S(\{s \in \mathcal{S}(\mathtt{M}') \mid s \sqsubset^+_{ES} s'\})\end{aligned}$$

Example 28 (Descendant Relation `desc_relation` and Entry Descendant Relation `entry_desc_relation`). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.3. The descendant relation mapping `desc_relation`, entry descendant relation mapping `entry_desc_relation` and their corresponding equations that are defined for SMMT specification $\mathtt{M}'$ are shown in Figures 6.8 and 6.9 respectively.

6.2.1.3 Transition Relation $\mathcal{T}$

We represent the transition relation of an SMMT specification $\mathtt{M}'$ in mCRL2 using mapping `transition_relation`. Mapping `transition_relation` maps an instance s of type `State` to a list of `Transition` instances consisting of all outgoing transitions of state s.

Each instance of type `Transition` has two arguments: onEvent of type `OnEvent` specifying the *OnEvent* for which the transition is defined and `target` of type `State` that specifies the target state of the transition. Figure 6.10 presents structured sort `Transition` in the mCRL2 language.

We define mapping function $\alpha_T(\mathbb{T})$ that translates a list of transitions $\mathbb{T}$ into a list of `Transition` instances in mCRL2. That is, let $\mathcal{T}(s) = [\langle e_1, s'_1\rangle, \ldots, \langle e_n, s'_n\rangle]$ be the list of tran-

```
map desc_relation : State → List(State);
eqn desc_relation(st_idle) = [];
    desc_relation(st_error) = [st_unresolved, st_resolved];
    desc_relation(st_unresolved) = [];
    desc_relation(st_resolved) = [];
    desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_post_cc,
                                  st_scaling, st_pre_scaling, st_post_scaling, st_printing_job];
    desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_post_cc, st_scaling,
                                       st_pre_scaling, st_post_scaling];
    desc_relation(st_color_correction) = [st_pre_cc, st_post_cc];
    desc_relation(st_pre_cc) = [];
    desc_relation(st_post_cc) = [];
    desc_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
    desc_relation(st_pre_scaling) = [];
    desc_relation(st_post_scaling) = [];
    desc_relation(st_printing_job) = [];
```

Figure 6.8: Mapping `desc_relation`

```
map entry_desc_relation : State → List(State);
eqn entry_desc_relation(st_idle) = [];
    entry_desc_relation(st_error) = [st_unresolved];
    entry_desc_relation(st_unresolved) = [];
    entry_desc_relation(st_resolved) = [];
    entry_desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
                                        st_pre_scaling];
    entry_desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_scaling,
                                             st_pre_scaling];
    entry_desc_relation(st_color_correction) = [st_pre_cc];
    entry_desc_relation(st_pre_cc) = [];
    entry_desc_relation(st_post_cc) = [];
    entry_desc_relation(st_scaling) = [st_pre_scaling];
    entry_desc_relation(st_pre_scaling) = [];
    entry_desc_relation(st_post_scaling) = [];
    entry_desc_relation(st_printing_job) = [];
```

Figure 6.9: Mapping `entry_desc_relation`

```
sort Transition = struct tra(onEvent : OnEvent, target : State);
```

Figure 6.10: Structured Sort `Transition`

sitions of an SMMT specification $\mathtt{M}'$ that are defined for state $s \in \mathcal{S}(\mathtt{M}')$. In the mCRL2 specification, list $\mathcal{T}(s)$ is translated into the following instance of type `List(Transition)`:

$$\alpha_T(\mathcal{T}(s)) = [\mathtt{tra}(\alpha_E(e_1), \alpha_S(s'_1)), \ldots, \mathtt{tra}(\alpha_E(e_n), \alpha_S(s'_n))]$$

For each state $s \in \mathcal{S}(\mathtt{M}')$ of SMMT specification $\mathtt{M}'$, we add the following equation to the mCRL2 specification to define the outgoing transitions of s:

$$\mathtt{transition\_relation}(\alpha_S(s)) = \alpha_T(\mathcal{T}(s))$$

Example 29 (Transition Relation `transition_relation`). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification as shown in Figure 5.3. Figure 6.11 shows mapping `transition_relation` and corresponding equations that are defined for SMMT specification $\mathtt{M}'$.

Let `t` be an instance of type `Transition`. We define projection functions `tra_o(t)` and `tra_s(t)` that return the *OnEvent* and target state of `Transition` instance `t` respectively. The projection functions of type `Transition` are shown in Figure 6.12.

```
map transition_relation : State → List(Transition);
eqn transition_relation(st_idle) = [tra(ev_submit_job, [], st_printing)];
    transition_relation(st_error) = [];
    transition_relation(st_unresolved) = [tra(ev_resolve_error, [], st_resolved)];
    transition_relation(st_resolved) = [tra(ev_reset_error, [], st_idle)];
    transition_relation(st_printing) = [];
    transition_relation(st_preparing_job) = [];
    transition_relation(st_color_correction) = [tra(ev_color_error, [], st_error)];
    transition_relation(st_pre_cc) = [tra(ev_finish_color, [], st_post_cc),
                                      tra(ev_finish_scaling, [], st_error)];
    transition_relation(st_post_cc) = [tra(ev_print_job, [], st_printing_job)];
    transition_relation(st_scaling) = [];
    transition_relation(st_pre_scaling) = [tra(ev_finish_scaling, [], st_post_scaling)];
    transition_relation(st_post_scaling) = [tra(ev_print_job, [], st_printing_job)];
    transition_relation(st_printing_job) = [tra(ev_finish_job, [], st_idle)];
```

Figure 6.11: Mapping `transition_relation`

```
map tra_o : Transition → OnEvent;
    tra_d : Transition → List(DoEvent);
var e : OnEvent;
    s : State;
eqn tra_o(tra(e, s)) = e;
    tra_s(tra(e, s)) = s;
```

Figure 6.12: Projection functions for instances of type `Transition`

6.2.1.4 Representing an SMMT Specification in mCRL2

Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. As mentioned before, an SMMT specification $\mathtt{M}'$ is represented in the mCRL2 specification as an instance of type Spec. We define mapping $\alpha_M(\mathtt{M}')$ that returns an instance of type Spec representing SMMT specification $\mathtt{M}'$, which is defined as follows:

$$\alpha_M(\mathtt{M}') = \mathtt{sm}\Big(\alpha_E(E), \alpha_{LS}(\mathtt{M}'), \alpha_S(ES), \mathtt{child\_relation}, \mathtt{desc\_relation}, \mathtt{entry\_desc\_relation}, \mathtt{transition\_relation}\Big),$$

where mapping functions `child_relation`, `desc_relation`, `entry_desc_relation` and `transition_relation` are constructed as discussed in their respective paragraphs in Section 6.2.1. In the mCRL2 specification we add mapping `smmt_spec` that returns the Spec instance representing the SMMT specification.

Example 30 (Representation of an SMMT specification in mCRL2). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be the SMMT specification as shown in Figure 5.3. In Figure 6.13 mapping `smmt_spec` and corresponding equation is defined that returns the instance of type Spec representing SMMT specification $\mathtt{M}'$.

Let sp be an instance of type Spec. We define projection functions `sm_o(sp)`, `sm_s(sp)`, `sm_ss(sp)`, `sm_es(sp)`, `sm_cr(sp)`, `sm_dr(sp)`, `sm_edr(sp)` and `sm_tr(sp)`. The projection functions of type Spec are shown in Figure 6.14. These projection functions return the list of *OnEvents*, list of all states, instance of type `LState`, the list of entry states, the child relation, the descendant relation, the entry descendant relation and the transition relation of instance sp respectively.

```
map smmt_spec : Spec;
eqn smmt_spec = sm(
        [ev_print_job, ev_finish_job, ev_finish_color, ev_finish_scaling, ev_resolve_error,
         ev_reset_error, ev_color_error, ev_submit_job],
        states(
            [st_idle, st_unresolved, st_resolved, st_pre_cc, st_post_cc, st_pre_scaling,
             st_post_scaling, st_printing_job],
            [st_error, st_printing, st_color_correction, st_scaling],
            [st_preparing_job]
        ),
        [st_idle, st_unresolved, st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
         st_pre_scaling],
        child_relation,       % Figure 6.7
        desc_relation,        % Figure 6.8
        entry_desc_relation,  % Figure 6.9
        transition_relation   % Figure 6.11
  );
```

Figure 6.13: mCRL2 Representation of SMMT Specification M as given in Figure 5.3

```
map sm_o : Spec → List(OnEvent);                    % List of OnEvents
    sm_s : Spec → List(State);                      % List of all states
    sm_ss : Spec → LState;                          % SimpleStates, CompositeStates and ParallelStates
    sm_es : Spec → List(State);                     % Entry state relation
    sm_cr : Spec → State → List(State);             % Child relation
    sm_dr : Spec → State → List(State);             % Descendant state relation
    sm_edr : Spec → State → List(State);            % Entry descendant relation
    sm_tr : Spec → State → List(Transition);        % Transition relation
var lo : List(OnEvent);
    s : LState;
    ls : List(State);
    cr, dr, edr : State → List(State);
    tr : State → List(Transition);
eqn sm_o(sm(lo, s, ls, cr, dr, edr, tr)) = lo;
    sm_d(sm(lo, s, ls, cr, dr, edr, tr)) = ld;
    sm_s(sm(lo, s, ls, cr, dr, edr, tr)) = ss_s(s) ++ ss_c(s) ++ ss_p(s);
    sm_ss(sm(lo, s, ls, cr, dr, edr, tr)) = s;
    sm_es(sm(lo, s, ls, cr, dr, edr, tr)) = ls;
    sm_cr(sm(lo, s, ls, cr, dr, edr, tr)) = cr;
    sm_dr(sm(lo, s, ls, cr, dr, edr, tr)) = dr;
    sm_edr(sm(lo, s, ls, cr, dr, edr, tr)) = edr;
    sm_tr(sm(lo, s, ls, cr, dr, edr, tr)) = tr;
```

Figure 6.14: Projection functions for instances of type Spec

6.2.2 Validation Checks on the SMMT Specification

In Chapter 5 we discussed the restrictions that must hold for the SMMT specifications. To ensure that these restrictions are not violated, we include validation checks in the mCRL2 specification. Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification and $\mathtt{sp} = \alpha_M(\mathtt{M}')$ be the instance of type Spec representing SMMT specification $\mathtt{M}'$. Table 6.1 shows how the restrictions from Section 5 are translated into validation checks in mCRL2.

Restriction	ID	Validation Check (mCRL2)
1	1	`val_non_empty_states(sp)`
2	2	`val_one_entry_root_state(sp)`
3	3	`val_cs_one_entry_child(sp)`
4	4	`val_ss_no_children(sp)`
5	5	`val_transition(sp)`
6	6	`val_ps_entry_children(sp)`
7	7	`val_ps_atleast_two_children(sp)`

Table 6.1: Translation of Restrictions to Mappings in mCRL2

Furthermore, we check whether the two constraints on the child relation $\sqsubset$ of the SMMT specification hold. Table 6.2 shows how the constraints from Definition 2 are translated into validation checks in mCRL2.

Constraint (Definition 2)	ID	Validation Check (mCRL2)
1	8	`val_child_relation_1_parent(sp)`
2	9	`val_child_relation_acyclic(sp)`

Table 6.2: Translation of Child Relation Constraints to Mappings in mCRL2

Each validation check mapping returns a list that is empty if the restriction/constraint is not violated. In case the restriction/constraint is violated, then the validation check returns a list consisting of the ID of the violated restriction/constraint as provided in Tables 6.1 and 6.2. The implementation of all validation checks can be found in Appendix B.

Example 31 (Validation Check `val_ps_entry_children(sp)`)**.** We determine whether Restriction 6 is violated using validation check `val_ps_entry_children(sp)` as shown in Figure 6.15. By Restriction 6 it must hold that all children of a *ParallelStates* are entry states. Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be an SMMT specification. Hence, it must hold that:

$$\underbrace{\forall_{s' \in S_P}}_{5} : (\underbrace{\forall_{s \in \mathcal{S}(\mathtt{M}')}}_{6} : \underbrace{s \sqsubset s'}_{7} \Rightarrow \underbrace{s \in ES)}_{8}$$

where the numbers below the braces correspond with the line numbers of mapping `val_ps_entry_children(sp)` in Figure 6.15.

To check whether the restrictions and constraints hold, we define a mapping `is_well_defined` that given an instance of type Spec returns the restrictions and constraints that are violated. The `is_well_defined` mapping joins the results of each validation check. Therefore, we obtain a list with the IDs of all restrictions and constraints that were violated. Mapping `is_well_defined` is defined as shown in Figure 6.16.

```
map val_ps_entry_children : Spec → List(Nat);
var sp : Spec;
eqn val_ps_entry_children(sp) = if(
    forall s, s' : State . (
        s' in ss_p(sm_ss(sp))          % s' ∈ S_P
        && s in sm_s(sp)               % s ∈ S(M')
        && s in sm_cr(sp)(s')          % s ⊏ s'
    ) ⇒ s in sm_es(sp),                % s ∈ ES
    [], [6]);
```

Figure 6.15: Validation Check `val_ps_entry_children(sp)`

```
map is_well_defined : Spec → List(Nat);
var sp : Spec;
eqn is_well_defined(sp) = val_non_empty_states(sp) ++ val_one_entry_root_state(sp)
        ++ val_cs_one_entry_child(sp) ++ val_ps_entry_children(sp) ++ val_ss_no_children(sp)
        ++ val_ps_atleast_two_children(sp) ++ val_transition(sp) ++ val_child_rel_1_parent(sp)
        ++ val_child_rel_acyclic(sp);
```

Figure 6.16: Mapping `is_well_defined`

6.2.3 Translation of Definitions to Mappings

Given the instance of type Spec that represents the mathematical model of an SMMT specification in mCRL2, we define mappings and equations in mCRL2 that correspond to the definitions that are defined in Chapter 5. These mappings and equations are used to define the operational semantics of an SMMT specification. Table 6.3 shows for each function as defined in Chapter 5 the mapping that corresponds to that function.

Def.	Function	Function mCRL2 Specification
12	$\mathcal{I}(\mathtt{M}')$	`init_state`($\alpha_M(\mathtt{M}')$)
13	$\mathcal{PT}(\mathtt{M}', EX)$	`get_prio_tr_event`($\alpha_M(\mathtt{M}'), \alpha_s(EX), \alpha_E(e)$)
	$\mathcal{PT}_e(\mathtt{M}', EX)$	`get_targets`($\mathcal{PT}(\mathtt{M}', EX)$)
20	$\mathcal{CS}(\mathtt{M}', s, s')$	`css`($\alpha_M(\mathtt{M}'), \alpha_s(s), \alpha_s(s')$)
21	$\mathcal{CTS}(\mathtt{M}', EX, e)$	`cts`($\alpha_M(\mathtt{M}'), \alpha_s(\mathcal{PT}_e(\mathtt{M}', EX))$)
23	$\mathcal{E}(\mathtt{M}', EX, e)$	`enabled`($\alpha_M(\mathtt{M}'), \alpha_s(EX), \alpha_E(e)$)
24	$\mathcal{ET}(\mathtt{M}', EX, e)$	`get_et`($\alpha_M(\mathtt{M}'), \alpha_T(T)$)
25	$\mathcal{XS}(\mathtt{M}', EX, e)$	`get_xs`($\alpha_M(\mathtt{M}'), \alpha_s(EX), \alpha_T(T)$)
27	$\mathcal{INIT}(\mathtt{M}', EX, e)$	`initiate`($\alpha_M(\mathtt{M}'), \alpha_s(X)$)
28	$\mathcal{ESU}(\mathtt{M}', EX, e)$	`esu`($\alpha_M(\mathtt{M}'), \alpha_s(EX), \alpha_T(T)$)

Table 6.3: Translation of Functions to Mappings in mCRL2

As mentioned before, the mCRL2 specification uses lists instead of sets, therefore the mappings iterate over lists instead of using set comprehension as defined in their respective definitions in Chapter 5. Most equations that have been defined in the mCRL2 specification consist of one or more helper functions that either increase the efficiency of the equations or improve readability of the mCRL2 specification.

For example, consider mapping `get_ax` that has been defined to determine the set of exited states as defined in Definition 25. The implementation of mapping `get_xs` is given in Figure 6.17. According to Definition 25, set $\mathcal{XS}(\mathtt{M}', EX, e)$ consist of all states that conflict

```
map get_xs : Spec # List(State) # List(Transition) → List(State);
var sp : Spec;
    s : State;
    ls : List(State);
    lt : List(Transition);
eqn get_xs(sp, [], lt) = [];
    xs_h(sp, s, lt) → get_xs(sp, s |> ls, lt) = [s] ++ get_xs(sp, ls, lt);
    !xs_h(sp, s, lt) → get_xs(sp, s |> ls, lt) = get_xs(sp, ls, lt);

map xs_h : Spec # State # List(Transition) → Bool;
var sp : Spec;
    s : State;
    t : Transition;
    lt : List(Transition);
eqn xs_h(sp, s, []) = false;
    css(sp, s, tra_s(t)) → xs_h(sp, s, t |> lt) = true;
    !css(sp, s, tra_s(t)) → xs_h(sp, s, t |> lt) = xs_h(sp, s, lt);
```

Figure 6.17: Mapping `get_ax`

with a state from the set of entered targets when *OnEvent* $e \in E$ is processed in execution state $EX \in \mathcal{EXS}(\mathtt{M}')$. Mapping `get_xs` takes as input the specification `sp`, the set of states `ls` of the specification `sp` and the list of transitions that will fire `lt`. For each state in set `ls`, the mapping checks using helper mapping `xs_h` whether the state conflicts with the target state of any of the transitions in list `lt`. The resulting list of states that is returned by mapping `get_xs` consists of all states that are conflicting with a state of the set of entered targets.

As shown in Table 6.3, functions $\mathcal{XS}(\mathtt{M}, EX, e)$, $\mathcal{ET}(\mathtt{M}, EX, e)$ and $\mathcal{ESU}(\mathtt{M}, EX, e)$ are represented by mappings `get_xs`, `get_et` and `esu` in the mCRL2 specifications respectively. Note that, the arguments of these functions and their corresponding mappings in the mCRL2 according to Table 6.3 do not correspond. When evaluating $\mathcal{ESU}(\mathtt{M}, EX, e)$, the set of prioritised transitions $\mathcal{PT}_e(EX)$ is calculated two times: when evaluating $\mathcal{XS}(\mathtt{M}, EX, e)$ and when evaluating $\mathcal{ET}(\mathtt{M}, EX, e)$. To save computations, the `esu` mapping in the mCRL2 specification does not take an `OnEvent` as third argument, but instead has the set of transitions that fire as the third argument. This allows us to forward the list of transitions as the last argument of both mappings `get_xs` and `get_et`, rather than passing the *OnEvent*. In case of the `get_et` mapping, we are no longer required to pass through the execution state as the set of prioritised transitions is already provided as an argument of the mapping.

6.2.4 mCRL2 Process Specification

To define the operational semantics of an SMMT specification $\mathtt{M}'$ in mCRL2, we define a process that corresponds to the LTS defined in Definition 29. As we defined the translations of an SMMT specification $\mathtt{M}'$ to an instance of type `Spec` and defined the translation of each definition that is used by Definition 29, we can define the process specification as follows:

$$
\begin{aligned}
&\mathtt{SM(sp : Spec,\ ex : List(State))} = \\
&\quad \textstyle\sum_{e \in E}(\mathtt{enabled(sp,\ ex,\ }\alpha_E(e)) \\
&\qquad \rightarrow \alpha_E(e).\mathtt{SM(sp,\ esu(sp,\ ex,\ get\_prio\_tr\_ev(sp,\ ex,\ }\alpha_E(e)))) \\
&\qquad <> \alpha_E(e).\mathtt{F());}
\end{aligned}
$$

Where process F is defined as follows: $\mathtt{F} = \mathtt{FAIL.F}()$. Process `SM` has two parameters: an instance `sp` of type `Spec` representing the SMMT specification and a list of states `ex` to maintain the execution state. The process allows the actions that are defined for each *OnEvent* to be performed. If the *OnEvent* of which the action was performed is enabled, then the execution state is updated according to the execution state update function. If the *OnEvent* is not enabled, we reach the failure state.

We extend process SM such that the behavior as defined by the operational semantics of the SMMT specification is only performed in case the SMMT does not violate any validation checks. We define an action for each validation check. In case a validation check is not satisfied, we only allow the process to perform the actions that correspond to the violated validation checks. After all violated restrictions and constraints have been handled we reach a deadlock. Process SM is extended as follows:

```
SM(sp : Spec, ex : List(State), valid : List(Nat)) =
  (valid = []) → (
    $\sum_{e \in E}$(
      (enabled(sp, ex, $\alpha_E(e)$)
        → $\alpha_E(e)$.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, $\alpha_E(e)$)))
        <> $\alpha_E(e)$.F())
  ) <> (
    (head(valid) = 1) → val_non_empty_states.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 2) → val_one_entry_root_state.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 3) → val_cs_one_entry_child.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 4) → val_ss_no_children.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 5) → val_transition.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 6) → val_ps_entry_children.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 7) → val_ps_atleast_two_children.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 8) → val_child_relation_1_parent.SM(sp, ex, tail(valid) ++ [0])
     + (head(valid) = 9) → val_child_relation_acyclic.SM(sp, ex, tail(valid) ++ [0])
  )
```

In this process, we add 0 to the list of violated validation checks that were not satisfied whenever an action is performed that corresponds to a validation check. As no action can fire when the head of list valid is equal to 0, this ensure that a deadlock is reached after all actions that correspond to the violated validation checks have been performed.

Process SM is initialised using mappings smmt_spec, init_state(sp) and is_well_defined(sp) as defined in Sections 6.2.1.4, 6.2.2 and 6.2.3 respectively. The initial state of process SM is defined as follows:

```
init SM(smmt_spec, init_state(smmt_spec), is_well_defined(smmt_spec));
```

We define an action for each *OnEvent* $e \in E$ of SMMT specification M′. That is, we define $\alpha_E(e)$ as action for each *OnEvent* $e \in E$.

Example 32 (Process Specification). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T} \rangle$ be the SMMT specification as shown in Figure 5.3. In Figure 6.18 process specification SM is defined that corresponds to the operational semantics of SMMT specification M′. The mCRL2 specification that is generated from the SMMT specification of Figure 5.3 by translation SMMT2MCRL2 can be found in Appendix B.

```
act FAIL, ev_print_job, ev_finish_job, ev_finish_color, ev_finish_scaling, ev_resolve_error,
     ev_reset_error, ev_color_error, ev_submit_job, val_non_empty_states, val_one_entry_root_state,
     val_cs_one_entry_child, val_ss_no_children, val_transition, val_ps_entry_children,
     val_ps_atleast_two_children, val_child_rel_1_parent, val_child_rel_acyclic;

proc SM(sp : Spec, ex : List(State), valid : List(Nat)) =
  (valid == []) → (
    (enabled(sp, ex, ev_print_job)
      → ev_print_job.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_print_job)), valid)
       <> ev_print_job.F())
    + (enabled(sp, ex, ev_finish_job)
      → ev_finish_job.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_finish_job)), valid)
      <> ev_finish_job.F())
    + (enabled(sp, ex, ev_finish_color)
      → ev_finish_color.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_finish_color)), valid)
      <> ev_finish_color.F())
    + (enabled(sp, ex, ev_finish_scaling)
      → ev_finish_scaling.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_finish_scaling)), valid)
        <> ev_finish_scaling.F())
    + (enabled(sp, ex, ev_resolve_error)
      → ev_resolve_error.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_resolve_error)), valid)
      <> ev_resolve_error.F())
    + (enabled(sp, ex, ev_reset_error)
      → ev_reset_error.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_reset_error)), valid)
      <> ev_reset_error.F())
    + (enabled(sp, ex, ev_color_error)
      → ev_color_error.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_color_error)), valid)
      <> ev_color_error.F())
    + (enabled(sp, ex, ev_submit_job)
      → ev_submit_job.SM(sp, esu(sp, ex, get_prio_tr_ev(sp, ex, ev_submit_job)), valid)
      <> ev_submit_job.F())
  ) <> (
    (head(valid) == 1) → val_non_empty_states.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 2) → val_one_entry_root_state.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 3) → val_cs_one_entry_child.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 4) → val_ss_no_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 5) → val_transition.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 6) → val_ps_entry_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 7) → val_ps_atleast_two_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 8) → val_child_rel_1_parent.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 9) → val_child_rel_acyclic.SM(sp, ex, tail(valid) ++ [0])
  );

proc F() = FAIL.F();

init SM(smmt_spec, init_state(smmt_spec), is_well_defined(smmt_spec));
```

Figure 6.18: mCRL2 Process Specification SM

Chapter 7

Experiments

In this chapter we discuss the experiments that have been performed to verify the correctness of the SMMT2MCRL2 translation and verify the correctness of the existing SMMT specifications. In addition to the translation defined in Chapter 6, we have formally defined the syntax and semantics of the complete set of constructs of SMMT in mCRL2. This formalisation excludes *SelfPost* and *Forward BehavioralActions*. The extended SMMT2MCRL2 translation allows us to run experiments on the 26 SMMT models that exist at the time of writing. When referring to the SMMT2MCRL2 translation in the remainder of this report, we refer to this extended SMMT2MCRL2 translation. The mCRL2 specification that has been generated by the SMMT2MCRL2 translation for the SMMT specification in Figure 5.3 can be found in Appendix C.

We discuss the test procedure that has been used to verify the correctness of the translation based on the existing SMMT specifications at Canon Production Printing in Section 7.1. We have verified several properties on the mCRL2 specifications that were generated using the SMMT2MCLR2 translation. We discuss the properties that have been verified on the mCRL2 specifications in Section 7.2.

7.1 Correctness of the SMMT2MCRL2 Translation

We discuss the approach that was used to verify the correctness of the SMMT2MCRL2 translation in Section 7.1.1. In Section 7.1.2 we discuss the results of the experiments that were performed to verify the correctness of the SMMT2MCRL2 translation.

7.1.1 Verification Approach

To verify the correctness of the SMMT2MCRL2 translation, we verify whether the behavior of the mCRL2 specifications as generated by the SMMT2MCRL2 translation corresponds to the behavior of the generated SCM C++ code. The SMMT2MCRL2 translation is said to be correct if the behavior of the SCM C++ code is strongly bisimilar to the behavior of the respective mCRL2 specification that is generated by translation SMMT2MCRL2 for all possible SMMT specifications. Strong bisimulation is defined below:

Definition 33 (Strong Bisimulation [1])**.** Let $(ST, L, \rightarrow, s_0)$ be a labelled transition system. A binary relation $R \subseteq ST \times ST$ is called a *strong bisimulation relation* if, and only if, for all $s, t \in ST$ such that $s\ R\ t$ holds, it also holds for all actions $a \in L$ that:

- if $s \xrightarrow{a} s'$, then there is a $t' \in ST$ such that $t \xrightarrow{a} t'$ with $s'\ R\ t'$
- if $t \xrightarrow{a} t'$, then there is an $s' \in ST$ such that $s \xrightarrow{a} s'$ with $s'\ R\ t'$

We say that two labelled transition systems are strongly bisimilar if, and only if, there exists a strong bisimulation relation that relates the initial states of the two systems. If the two labelled transition systems are strongly bisimilar, then this means that every action that is performed in one of the labelled transition systems can be matched by the same action in the other labelled transition system and vice versa.

To analyse the behavior of the executable SCM C++ code, we define a translation SCM2MCRL2 that generates an mCRL2 specification from the executable SCM C++ code. Using the SCM2MCRL2 translation, we check the correctness of the SMMT2MCRL2 translation by verifying whether the mCRL2 specifications that are obtained by both translations have the same behavior. That is, we convert the mCRL2 specifications that are obtained by the two translations for each SMMT specification to labelled transition systems and evaluate whether the two LTSs that are obtained for each SMMT specification are strongly bisimilar using the `ltscompare` tool [23] of the mCRL2 toolset. An overview of the testing procedure is shown in Figure 7.1.

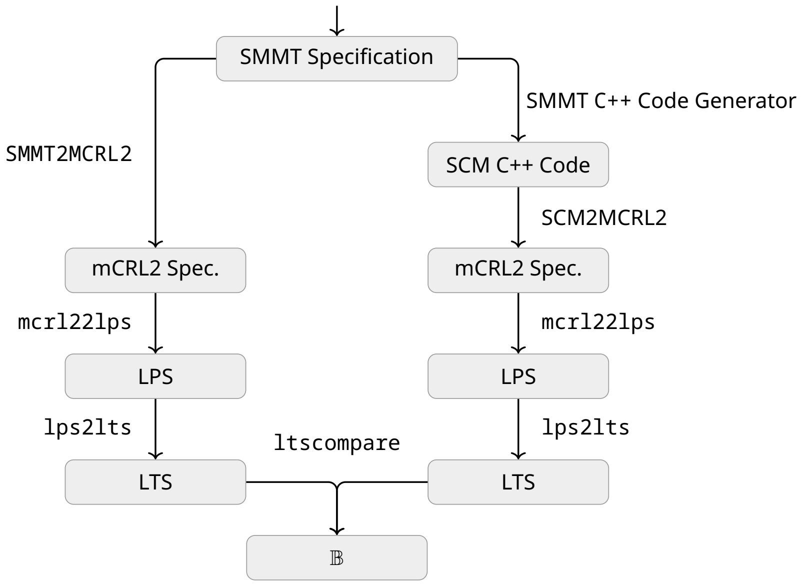


Figure 7.1: Overview of Testing Procedure

In the remainder of this section we go into more detail on the SCM2MCRL2 translation (Section 7.1.1.1) and the tools of the mCRL2 toolset that are used to determine whether the behavior of the mCRL2 specifications that are generated by the two translations are strongly bisimilar (Section 7.1.1.2). We discuss how the testing procedure has been automated using Python scripts in Section 7.1.1.3.

7.1.1.1 SCM2MCRL2

To generate an mCRL2 specification that models the behavior of the executable SCM C++ code, we first analyse the generated SCM C++ code. We generate a C++ generator to explore the state space of the generated SCM C++ code. That is, we determine for each reachable execution state what *DoEvents* are produced and which execution state is reached after each *OnEvent* is processed. Finally, we convert the explored state space into an mCRL2 specification. This procedure has been automated using Python. We discuss the SCM2MCRL2 translation in the remainder of this section.

Using Python scripts, the generated SCM C++ files are analysed to obtain the states, *OnEvents*, *DoEvents*, transitions, entry handlers, exit handlers and custom types that are defined for the SMMT specification. The Python script performing the SCM2MCRL2 translation generates two files:

- `x_state_machine_custom_types.h`, defining the custom types that are used in SMMT specification, where x denotes the name of the SMMT specification;
- `x_state_machine_generator.cpp`, a script to explore the state space of the SMMT specification and generate an mCRL2 specification for this SMMT specification, where x denotes the name of the SMMT specification.

In the remainder of this section we discuss how the above-mentioned C++ files are generated and how the C++ files are compiled and run to generate the mCRL2 specification that models the behavior of the executable SCM C++ code.

Custom Types The `SCM2MCRL2` translation generates a C++ header file to define the custom types of the SMMT specification that is translated: `x_state_machine_custom_types.h`, where x denotes the name of the SMMT specification. This header file contains the definitions of custom types that are not basic types that are implemented in C++. All custom types that occur in the SMMT specification correspond to the following format: `namespaceName::className`, as shown in Figure 7.2.

```
#pragma once
#include <string>

namespace namespaceName {
  class className {
  public:
    std::string val;
  };
}
```

Figure 7.2: Definition of Custom Type `namespaceName::className`

As mentioned before, SMMT contains no comparison operators that can be used to compare the values of custom typed arguments in guards. As the custom typed arguments of an *OnEvent* e cannot be used in the guards of a transition, it follows that the custom typed arguments can only be used in the *DoEvents* that are produced as a response to *OnEvent* e. We define each custom type as a class consisting of a single variable `val` of type `std::string` for which only a default value `"a"` is used during the exploration of the state space.

State Space Exploration The generated executable SCM C++ code allows us to interact with the state machine by, among others, sending *OnEvents* and checking the state of the state machine. In Figure 7.3 a C++ class is shown that can be used to interact with the generated SCM C++ code of an SMMT specification called `test_model`.

Two methods that can be used to interact with a state machine instance are the `process<OnEvent>()` and `isInState<State>()` methods that respectively send an *OnEvent* to the state machine instance and check whether a state is active in the current execution state of the state machine instance. Using these two methods, we can explore the state space of the SMMT specification from the generated SCM C++ code.

To generate the state space of the generated SCM C++ code, a depth-first search based algorithm is used. The algorithm uses a queue to store all execution states that have been reached but require to be explored further. Each pair (EX, sq) in the queue consist of an execution state EX that has been reached by performing the *OnEvents* in the sequence of *OnEvents* sq. The SCM library does not provide functionality to set the execution state of a state machine instance. Hence, to reach an execution state, we must perform the sequence of *OnEvents* that was performed to reach this execution state, starting from the initial state. Initially, the queue only contains a pair (I, ϵ) where I denotes the initial execution state of the state machine and ϵ denotes empty sequence of *OnEvents*.

```cpp
#include "test_model_state_machine_impl.h"
using namespace cpp::test_model;
using namespace std;

class ReplyClass final : public ITestModelStateMachineReplies {
    // Class to handle replies of DoEvents (Example)
    void doEventA() override {
        cout << "DoEvent A has been produced" << endl;
    }
};

int main()
{
    auto replies = make_shared<ReplyClass>();
    TestModelStateMachinePtr chartPtr = ITestModelStateMachine::create(replies);
    auto& chart = static_cast<TestModelStateMachin>(*chartPtr);

    chart.initiate();

    // Interactions with State Machine (Examples)
    chart.process<EventA>();
    cout << "StateA is " + (chart.isInState<StateA>() ? "" : "not ") + "active" << endl;

    chart.terminate();
    return 0;
}
```

Figure 7.3: C++ Class to interact with the generated SCM C++ code of SMMT specification `test_model`

The generator stores each execution state EX as a set of `std::string` instances. A map `stateSpace` of type `map<set<string>, map<string, set<string»>` stores for each reachable execution state EX and for each *OnEvent* e the execution state that is reached after *OnEvent* e is processed from execution state EX. Similarly, a map `doEvents` of type `map<set<string>, map<string, vector<string»>` is used to store the *DoEvents* that are produced when *OnEvent* e is processed from execution state EX.

For each pair (EX, sq) in the queue, the algorithm evaluates the *DoEvents* that are produced and the execution state that is reached when each *OnEvent* e is processed in execution state EX. To do so, the algorithm first initializes a state machine instance and processes all *OnEvents* in sequence sq after which *OnEvent* e is processed in execution state EX. After *OnEvent* e is processed, the algorithm stores the execution state in the `stateSpace` map. Furthermore, the algorithm stores the *DoEvents* that are produced after *OnEvent* e is processed in state EX in the `doEvents` map. Whenever an *OnEvent* is processed that is not enabled, a failure is recorded in the `stateSpace` map for execution state EX and *OnEvent* e. When an execution state is reached that is not yet explored, a pair consisting of that execution state and the sequence of *OnEvents* that led to the execution state is added to the queue of execution states that are required to be explored.

Generating an mCRL2 Specification from the State Space We generate an mCRL2 specification using the state space that has been explored by the C++ generator. In this subsection we discuss how the `stateSpace` and `doEvents` maps are translated into an mCRL2 specification that corresponds to the behavior of the SCM C++ code.

To represent the reachable execution states of the executable SCM C++ code, we define a structured sort `State` that consists of an element for each reachable execution state. Each execution state EX in map `stateSpace` is represented as a set of strings consisting of the names of the states in execution state EX. We define function $\mathcal{ORD}(X)$ that converts set X into a list that contains the elements of X in an user-defined order. This ordering function is used to ensure that each set is always mapped to the same list of states. We define mapping

$\alpha_{SN}(X)$ that translates each non-empty set of strings X, representing an execution state, into an instance of type `State` as follows:

$$\alpha_{SN}(X) = \texttt{"st"} + \texttt{"\_"} + \mathbb{X}[0] + \texttt{"\_"} + \mathbb{X}[1] + \texttt{"\_"} + \ldots + \texttt{"\_"} + \mathbb{X}[|\mathbb{X}| - 1]$$

where $s + s'$ represents the concatenation of strings s and s', $\mathbb{X} = \mathcal{ORD}(X)$ and $\mathbb{X}[i]$ denotes the element of list $\mathbb{X}$ at index i for $0 \leq i < |\mathbb{X}|$. Let $Y = \{\text{EX}_1, \text{EX}_2, \ldots, \text{EX}_n\}$ be the reachable execution states of the SMMT specification as generated by the C++ generator. Structured sort `State` is defined as follows in the generated mCRL2 specification:

$$\texttt{sort State = struct } \alpha_{SN}(EX_1) \mid \alpha_{SN}(EX_2) \mid \ldots \mid \alpha_{SN}(EX_n);$$

All *OnEvents* and *DoEvents* of the SMMT specification are added as actions to the mCRL2 specification. We define mapping $\alpha_{EV}(e, \mathbb{D})$ to map *OnEvent* e and a list of *DoEvents* $\mathbb{D}$ to a sequence of actions in the mCRL2 specification. Let $\mathbb{D} = [d_1, d_2, \ldots, d_n]$. Mapping $\alpha_{EV}(e, \mathbb{D})$ is defined as follows:

$$\alpha_{EV}(e, \mathbb{D}) = \alpha_E(e) \cdot \alpha_E(d_1) \cdot \alpha_E(d_2) \cdot \ldots \cdot \alpha_E(d_n)$$

We define a process `p(s : State)` that models the behavior of the SCM C++ code. Process p consists of a single argument s of type `State` that represents the execution state of the SMMT specification. Process p is defined as follows:

$$\texttt{p(s : State)} = \sum_{\mathsf{s}' \in \mathsf{Y}} \sum_{\mathsf{e} \in \mathsf{E}} (\mathsf{s} \approx \alpha_{SN}(\mathsf{s}')) \rightarrow \alpha_{EV}(e, \texttt{doEvents}[s][e]) \cdot \mathsf{p}(\alpha_{SN}(\texttt{stateSpace}[s][e]))$$

For each reachable execution state $EX \in \mathcal{EXS}(\mathtt{M})$ and each *OnEvent* $e \in E$, process p contains an equation that specifies the *DoEvents* that are performed and the state that is reached after *OnEvent* e is processed in execution state EX. This state is either an execution state of the SMMT specification or the failure state.

7.1.1.2 mCRL2 Toolset

In this section we discuss the tools of the mCRL2 toolset [2] that are used to convert the mCRL2 specifications into labelled transitions systems and compare whether two labelled transition systems are strongly bisimilar. As shown in Figure 7.1, each mCRL2 specification is first translated to a linear process specification (LPS) using the `mcrl22lps` tool [24]. The following command is executed to translate an mCRL2 specification `x.mCRL2` to a linear process specification `x.lps`:

```
mcrl22lps -o x.mcrl2 x.lps
```

The no rewriting option (`-o`) has been used to reduce the runtime of the conversion from mCRL2 specifications to LPSs. The runtime of converting the LPS into an LTS does increase when using the no rewriting option. However, the overall runtime of converting the mCRL2 specification into an LTS decreases when the no rewriting option is used.

Next, the LPS is translated to an LTS using the `lps2lts` tool [25]. The following command is executed to translate an LPS `x.lps` to an LTS `x.lts`:

```
lps2lts --cached x.lps x.lts
```

The caching option (`--cached`) is used to speed up the state space generation. The `lps2lts` tool allows for multi-threading. We performed an experiment to determine the number of cores that should be used to generate the LTSs for each SMMT specification. We ran the `lps2lts` command to generate the LTSs of SMMT specifications B, U and W (Table 3.2) using 1 to 128 cores. The generation of the LTSs for SMMT specifications B, U and W has been performed 10 times for each number of cores. The average runtime, as measured by the `time` command of Linux, for SMMT specifications B, U and W is shown in Figure 7.4. The experiments show that using 1 core results in the fastest runtime. Hence, this shows that the mCRL2 specifications that are generated using the `SMMT2MCRL2` translation are not effectively run in parallel. The overhead of running the task in parallel exceeds the time that is saved by dividing the computations over multiple cores.

The mCRL2 toolset consist of a compiling rewriter JITty that can be used by both `mcrl22lps` and `lps2lts` to speed up the rewriting. Unfortunately, the mCRL2 specifications that are generated by translation `SMMT2MCRL2` cannot be converted using the JITty compiling rewriter as these specifications exceed the template depth of the compiler rewriter.

We use the `ltscompare` tool [23] of the mCRL2 toolset to determine whether two LTSs are strongly bisimilar (Definition 33). The following command is executed to evaluate whether two LTSs `x.lts` and `y.lts` are strongly bisimilar:

```
ltscompare -v x.lts y.lts --equivalence=bisim --counter-example
            --counter-example-file=counter-example.mcf
```

The `ltscompare` tool outputs whether the transition systems are strongly bisimilar. If the transition systems are not strongly bisimilar, a counter-example is generated. This counter example consist of a modal mu-calculus formulae that is satisfied by LTS x but is not satisfied by LTS y.

7.1.1.3 Automating the Generation of the mCRL2 Specifications

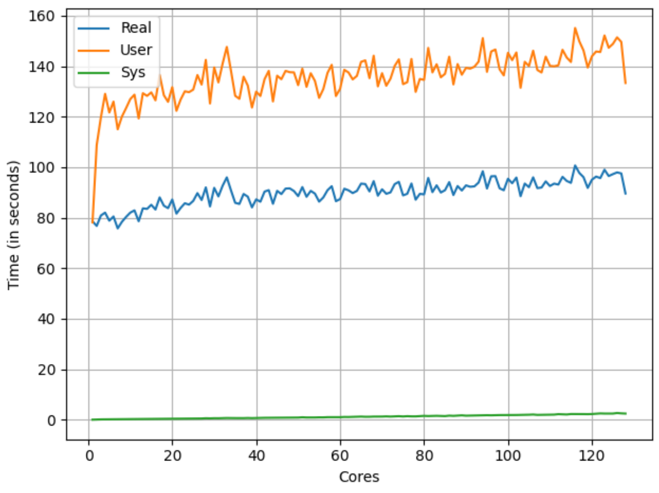
In this section we discuss the Python scripts that were developed to automate the procedure as shown in Figure 7.1. For each SMMT specification, the following files are used to generate the mCRL2 specifications for the SMMT specification:

- `x.mps`: The XML representation of SMMT specification x that is used by MPS as internal structure to store the SMMT specification. This file is used by the `SMMT2MCRL2` translation to generate an mCRL2 specification.
- `x_state_machine.cpp`, `x_state_machine.h` and `x_state_machine_impl.h`: The generated SCM C++ source and header files of SMMT specification x that are generated by SMMT. Translation `SCM2MCRL2` makes use of the generated SCM C++ file to generate an mCRL2 specification.

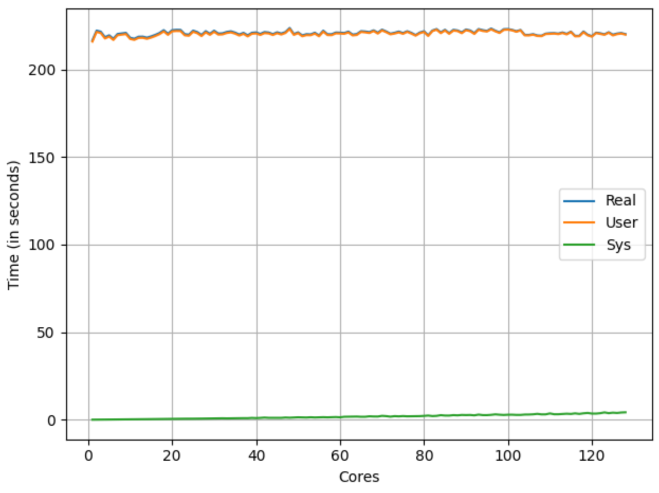
We discuss how the mCRL2 specifications are generated using the `SMMT2MCRL2` and `SCM2MCRL2` translations. Furthermore, we discuss the shell script that is generated to convert the generated mCRL2 specifications of each SMMT specification to LTSs and compare whether the LTSs that are generated for each SMMT specification are strongly bisimilar.

`SMMT2MCRL2` To generate an mCRL2 specification using the `SMMT2MCRL2` translation, we first parse the XML representation in the `.mps` file. The SMMT specification is stored as an object of the `StateMachine` Python class. The states of the SMMT specification are hierarchically stored in an instance of the `StateMachine` class. Each state is stored as an instance of class `State`. The `StateMachine` and `State` Python classes are shown in Figure 7.5.

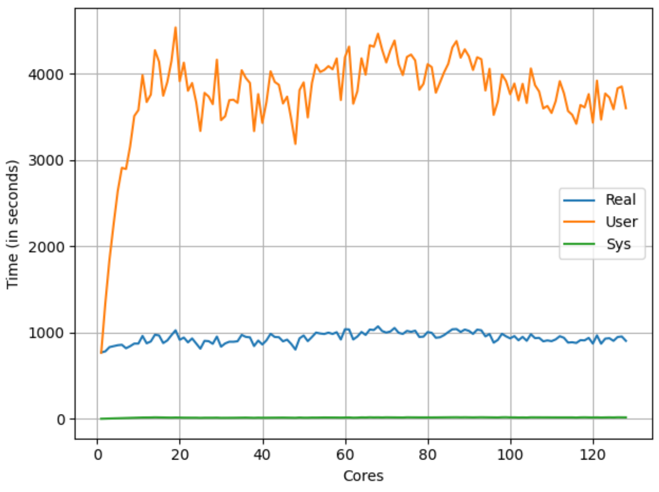
The `SMMT2MCRL2` translation converts the instance of the `StateMachine` class to an mCRL2 specification. As described in Chapter 6, all mCRL2 specifications that are generated consist of four sections. The Python script that performs this translation first generates the mCRL2 representation of the SMMT specification using the instance of the `StateMachine` class. Next, the script appends the validation checks and mappings that are used in the process equations of the mCRL2 specifications. Note that these are independent of the SMMT specification. Finally, the instance of the `StateMachine` class is used to generate the process equations of the mCRL2 specification.



(a) SMMT Specification B



(b) SMMT Specification U



(c) SMMT Specification W

Figure 7.4: Runtime of LPS2LTS for SMMT specifications B, U and W

```
class StateMachine:
    def __init__(self):
        self.name;           # Name of the SMMT specification
        self.namespace;      # Namespace of the SMMT specification
        self.OE = [];        # List of OnEvents of the SMMT specification
        self.DE = [];        # List of DoEvents of the SMMT specification
        self.region = [];    # Region of the SMMT specification

class State:
    def __init__(self, type):
        self.type = type;        # Type of the state (SimpleState, CompositeState, ...)
        self.name = "";          # Name of the state
        self.entry = False;      # Whether the state is an entry state
        self.children = [];      # Children of the state (instances of type State)
        self.transitions = [];   # Outgoing transitions of the states
        self.joins = [];         # Names of the states to which the state refers
        self.ceh = [];           # Conditional Entry Handlers of the state
        self.oeh = [];           # Otherwise Entry Handlers of the state
        self.exh = [];           # Exit Handlers of the state
```

Figure 7.5: Python Classes `StateMachine` and `State`

`SCM2MCRL2` The `SCM2MCRL2` translation generates an mCRL2 specification by exploring the state space of the generated SCM C++ code. First, the Python script that performs the `SCM2MCRL2` translation analyses the `x_state_machine_impl.h` C++ header file to obtain the name, namespace, states, *OnEvents* and *DoEvents* of the SMMT specification. Using the *OnEvents* and *DoEvents*, the Python script generates the C++ header file `x_state_machine_custom_types.h` to define the custom types that are used by the parameters of the *OnEvents* and *DoEvents*. Next, the Python script generates the C++ generator for the mCRL2 specification: `x_state_machine_generator.cpp`. As discussed in Section 7.1.1.1, this C++ generator explores the state space of the SMMT specification and converts the state space into an mCRL2 specification. When the C++ generator has been generated by the Python script, the Python script compiles and executes the C++ generator, resulting in the mCRL2 specification of the SMMT specification that was translated.

Shell Script The Python scripts generate a Shell script that can be used to automatically convert the mCRL2 specifications that are generated for each SMMT specification into LTSs. Furthermore, the Shell script evaluates for each SMMT specification whether the two LTSs that were obtained are strongly bisimilar. The Shell script that is generated for an SMMT specification x is shown in Figure 7.6.

```
#!/bin/bash

# Convert the mCRL2 Specification obtained by translation SMMT2MCRL2 into an LPS
mcrl22lps -o x.mcrl2 x.lps

# Convert the mCRL2 Specification obtained by translation SCM2MCRL2 into an LPS
mcrl22lps -o x-SCM.mcrl2 x-SCM.lps

# Converting the LPSs obtained from both translations into LTS
lps2lts --threads=1 --cached x.lps x.lts
lps2lts --threads=1 --cached x-SCM.lps x-SCM.lts

# Determining whether the LTSs are strongly bisimilar
ltscompare x.lts x-SCM.lts --equivalence=bisim
  --counter-example --counter-example-file=x.mcf
```

Figure 7.6: Shell Script generated for SMMT specification x

7.1.2 Results Correctness Verification SMMT2MCRL2

In this section we present the results of the experiments that we ran to verify the correctness of the SMMT2MCRL2 translation. We generated the mCRL2 specifications of all existing SMMT specification using both the SMMT2MCRL2 and the SCM2MCRL2 translation on a laptop with Windows 10 Pro, an Intel Core i7-6500U 2.50 GHz processor and 16 GB of RAM. The Shell script that has been generated for each existing SMMT specifications has been run on the High-Performance Cluster at Canon Production Printing. The High-Performance Cluster operates over Rocky Linux 8.6 and consist of 5 computing nodes that each are equipped with two EPYC 7763 processors and 256 GB of RAM. Version 202307.0 of the mCRL2 toolset was used to run various tools of the toolset on the High-Performance Cluster. The results of our experiments to verify the correctness of the SMMT2MCRL2 translation are shown in Tables 7.1 (SMMT Specifications A to I) and 7.2 (SMMT Specifications J to Z). Tables 7.1 to 7.5 use format `days : hours : minutes : seconds` to denote the execution times. The number of states and transitions that are given in these tables correspond to the number of states and transitions in the generated LTSs.

	Translation	States	Transition	Generation	mcrl22lps	lps2lts	Total Time	Strongly Bisimilar
A	SMMT2MCRL2	205	2701	00:00:00:02	00:00:00:02	00:02:31:10	00:02:31:14	True
	SCM2MCRL2	205	2701	00:00:00:25	00:00:00:02	00:00:00:03	00:00:00:30	
B	SMMT2MCRL2	37	389	00:00:00:01	00:00:00:01	00:00:01:01	00:00:01:03	True
	SCM2MCRL2	37	389	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	
C	SMMT2MCRL2	39	379	00:00:00:01	00:00:00:01	00:00:01:05	00:00:01:07	True
	SCM2MCRL2	39	379	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	
D	SMMT2MCRL2	46	446	00:00:00:01	00:00:00:01	00:00:01:51	00:00:01:53	True
	SCM2MCRL2	46	446	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	
E	SMMT2MCRL2	39	305	00:00:00:01	00:00:00:01	00:00:00:42	00:00:00:44	True
	SCM2MCRL2	39	305	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	
F	SMMT2MCRL2	34	268	00:00:00:01	00:00:00:01	00:00:00:32	00:00:00:34	True
	SCM2MCRL2	34	268	00:00:00:17	00:00:00:01	00:00:00:01	00:00:00:19	
G	SMMT2MCRL2	21	120	00:00:00:01	00:00:00:01	00:00:00:10	00:00:00:12	True
	SCM2MCRL2	21	120	00:00:00:15	00:00:00:02	00:00:00:01	00:00:00:18	
H	SMMT2MCRL2	19	109	00:00:00:01	00:00:00:01	00:00:00:20	00:00:00:22	True
	SCM2MCRL2	19	109	00:00:00:20	00:00:00:01	00:00:00:01	00:00:00:22	
I	SMMT2MCRL2	11	36	00:00:00:01	00:00:00:01	00:00:00:02	00:00:00:04	True
	SCM2MCRL2	11	36	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	

Table 7.1: Experiment Verifying Correctness of the SMMT2MCRL2 Translation (SMMT Specifications A-I)

	Translation	States	Transition	Generation	mcrl22lps	lps2lts	Total Time	Strongly Bisimilar
J	SMMT2MCRL2	24706	185221	00:00:00:01	00:00:00:01	07:14:51:40	07:14:51:42	True
	SCM2MCRL2	24706	185221	00:00:14:56	01:03:47:43	-[1]	-[1]	
K	SMMT2MCRL2	6636	53330	00:00:00:01	00:00:00:01	01:19:57:40	01:19:57:42	True
	SCM2MCRL2	6636	53330	00:00:03:51	00:00:51:54	00:12:52:05	00:13:47:50	
L	SMMT2MCRL2	30	258	00:00:00:01	00:00:00:02	00:00:00:23	00:00:00:26	True
	SCM2MCRL2	30	258	00:00:00:17	00:00:00:01	00:00:00:01	00:00:00:19	
M	SMMT2MCRL2	-[1]	-[1]	00:00:00:01	00:00:00:01	-[1]	-[1]	Error
	SCM2MCRL2	-	-	-	-	-	-	
N	SMMT2MCRL2	53	270	00:00:00:01	00:00:00:01	00:00:01:38	00:00:01:40	True
	SCM2MCRL2	53	270	00:00:00:17	00:00:00:01	00:00:00:01	00:00:00:19	
O	SMMT2MCRL2	33	313	00:00:00:01	00:00:00:02	00:00:00:35	00:00:00:38	True
	SCM2MCRL2	33	313	00:00:00:17	00:00:00:01	00:00:00:01	00:00:00:19	
P	SMMT2MCRL2	24	132	00:00:00:01	00:00:00:02	00:00:00:08	00:00:00:11	True
	SCM2MCRL2	24	132	00:00:00:17	00:00:00:01	00:00:00:01	00:00:00:19	
Q	SMMT2MCRL2	9	41	00:00:00:01	00:00:00:02	00:00:00:02	00:00:00:05	True
	SCM2MCRL2	9	41	00:00:00:15	00:00:00:02	00:00:00:01	00:00:00:18	
R	SMMT2MCRL2	4	10	00:00:00:01	00:00:00:02	00:00:00:02	00:00:00:05	True
	SCM2MCRL2	4	10	00:00:00:15	00:00:00:01	00:00:00:01	00:00:00:17	
S	SMMT2MCRL2	18	99	00:00:00:01	00:00:00:01	00:00:00:10	00:00:00:12	True
	SCM2MCRL2	18	99	00:00:00:15	00:00:00:01	00:00:00:01	00:00:00:17	
T	SMMT2MCRL2	1038	6342	00:00:00:01	00:00:00:02	00:05:13:52	00:05:13:55	Error
	SCM2MCRL2	-	-	-	-	-	-	
U	SMMT2MCRL2	2	1	00:00:00:01	00:00:00:02	00:00:00:45	00:00:00:48	Error
	SCM2MCRL2	-	-	-	-	-	-	
V	SMMT2MCRL2	2	1	00:00:00:01	00:00:00:02	00:00:01:21	00:00:01:24	False
	SCM2MCRL2	219	1254	00:00:00:19	00:00:00:03	00:00:00:07	00:00:00:29	
W	SMMT2MCRL2	132	822	00:00:00:01	00:00:00:02	00:00:10:52	00:00:10:55	True
	SCM2MCRL2	132	822	00:00:00:19	00:00:00:01	00:00:00:02	00:00:00:22	
X	SMMT2MCRL2	10	30	00:00:00:01	00:00:00:01	00:00:00:02	00:00:00:04	True
	SCM2MCRL2	10	30	00:00:00:16	00:00:00:01	00:00:00:01	00:00:00:18	
Y	SMMT2MCRL2	19	49	00:00:00:01	00:00:00:02	00:00:00:09	00:00:00:12	True
	SCM2MCRL2	19	49	00:00:00:16	00:00:00:01	00:00:00:02	00:00:00:19	
Z	SMMT2MCRL2	48	265	00:00:00:01	00:00:00:01	00:00:02:02	00:00:02:04	True
	SCM2MCRL2	48	265	00:00:00:19	00:00:00:01	00:00:00:02	00:00:00:22	

Table 7.2: Experiment Verifying Correctness of the SMMT2MCRL2 Translation (SMMT Specifications J-Z)

[1] The files containing these values were overwritten by accident when performing the experiments.

In the remainder of this section, we discuss the results as shown in Tables 7.1 and 7.2. First, we discuss the problems that occurred during the generation of the mCRL2 specification for each SMMT specification using translations `SMMT2MCRL2` and `SCM2MCRL2`. Next, we discuss whether the behavior of the mCRL2 specification that is generated by the `SMMT2MCRL2` translation is strongly bisimilar to the behavior of the mCRL2 specification that is obtained by the `SCM2MCRL2` translation.

7.1.2.1 Generation of the mCRL2 Specifications

An mCRL2 specification has been generated for each of the 26 existing SMMT specifications using the `SMMT2MCRL2` translation. Using the `SCM2MCRL2` translation, we were only able to generate an mCRL2 specification for 23 out of the 26 existing SMMT specifications. The `SCM2MCRL2` translation could not generate an mCRL2 specification for SMMT specifications M, T and U. We were not able to generate an mCRL2 specification using translation `SCM2MCRL2` for SMMT specification M as the generation of the mCRL2 specification exceeded the available RAM. Hence, we have not been able to verify the correctness of the `SMMT2MCRL2` translation for SMMT specification M.

The SCM C++ code that was generated for SMMT specifications T and U threw an exception during the compilation of the SCM C++ classes that were generated by SMMT. The SCM C++ classes that are generated by SMMT throw a read access violation exception when a *JointState* is initiated that refers to a state that is not yet initiated at that point in time. In the C++ classes that are generated by SMMT, it must hold that all states that are referred to by a *JointState* must be initiated before the *JointState* itself is initiated. This restriction is only required for the C++ code that is generated for an SMMT specification by MPS, not for the generated C# code. The engineers that designed SMMT specifications T and U mentioned that these two specifications were designed to generate C# code. Therefore, no problems occurred when the generated C# classes were compiled and executed. This is an issue in the C++ code generator of SMMT which has been reported to the developer of the SCM C++ library.

To verify that the exception is indeed thrown when a *JointState* is initiated that refers to another state that is not yet initiated, we created a copy of both SMMT specifications T and U: specifications T′ and U′ respectively. In SMMT specifications T′ and U′, we replaced the order in which the states are initiated to ensure that no *JointState* is initiated that refers to a state that is not yet initiated. We ran the experiment for SMMT specifications T′ and U′. The results of the experiments for SMMT specifications T′ and U′ are shown in Table 7.3.

	Translation	**States**	**Transition**	**Generation**	**mcrl22lps**	**lps2lts**	**Total Time**	**Strongly Bisimilar**
T′	SMMT2MCRL2	1038	6342	00:00:00:01	00:00:00:01	00:04:22:34	00:04:22:36	🟩 True
	SCM2MCRL2	1038	6342	00:00:00:37	00:00:00:08	00:00:03:30	00:00:04:15	
U′	SMMT2MCRL2	2	1	00:00:00:01	00:00:00:01	00:00:00:44	00:00:00:46	🟥 False
	SCM2MCRL2	8125	54691	00:00:06:18	00:01:30:39	01:22:48:10	02:00:25:07	

Table 7.3: Experiment Verifying Correctness of the `SMMT2MCRL2` Translation (SMMT Specifications T′ and U′)

The `SCM2MCRL2` translation was able to generate an mCRL2 specification for both SMMT specifications T′ and U′. Hence, this shows that the exception during the compilation of the SCM C++ code is indeed caused by a *JointState* referring to a state that is not yet initiated.

7.1.2.2 Correctness Translations SMMT Specification

Out of the 25 SMMT specifications for which we were able to generate an mCRL2 specification using both translations (A to L, N to S, T′, U′ and V to Z), there are 23 SMMT specifications of which the LTSs that were obtained by the two translations (SMMT2MCRL2 and SCM2MCRL2) are strongly bisimilar. Only the LTSs that were generated for SMMT specifications U′ and V are not strongly bisimilar. We discuss why the LTSs that are generated using the two translations for both SMMT specification U′ and V are not strongly bisimilar.

SMMT Specification U′ Using the `ltsgraph` tool of the mCRL2 toolset, we analysed the LTS that was generated for SMMT specification U′ by translation SMMT2MCRL2 (Figure 7.7).

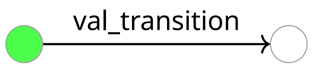


Figure 7.7: LTS generated for SMMT specification U′ using translation SMMT2MCRL2 and the `mcrl22lps`, `lps2lts` and `ltsgraph` tools of the mCRL2 toolset

The LTS shown in Figure 7.7 shows a transition with label `val_transition`. According to Table 6.1 this action corresponds to a violation of Restriction 5 by SMMT specification U′. Hence, there must exist one or more state in SMMT specification U′ that each contain two or more transitions that are defined for the same *OnEvent*. SMMT specification U′ contains two states that, like `state_a` in Figure 7.8, have an internal and external transition defined for the same *OnEvent*.

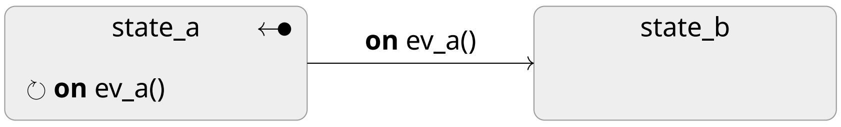


Figure 7.8: Example SMMT specification violation Restriction 5

The engineers that worked on SMMT specification U mentioned that the internal transitions were temporarily added to be able to test the model during development. When the engineers extended the SMMT specification, the engineers added the external transition but forgot to remove the temporarily added internal transition. We create a copy of SMMT specification U′: U″. In SMMT specification U″, we removed the internal transitions of the two states that had both an internal and external transition defined for the same *OnEvent*. We performed the experiment for SMMT specification U″, the result of the experiment is shown in Table 7.4.

	Translation	States	Transition	Generation	mcrl22lps	lps2lts	Total Time	Strongly Bisimilar
U″	SMMT2MCRL2	8125	54691	00:00:00:01	00:00:00:01	03:10:02:18	03:10:02:20	True
	SCM2MCRL2	8125	54691	00:00:06:18	00:01:30:39	01:22:48:10	02:00:25:07	

Table 7.4: Experiment Verifying Correctness of the Extended SMMT2MCRL2 Translation (SMMT Specification U″)

As shown in Table 7.4, the LTSs that are generated for SMMT specification U″ by translations SMMT2MCRL2 and SCM2MCRL2 are strongly bisimilar. Hence, this shows that the behavior of the SMMT specification was not affected by removing the previously mentioned internal transitions.

SMMT Specification V The engineers that developed SMMT mentioned that for each state of an SMMT specification it must holds that the guards of all transitions that are defined for the same *OnEvent* are mutually exclusive. That is, for any valuation of the arguments of the *OnEvents*, there exists at most one transition that is defined for the *OnEvent* without a guard or of which the guard is satisfied. SMMT does not contain a checking rule to check whether this restriction is violated or not. Hence, engineers are not alerted of states for which multiple transitions are defined for the same *OnEvent* that are not mutually exclusive. SMMT should be extended with a checking rule that determines whether this restriction is violated during the creation of the SMMT specification in MPS.

The LTS that is generated from the mCRL2 specification obtained by translation `SMMT2MCRL2` shows that there exist one or more states in SMMT specification V of which the transitions for an *OnEvent* are not mutually exclusive. SMMT specification V has two states that each have an internal and two external transitions defined for the same *OnEvent*. Figure 7.9 shows one of these two states, `state_a`, and shows the outgoing transitions that are defined for the same *OnEvent* for which the guards are not mutually exclusive.

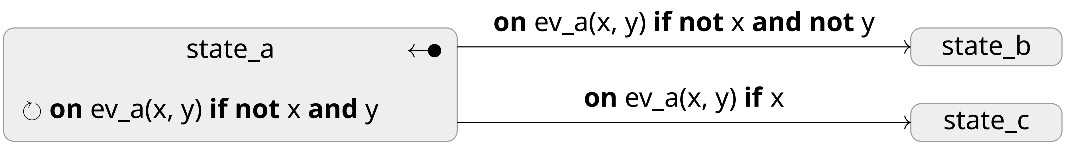

state_a
on ev_a(x, y) **if not** x **and** y
on ev_a(x, y) **if not** x **and not** y
state_b
on ev_a(x, y) **if** x
state_c

Figure 7.9: Violation: Guards not mutually exclusive

By the order of precedence of the logical connectives [26], it follows that the `not` connective binds stronger than the and and `or` connectives. Hence, when *OnEvent* `ev_a(x, y)` is processed, we have that:

- The internal transition of `state_a` fires if, and only if, x is false and y is true.
- The transition from `state_a` to `state_b` fires if, and only if, x and y are both false.
- The transition from `state_a` to `state_c` fires if, and only if, x is true.

Therefore, the guards of the transitions in Figure 7.9 are mutually exclusive. This contradicts with the results that were obtained using translation `SMMT2MCRL2`.

Using the reflective editing mode of JetBrains MPS, we are able to directly view the guard of each transition using the underlying abstract syntax tree. For example, the guard of the internal transition of `state_a` shows the following abstract syntax tree:

```
if not {
    and {
        left: x
        right: y
    }
}
```

According to the abstract syntax tree the guard of the internal transition of `state_a` is defined as **not** (x **and** y). Hence, the guards are not visualized conform to the order of precedence as **not** (x **and** y) is trivially not equivalent to (**not** x) **and** y. Therefore, the transitions that are defined in the example of Figure 7.9 are indeed not mutually exclusive. Hence, this shows that the representation of guards is not correctly implemented when using the regular editing mode of MPS. The representation of guards in SMMT needs to be adopted such that:

- Parentheses are added to the guard to ensure that the guard is correctly visualized in regular editing mode of MPS.
- A checking rule must be implemented that ensures that the engineers may only add guards of which the representation as shown using both the regular and reflective editing mode are equivalent. That is, if we format the guard as shown in the regular editing mode as a tree conform the order of precedence, this tree should be equivalent to the guard in the abstract syntax tree as shown in the reflective editing mode.

The code generator of SMMT does not insert parentheses when a guard is converted to either C++ or C# code. Therefore, the guard in the generated executable code always corresponds to the guard as shown in the SMMT specification in the regular editing mode. Hence, the guards in the generated C++ and C# code are mutually exclusive.

We create a copy of SMMT specification V: V′. We changed all guards in SMMT specification V′ such that the representation of each guard is equivalent in both the regular and reflective editor. We performed the experiment for SMMT specification V′, the result of the experiment for SMMT specification V′ is shown in Table 7.5.

	Translation	States	Transition	Generation	mcrl22lps	lps2lts	Total Time	Strongly Bisimilar
V′	SMMT2MCRL2	219	1254	00:00:00:01	00:00:00:01	00:00:30:42	00:00:30:44	True
	SCM2MCRL2	219	1254	00:00:00:20	00:00:00:03	00:00:00:07	00:00:00:30	

Table 7.5: Experiment Verifying Correctness of the SMMT2MCRL2 Translation (SMMT Specification V′)

As we adapted the guards such that the guards as shown by the regular and reflective editor are equivalent, it follows that the guards in the MPS XML file and the generated C++ or C# must be equivalent. Therefore, the modifications that are made to the guard ensure that the mutually exclusive guards are used in the mCRL2 specification for both translations.

7.2 Property Verification

In this section we discuss the properties that we have verified on the existing SMMT specifications. Note that we have verified these properties on the corrected versions of the SMMT specification. We first discuss the properties that have been verified for each SMMT specification. Next, we discuss how the properties have been checked on each SMMT specification. Finally, we present the results of verifying these properties on the SMMT specifications.

7.2.1 Properties

We have verified 5 properties on each of the 26 SMMT specifications that exist at the time of writing. In this section we introduce the properties that were verified and discuss how the properties are expressed as mu-calculus formulae. These properties determine whether the SMMT specifications can be reduced without affecting the behavior of the SMMT specification. That is, they check whether there exist states that never become active, transitions that can never fire, *OnEvents* that are never enabled and *DoEvents* that are never produced. Furthermore, we determine whether there exist execution states for which no *OnEvent* is enabled and hence always leads to the failure state. Unfortunately, due to a lack of time, we were not able to define SMMT specification specific properties for the existing SMMT specifications.

1. Enabled States The enabled states property determines whether for all states of the mCRL2 specification, except for the failure state, we can always perform an action that does not lead to the failure state. That is, from any state except for the failure state there should be an action that can be performed that cannot be followed by the `FAIL` action. By construction of the process equations of the mCRL2 specification, a `FAIL` action can only occur after an *OnEvent* is processed that is not enabled. Property 1 can be expressed using the following mu-calculus formula:

```
[true*](<FAIL>true || <true>[FAIL]false)
```

This formula states that after any trace of actions, we either reached the failure state and thus we can perform the `FAIL` action or we can perform an action after which we do not reach the failure state and therefore the `FAIL` action cannot occur.

2. Activeness of States The activeness of states property determines whether all states of the SMMT specification can become active during execution. For each state s of the SMMT specification, there should exist a sequence of *OnEvents* starting from the initial state that lead to an execution state in which state s is active. In the mCRL2 specification that is generated for each SMMT specification, there are no actions that can be used to determine the execution state of the mCRL2 specification. Hence, to determine whether this property holds for an SMMT specification, we are required to add actions to indicate when a state is active.

We have generated a reachability mCRL2 specification for each SMMT specification that is an extension of the mCRL2 specification that is obtained using translation `SMMT2MCLR2`. In the reachability mCRL2 specification that is generated for an SMMT specification, we define an action for each state of the SMMT specification. In the process equation of the reachability mCRL2 specification we allow the action that has been defined for a state to be performed if, and only if, the state to which the action corresponds is active. After the action is performed, the arguments of the process are not altered. Therefore, the actions that indicate the activeness of states are defined as self-loops.

To determine whether a state can become active during execution, it therefore suffices to determine whether there exists a trace in which the action that was defined for that state occurs. The activeness of states property can be expressed for a state `state_a`, for which reachability action `st_state_a` has been defined, using the following mu-calculus formula:

```
<true*.st_state_a>true
```

This property must be verified for all states of the SMMT specification.

3. All *OnEvents* Enabled This property determines whether there exists a state for each *OnEvent* of the mCRL2 specification such that the *OnEvent* does not lead to the failure state. That is, this property checks for each *OnEvent* of the mCRL2 specification whether there exists a trace in which the *OnEvent* action occurs after which a `FAIL` action cannot be performed. Hence, for each *OnEvent* `ev_a` of an mCRL2 specification, this property can be expressed using the following mu-calculus formula:

```
<true*.ev_a>[FAIL]false
```

This formula states that there exists a trace in which *OnEvent* `ev_a` occurs after which we do not reach the failure state.

4. All *DoEvents* Produced Similar to property 3, we define a property to determine whether each *DoEvent* can be produced during execution of the mCRL2 specification. This property checks for each *DoEvent* of the mCRL2 specification whether there exists a trace in which the *DoEvent* action occurs. Hence, for each *DoEvent* `re_a` of an mCRL2 specification, this property can be expressed using the following mu-calculus formula:

```
<true*.re_a>true
```

5. All Transitions Fire Finally, we define a property to determine whether all transitions that are defined in the SMMT specification can fire at least once during the execution. There are no actions present in the mCRL2 specification to indicate that a specific transition has fired. We assign a unique integer `i` to each transition and add a *DoEvent* `tr_reach(i)` to the list of *DoEvents* that are produced when the transition is fired in the reachability mCRL2 specification. Hence, to determine whether a transition to which unique integer `i` has assigned fires in the reachability mCRL2 specification, it suffices to check whether there exists a trace in which action `tr_reach(i)` occurs. For a transition to which unique integer `i` has been assigned, the property can be expressed using the following mu-calculus formula:

```
<true*.tr_reach(i)>true
```

This property must be verified for all transitions of the SMMT specification.

7.2.2 Approach

The mu-calculus formulae to check the properties for each SMMT specification are generated during the generation of the mCRL2 model by translation `SMMT2MCRL2`. To determine whether a property is satisfied by an mCRL2 specification, the `lts2pbes` [27] and `pbessolve` [28] tools of the mCRL2 toolset are used. Figure 7.10 shows the commands that are executed to convert the previously generated LTS and mu-calculus formula into a Parameterized Boolean Equation System (PBES) and to solve the generated PBES.

```
% Replace x.lts with x—reachability.lts if property p corresponds to property 2 or 5
lts2pbes —c —{}—formula=p.mcf x.lts x_p.pbes
pbessolve —{}—file=x.lts —{}—evidence—file=x_evidence_p.lts x_p.pbes
```

Figure 7.10: Commands to verify a property p on SMMT specification x

The `-c` option is used in the `lts2pbes` command to add counter example equations to the generated PBES. Using the `--evidence-file` option, an evidence file is generated that either shows the evidence that the property holds or a counterexample showing how the property is violated. This evidence file is an LTS.

7.2.3 Results

In this section we discuss whether the properties as discussed in Section 7.2.1 are satisfied by the 26 existing SMMT specifications. Table 7.6 shows for each property and each SMMT specification whether the property is satisfied by the mCRL2 specification that is generated by translation `SMMT2MCRL2`. All properties have been evaluated on a laptop with Windows 10 Pro, an Intel Core i7-6500U 2.50 GHz processor and 16 GB of RAM on which version 202307.0 of the mCRL2 toolset is installed. As shown in Table 7.6, there are only four SMMT specifications that do not satisfy all five properties. First, we discuss why SMMT specifications N, P and Z do not satisfy property 1. Next, we discuss the reason that SMMT specification I violates properties 2, 3 and 5.

SMMT Spec.	1. Enabled States	2. Activeness of States	3. All *OnEvents* Enabled	4. All *DoEvents* Produced	5. All Transitions Fire
A	True	True	True	True	True
B	True	True	True	True	True
C	True	True	True	True	True
D	True	True	True	True	True
E	True	True	True	True	True
F	True	True	True	True	True
G	True	True	True	True	True
H	True	True	True	True	True
I	True	False	False	True	False
J	True	True	True	True	True
K	True	True	True	True	True
L	True	True	True	True	True
M	True	True	True	True	True
N	False	True	True	True	True
O	True	True	True	True	True
P	False	True	True	True	True
Q	True	True	True	True	True
R	True	True	True	True	True
S	True	True	True	True	True
T′	True	True	True	True	True
U″	True	True	True	True	True
V′	True	True	True	True	True
W	True	True	True	True	True
X	True	True	True	True	True
Y	True	True	True	True	True
Z	False	True	True	True	True

Table 7.6: Verification of the properties (Section 7.2.1) on the SMMT specifications

SMMT Specifications N, P and Z As shown in Table 7.6, there are three SMMT specifications that do not satisfy property 1: N, P and Z. Hence, each of these three SMMT specifications has an execution state EX in which all *OnEvents* that are processed lead to the failure state. SMMT specifications N, P and Z each contain a state to denote that the machine has finished or has shut down. It is expected that no more *OnEvents* are handled whenever these states are reached. Let $E' \subseteq E$ be the set of all *OnEvents* for which a transition is defined after which execution state EX is reached directly. We define an additional property for SMMT specifications N, P and Z that denotes that there always exists an action that does not lead to the failure state, unless an *OnEvent* that is contained in E' is processed. This property can be expressed using the following mu-calculus formula:

$$\texttt{[(!}E'\texttt{)*](<FAIL>true || <true>[FAIL]false)}$$

This formula states that as long as no action $e \in E'$ occurs, we either reached the failure state and thus we can perform the `FAIL` action or we can perform an action after which we do not reach the failure state. This property is satisfied by SMMT specifications N, P and Z. Hence, this shows that for each execution state of SMMT specifications N, P and Z there always exists an *OnEvent* that is enabled in that execution state unless an action $e \in E'$ is performed.

SMMT Specification I As shown in Table 7.6, properties 2, 3 and 5 are not satisfied by SMMT specification I. Hence, there exists a state of the SMMT specification that can never be active during execution, there exist an *OnEvent* that is never enabled and there exists a transition that cannot fire. SMMT specification I is shown in Figure 7.11.

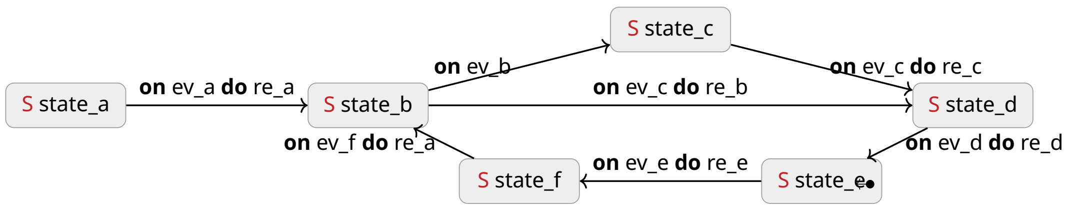


Figure 7.11: SMMT Specification I

As `state_e` is the initial state of SMMT specification I and there is no transition that leads to `state_a`, we have that `state_a` can never be active during execution. Hence, the transition from `state_a` to `state_b` cannot fire either. The engineers that developed SMMT specification I mentioned that `state_a` and the outgoing transition of `state_a` should have been removed from the SMMT specification.

7.3 Conclusion

In this chapter we discussed the experiments that were performed to verify the correctness of the SMMT2MCRL2 translation and to prove the correctness of the existing SMMT specifications. The experiments that were performed show that the behavior of the mCRL2 specification that is generated for each of the 26 existing SMMT specifications by translation SMMT2MCRL2 is strongly bisimilar to the behavior of the mCRL2 specification that is generated by the SCM2MCRL2 translation, with the exception of SMMT specifications M, T, U and V. For SMMT specification M, we were not able to generate an mCRL2 specification using translation SCM2MCRL2 as this exceeded the available RAM. Hence, we were not able to verify the correctness of translation SMMT2MCRL2 for SMMT specification M.

Through the experiments that have been performed, we found two SMMT specification for which the C++ code generator of SMMT generated C++ classes that could not be compiled. The generated C++ classes that were generated for SMMT specifications T and U each contain a *JointState* that is initiated before all states that are referred to by this *JointState* are initiated. This issue has been reported to the developer of the SCM C++ library. We have shown how SMMT specifications T and U were affected by this issue and discussed a temporary solution to compile and run the generated SCM C++ code.

Using the validation checks that are contained in the mCRL2 specifications as generated by the SMMT2MCRL2 translation, we have found two specifications that did not satisfy all requirements: SMMT specifications U and V. We found several states in SMMT specification U for which more than one transition were defined for the same *OnEvent*. We have shown which transitions of the SMMT specification should have been removed and shown that removing these transitions did not affect the behavior of the SMMT specification.

Furthermore, we found an issue with the representation of guards in SMMT. The representation of guards in the regular editing mode of MPS has been shown to be incorrect. We have shown that the guard as shown in the reflective editing mode, showing the abstract syntax tree, does not correspond to the guard as shown in the regular editing mode. The engineers that implemented the guards of SMMT specification V did not use the reflective editing mode when defining the guard. Therefore, the guards that were defined in SMMT specification V were assumed to be defined as desired by the engineer. The code that has been generated for SMMT specification V contains the guards as shown in the regular editing mode of MPS. Hence, the code contains the guard as desired by the engineers of SMMT specification V. We have replaced the guards in SMMT specification V such that each guard is equivalent in both editing modes of MPS. Using the SMMT2MCRL2 and SCM2MCRL2 translation we have shown that we correctly modified the guards.

Using the SMMT2MCRL2 translation, mCRL2 specifications have been generated for the corrected versions of SMMT specifications T, U and V. The behavior of these mCRL2 specifications has been shown to be strongly bisimilar to the behavior of the SCM C++ code that was generated for the corresponding SMMT specification. Hence, we can conclude that the mCRL2 specifications that are generated by translation SMMT2MCRL2 for SMMT specifications A to L and N to Z correctly model the behavior of the respective SMMT specification.

As we have only shown that the SMMT2MCRL2 translation can generate an mCRL2 specification that correctly models the behavior of the corresponding SMMT specification for 25 SMMT specifications (A to L and N to Z), we did not prove the complete correctness of the SMMT2MCRL2 translation. As shown in Chapter 3, the 25 SMMT specifications are diverse and each construct of SMMT is contained in several SMMT specifications. The SMMT2MCRL2 translation has been defined using the information that was obtained through several meetings with the engineers that are experienced with SMMT and obtained through analysing the code generator and the SCM library. Hence, we conclude with high confidence that the SMMT2MCRL2 translation is able to correctly generate an mCRL2 specification for any SMMT specification.

As shown in Tables 7.1 to 7.5, the generation of the LTSs for the existing SMMT specifications may take up to several days. Unfortunately, the runtime of generating the LTSs for each SMMT specification did not decrease when the LTSs were generated using more than one thread. As discussed before, the JITty compiler rewriter could not be used to speed up the generation of the LTSs either, as this resulted in an exception. Various improvements can be made to the mCRL2 specification to improve the efficiency of the mCRL2 specification and therefore decrease the runtime of generating an LTS from the mCRL2 specification. Due to time constraints the efficiency of the generated mCRL2 specifications has not been optimized.

We have defined and verified five properties to check the correctness of each SMMT specification. Using these properties, we have found four SMMT specifications that did not satisfy all properties. SMMT specification I has been shown to contain a state that never becomes active, a transition that never fires and an *OnEvent* that is not enabled in any execution state of the SMMT specification. According to the engineers that developed SMMT specification I, this state should have been removed from the SMMT specification. Furthermore, we found that property 1 was not satisfied by SMMT specifications N, P and Z. We have shown that these three SMMT specifications contain an execution state in which all produced *OnEvents* lead to the failure state. However, these SMMT specifications were intentionally designed to have a state in which the execution of the SMMT specification should be terminated, that is, in which no more *OnEvents* should be processed.

Chapter 8

Conclusion

In this graduation report we presented the State Machine Modelling Tool (SMMT) that is used to model the behavior of software components using state machines. In this chapter we answer the research question that has been presented in Chapter 1:

"To what extent can we formalise and prove the correctness of SMMT specifications using the mCRL2 model checker?"

We formally defined the syntax, static semantics and operational semantics of SMMT specifications that consist of *SimpleStates*, *CompositeStates*, *ParallelStates* and transitions without guards or *BehavioralActions*. The formal definition of SMMT that is presented in this graduation report is derived from the discussions with the engineers at Canon Production Printing and the analysis of the code generator, the generated SCM code and the SCM library. First, we formalised SMMT specifications consisting of only *SimpleStates*, *CompositeStates* and transitions without guards and *BehavioralActions*. We defined the abstract syntax of an SMMT specification, the static semantics as a number of restrictions and defined the operational semantics of SMMT as a labelled transition system. Next, we extended our formal definition to include *ParallelStates*. We have shown how the formal definition of SMMT should be extended to include *ParallelStates* by redefining the abstract syntax, adding several additional restrictions and redefining the operational semantics.

We have defined a translation `SMMT2MCRL2` to translate SMMT specifications into mCRL2 specifications. This translation is defined using the formal definition of SMMT. The mCRL2 specifications that are generated by the `SMMT2MCRL2` closely correlate to the formal definition as presented in this report. That is, each generated mCRL2 specification consists of a representation of the mathematical model of the SMMT specification, validation checks to determine whether the SMMT specification satisfies all restrictions, mappings and equations that represent the definitions that are used to define the operational semantics of SMMT and the process equations representing the operational semantics of the SMMT specification.

Due to a lack of time, we were unfortunately not able to formally define SMMT for the complete set of constructs in Chapter 5. However, we have defined the syntax and semantics of the complete set of constructs in mCRL2, excluding *Forward* and *SelfPost BehavioralActions*. The formal definition consisting of all constructs of SMMT, excluding *SelfPost* and *Forward BehavioralActions*, can be derived from this mCRL2 specification.

We have defined the `SCM2MCRL2` translation that explores the state space of the executable SCM C++ code that is generated by SMMT and translates the explored state space into an mCRL2 specification. We have shown that the `SMMT2MCRL2` translation correctly generates an mCRL2 specification for 25 out of the 26 SMMT specifications that exist at the time of writing. That is, we have shown that the behavior of the generated SCM C++ code is strongly bisimilar to the behavior of the generated mCRL2 specification by translation `SMMT2MCRL2` for 25 out of the 26 existing SMMT specifications. For the one remaining SMMT specification, we were not able to generate an mCRL2 specification using the SCM2MCRL2 translation as the state space exploration exceeded the available RAM. Hence, we cannot determine the correctness of the SMMT2MCRL2 translation for this SMMT specification.

Through the experiments that we performed, we found an issue with the C++ code generator concerning the order in which states are initiated. Due to this issue, the C++ classes that are generated for two SMMT specifications cannot be compiled. Using the validation checks that are implemented in the mCRL2 specifications that are generated by the SMMT2MCRL2 translation, we detected two SMMT specifications that violated a validation check. Furthermore, we found an issue with the representation of the guards of transitions. We have shown that the representation of the guards as shown in the reflective and regular editing mode of MPS are not always equivalent. We have discussed these issues with the engineers at Canon Production Printing and proposed solutions to resolve these issues.

Finally, we have shown how the mCRL2 specifications that are generated by the SMMT2MCRL2 translation can be used to verify whether the SMMT specifications satisfy certain properties. We have shown that there exist three SMMT specifications that contain states for which all *OnEvents* that are processed lead to the failure state. Furthermore, we found one SMMT specification that contains a state that can never become active, an *OnEvent* that never is enabled and a transition that can never fire. Unfortunately, due to a lack of time, we have not been able to prove whether SMMT specification specific properties hold on the existing SMMT specifications. These results demonstrate the power of using the mCRL2 model checker to determine the correctness of SMMT specifications.

Chapter 9

Future Research

In this chapter we present some open issues that deserve further research.

Integration of the translation and verification in MPS The `SMMT2MCRL2` translation that is defined to generate an mCRL2 specification from an SMMT specification and all steps to analyse properties on the generated mCRL2 specifications are performed using Python scripts. These scripts are not integrated with the implementation of SMMT in JetBrains MPS. It could be interesting to integrate the generation of the mCRL2 specification and verification of the properties with the implementation of SMMT in MPS. This would allow the engineer to automatically get feedback on the modelled SMMT specifications in JetBrains MPS.

Extending the Formal Definition of SMMT Due to a lack of time we have not been able to formally define the syntax and semantics of the complete set of constructs of SMMT. The syntax and semantics of SMMT specifications that include all constructs, except for *Forward* and *SelfPost BehavioralActions*, are defined in mCRL2 and can be found in Appendix C. Using the mCRL2 specification in Appendix C, the formal definition as presented in Chapter 5 can be extended with *JointStates*, Guards, *DoEvents*, entry handlers and exit handlers.

Out of the 26 SMMT specification that exist at the time of writing, no SMMT specification contains either an *Forward* or *SelfPost BehaviorAction*. It would be interesting to analyse why these two constructs do not occur in the existing SMMT specifications. Depending on this analysis, the formal definition and translation of SMMT specifications to mCRL2 specifications can be extended with *Forward* and *SelfPost BehaviorActions*.

Multi-threaded conversion of the generated mCRL2 specifications The experiments that we performed in Chapter 7 show that the runtime of the conversion of the generated mCRL2 specifications into LPSs and LTSs is not reduced when multiple cores are used to perform the conversions. It could be interesting to investigate why the runtime of these conversions is not reduced when multiple cores are used.

SMMT specification specific properties Due to a lack of time we have not been able to verify SMMT specification specific properties. It could be interesting to define and verify a number of properties that must hold for the existing SMMT specifications.

References

[1] Jan Friso Groote and Mohammad R. Mousavi. *Modeling and analysis of communicating systems*. English. MIT Press, 2014. ISBN: 978-0-262-02771-7.

[2] Olav Bunte et al. "The mCRL2 Toolset for Analysing Concurrent Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2019, LNCS vol. 11428. Cham: Springer International Publishing, 2019, pp. 21–39. ISBN: 978-3-030-17465-1.

[3] Jan A. Bergstra and Jan W. Klop. "Process algebra for synchronous communication". In: *Information and Control* 60.1 (1984), pp. 109–137. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(84)80025-X.

[4] *Boost C++ Libraries*. Accessed on 19 October 2023. URL: https://www.boost.org/.

[5] *Boost Statechart Documentation*. Accessed on 19 October 2023. URL: https://www.boost.org/doc/libs/1_83_0/libs/statechart/doc/index.html.

[6] Olav Bunte, Louis C.M. van Gool, and Tim A.C. Willemse. "Formal verification of OIL component specifications using mCRL2". In: *International Journal on Software Tools for Technology Transfer* 24.3 (June 2022), pp. 441–472. ISSN: 1433-2787. DOI: 10.1007/s10009-022-00658-y. URL: https://doi.org/10.1007/s10009-022-00658-y.

[7] Jasper Denkers, Louis C.M. van Gool, and Eelco Visser. "Migrating Custom DSL Implementations to a Language Workbench (Tool Demo)". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 205–209. ISBN: 9781450360296. DOI: 10.1145/3276604.3276608. URL: https://doi.org/10.1145/3276604.3276608.

[8] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 444–463. ISSN: 0362-1340. DOI: 10.1145/1932682.1869497. URL: https://doi.org/10.1145/1932682.1869497.

[9] Mark H.M. Frenken. "Code Generation and Model-Based Testing in Context of OIL". Master's thesis. Dec. 2019. URL: https://research.tue.nl/nl/studentTheses/code-generation-and-model-based-testing-in-context-of-oil.

[10] Tom Buskens. "Optimizing the Code Generator for OIL". Master's thesis. Oct. 2021. URL: https://research.tue.nl/en/studentTheses/optimizing-the-code-generator-for-oil.

[11] Rutger van Beusekom et al. "Dezyne: Paving the Way to Practical Formal Software Engineering". In: *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021*. Ed. by José Proença and Andrei Paskevich. Vol. 338. EPTCS. 2021, pp. 19–30. DOI: 10.4204/EPTCS.338.4. URL: https://doi.org/10.4204/EPTCS.338.4.

[12] Rutger van Beusekom et al. "Formalising the Dezyne Modelling Language in mCRL2". In: Aug. 2017, pp. 217–233. ISBN: 978-3-319-67112-3. DOI: 10.1007/978-3-319-67113-0_14.

[13] International Organization for Standardization, ed. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[14] Thomas Gibson-Robinson and Philippa Hopcroft. *Coco Platform*. URL: `https://cocotec.io/`.

[15] Thomas Gibson-Robinson et al. "FDR3 — A Modern Refinement Checker for CSP". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 187–201.

[16] K. Morris et al. "Formal verification and validation of run-to-completion style state charts using Event-B". In: *Innovations in Systems and Software Engineering* 18.4 (Dec. 2022), pp. 523–541. ISSN: 1614-5054. DOI: `10.1007/s11334-021-00416-4`. URL: `https://doi.org/10.1007/s11334-021-00416-4`.

[17] Yi Ling Hwong et al. "Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider". In: *Science of Computer Programming* 78.12 (2013). Special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011), pp. 2435–2452. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2012.11.009`. URL: `https://www.sciencedirect.com/science/article/pii/S0167642312002365`.

[18] Anna Stramaglia and Jeroen J.A. Keiren. "Formal Verification of an Industrial UML-like Model using mCRL2". In: *Formal Methods for Industrial Critical Systems*. Ed. by Jan Friso Groote and Marieke Huisman. Cham: Springer International Publishing, 2022, pp. 86–102. ISBN: 978-3-031-15008-1.

[19] Helle H. Hansen et al. "Towards model checking executable UML specifications in mCRL2". In: *Innovations in Systems and Software Engineering* 6.1 (Mar. 2010), pp. 83–90. ISSN: 1614-5054. DOI: `10.1007/s11334-009-0116-1`. URL: `https://doi.org/10.1007/s11334-009-0116-1`.

[20] Michele Pasqua, Massimo Comuzzo, and Marino Miculan. "The AbU Language: IoT Distributed Programming Made Easy". In: *IEEE Access* 10 (2022), pp. 132763–132776. DOI: `10.1109/ACCESS.2022.3230287`.

[21] *JetBrains MPS*. URL: `https://www.jetbrains.com/mps/`.

[22] *MPS Sequence Type*. URL: `https://www.jetbrains.com/help/mps/sequence.html`.

[23] *mCRL2 Tool Documentation: ltscompare*. Accessed on 23 October 2023. URL: `https://www.mcrl2.org/web/user_manual/tools/release/ltscompare.html`.

[24] *mCRL2 Tool Documentation: mcrl22lps*. Accessed on 23 October 2023. URL: `https://www.mcrl2.org/web/user_manual/tools/release/mcrl22lps.html`.

[25] *mCRL2 Tool Documentation: lps2lts*. Accessed on 23 October 2023. URL: `https://www.mcrl2.org/web/user_manual/tools/release/lps2lts.html`.

[26] Rob P. Nederpelt and Fairouz D. Kamareddine. "Precedence of Logical Connectives". In: *Logical reasoning A first course*. College Publications, 2011, pp. 13–80.

[27] *mCRL2 Tool Documentation: lts2pbes*. Accessed on 23 October 2023. URL: `https://www.mcrl2.org/web/user_manual/tools/release/lts2pbes.html`.

[28] *mCRL2 Tool Documentation: pbessolve*. Accessed on 23 October 2023. URL: `https://www.mcrl2.org/web/user_manual/tools/release/pbessolve.html`.

Appendix A

Proofs for Chapter 5

Lemma 1 (Entry Child Relation $\sqsubset_{ES}$). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The entry child relation $\sqsubset_{ES}$ as defined in Definition 9 is equivalent to $\sqsubset \cap (ES \times S_C)$, that is:

$$\sqsubset_{ES} \equiv \sqsubset \cap (ES \times S_C)$$

Proof. Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. By Definition 9 we have that:

$$\forall_{s,s' \in \mathcal{S}(\mathtt{M})} : s \sqsubset_{ES} s' \Leftrightarrow s \in \mathcal{EC}(\mathtt{M}, s')$$

Hence, it follows that:

$$\sqsubset_{ES} = \{(s, s') \in (\mathcal{S}(\mathtt{M}) \times \mathcal{S}(\mathtt{M})) \mid s \in \mathcal{EC}(\mathtt{M}, s')\}$$

We show that $\sqsubset_{ES} \equiv \sqsubset \cap (ES \times S_C)$:

$$\begin{aligned}
\sqsubset_{ES} &= \{(s, s') \in (\mathcal{S}(\mathtt{M}) \times \mathcal{S}(\mathtt{M})) \mid s \in \mathcal{EC}(\mathtt{M}, s')\} \\
&\equiv \{(s, s') \in (\mathcal{S}(\mathtt{M}) \times \mathcal{S}(\mathtt{M})) \mid s \in \{x \in ES \mid x \sqsubset s'\}\} && \text{(Definition 8)} \\
&\equiv \{(s, s') \in (\mathcal{S}(\mathtt{M}) \times \mathcal{S}(\mathtt{M})) \mid s \in ES \land s \sqsubset s'\} \\
&\equiv \{(s, s') \in (ES \times \mathcal{S}(\mathtt{M})) \mid s \sqsubset s'\} && (ES \subseteq \mathcal{S}(\mathtt{M})) \\
&\equiv \{(s, s') \in (ES \times S_C) \mid s \sqsubset s'\} && \text{(Restriction 4)} \\
&\equiv \{(s, s') \in (\sqsubset \cap (ES \times S_C)) \mid \mathtt{true}\} \\
&\equiv \sqsubset \cap (ES \times S_C)
\end{aligned}$$

Hence, this proves that $\sqsubset_{ES} \equiv \sqsubset \cap (ES \times S_C)$ □

Lemma 2 (Execution State). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The set of execution states $\mathcal{EXS}(\mathtt{M})$ consists of exactly one execution state for each *SimpleState* $s \in S_S$ that consists of state s and all ancestors of state s. Hence, the set of execution states can be derived as follows:

$$\mathcal{EXS}(\mathtt{M}) \equiv \bigcup_{s \in S_S} \{\{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^* s'\}\}$$

Proof. Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. To prove that the claim holds, we show the that:

1. All execution states $EX \in \mathcal{EXS}(\mathtt{M})$ consist of exactly one *SimpleState* $s \in S_S$ and all ancestors of state s, that is:

$$\forall_{EX \in \mathcal{EXS}(\mathtt{M})} : \exists!_{s \in S_S} : EX = \{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^* s'\}$$

2. For each *SimpleState* $s \in S_S$, $\mathcal{EXS}(\mathtt{M})$ contains exactly one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ such that $s \in EX$, that is:

$$\forall_{s \in S_S} : \exists!_{EX \in \mathcal{EXS}(\mathtt{M})} : s \in EX$$

We first prove that each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ consists of exactly one *SimpleState* $s \in S_S$ and all ancestors of s. Let $EX \in \mathcal{EXS}(\mathtt{M})$ be an execution state of SMMT specification $\mathtt{M}$. We show that execution state EX contains exactly one *SimpleState* $s \in S_S$ and all ancestors of state s. By Definition 11, it follows that EX contains exactly one root state. We distinguish two cases:

- Assume that execution state EX consists of only one state, that is, $EX = \{t\}$ for some $t \in \mathcal{S}(\mathtt{M})$. By Definition 11 it follows that state $t \in \mathcal{R}(\mathtt{M})$. Furthermore, as execution state EX contains exactly one state and since execution state EX must contain a child $c \in \mathcal{S}(\mathtt{M})$ for each *CompositeState* $c' \in (S_C \cap EX)$, it follows that state t is a *SimpleState*. By the definition of a root state, state t has no parents. Hence, it trivially follows that execution state EX contains of exactly one *SimpleState* t and all ancestors of state t.

- Assume that execution state EX consists of more than one state, that is $|EX| > 1$. Therefore, it follows that the root state of EX must be a *CompositeState*. By Definition 11, it follows that exactly one child $c \in \mathcal{S}(\mathtt{M})$ of each *CompositeStates* $c' \in (EX \cap S_C)$ must be contained in EX. As the set of *CompositeStates* is finite and the child relation is acyclic, it follows that there exist exactly one *CompositeState* $c'' \in (EX \cap S_C)$ of which a child is contained in EX that is a *SimpleState*. Furthermore, it follows that EX contains exactly one *SimpleState* $s \in S_S$ and that all *CompositeStates* in EX must be an ancestor of *SimpleState* s.

Hence, in both cases we have shown that each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ consists of exactly one *SimpleState* $s \in S_S$ and all ancestors of state s. It remains to prove that $\mathcal{EXS}(\mathtt{M})$ contains exactly one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$. We show that there exists at least and at most one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$.

- We show that there exists at most one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$. Assume, to derive a contradiction, there exist two distinct execution states $EX, EX' \in \mathcal{EXS}(\mathtt{M})$ such that $s \in EX \land s \in EX'$. By the child relation (Definition 2) it follows that each state has at most one parent. As we have shown that each execution state consist of exactly one *SimpleState* and all ancestors of that *SimpleState*, it follows that $EX = EX'$. We derive a contradiction. There exists at most one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$.

- We show that there exists at least one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$. Assume, to derive a contradiction, there exists a *SimpleState* $s' \in S_S$ for which there does not exist an execution state $EX \in \mathcal{EXS}(\mathtt{M})$ such that $s' \in EX$. Consider set of state $X \subseteq \mathcal{S}(\mathtt{M})$ that is defined as follows:

$$X = \{s'' \in \mathcal{S}(\mathtt{M}) \mid s' \sqsubset^* s''\}$$

We show that $X \in \mathcal{EXS}(\mathtt{M})$. By Definition 11, we have that $X \in \mathcal{EXS}(\mathtt{M})$ if, and only if:

P1: $X \subseteq \mathcal{S}(\mathtt{M})$
P2: $\exists!_{r \,\in\, X} : r \in \mathcal{R}(\mathtt{M})$
P3: $\forall_{s' \,\in\, (S_C \,\cap\, X)} : (\exists!_{s \,\in\, \mathcal{S}(\mathtt{M})} : s \sqsubset s' \wedge s \in X))$
P4: $\forall_{t \,\in\, X} : (\forall_{t' \,\in\, \mathcal{S}(\mathtt{M})} : (t \sqsubset^+ t' \Rightarrow t' \in X))$

Property P1 trivially holds by the definition of X. As each state has at most one parent and the set of states is finite, there exists exactly one ancestor of s' that has no parent. Therefore, property P2 is satisfied. By definition of X, we have that X consists of *SimpleState* s' and all ancestor thereof. As all states have at most one parent, it trivially follows that exactly one child of each *CompositeState* ancestor of s' is contained in X. Therefore, property P3 holds for set X. Property P4 trivially holds by the definition of X. Hence, this shows that $X \in \mathcal{EXS}(\mathtt{M})$. We derive a contradiction as $s' \in X \wedge X \in \mathcal{EXS}(\mathtt{M})$.

Hence, we have shown that for each *SimpleState* $s \in S_S$, $\mathcal{EXS}(\mathtt{M})$ contains exactly one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ such that $s \in EX$.
As we have shown that each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ consists of exactly one *SimpleState* $s \in S_S$ and all ancestors of s and since we have shown that there exists exactly one execution state $EX \in \mathcal{EXS}(\mathtt{M})$ for each *SimpleState* $s \in S_S$ such that $s \in EX$, it trivially follows that the claim holds. That is, it holds that:

$$\mathcal{EXS}(\mathtt{M}) \equiv \bigcup_{s \,\in\, S_S} \{\{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^* s'\}\}$$

□

Lemma 3 (Initial Execution State is an Execution State). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The *initial execution state* $\mathcal{I}(\mathtt{M})$ is an execution state, that is:

$$\mathcal{I}(\mathtt{M}) \in \mathcal{EXS}(\mathtt{M})$$

Proof. By Lemma 2 it follows that $\mathcal{I}(\mathtt{M}) \in \mathcal{EXS}(\mathtt{M})$ if, and only if, there exists a *SimpleState* $s \in (S_S \cap \mathcal{I}(\mathtt{M}))$ such that all states in initial execution state $\mathcal{I}(\mathtt{M})$ are either state s or an ancestor of state s, that is:

$$\exists_{s \,\in\, (S_S \,\cap\, \mathcal{I}(\mathtt{M}))} : \mathcal{I}(\mathtt{M}) = \{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^* s'\}$$

By Restrictions 2 and 3 we have that there exists exactly one entry root state and that each *CompositeState* has exactly one entry child. Therefore, it trivially follows that there exists exactly one *SimpleState* $t \in (S_S \cap ES)$ of which all ancestors are entry states. By Definition 12 we have that:

$$\mathcal{I}(\mathtt{M}) = \{s \in ES \mid \forall_{s' \,\in\, \mathcal{S}(\mathtt{M})} : s \sqsubset^+ s' \Rightarrow s' \in ES\}$$

Hence, it trivially follows that initial execution state $\mathcal{I}(\mathtt{M})$ consists of state t and all ancestors of state t. Therefore, it follows that $\mathcal{I}(\mathtt{M}) \in \mathcal{EXS}(\mathtt{M})$.

□

Lemma 4 (Exactly One Prioritised Transition for an *Enabled OnEvent*). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The set of prioritised transitions $\mathcal{PT}_e(\mathtt{M}, EX)$ contains exactly one target state if *OnEvent* $e \in E$ is *enabled* in execution state $EX \in \mathcal{EXS}(\mathtt{M})$, that is:

$$\forall_{EX \,\in\, \mathcal{EXS}(\mathtt{M})} : (\forall_{e \,\in\, E} : (\mathcal{E}(\mathtt{M}, EX, e) \Rightarrow (|\mathcal{PT}_e(\mathtt{M}, EX)| = 1)))$$

Proof. Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M})$ be an execution state of SMMT specification $\mathtt{M}$. Let *OnEvent* $e \in E$ be an *enabled OnEvent*. We prove that $\mathcal{PT}_e(M, EX)$ contains exactly one state by proving that $\mathcal{PT}_e(M, EX)$ contains at least and at most one state.

- ***Set** $\mathcal{PT}_e(M, EX)$ **contains at least 1 state, that is:***

  $$|\mathcal{PT}_e(M, EX)| \geq 1$$

  Since *OnEvent* e is an *enabled onEvent*, it follows by Definition 14 that: $\mathcal{PT}_e(M, EX) \neq \emptyset$. Hence, it trivially follows that $|\mathcal{PT}_e(M, EX)| \geq 1$.

- ***Set** $\mathcal{PT}_e(M, EX)$ **contains at most 1 state, that is:***

  $$|\mathcal{PT}_e(M, EX)| \leq 1$$

  Assume, to derive a contradiction, that the set of prioritised transitions $\mathcal{PT}(\mathtt{M}, EX)$ contains two or more transitions that are defined for *OnEvent* e. By Restriction 5 it follows that each state $s \in \mathcal{S}(\mathtt{M})$ has at most one transition defined for each *OnEvent* e. Hence, there must exist two distinct active states $s, s' \in EX$ that have a transition defined for *OnEvent* e. Let $\langle e, t\rangle \in \mathcal{T}(s)$ and $\langle e, t'\rangle \in \mathcal{T}(s')$, where $t, t' \in \mathcal{S}(\mathtt{M}')$.

  By Definition 13 we have that:

  $$\begin{array}{lll} \langle e, t\rangle \in \mathcal{PT}(\mathtt{M}, EX) & \text{if, and only if,} & \neg\exists_{x \,\in\, EX} : (x \sqsubset^+ s \wedge (\exists_{x' \,\in\, \mathcal{S}(\mathtt{M})} : \langle e, x'\rangle \in \mathcal{T}(x))) \\ \langle e, t'\rangle \in \mathcal{PT}(\mathtt{M}, EX) & \text{if, and only if,} & \neg\exists_{y \,\in\, EX} : (y \sqsubset^+ s' \wedge (\exists_{y' \,\in\, \mathcal{S}(\mathtt{M})} : \langle e, y'\rangle \in \mathcal{T}(y))) \end{array}$$

  By Lemma 2, it follows that execution state EX consists of exactly one *SimpleState* $t \in S_S$ and all ancestors of state t. Therefore, all states in EX are related by descendant relation $\sqsubset^+$. As s and s' are distinct states it follows that either $s \sqsubset^+ s'$ or $s' \sqsubset^+ s$ holds. We distinguish two cases:

  - If $s \sqsubset^+ s'$ then it follows that transition $\langle e, t'\rangle \notin \mathcal{PT}(\mathtt{M}, EX)$. This holds as $s \sqsubset^+ s' \wedge \langle e, t\rangle \in \mathcal{T}(s)$. We derive a contradiction. If $s \sqsubset^+ s'$ then $\mathcal{PT}(\mathtt{M}, EX)$ contains at most one transition that is defined for *OnEvent* e.
  - If $s' \sqsubset^+ s$ then it follows that transition $\langle e, t\rangle \notin \mathcal{PT}(\mathtt{M}, EX)$. This holds as $s' \sqsubset^+ s \wedge \langle e, t'\rangle \in \mathcal{T}(s')$. We derive a contradiction. If $s' \sqsubset^+ s$ then $\mathcal{PT}(\mathtt{M}, EX)$ contains at most one transition that is defined for *OnEvent* e.

  Hence, in both cases $\mathcal{PT}(\mathtt{M}, EX)$ contains at most one transition that is defined for *OnEvent* e. As $\mathcal{PT}_e(\mathtt{M}, EX)$ is defined as the set consisting of all target states of the transitions that are defined in $\mathcal{PT}(\mathtt{M}, EX)$ for *OnEvent* e, it trivially follows that $|\mathcal{PT}_e(\mathtt{M}, EX)| \leq 1$

As we have shown that $|\mathcal{PT}_e(M, EX)| \geq 1$ and $|\mathcal{PT}_e(M, EX)| \leq 1$ hold, it trivially follows that $|\mathcal{PT}_e(M, EX)| = 1$. Hence, The set of target states of the prioritised transitions that are defined for *OnEvent* e in $\mathcal{PT}(\mathtt{M}, EX)$ contains exactly one state, that is

$$|\mathcal{PT}_e(M, EX)| = 1$$

□

Lemma 5 (Execution State Update returns an Execution State). Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. For each execution state $EX \in \mathcal{EXS}(\mathtt{M})$ and each enabled *OnEvent* $e \in E$, $\mathcal{ESU}(\mathtt{M}, EX, e)$ is an execution state, that is:

$$\forall_{EX \,\in\, \mathcal{EXS}(\mathtt{M})} : (\forall_{e \,\in\, E} : (\mathcal{E}(\mathtt{M}, EX, e) \Rightarrow (\mathcal{ESU}(\mathtt{M}, EX, e) \in \mathcal{EXS}(\mathtt{M}))))$$

Proof. Let $\mathtt{M} = \langle E, S_S, S_C, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification, $EX \in \mathcal{EXS}(\mathtt{M})$ be an execution state of SMMT specification $\mathtt{M}$ and $e \in E$ be an enabled *OnEvent*. By Definition 15 we have that:

$$\mathcal{ESU}(\mathtt{M}, EX, e) = \{s \in \mathcal{S}(\mathtt{M}) \mid \exists_{s' \in \mathcal{PT}_e(\mathtt{M}, EX)} : s' \sqsubset^{+} s \vee s \sqsubset^{*}_{ES} s'\}$$

As *OnEvent* e is enabled, it follows by Lemma 4 that $|\mathcal{PT}_e(\mathtt{M}, EX)| = 1$. Let $t \in \mathcal{S}(\mathtt{M})$ be the target state in $\mathcal{PT}_e(\mathtt{M}, EX)$, that is, $\mathcal{PT}_e(\mathtt{M}, E) = \{t\}$. Hence, we have that:

$$\begin{aligned}
\mathcal{ESU}(\mathtt{M}, EX, e) &= \{s \in \mathcal{S}(\mathtt{M}) \mid \exists_{s' \in \mathcal{PT}_e(\mathtt{M}, EX)} : s' \sqsubset^{+} s \vee s \sqsubset^{*}_{ES} s'\} \\
&= \{s \in \mathcal{S}(\mathtt{M}) \mid \exists_{s' \in \{t\}} : s' \sqsubset^{+} s \vee s \sqsubset^{*}_{ES} s'\} \\
&= \{s \in \mathcal{S}(\mathtt{M}) \mid t \sqsubset^{+} s \vee s \sqsubset^{*}_{ES} t\}
\end{aligned}$$

By Lemma 2 it follows that $\mathcal{ESU}(\mathtt{M}, EX, e) \in \mathcal{EXS}(\mathtt{M})$ if, and only if, there exists a *SimpleState* $s \in (S_S \cap \mathcal{ESU}(\mathtt{M}, EX, e))$ such that all states in $\mathcal{ESU}(\mathtt{M}, EX, e)$ are either state s or an ancestor of state s, that is:

$$\mathcal{ESU}(\mathtt{M}, EX, e) = \{s' \in \mathcal{S}(\mathtt{M}) \mid s \sqsubset^{*} s'\}$$

We distinguish two cases:

- Assume that state t is a *SimpleState*. By Restriction 4 it follows that state t has no children. Hence, it follows that:

$$\mathcal{ESU}(\mathtt{M}, EX, e) = \{s' \in \mathcal{S}(\mathtt{M}) \mid t \sqsubset^{*} s'\}$$

- Assume that state t is a *CompositeState*. By Restriction 4 it follows that all ancestors of state t are *CompositeStates*. As all *CompositeStates* have exactly one entry child (Restriction 3), the set of *CompositeStates* is finite and the child relation is acyclic, it follows that state t has exactly one entry descendant that is a *SimpleState*. Hence, $\mathcal{ESU}(\mathtt{M}, EX, e)$ contains exactly one *SimpleState* $x \in S_S$. Trivially, it follows that all states $r \in \mathcal{ESU}(\mathtt{M}, EX, e)$ are either equal to state x or an ancestor of state x. Hence, it follows that:

$$\mathcal{ESU}(\mathtt{M}, EX, e) = \{s' \in \mathcal{S}(\mathtt{M}) \mid x \sqsubset^{*} s'\}$$

We have shown in both cases that the set of states $\mathcal{ESU}(\mathtt{M}, EX, e)$ can be expressed as a *SimpleState* $s \in S_S$ and all ancestors of state s. Therefore, it follows by Lemma 2 that $\mathcal{ESU}(\mathtt{M}, EX, e) \in \mathcal{EXS}(\mathtt{M})$.

□

Lemma 6 (Entry Child Relation $\sqsubset_{ES}$ (SMMT Specifications with *ParallelStates*)). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The entry child relation $\sqsubset_{ES}$ as defined in Definition 9 where all occurrences of M are replaced by M′ is equivalent to $\sqsubset \cap (ES \times (S_C \cup S_P))$, that is:

$$\sqsubset_{ES} \equiv \sqsubset \cap (ES \times (S_C \cup S_P))$$

Proof. Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The definition of the entry child relation of SMMT specification M′ corresponds to Definition 9 where all occurrences of M are replaced by M′, hence we have that:

$$\forall_{s,s' \in \mathcal{S}(\mathtt{M}')} : s \sqsubset_{ES} s' \Leftrightarrow s \in \mathcal{EC}(\mathtt{M}', s')$$

Hence, it follows that:

$$\sqsubset_{ES} = \{(s,s') \in (\mathcal{S}(\mathtt{M}') \times \mathcal{S}(\mathtt{M}')) \mid s \in \mathcal{EC}(\mathtt{M}', s')\}$$

We show that $\sqsubset_{ES} \equiv \sqsubset \cap (ES \times (S_C \cup S_P))$:

$$\begin{aligned}
\sqsubset_{ES} &= \{(s,s') \in (\mathcal{S}(\mathtt{M}') \times \mathcal{S}(\mathtt{M}')) \mid s \in \mathcal{EC}(\mathtt{M}', s')\} \\
&\equiv \{(s,s') \in (\mathcal{S}(\mathtt{M}') \times \mathcal{S}(\mathtt{M}')) \mid s \in \{x \in ES \mid x \sqsubset s'\}\} && \text{(Definition 8)} \\
&\equiv \{(s,s') \in (\mathcal{S}(\mathtt{M}') \times \mathcal{S}(\mathtt{M}')) \mid s \in ES \wedge s \sqsubset s'\} \\
&\equiv \{(s,s') \in (ES \times \mathcal{S}(\mathtt{M}')) \mid s \sqsubset s'\} && (ES \subseteq \mathcal{S}(\mathtt{M}')) \\
&\equiv \{(s,s') \in (ES \times (S_C \cup S_P)) \mid s \sqsubset s'\} && \text{(Restriction 4)} \\
&\equiv \{(s,s') \in (\sqsubset \cap (ES \times (S_C \cup S_P))) \mid \mathtt{true}\} \\
&\equiv \sqsubset \cap (ES \times (S_C \cup S_P))
\end{aligned}$$

Hence, this proves that $\sqsubset_{ES} \equiv \sqsubset \cap (ES \times (S_C \cup S_P))$ ☐

Lemma 7 (Initial Execution State is an Execution State (SMMT Specifications with *ParallelStates*)). Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification. The *initial execution state* $\mathcal{I}(\mathtt{M}')$ is an execution state, that is:

$$\mathcal{I}(\mathtt{M}') \in \mathcal{EXS}(\mathtt{M}')$$

Proof. By Definition 19, we have that $\mathcal{I}(\mathtt{M}') \in \mathcal{EXS}(\mathtt{M}')$ if, and only if:

P1: $\mathcal{I}(\mathtt{M}') \subseteq \mathcal{S}(\mathtt{M}')$

P2: $\exists!_{r \,\in\, \mathcal{I}(\mathtt{M}')} : r \in \mathcal{R}(\mathtt{M}')$

P3: $\forall_{s' \,\in\, (S_C \,\cap\, \mathcal{I}(\mathtt{M}'))} : (\exists!_{s \,\in\, \mathcal{S}(\mathtt{M}')} : s \sqsubset s' \wedge s \in \mathcal{I}(\mathtt{M}'))$

P4: $\forall_{t' \,\in\, (S_P \,\cap\, \mathcal{I}(\mathtt{M}'))} : (\forall_{t \,\in\, \mathcal{S}(\mathtt{M}')} : t \sqsubset t' \Rightarrow t \in \mathcal{I}(\mathtt{M}'))$

P5: $\forall_{u \,\in\, \mathcal{I}(\mathtt{M}')} : (\forall_{u' \,\in\, \mathcal{S}(\mathtt{M}')} : (u \sqsubset^{+} u' \Rightarrow u' \in \mathcal{I}(\mathtt{M}')))$

The initial execution state $\mathcal{I}(\mathtt{M}')$ is defined as in Definition 12 where all occurrences of $\mathtt{M}$ are replaced by $\mathtt{M}'$, that is:

$$\mathcal{I}(\mathtt{M}') = \{s \in ES \mid \forall_{s' \in \mathcal{S}(\mathtt{M}')} : s \sqsubset^{+} s' \Rightarrow s' \in ES\}$$

To prove that the initial execution state $\mathcal{I}(\mathtt{M}')$ is an execution state, we show that properties P1 to P5 hold on $\mathcal{I}(\mathtt{M}')$.

P1. By the definition of $\mathcal{I}(\mathtt{M}')$ we have that $\mathcal{I}(\mathtt{M}') \subseteq ES$. Since $ES \subseteq \mathcal{S}(\mathtt{M}')$ (Definitions 17 and 18), it follows that $\mathcal{I}(\mathtt{M}') \subseteq \mathcal{S}(\mathtt{M}')$.

P2. We prove that $\mathcal{I}(\mathtt{M}')$ contains exactly one root state, that is:

$$\exists!_{r \,\in\, \mathcal{I}(\mathtt{M}')} : r \in \mathcal{R}(\mathtt{M}')$$

By Restriction 2 we have that there exists exactly one entry root state, that is, $|ES \cap \mathcal{R}(\mathtt{M}')| = 1$. Let $r \in \mathcal{S}(\mathtt{M}')$ be the entry root state of SMMT specification $\mathtt{M}'$, that is, $ES \cap \mathcal{R}(\mathtt{M}') = \{r\}$. By the definition of a root state, it follows that root state r has no parent. Hence, it trivially follows that all ancestors of root state r are entry states, that is:

$$\forall_{r' \,\in\, \mathcal{S}(\mathtt{M}')} : r \sqsubset^{+} r' \Rightarrow r' \in ES$$

Therefore, it follows by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ that root state r is contained in initial execution state $\mathcal{I}(\mathtt{M}')$, that is: $r \in \mathcal{I}(\mathtt{M}')$. As $r \in \mathcal{I}(\mathtt{M}')$, $\mathcal{I}(\mathtt{M}') \subseteq ES$ and $ES \cap \mathcal{R}(\mathtt{M}') = \{r\}$ it follows that $\mathcal{I}(\mathtt{M}')$ contains exactly one root state.

P3. We show that exactly one child $s \in \mathcal{I}(\mathtt{M}')$ of each *CompositeStates* $s' \in (S_C \cap \mathcal{I}(\mathtt{M}'))$ is contained in $\mathcal{I}(\mathtt{M}')$, that is:

$$\forall_{s' \,\in\, (S_C \,\cap\, \mathcal{I}(\mathtt{M}'))} : (\exists!_{s \,\in\, \mathcal{S}(\mathtt{M}')} : s \sqsubset s' \wedge s \in \mathcal{I}(\mathtt{M}'))$$

Let $c' \in (S_C \cap \mathcal{I}(\mathtt{M}'))$ be a *CompositeState* that is contained in initial execution state $\mathcal{I}(\mathtt{M}')$. We prove that exactly one child $c \in \mathcal{S}(\mathtt{M}')$ of *CompositeState* c' is contained in $\mathcal{I}(\mathtt{M}')$.

By Restriction 3 we have that each *CompositeState* has exactly one entry child. Let $c'' \in \mathcal{S}(\mathtt{M}')$ be the entry child of *CompositeState* c'. As $c' \in \mathcal{I}(\mathtt{M}')$, it follows by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ that all ancestors of state c' are entry states, that is:

$$\forall_{x \,\in\, \mathcal{S}(\mathtt{M}')} : (c' \sqsubset^{+} x \Rightarrow x \in ES)$$

As $c'' \sqsubset_{ES} c' \wedge c' \in ES$, it follows that:

$$\forall_{x \in \mathcal{S}(\mathtt{M}')} : (c'' \sqsubset^{+} x \Rightarrow x \in ES)$$

Hence, by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ it follows that $c'' \in \mathcal{I}(\mathtt{M}')$. As $c'' \in \mathcal{I}(\mathtt{M}')$ and state c'' is the only entry child of *CompositeState* c' it follows that initial execution state $\mathcal{I}(\mathtt{M}')$ contains exactly one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{I}(\mathtt{M}'))$.

P4. We show that all children $t \in \mathcal{S}(\mathtt{M}')$ of each *ParallelState* $t' \in (S_P \cap \mathcal{I}(\mathtt{M}'))$ are contained in initial execution state $\mathcal{I}(\mathtt{M}')$, that is:

$$\forall_{t' \in (S_P \cap \mathcal{I}(\mathtt{M}'))} : (\forall_{t \in \mathcal{S}(\mathtt{M}')} : t \sqsubset t' \Rightarrow t \in \mathcal{I}(\mathtt{M}'))$$

Let $p' \in (S_P \cap \mathcal{I}(\mathtt{M}'))$ be a *ParallelState* that is contained in initial execution state $\mathcal{I}(\mathtt{M}')$. We prove that all children $p \in \mathcal{S}(\mathtt{M}')$ of *ParallelState* p' is contained in initial execution state $\mathcal{I}(\mathtt{M}')$.

By Restriction 6 we have that all children of a *ParallelState* are entry states. Hence, for each child $p \in \mathcal{S}(\mathtt{M}')$ of *ParallelState* p' it follows that $p \sqsubset_{ES} p'$. By the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ it follows that p' and all ancestors of state p' are entry states. Therefore, it trivially follows that all ancestors of each child p are entry states, that is:

$$\forall_{p \in \mathcal{S}(\mathtt{M}')} : (p \sqsubset_{ES} p' \Rightarrow \forall_{p'' \in \mathcal{S}(\mathtt{M}')} : (p \sqsubset^{+} p'' \Rightarrow p'' \in ES))$$

Hence, it follows by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ that all children $p \in \mathcal{S}(\mathtt{M}')$ of a *ParallelState* $p' \in (S_P \cap \mathcal{I}(\mathtt{M}'))$ are contained in initial execution state $\mathcal{I}(\mathtt{M}')$.

P5. We show that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of each state $u \in \mathcal{I}(\mathtt{M}')$ are contained in initial execution state $\mathcal{I}(\mathtt{M}')$, that is:

$$\forall_{u \in \mathcal{I}(\mathtt{M}')} : (\forall_{u' \in \mathcal{S}(\mathtt{M}')} : (u \sqsubset^{+} u' \Rightarrow u' \in \mathcal{I}(\mathtt{M}')))$$

Let $u \in \mathcal{S}(\mathtt{M}')$ be a state that is contained in initial execution state $\mathcal{I}(\mathtt{M}')$. We prove that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of state u are contained in initial execution state $\mathcal{I}(\mathtt{M}')$.

Assume, to derive a contradiction, that there exists a state $s \in \mathcal{I}(\mathtt{M}')$ that has an ancestor $s' \in \mathcal{S}(\mathtt{M}')$ that is not contained in initial execution state $\mathcal{I}(\mathtt{M}')$. As $s \in \mathcal{I}(\mathtt{M}')$, it follows by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$ that state s and all ancestors of state s are entry states, that is:

$$\forall_{s'' \in \mathcal{S}(\mathtt{M}')} : s \sqsubset^{+} s'' \Rightarrow s'' \in ES$$

As $s \sqsubset^{+} s'$ it follows that $s' \in ES$ and that $\forall_{s'' \in \mathcal{S}(\mathtt{M}')} : s' \sqsubset^{+} s'' \Rightarrow s'' \in ES$. Therefore, by the definition of initial execution state $\mathcal{I}(\mathtt{M}')$, it follows that $s' \in \mathcal{I}(\mathtt{M}')$. Hence, this shows that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of each state $u \in \mathcal{I}(\mathtt{M}')$ are contained in initial execution state $\mathcal{I}(\mathtt{M}')$.

As we have shown that properties P1 to P5 hold for initial execution state $\mathcal{I}(\mathtt{M}')$, it follows that the initial execution state $\mathcal{I}(\mathtt{M}')$ is an execution state. □

Lemma 8 (Conflicting Targets).Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification, $EX \in \mathcal{EXS}(\mathtt{M}')$ be an execution state and $e \in E$ be an *OnEvent*. The target states in $\mathcal{PT}_e(\mathtt{M}', EX)$ conflict if, and only if, there does not exist an execution state $EX' \in \mathcal{EXS}(\mathtt{M}')$ such that $\mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'$. Hence, it follows that:

$$\mathcal{CT}(\mathtt{M}', EX, e) \equiv \neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'$$

Proof. Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M}')$ be an execution state of SMMT specification $\mathtt{M}'$. We prove that

$$\mathcal{CT}(\mathtt{M}', EX, e) \equiv \neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'$$

$$\begin{aligned}
\mathcal{CT}(\mathtt{M}', EX, e) &= \exists_{s,s' \,\in\, \mathcal{PT}_e(\mathtt{M}',EX)} : \mathcal{CS}(\mathtt{M}', s, s') && \text{(Definition 21)}\\
&\equiv \exists_{s,s' \,\in\, \mathcal{PT}_e(\mathtt{M}',EX)} : (\neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \{s, s'\} \subseteq EX') && \text{(Definition 20)}\\
&\equiv \neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'
\end{aligned}$$

□

Lemma 9 (Execution State Update returns an Execution State (SMMT Specifications with *ParallelStates*)). For all SMMT specifications $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$, for each execution state $EX \in \mathcal{EXS}(\mathtt{M}')$ and for each *enabled OnEvent* $e \in E$, the execution state update function $\mathcal{ESU}(\mathtt{M}', EX, e)$ returns an execution state, that is:

$$\mathcal{E}(\mathtt{M}', EX, e) \Rightarrow \mathcal{ESU}(\mathtt{M}', EX, e) \in \mathcal{EXS}(\mathtt{M}')$$

Proof. Let $\mathtt{M}' = \langle E, S_S, S_C, S_P, ES, \sqsubset, \mathcal{T}\rangle$ be an SMMT specification and $EX \in \mathcal{EXS}(\mathtt{M}')$ be an execution state of SMMT specification $\mathtt{M}'$. Let $e \in E$ be an enabled *OnEvent*. By Definition 28 we have that:

$$\mathcal{ESU}(\mathtt{M}', EX, e) = \mathcal{INIT}\Big(\mathtt{M}', (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{ET}(\mathtt{M}', EX, e)\Big)$$

By Definition 19, we have that $\mathcal{ESU}(\mathtt{M}', EX, e) \in \mathcal{EXS}(\mathtt{M}')$ if, and only if:

P1: $\mathcal{ESU}(\mathtt{M}', EX, e) \subseteq \mathcal{S}(\mathtt{M}')$

P2: $\exists!_{r \,\in\, \mathcal{ESU}(\mathtt{M}', EX, e)} : r \in \mathcal{R}(\mathtt{M}')$

P3: $\forall_{s' \,\in\, (S_C \,\cap\, \mathcal{ESU}(\mathtt{M}', EX, e))} : (\exists!_{s \,\in\, \mathcal{S}(\mathtt{M}')} : s \sqsubset s' \wedge s \in \mathcal{ESU}(\mathtt{M}', EX, e))$

P4: $\forall_{t' \,\in\, (S_P \,\cap\, \mathcal{ESU}(\mathtt{M}', EX, e))} : (\forall_{t \,\in\, \mathcal{S}(\mathtt{M}')} : t \sqsubset t' \Rightarrow t \in \mathcal{ESU}(\mathtt{M}', EX, e))$

P5: $\forall_{u \,\in\, \mathcal{ESU}(\mathtt{M}', EX, e)} : (\forall_{u' \,\in\, \mathcal{S}(\mathtt{M}')} : (u \sqsubset^{+} u' \Rightarrow u' \in \mathcal{ESU}(\mathtt{M}', EX, e)))$

We show that $\mathcal{ESU}(\mathtt{M}', EX, e) \in \mathcal{EXS}(\mathtt{M}')$ by proving that properties P1 to P5 hold for $\mathcal{ESU}(\mathtt{M}', EX, e)$.

P1. We prove that $\mathcal{ESU}(\mathtt{M}', EX, e)$ is a subset of set of states $\mathcal{S}(\mathtt{M}')$, we have that:

$$\begin{aligned}
&\mathcal{ESU}(\mathtt{M}', EX, e) \\
&= \mathcal{INIT}\Big(\mathtt{M}', (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{ET}(\mathtt{M}', EX, e)\Big) && \text{(Definition 28)} \\
&= ((EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{ET}(\mathtt{M}', EX, e)) \cup \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^{*}_{ES} s'\} && \text{(Definition 27)} \\
&\subseteq ((EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{ET}(\mathtt{M}', EX, e)) \cup \mathcal{S}(\mathtt{M}') \\
&= ((EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{PT}_e(\mathtt{M}', EX)} : s' \in \mathcal{SR}(\mathtt{M}', s)\}) \cup \mathcal{S}(\mathtt{M}') && \text{(Definition 24)} \\
&\subseteq (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)) \cup \mathcal{S}(\mathtt{M}') \\
&\subseteq EX \cup \mathcal{S}(\mathtt{M}') \\
&= \mathcal{S}(\mathtt{M}') && (EX \subseteq \mathcal{S}(\mathtt{M}'))
\end{aligned}$$

Hence, we have that $\mathcal{ESU}(\mathtt{M}', EX, e) \subseteq \mathcal{S}(\mathtt{M}')$.

P2. We prove that $\mathcal{ESU}(\mathtt{M}', EX, e)$ contains exactly one root state $r \in \mathcal{R}(\mathtt{M}')$, that is:

$$\exists!_{r \,\in\, \mathcal{ESU}(\mathtt{M}', EX, e)} : r \in \mathcal{R}(\mathtt{M}')$$

As $EX \in \mathcal{EXS}(\mathtt{M}')$, it follows by Definition 19 that EX contains exactly one root state $r \in \mathcal{R}(\mathtt{M}')$, that is:

$$\exists_{r \,\in\, EX} : r \in \mathcal{R}(\mathtt{M}')$$

We first prove that $\mathcal{ET}(\mathtt{M}', EX, e)$ has exactly one root state, that is:

$$\exists!_{r \in \mathcal{R}(\mathtt{M}')} : r \in \mathcal{ET}(\mathtt{M}', EX, e)$$

- Since *OnEvent* e is enabled in execution state EX, we have by Definition 23 that, among others:

  $$\mathcal{PT}_e(\mathtt{M}', EX) \neq \emptyset \wedge \neg\mathcal{CT}(\mathtt{M}', EX, e)$$

  As $\mathcal{PT}_e(\mathtt{M}', EX) \neq \emptyset$, it follows that there exists a state $s \in \mathcal{PT}_e(\mathtt{M}', EX)$. By definition of the set of entered targets (Definition 24), it follows that s and all ancestors of s are contained in $\mathcal{ET}(\mathtt{M}', EX, e)$. Hence, it trivially follows that $\mathcal{ET}(\mathtt{M}', EX, e)$ contains at least one root state.

  As $\neg\mathcal{CT}(\mathtt{M}', EX, e)$, it follows by Definition 21 that there exists an execution state $EX' \in \mathcal{EXS}(\mathtt{M}')$ such that $\mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'$. By Definition 19, it follows that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of each state $u \in EX'$ are contained in execution state EX'. Therefore, as the set of entered targets is defined as the states in $\mathcal{PT}_e(\mathtt{M}', EX)$ and all ancestors thereof, it trivially follows that $\mathcal{ET}(\mathtt{M}', EX, e) \subseteq EX'$. By Definition 19 it follows that EX' has at most one root state. As $\mathcal{ET}(\mathtt{M}', EX, e) \subseteq EX'$, it therefore follows that $\mathcal{ET}(\mathtt{M}', EX, e)$ has at most one root state.

  As we have shown that $\mathcal{ET}(\mathtt{M}', EX, e)$ contains at least and at most one root state, we can conclude that $\mathcal{ET}(\mathtt{M}', EX, e)$ contains exactly one root state.

Let $X = (EX \setminus \mathcal{XS}(\mathtt{M}')) \cup \mathcal{ET}(\mathtt{M}', EX, e)$. We show that set X contains exactly one root state, that is:

$$\exists!_{r \,\in\, X} : r \in \mathcal{R}(\mathtt{M}')$$

As mentioned before, both EX and $\mathcal{ET}(\mathtt{M}', EX, e)$ have each exactly one root state. Let $t \in \mathcal{R}(\mathtt{M}')$ be the root state that is contained in execution state EX and let $t' \in \mathcal{R}(\mathtt{M}')$ be the root state that is contained in the set of entered targets $\mathcal{ET}(\mathtt{M}', EX, e)$. We distinguish two cases:

- Execution state EX and the set of entered targets $\mathcal{ET}(\mathtt{M}', EX, e)$ contain the same root state, that is $t = t'$. We show that the set of exited states $\mathcal{XS}(\mathtt{M}', EX, e)$ has no root states, that is $(\mathcal{XS}(\mathtt{M}', EX, e) \cap \mathcal{R}(\mathtt{M}')) = \emptyset$. As $\mathcal{XS}(\mathtt{M}', EX, e) \subseteq EX$, it follows that $\mathcal{XS}(\mathtt{M}', EX, e)$ has at most one root state, namely root state t. Since *OnEvent* e is enabled in execution state EX, it follows by Definition 23 that $\neg\mathcal{CT}(\mathtt{M}', EX, e)$. Therefore, it follows by Definition 21 that there must exist an execution state $EX' \in \mathcal{EXS}(\mathtt{M}')$ such that $\mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'$, that is:

  $$\begin{aligned}&\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \mathcal{PT}_e(\mathtt{M}', EX) \subseteq EX'\\ &\quad\Leftrightarrow \neg\exists_{s' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : (\neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \{t', s'\} \subseteq EX'\end{aligned}$$

  Hence, it follows by the definition of the set of exited states $\mathcal{XS}(\mathtt{M}', EX, e)$ (Definition 25) that:

  $$\begin{aligned}t \in \mathcal{XS}(\mathtt{M}', EX, e) &\Leftrightarrow \exists_{s' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', t, s')\\ &\Leftrightarrow \exists_{s' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : (\neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \{t', s'\} \subseteq EX')\\ &\Leftrightarrow \mathtt{false}\end{aligned}$$

  Hence, $\mathcal{XS}(\mathtt{M}', EX, e) \cap \mathcal{R}(\mathtt{M}') = \emptyset$. Therefore, as $EX \cap \mathcal{R}(\mathtt{M}') = \mathcal{ET}(\mathtt{M}', EX, e) \cap \mathcal{R}(\mathtt{M}') = \{t\}$, it follows that $X \cap \mathcal{R}(\mathtt{M}') = \{t\}$. Hence X contains exactly one root state.

- Execution state EX and the set of entered targets $\mathcal{ET}(\mathtt{M}', EX, e)$ have distinct root states, that is $t \neq t'$. As $\mathcal{XS}(\mathtt{M}', EX, e) \subseteq EX$, it follows that $\mathcal{XS}(\mathtt{M}', EX, e)$ has at most one root state, namely root state t. We show that $t \in \mathcal{XS}(\mathtt{M}', EX, e)$. By Definition 19, we have that there does not exist an execution state that contains more than one root states, therefore we have that there does not exist an execution state that contains both t and t', that is:

  $$\neg\exists_{EX' \,\in\, \mathcal{EXS}(\mathtt{M}')} : \{t, t'\} \subseteq EX'$$

As $t' \in \mathcal{ET}(\texttt{M}', EX, e) \wedge \neg\exists_{EX' \in \mathcal{EXS}(\texttt{M}')} : \{t, t'\} \subseteq EX'$, it follows that states t and t' are conflicting, that is:

$$\mathcal{CS}(\texttt{M}', t, t')$$

Hence, we have that $\exists_{s' \in \mathcal{ET}(\texttt{M}', EX, e)} : \mathcal{CS}(\texttt{M}', t, s')$. By Definition 25, it therefore follows that $t \in \mathcal{XS}(\texttt{M}', EX, e)$.

Hence, as $EX \cap \mathcal{R}(\texttt{M}') = \{t\}$, $\mathcal{ET}(\texttt{M}', EX, e) \cap \mathcal{R}(\texttt{M}') = \{t'\}$ and $t \in \mathcal{XS}(\texttt{M}', EX, e)$, it follows that $X \cap \mathcal{R}(\texttt{M}') = \{t'\}$. Therefore, we conclude that X contains exactly one root state.

As set X contains exactly one root state, it follows that $\mathcal{ESU}(\texttt{M}', EX, e)$ has exactly one root state if $(\mathcal{INIT}(\texttt{M}', X) \setminus X) \cap \mathcal{R}(\texttt{M}') = \emptyset$. That is:

$$\neg\exists_{r \in \mathcal{R}(\texttt{M}')} : r \in \{s \in \mathcal{S}(\texttt{M}') \mid \exists_{s' \in \mathcal{MC}(\texttt{M}', X)} : s \sqsubset^*_{ES} s'\}$$

Assume, to derive a contradiction, that there exists a root state $r \in \mathcal{R}(\texttt{M}')$ in $(\mathcal{INIT}(\texttt{M}', X) \setminus X)$. Hence, we have that:

$$\exists_{s' \in \mathcal{MC}(\texttt{M}', X)} : r \sqsubset^*_{ES} s'$$

As r is a root state, we have that there does not exist a state $r' \in \mathcal{S}(\texttt{M}')$ that is a parent of r. That is, we have that:

$$\neg\exists_{s' \in \mathcal{MC}(\texttt{M}', X)} : r \sqsubset^+_{ES} s'$$

Hence, it follows that:

$$r \in \mathcal{MC}(\texttt{M}', X)$$

By Definition 26, it follows that $r \in \mathcal{MC}(\texttt{M}', X)$ if, among others, there exists a *CompositeState* or *ParallelState* that is a parent of state r. As r is a root state, it trivially follows that r has no parent. Hence, $r \notin \mathcal{MC}(\texttt{M}', X)$. We derive a contradiction. Hence, we conclude that $\mathcal{ESU}(\texttt{M}', EX, e)$ contains exactly one root state.

P3. We prove that exactly one child $s \in \mathcal{S}(\texttt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{ESU}(\texttt{M}', EX, e))$ is contained in the set $\mathcal{ESU}(\texttt{M}', EX, e)$, that is:

$$\forall_{s' \in (S_C \cap \mathcal{ESU}(\texttt{M}', EX, e))} : (\exists!_{s \in \mathcal{S}(\texttt{M}')} : s \sqsubset s' \wedge s \in \mathcal{ESU}(\texttt{M}', EX, e))$$

Let $X = (EX \setminus \mathcal{XS}(\texttt{M}')) \cup \mathcal{ET}(\texttt{M}', EX, e)$. In order to prove that $\mathcal{ESU}(\texttt{M}', EX, e)$ contains exactly one child $s \in \mathcal{S}(\texttt{M}')$ of each *CompositeState* $s' \in S_C \cap \mathcal{ESU}(\texttt{M}', EX, e)$, we show that:

R1. Set X contains at most one child $t \in \mathcal{S}(\texttt{M}')$ of each *CompositeState* $t' \in (S_C \cap X)$.

R2. All ancestors $s' \in \mathcal{S}(\texttt{M}')$ of each state $s \in X$ are contained in X

R3. Set $\mathcal{INIT}(\texttt{M}', X)$ contains exactly one child $u \in \mathcal{S}(\texttt{M}')$ given that R1 and R2 hold.

We prove requirements R1 to R3:

R1. As $EX \in \mathcal{EXS}(\texttt{M}')$ we have by Definition 19 that:

$$\forall_{s' \in (S_C \cap EX)} : (\exists!_{s \in EX} : s \sqsubset s')$$

Hence, it trivially follows that $EX \setminus \mathcal{XS}(\texttt{M}', EX, e)$ contains at most one child of each *CompositeState* in $EX \setminus \mathcal{XS}(\texttt{M}', EX, e)$, that is:

$$\forall_{s' \in (S_C \cap (EX \setminus \mathcal{XS}(\texttt{M}', EX, e)))} : (\neg\exists_{s, s'' \in (EX \setminus \mathcal{XS}(\texttt{M}', EX, e))} : s \neq s'' \wedge s \sqsubset s' \wedge s'' \sqsubset s')$$

Furthermore, as *OnEvent* e is *enabled*, it follows that $\neg\mathcal{CT}(\texttt{M}', EX, e)$. Hence, by Definition 21 we have that:

$$\exists_{EX' \in \mathcal{EXS}(\texttt{M}')} : \mathcal{PT}_e(\texttt{M}', EX) \subseteq EX'$$

By Definition 19, it therefore follows that at most one child of each *CompositeState* in $\mathcal{ET}(\mathtt{M}', EX, e)$ is contained in $\mathcal{ET}(\mathtt{M}', EX, e)$:

$$\forall_{s' \,\in\, (S_C \,\cap\, \mathcal{ET}(\mathtt{M}', EX, e))} : (\neg\exists_{s,s'' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e))} : s \neq s'' \wedge s \sqsubset s' \wedge s'' \sqsubset s')$$

Assume, to derive a contradiction, that there exists a *CompositeState* $c \in (X \cap S_C)$ for which two or more children are contained in X. Let $c', c'' \in X$ be distinct children of *CompositeState* c that are contained in X.

As sets $EX \setminus \mathcal{XS}(\mathtt{M}', EX, e)$ and $\mathcal{ET}(\mathtt{M}', EX, e)$ both contain at most one child per *CompositeState*, it follows that $\{c, c'\} \subseteq (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e))$ and $c'' \in \mathcal{ET}(\mathtt{M}', EX, e)$. As $\{c, c'\} \subseteq (EX \setminus \mathcal{XS}(\mathtt{M}', EX, e))$, it follows that $c' \notin \mathcal{XS}(\mathtt{M}', EX, e)$. Hence, by Definition 25 we have that:

$$\neg\exists_{s \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', s, c')$$

By Definition 20 it follows that states c' and c'' are conflicting as they are both contained in X and are a child of *CompositeState* c, that is:

$$\exists_{s \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', s, c')$$

Hence, we derive a contradiction. Each *CompositeState* $c \in (X \cap S_C)$ has at most one child $c \in \mathcal{S}(\mathtt{M}')$ in set X.

R2. We show that all ancestors $s' \in \mathcal{S}(\mathtt{M}')$ of each state $s \in X$ are contained in X, that is:

$$\forall_{s \,\in\, X} : \forall_{s' \,\in\, \mathcal{S}(\mathtt{M}')} : (s \sqsubset^{+} s' \Rightarrow s' \in X))$$

Assume, to derive a contradiction, that there exists a state $x \in X$ of which an ancestor $x' \in \mathcal{S}(\mathtt{M}')$ is not contained in X. As $EX \in \mathcal{EXS}(\mathtt{M}')$, it follows that all ancestors $r' \in \mathcal{S}(\mathtt{M}')$ of each state $r \in EX$ are contained in EX, that is:

$$\forall_{r \,\in\, EX} : \forall_{r' \,\in\, \mathcal{S}(\mathtt{M}')} : (r \sqsubset^{+} r' \Rightarrow r' \in EX))$$

We distinguish two cases:

* State x is contained in execution state EX, that is $x \in EX$. By Definition 19 it follows that $x' \in EX$ as $EX \in \mathcal{EXS}(\mathtt{M}')$ and $x \sqsubset^{+} x'$. Since $x' \notin X$, it follows that x' is an exited state but it not contained in the set of entered targets, that is:

  $$x' \in \mathcal{XS}(\mathtt{M}', EX, e) \wedge x' \notin \mathcal{ET}(\mathtt{M}', EX, e)$$

  As $x' \in \mathcal{XS}(\mathtt{M}', EX, e)$, it follows by Definition 25 that there exists a state $x'' \in \mathcal{ET}(\mathtt{M}', EX, e)$ such that state x' conflicts with x'', that is:

  $$\exists_{s'' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', x', x'')$$

  As state x is an descendant of state x', it therefore follows that states x and x'' are also conflicting. Therefore, it follow that:

  $$\exists_{s'' \,\in\, \mathcal{ET}(\mathtt{M}', EX, e)} : \mathcal{CS}(\mathtt{M}', x, x'')$$

  By Definition 25 it follows that $x \in \mathcal{XS}(\mathtt{M}', EX, e)$. Hence, $x \notin (EX \setminus \mathcal{ET}(\mathtt{M}', EX, e))$. Since $x \in X$, it follows by the definition of X that x must be contained in the set of entered targets, that is $x \in \mathcal{ET}(\mathtt{M}', EX, e)$. Hence, by Definition 24 it follows that:

  $$\exists_{t \,\in\, \mathcal{PT}_e(\mathtt{M}', EX)} : t \sqsubset^{*} x$$

  As state x is an descendant of state x', it therefore follows that:

  $$\exists_{t \,\in\, \mathcal{PT}_e(\mathtt{M}', EX)} : t \sqsubset^{*} x'$$

  Hence, by Definition 24 it follows that $x' \in \mathcal{ET}(\mathtt{M}', EX, e)$ and thus by the definition of X that $x' \in X$. We derive a contradiction.

  * State x is not contained in execution state EX, that is $x \notin EX$. If $x \in X \wedge x \notin EX$, then it follows by the definition of X that $x \in \mathcal{ET}(\mathtt{M}', EX, e)$. As shown in the previous case, it follows that $x' \in X$ if $x \in \mathcal{ET}(\mathtt{M}', EX, e) \wedge x \sqsubset x'$. Hence, we derive a contradiction.

  As both cases lead to a contradiction, it follows that all ancestors $s' \in \mathcal{S}(\mathtt{M}')$ of each state $s \in X$ are contained in X.

R3. We prove that set $\mathcal{INIT}(\mathtt{M}', X)$ contains exactly one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{ESU}(\mathtt{M}', EX, e)$, given that requirements R1 and R2 hold.

  * We first prove that set $\mathcal{INIT}(\mathtt{M}', X)$ contains at least one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{ESU}(\mathtt{M}', EX, e))$. Assume, to derive a contradiction, that there exists a *CompositeState* $c' \in \mathcal{ESU}(\mathtt{M}', EX, e)$ of which no children are contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$, that is:

$$\neg(\exists_{s'' \,\in\, \mathcal{ESU}(\mathtt{M}', EX, e)} : s'' \sqsubset c)$$

  By Definitions 27 and 28 it therefore follow that:

$$\neg\exists_{c \,\in\, \mathcal{S}(\mathtt{M}')} : c \sqsubset c' \wedge (c \in X \vee c \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^*_{ES} s'\})$$

  Hence, it holds that:

$$\neg\exists_{c \,\in\, \mathcal{S}(\mathtt{M}')} : c \sqsubset c' \wedge \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : c \sqsubset^*_{ES} s'$$

  By Restriction 3, we have that each *CompositeState* has exactly one entry state. Let $c'' \in \mathcal{S}(\mathtt{M}')$ be the entry state of *CompositeState* c', that is, $c'' \sqsubset_{ES} c'$. As $c' \in (S_C \cap \mathcal{ESU}(\mathtt{M}', EX, e))$, $c'' \sqsubset_{ES} c'$ and $\neg(\exists_{s'' \,\in\, \mathcal{ESU}(\mathtt{M}', EX, e)} : s'' \sqsubset c)$, it follows by Definition 26 that state c'' is a missing child of state c'. Hence, it follows that:

$$\exists_{c'' \,\in\, \mathcal{S}(\mathtt{M}')} : c'' \sqsubset c' \wedge \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : c'' \sqsubset^*_{ES} s'$$

  We derive a contradiction, therefore set $\mathcal{INIT}(\mathtt{M}', X)$ contains at least one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{ESU}(\mathtt{M}', EX, e))$.

  * We prove that set $\mathcal{INIT}(\mathtt{M}', X)$ contains at most one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in (S_C \cap \mathcal{ESU}(\mathtt{M}', EX, e))$, given that requirements R1 and R2 hold. Assume, to derive a contradiction, there exists a *CompositeState* $c' \in \mathcal{ESU}(\mathtt{M}', EX, e)$ that has children $c, c'' \in \mathcal{S}(\mathtt{M}')$ that are contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$. We distinguish four cases:

    · Both states c and c'' are contained in X, that is $\{c, c''\} \subseteq X$. By requirement R1 we have that:

$$\forall_{s' \,\in\, (S_C \cap X)} : (\neg\exists_{s, s'' \,\in\, \mathcal{S}(\mathtt{M}')} : (s \neq s'' \wedge s \sqsubset s' \wedge s'' \sqsubset s'))$$

    Hence, it trivially follows that c and c'' cannot both be contained in X.

    · State c is contained in X and state c'' is not contained in X, that is, $c \in X \wedge c'' \notin X$. Since $c'' \notin X \wedge c'' \in \mathcal{ESU}(\mathtt{M}', EX, e)$ it follows by Definition 27 that:

$$c'' \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^*_{ES} s'\}$$

    Hence, it follows that:

$$\exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : c'' \sqsubset^*_{ES} s'$$

    Let $r \in \mathcal{MC}(\mathtt{M}', X)$ be a missing child such that $c'' \sqsubset^*_{ES} r$. By requirement R2 we have that all ancestors of c are contained in X, that is:

$$\forall_{s' \,\in\, \mathcal{S}(\mathtt{M}')} : c \sqsubset^+ s' \Rightarrow s' \in X$$

Hence, as $c \sqsubset c' \wedge c'' \sqsubset c'$, it follows that all ancestors of c'' are contained in X, that is:

$$\forall_{s' \in \mathcal{S}(\mathtt{M}')} : c'' \sqsubset^{+} s' \Rightarrow s' \in X$$

Hence, it follows that all ancestors of c'' have at least one child that is either c'' or an ancestor of c''. Therefore, it follows by Definition 26 that there does not exist a missing child that is an ancestor of c'', that is:

$$\neg\exists_{s' \in \mathcal{MC}(\mathtt{M}', EX)} : c'' \sqsubset^{*}_{ES} s'$$

Hence, we derive a contradiction.

· State c is not contained in X and state c'' is contained in X, that is, $c \notin X \wedge c'' \in X$. The proof of this case is equal to the proof of the previous case, where the occurrence of c and c'' are swapped.

· Both states c and c'' are not contained in X, that is $c \notin X \wedge c'' \notin X$. As $\{c, c''\} \subseteq \mathcal{ESU}(\mathtt{M}', EX, e)$, it follows by Definition 26 that:

$$(\exists_{s \in \mathcal{MC}(\mathtt{M}', X)} : c \sqsubset^{*}_{ES} s) \wedge (\exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : c'' \sqsubset^{*}_{ES} s')$$

As states c and c'' are both children of state c' and each *CompositeState* has exactly one entry state (Restriction 3), it follows by Definition 26 that at most one child of each *CompositeState* can be a missing child. Therefore, it follows that at most one child of *CompositeState* c' is contained in set $\{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^{*}_{ES} s'\}$. Hence, we derive a contradiction.

As all four cases lead to a contradiction, we can conclude that set $\mathcal{INIT}(\mathtt{M}', X)$ has at most one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in \mathcal{INIT}(\mathtt{M}', X)$ in $\mathcal{INIT}(\mathtt{M}', X)$.

As we showed that set $\mathcal{INIT}(\mathtt{M}', X)$ contains at least and at most one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in \mathcal{INIT}(\mathtt{M}', X)$ in $\mathcal{INIT}(\mathtt{M}', X)$, we can conclude that set $\mathcal{INIT}(\mathtt{M}', X)$ contains exactly one child $c \in \mathcal{S}(\mathtt{M}')$ of each *CompositeState* $c' \in \mathcal{INIT}(\mathtt{M}', X)$ in $\mathcal{INIT}(\mathtt{M}', X)$.

P4. Assume to derive a contradiction, that there exists a *ParallelState* $t' \in (S_P \cap \mathcal{ESU}(\mathtt{M}', EX, e))$ with child $t \in \mathcal{S}(\mathtt{M}')$ that is not contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$, that is, $t \notin \mathcal{ESU}(\mathtt{M}', EX, e)$. We distinguish two cases:

- State t' is contained in set X, that is, $t' \in X$. As $t \notin \mathcal{ESU}(\mathtt{M}', EX, e)$, it follows by Definition 27 that:

$$t \notin X \wedge t \notin \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^{*}_{ES} s'\}$$

By Restriction 6, it follows that all children of a *ParallelState* are entry children. Hence, state t is an entry child of state t': $t \sqsubset_{ES} t'$. As $t \sqsubset_{ES} t' \wedge t' \in (S_P \cap X) \wedge t \notin X$, it follows by Definition 26 that $t \in \mathcal{MC}(\mathtt{M}', X)$.

Since $t \in \mathcal{MC}(\mathtt{M}', X) \wedge t \sqsubset^{*}_{ES} t$ it follows by Definition 27 that:

$$t \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^{*}_{ES} s'\}$$

Therefore, $t \in \mathcal{ESU}(\mathtt{M}', EX, e)$. We derive a contradiction.

- State t' is not contained in set X, that is $t' \notin X$. It follows by Definition 27 that:

$$t' \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^{*}_{ES} s'\}$$

Hence, we have that:

$$\exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : t' \sqsubset^{*}_{ES} s'$$

As $t \sqsubset^{*}_{ES} t'$, it follows that:

$$\exists_{s' \in \mathcal{MC}(\mathtt{M}', X)} : t \sqsubset^{*}_{ES} s'$$

Hence, by Definition 27 we have that $t \in \mathcal{ESU}(\mathtt{M}', EX, e)$. We derive a contradiction.

As both cases lead to a contradiction, we can conclude that all children of each *ParallelState* $t' \in (S_P \cap \mathcal{ESU}(\mathtt{M}', EX, e))$ are contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$

P5. We show that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of each state $u \in \mathcal{ESU}(\mathtt{M}', EX, e)$ are contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$. Let $u' \in \mathcal{S}(\mathtt{M}')$ be an ancestor of state u. We show that $u' \in \mathcal{ESU}(\mathtt{M}', EX, e)$ if $u \in \mathcal{ESU}(\mathtt{M}', EX, e)$. We distinguish two cases:

- State u is contained in X, that is, $u \in X$. When proving property P3, we proved that all ancestors $s' \in \mathcal{S}(\mathtt{M}')$ of each state $s \in X$ are contained in X. Hence, if $u \in X$, it trivially follows that ancestor u' is contained in X. Therefore, by Definitions 27 and 28 it follows that $u' \in \mathcal{ESU}(\mathtt{M}', EX, e)$.
- State u is not contained in X, that is, $u \notin X$. By Definitions 27 and 28 it therefore follows that:
$$u \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^*_{ES} s'\}$$
Hence, we have that:
$$\exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : u \sqsubset^*_{ES} s'$$
Let $x \in \mathcal{MC}(\mathtt{M}', X)$ be the missing child for which holds that $u \sqsubset^*_{ES} x$. We distinguish two cases:
  * State u' is either equal to state x or u' is an ancestor of state x, that is $u' \sqsubset^*_{ES} x$. As $x \in \mathcal{MC}(\mathtt{M}', X) \wedge u' \sqsubset^*_{ES} x$ it follows that:
  $$\exists_{s' \,\in\, \mathcal{S}(\mathtt{M}')} : s' \in \mathcal{MC}(\mathtt{M}', X) \wedge u' \sqsubset^*_{ES} s'$$
  Hence, it follows that:
  $$u' \in \{s \in \mathcal{S}(\mathtt{M}') \mid \exists_{s' \,\in\, \mathcal{MC}(\mathtt{M}', X)} : s \sqsubset^*_{ES} s'\}$$
  Therefore, by Definitions 27 and 28 we have that $u' \in \mathcal{ESU}(\mathtt{M}', EX, e)$.
  * State u' is an ancestor of state x. As $x \in \mathcal{MC}(\mathtt{M}', EX)$ it follows by Definition 26 that there exists a parent of x that is contained in X, that is:
  $$\exists_{s' \,\in\, X} : x \sqsubset s'$$
  Let $x' \in X$ be the parent of missing child x. As $x \sqsubset^+ u' \wedge x \sqsubset x'$, it follows that $x' \sqsubset^* u'$. Since $x' \in X$, it follows that $u' \in X$. Therefore, by Definitions 27 and 28 we have that $u' \in \mathcal{ESU}(\mathtt{M}', EX, e)$.

In both cases we have shown that all ancestors $u' \in \mathcal{S}(\mathtt{M}')$ of each state $u \in \mathcal{ESU}(\mathtt{M}', EX, e)$ are contained in $\mathcal{ESU}(\mathtt{M}', EX, e)$. Hence, property P5 hold for set $\mathcal{ESU}(\mathtt{M}', EX, e)$.

As we have shown that properties P1 to P5 hold for X and as $X = \mathcal{ESU}(\mathtt{M}', EX, e)$, it follows that the set of active states after *enabled OnEvent* e is processed in execution state EX, denoted by $\mathcal{ESU}(\mathtt{M}', EX, e)$, is an execution state. □

Appendix B

mCRL2 Specification of Figure 5.3

```
% mCRL2 Specification representing SMMT Specification printer (Namespace: cpp.jordi.examples.printer)

% mCRL2 Representation of the SMMT Specification

sort
  OnEvent = struct ev_print_job | ev_finish_job | ev_finish_color | ev_finish_scaling |
                   ev_resolve_error | ev_reset_error | ev_color_error | ev_submit_job;
  State = struct st_idle | st_error | st_unresolved | st_resolved | st_printing | st_preparing_job |
                 st_color_correction | st_pre_cc | st_post_cc | st_scaling | st_pre_scaling |
                  st_post_scaling | st_printing_job;
  Transition = struct tra(onEvent : OnEvent, target : State);
  LState = struct states(ss_ss : List(State), ss_cs : List(State), ss_ps : List(State));
  Spec = struct sm(
    OnEvents : List(OnEvent),
    States : LState,
    EntryStates : List(State),
    Children : State → List(State),
    Descendants : State → List(State),
    EntryDescendants : State → List(State),
    Transitions: State → List(Transition)
  );

map child_relation : State → List(State);
eqn child_relation(st_idle) = [];
  child_relation(st_error) = [st_unresolved, st_resolved];
  child_relation(st_unresolved) = [];
  child_relation(st_resolved) = [];
  child_relation(st_printing) = [st_preparing_job, st_printing_job];
  child_relation(st_preparing_job) = [st_color_correction, st_scaling];
  child_relation(st_color_correction) = [st_pre_cc, st_post_cc];
  child_relation(st_pre_cc) = [];
  child_relation(st_post_cc) = [];
  child_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
  child_relation(st_pre_scaling) = [];
  child_relation(st_post_scaling) = [];
  child_relation(st_printing_job) = [];

map desc_relation : State → List(State);
eqn desc_relation(st_idle) = [];
  desc_relation(st_error) = [st_unresolved, st_resolved];
  desc_relation(st_unresolved) = [];
  desc_relation(st_resolved) = [];
  desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_post_cc,
                               st_scaling, st_pre_scaling, st_post_scaling, st_printing_job];
  desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_post_cc, st_scaling,
                                     st_pre_scaling, st_post_scaling];
  desc_relation(st_color_correction) = [st_pre_cc, st_post_cc];
  desc_relation(st_pre_cc) = [];
  desc_relation(st_post_cc) = [];
  desc_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
  desc_relation(st_pre_scaling) = [];
  desc_relation(st_post_scaling) = [];
  desc_relation(st_printing_job) = [];

map entry_desc_relation : State → List(State);
eqn entry_desc_relation(st_idle) = [];
  entry_desc_relation(st_error) = [st_unresolved];
  entry_desc_relation(st_unresolved) = [];
  entry_desc_relation(st_resolved) = [];
  entry_desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
                                      st_pre_scaling];
  entry_desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_scaling,
                                          st_pre_scaling];
  entry_desc_relation(st_color_correction) = [st_pre_cc];
  entry_desc_relation(st_pre_cc) = [];
  entry_desc_relation(st_post_cc) = [];
  entry_desc_relation(st_scaling) = [st_pre_scaling];
  entry_desc_relation(st_pre_scaling) = [];
  entry_desc_relation(st_post_scaling) = [];
  entry_desc_relation(st_printing_job) = [];

map transition_relation : State → List(Transition);
eqn transition_relation(st_idle) = [tra(ev_submit_job, st_printing)];
  transition_relation(st_error) = [];
  transition_relation(st_unresolved) = [tra(ev_resolve_error, st_resolved)];
  transition_relation(st_resolved) = [tra(ev_reset_error, st_idle)];
  transition_relation(st_printing) = [];
  transition_relation(st_preparing_job) = [];
  transition_relation(st_color_correction) = [tra(ev_color_error, st_error)];
  transition_relation(st_pre_cc) = [tra(ev_finish_color, st_post_cc),
                                    tra(ev_finish_scaling, st_error)];
  transition_relation(st_post_cc) = [tra(ev_print_job, st_printing_job)];
  transition_relation(st_scaling) = [];
  transition_relation(st_pre_scaling) = [tra(ev_finish_scaling, st_post_scaling)];
  transition_relation(st_post_scaling) = [tra(ev_print_job, st_printing_job)];
  transition_relation(st_printing_job) = [tra(ev_finish_job, st_idle)];

map smmt_spec : Spec;
eqn smmt_spec = sm(
  [ev_print_job, ev_finish_job, ev_finish_color, ev_finish_scaling, ev_resolve_error, ev_reset_error,
      ev_color_error, ev_submit_job],
  states(
    [st_idle, st_unresolved, st_resolved, st_pre_cc, st_post_cc, st_pre_scaling, st_post_scaling,
      st_printing_job],
    [st_error, st_printing, st_color_correction, st_scaling],
    [st_preparing_job]
  ),
  [st_idle, st_unresolved, st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
      st_pre_scaling],
  child_relation,
  desc_relation,
  entry_desc_relation,
  transition_relation
);

map ss_s : LState → List(State);
  ss_c : LState → List(State);
  ss_p : LState → List(State);
var ss, cs, ps : List(State);
eqn ss_s(states(ss, cs, ps)) = ss;
  ss_c(states(ss, cs, ps)) = cs;
  ss_p(states(ss, cs, ps)) = ps;

map sm_o : Spec → List(OnEvent);
  sm_s : Spec → List(State);
  sm_ss : Spec → LState;
  sm_es : Spec → List(State);
  sm_cr : Spec → State → List(State);
  sm_dr : Spec → State → List(State);
  sm_edr : Spec → State → List(State);
  sm_tr : Spec → State → List(Transition);
var lo : List(OnEvent);
  s : LState;
  ls2: List(State);
  cr : State → List(State);
  dr : State → List(State);
  edr : State → List(State);
  tr : State → List(Transition);
eqn sm_o(sm(lo, s, ls2, cr, dr, edr, tr)) = lo;
  sm_s(sm(lo, s, ls2, cr, dr, edr, tr)) = ss_s(s) ++ ss_c(s) ++ ss_p(s);
  sm_ss(sm(lo, s, ls2, cr, dr, edr, tr)) = s;
  sm_es(sm(lo, s, ls2, cr, dr, edr, tr)) = ls2;
  sm_cr(sm(lo, s, ls2, cr, dr, edr, tr)) = cr;
  sm_dr(sm(lo, s, ls2, cr, dr, edr, tr)) = dr;
  sm_edr(sm(lo, s, ls2, cr, dr, edr, tr)) = edr;
  sm_tr(sm(lo, s, ls2, cr, dr, edr, tr)) = tr;

```

```
map tra_o : Transition → OnEvent;
  tra_s : Transition → State;
var e : OnEvent;
  s : State;
eqn tra_o(tra(e, s)) = e;
  tra_s(tra(e, s)) = s;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Validation Checks

map is_well_defined : Spec → List(Nat);
var sp : Spec;
eqn is_well_defined(sp) = val_non_empty_states(sp)
              ++ val_one_entry_root_state(sp)
              ++ val_cs_one_entry_child(sp)
              ++ val_ss_no_children(sp)
              ++ val_transition(sp)
              ++ val_ps_entry_children(sp)
              ++ val_ps_atleast_two_children(sp)
              ++ val_child_rel_1_parent(sp)
              ++ val_child_rel_acyclic(sp);

% Validation Check 1
map val_non_empty_states : Spec → List(Nat);
var sp : Spec;
eqn val_non_empty_states(sp) = if(sm_s(sp) == [], [1], []);

% Validation Check 2
map val_one_entry_root_state : Spec → List(Nat);
var sp : Spec;
eqn val_one_entry_root_state(sp) = if(exists s : State . is_entry_root_state(sp, s) &&
      !(exists s' : State . s != s' && is_entry_root_state(sp, s')), [], [2]);

% Validation Check 3
map val_cs_one_entry_child : Spec → List(Nat);
var sp : Spec;
eqn val_cs_one_entry_child(sp) = if(forall s' : State . s' in ss_c(sm_ss(sp))  => (
    exists s : State . s in sm_s(sp) && s in sm_cr(sp)(s') && s in sm_es(sp)
      && !(exists r : State . r in sm_s(sp) && r in sm_cr(sp)(s') && r in sm_es(sp) && s != r)
  ), [], [3]);

% Validation Check 4
map val_ss_no_children : Spec → List(Nat);
var sp : Spec;
eqn val_ss_no_children(sp) = if(forall s' : State . s' in ss_s(sm_ss(sp)) =>
      !(exists s : State . s in sm_s(sp) && s in sm_cr(sp)(s')), [], [4]);

% Validation Check 5
map val_transition : Spec → List(Nat);
var sp : Spec;
eqn val_transition(sp) = if(forall s : State . s in sm_s(sp) =>
      (forall e : OnEvent . e in sm_o(sp) => (tr_count(sm_tr(sp)(s), e) < 2)), [], [5]);

% Validation Check 6
map val_ps_entry_children : Spec → List(Nat);
var sp : Spec;
eqn val_ps_entry_children(sp) = if(forall s, s' : State . (s' in ss_p(sm_ss(sp)) &&
      s in sm_s(sp) && s in sm_cr(sp)(s')) => s in sm_es(sp), [], [6]);

% Validation Check 7
map val_ps_atleast_two_children : Spec → List(Nat);
var sp : Spec;
eqn val_ps_atleast_two_children(sp) = if(forall s'' : State . s'' in ss_p(sm_ss(sp)) => (
    exists s, s' : State . s in sm_s(sp) && s' in sm_s(sp) && s != s' && s in sm_cr(sp)(s'') &&
      s' in sm_cr(sp)(s'')
  ), [], [7]);

map tr_count : List(Transition) # OnEvent → Nat;
var e, e' : OnEvent;
  s' : State;
  T : List(Transition);
eqn tr_count([], e') = 0;
  (e == e') → tr_count(tra(e, s') |> T, e') = 1 + tr_count(T, e');
  (e != e') → tr_count(tra(e, s') |> T, e') = tr_count(T, e');

% Validation Check 8
map val_child_rel_1_parent : Spec → List(Nat);
```

```
var sp : Spec;
eqn val_child_rel_1_parent(sp) = if(forall x1, x2, x3 : State . (x1 in sm_cr(sp)(x2) &&
      x1 in sm_cr(sp)(x3)) => (x2 == x3), [], [8]);

% Validation Check 9
map val_child_rel_acyclic : Spec → List(Nat);
var sp : Spec;
eqn val_child_rel_acyclic(sp) = if(forall s, s' : State . !(is_desc_of(sp, s, s') &&
      is_desc_of(sp, s', s)), [], [9]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Definitions specifying the semantics in mCRL2


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List Interaction

map subset : List(State) # List(State) → Bool;
var s : State;
  ls, ls' : List(State);
eqn subset([], ls') = true;
  !(s in ls') → subset(s |> ls, ls') = false;
  (s in ls') → subset(s |> ls, ls') = subset(ls, ls');

map get_unique : List(State) → List(State);
var S : List(State);
eqn get_unique(S) = get_unique_helper(S, []);

map get_unique_helper : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn get_unique_helper([], ls2) = [];
  !(s in ls2) → get_unique_helper(s |> ls1, ls2) = [s] ++ get_unique_helper(ls1, ls2 ++ [s]);
  (s in ls2) → get_unique_helper(s |> ls1, ls2) = get_unique_helper(ls1, ls2);

map list_minus_state : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn list_minus_state([], ls2) = [];
  !(s in ls2) → list_minus_state(s |> ls1, ls2) = [s] ++ list_minus_state(ls1, ls2);
  (s in ls2) → list_minus_state(s |> ls1, ls2) = list_minus_state(ls1, ls2);

map sort_states : List(State) # List(State) → List(State);
var s : State;
  ls, ex : List(State);
eqn sort_states(ex, []) = [];
  s in ex → sort_states(ex, s |> ls) = [s] ++ sort_states(ex, ls);
  !(s in ex) → sort_states(ex, s |> ls) = sort_states(ex, ls);

map list_union : List(State) # List(State) → List(State);
var ls, ls' : List(State);
  s : State;
eqn list_union(ls, ls') = get_unique(ls ++ ls');

map list_intersect : List(State) # List(State) → List(State);
var ls, ls' : List(State);
  s : State;
eqn list_intersect([], ls') = [];
  (s in ls') → list_intersect(s |> ls, ls') = [s] ++ list_intersect(ls, ls');
  !(s in ls') → list_intersect(s |> ls, ls') = list_intersect(ls, ls');

map overlap : List(State) # List(State) → Bool;
var s : State;
  ls, ls' : List(State);
eqn overlap([], ls') = false;
  (s in ls') → overlap(s |> ls, ls') = true;
  !(s in ls') → overlap(s |> ls, ls') = overlap(ls, ls');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% IsSimpleState

map is_ss : Spec # State → Bool;
var sp : Spec;
  s : State;
```

```
eqn is_ss(sp, s) = s in ss_s(sm_ss(sp));

% IsCompositeState
map is_cs : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_cs(sp, s) = s in ss_c(sm_ss(sp));

% IsParallelState
map is_ps : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_ps(sp, s) = s in ss_p(sm_ss(sp));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% IsRootState
map is_root_state : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_root_state(sp, s) = exists s' : State . s' in sm_s(sp) &&
      !(exists s'' : State . s in sm_cr(sp)(s''));

% IsEntryRootState
map is_entry_root_state : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_entry_root_state(sp, s) = is_root_state(sp, s) && s in sm_es(sp);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% IsDescendantOf
map is_desc_of : Spec # State # State → Bool;
var sp : Spec;
    s, s' : State;
eqn is_desc_of(sp, s, s') = s in sm_dr(sp)(s');

% IsEntryChildOf
map is_entry_child_of : Spec # State # State → Bool;
var sp : Spec;
  s, s' : State;
eqn is_entry_child_of(sp, s, s') = s in sm_cr(sp)(s') && s in sm_es(sp);

% IsEntryDescendantOf
map is_entry_descendant_of : Spec # State # State → Bool;
var sp : Spec;
  s, s' : State;
eqn is_entry_descendant_of(sp, s, s') = s in sm_edr(sp)(s');

% GetEntryChildren
map get_entry_children : Spec # State → List(State);
var sp : Spec;
  s : State;
eqn get_entry_children(sp, s) = get_entry_children_helper(sm_cr(sp)(s), sm_es(sp));

map get_entry_children_helper : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn get_entry_children_helper([], ls2) = [];
  (s in ls2) →
      get_entry_children_helper(s |> ls1, ls2) = [s] ++ get_entry_children_helper(ls1, ls2);
  !(s in ls2) →
      get_entry_children_helper(s |> ls1, ls2) = get_entry_children_helper(ls1, ls2);

% getAncestors
map get_ancestors : Spec # State → List(State);
var sp : Spec;
  s : State;
eqn get_ancestors(sp, s) = get_ancestors_helper(sp, sm_s(sp), s);

map get_ancestors_helper : Spec # List(State) # State → List(State);
var sp : Spec;
  s, s' : State;
  S : List(State);
eqn get_ancestors_helper(sp, [], s') = [];
  (is_desc_of(sp, s', s)) →
      get_ancestors_helper(sp, s |> S, s') = [s] ++ get_ancestors_helper(sp, S, s');
  !(is_desc_of(sp, s', s)) →
      get_ancestors_helper(sp, s |> S, s') = get_ancestors_helper(sp, S, s');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% InitState
map init_state : Spec → List(State);
var sp : Spec;
eqn init_state(sp) = init_state_helper(sp, sm_s(sp));

% Init State Helper
map init_state_helper : Spec # List(State) → List(State);
var sp : Spec;
  s : State;
  ls : List(State);
eqn init_state_helper(sp, []) = [];
  (is_entry_root_state(sp, s) || (exists s′ : State . is_entry_root_state(sp, s′) &&
      is_entry_descendant_of(sp, s, s′))) →
          init_state_helper(sp, s |> ls) = [s] ++ init_state_helper(sp, ls);
  !(is_entry_root_state(sp, s) || (exists s′ : State . is_entry_root_state(sp, s′) &&
      is_entry_descendant_of(sp, s, s′))) →
          init_state_helper(sp, s |> ls) = init_state_helper(sp, ls);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% HasOutgoingTransitionForEvent
map has_out_tr_for_event : Spec # State # OnEvent → Bool;
var sp : Spec;
  s : State;
  e : OnEvent;
eqn has_out_tr_for_event(sp, s, e) = exists s′ : State . tra(e, s′) in sm_tr(sp)(s);

% isTransitionDefinedForEvent
map is_tr_def_ev : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  s : State;
  ex : List(State);
  e : OnEvent;
eqn is_tr_def_ev(sp, ex, e) = exists s : State . s in ex && has_out_tr_for_event(sp, s, e);
    is_tr_def_ev(sp, [], e) = false;
  has_out_tr_for_event(sp, s, e) → is_tr_def_ev(sp, s |> ex, e) = true;
  !has_out_tr_for_event(sp, s, e) → is_tr_def_ev(sp, s |> ex, e) = is_tr_def_ev(sp, ex, e);

% GetTransition
map get_transition : Spec # State # OnEvent → Transition;
var sp : Spec;
  s : State;
  e : OnEvent;
eqn get_transition(sp, s, e) = get_transition_helper(sm_tr(sp)(s), e);

map get_transition_helper : List(Transition) # OnEvent → Transition;
var T : List(Transition);
  t : Transition;
  e : OnEvent;
eqn tra_o(t) == e → get_transition_helper(t |> T, e) = t;
  tra_o(t) != e → get_transition_helper(t |> T, e) = get_transition_helper(T, e);

% GetPrioritisedTransitionsEvent
map get_prio_tr_event : Spec # List(State) # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  ex : List(State);
  e : OnEvent;
eqn !is_tr_def_ev(sp, ex, e) → get_prio_tr_event(sp, ex, e) = [];
  is_tr_def_ev(sp, ex, e) → get_prio_tr_event(sp, ex, e) = get_prio_tr_event_helper(sp, ex, ex, e);

map get_prio_tr_event_helper : Spec # List(State) # List(State) # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  ex : List(State);
  ss : List(State);
  e : OnEvent;
eqn get_prio_tr_event_helper(sp, ex, [], e) = [];
  (has_out_tr_for_event(sp, s, e) && !(exists s′ : State . s′ in ex && is_desc_of(sp, s′, s) &&
      has_out_tr_for_event(sp, s′, e))) → get_prio_tr_event_helper(sp, ex, s |> ss, e) =
          [get_transition(sp, s, e)] ++ get_prio_tr_event_helper(sp, ex, ss, e);
  !(has_out_tr_for_event(sp, s, e) && !(exists s′ : State . s′ in ex && is_desc_of(sp, s′, s) &&
      has_out_tr_for_event(sp, s′, e))) →
          get_prio_tr_event_helper(sp, ex, s |> ss, e) = get_prio_tr_event_helper(sp, ex, ss, e);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map css : Spec # State # State → Bool;
```

```
var sp : Spec;
  s, s′ : State;
eqn css(sp, s, s′) = cs_rs(sp, sm_s(sp), s, s′) || cs_sr(sp, sm_s(sp), s, s′);

map cs_rs : Spec # List(State) # State # State → Bool;
var sp : Spec;
  s, s′, r : State;
  ls : List(State);
eqn cs_rs(sp, [], s, s′) = true;
  (is_root_state(sp, r) && s in sr(sp, r) && s′ in sr(sp, r)) →
      cs_rs(sp, r |> ls, s, s′) = false;
  !(is_root_state(sp, r) && s in sr(sp, r) && s′ in sr(sp, r)) →
      cs_rs(sp, r |> ls, s, s′) = cs_rs(sp, ls, s, s′);

map cs_sr : Spec # List(State) # State # State → Bool;
var sp : Spec;
  c, s, s′ : State;
  ls : List(State);
eqn cs_sr(sp, [], s, s′) = false;
  (is_cs(sp, c) && is_desc_of(sp, s, c) && is_desc_of(sp, s′, c)) →
      cs_sr(sp, c |> ls, s, s′) =
          cs_sr(sp, ls, s, s′) || cs_sr_h(sp, sm_cr(sp)(c), c, s, s′);
  !(is_cs(sp, c) && is_desc_of(sp, s, c) && is_desc_of(sp, s′, c)) →
      cs_sr(sp, c |> ls, s, s′) = cs_sr(sp, ls, s, s′);

map cs_sr_h : Spec # List(State) # State # State # State → Bool;
var sp : Spec;
  ls : List(State);
  c, c′, s, s′ : State;
eqn cs_sr_h(sp, [], c′, s, s′) = false;
  (s in sr(sp, c)) → cs_sr_h(sp, c |> ls, c′, s, s′) =
      cs_sr_hh(sp, sm_cr(sp)(c′), c, s, s′);
  !(s in sr(sp, c)) → cs_sr_h(sp, c |> ls, c′, s, s′) =
      cs_sr_h(sp, ls, c′, s, s′);

map cs_sr_hh : Spec # List(State) # State # State # State → Bool;
var sp : Spec;
  ls : List(State);
  c, c′, s, s′ : State;
eqn cs_sr_hh(sp, [], c′, s, s′) = false;
  (c != c′ && s′ in sr(sp, c)) →
      cs_sr_hh(sp, c |> ls, c′, s, s′) = true;
  !(c != c′ && s′ in sr(sp, c)) →
      cs_sr_hh(sp, c |> ls, c′, s, s′) = cs_sr_hh(sp, ls, c′, s, s′);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map sr : Spec # State → List(State);
var sp : Spec;
  s  : State;
eqn sr(sp, s) = [s] ++ sm_dr(sp)(s);

map cuo : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  ex : List(State);
  e : OnEvent;
eqn cuo(sp, ex, e) = cuoh(sp, ex, ex, e);

map cuoh : Spec # List(State) # List(State) # OnEvent → Bool;
var sp : Spec;
  ex, ex′ : List(State);
  s : State;
  e : OnEvent;
eqn cuoh(sp, ex, [], e) = false;
  (has_out_tr_for_event(sp, s, e) && cuo_ps_rg(sp, ex, ps_anc(sp, get_ancestors(sp, s)), s, e)) →
      cuoh(sp, ex, s |> ex′, e) = true;
  !(has_out_tr_for_event(sp, s, e) && cuo_ps_rg(sp, ex, ps_anc(sp, get_ancestors(sp, s)), s, e)) →
      cuoh(sp, ex, s |> ex′, e) = cuoh(sp, ex, ex′, e);

map ps_anc : Spec # List(State) → List(State);
var sp : Spec;
  ls : List(State);
  s : State;
eqn ps_anc(sp, []) = [];
  is_ps(sp, s) → ps_anc(sp, s |> ls) = [s] ++ ps_anc(sp, ls);
  !is_ps(sp, s) → ps_anc(sp, s |> ls) = ps_anc(sp, ls);

map cuo_ps_rg : Spec # List(State) # List(State) # State # OnEvent → Bool;
var sp : Spec;
```

```
  ex, ps : List(State);
  s, s′ : State;
  e : OnEvent;
eqn cuo_ps_rg(sp, ex, [], s′, e) = false;
  cuo_ps_one(sp, ex, sm_cr(sp)(s), e) → cuo_ps_rg(sp, ex, s |> ps, s′, e) = true;
  !cuo_ps_one(sp, ex, sm_cr(sp)(s), e) → cuo_ps_rg(sp, ex, s |> ps, s′, e) =
      cuo_ps_rg(sp, ex, ps, s′, e);

map cuo_ps_one : Spec # List(State) # List(State) # OnEvent → Bool;
var sp : Spec;
  ex, cr : List(State);
  s, s′ : State;
  e : OnEvent;
eqn cuo_ps_one(sp, ex, [], e) = false;
  unhandled_subregion(sp, ex, sr(sp, s), e, false) → cuo_ps_one(sp, ex, s |> cr, e) = true;
  !unhandled_subregion(sp, ex, sr(sp, s), e, false) → cuo_ps_one(sp, ex, s |> cr, e) =
      cuo_ps_one(sp, ex, cr, e);

map unhandled_subregion : Spec # List(State) # List(State) # OnEvent # Bool → Bool;
var sp : Spec;
  ex, st : List(State);
  s : State;
  e : OnEvent;
  b : Bool;
eqn unhandled_subregion(sp, ex, [], e, b) = b;
  (has_out_tr_for_event(sp, s, e) && (s in ex)) →
      unhandled_subregion(sp, ex, s |> st, e, b) = false;
  (has_out_tr_for_event(sp, s, e) && !(s in ex)) →
      unhandled_subregion(sp, ex, s |> st, e, b) = unhandled_subregion(sp, ex, st, e, true);
  !has_out_tr_for_event(sp, s, e) →
      unhandled_subregion(sp, ex, s |> st, e, b) = unhandled_subregion(sp, ex, st, e, b);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% getEnteredTargets
map get_et : Spec # List(Transition) → List(State);
var sp : Spec;
  e : OnEvent;
  s′ : State;
  T : List(Transition);
eqn get_et(sp, []) = [];
  get_et(sp, tra(e, s′) |> T) = get_unique([s′] ++ get_ancestors(sp, s′) ++ get_et(sp, T));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map get_as : Spec # List(State) # List(Transition) → List(State);
var sp : Spec;
  s : State;
  ls : List(State);
  lt : List(Transition);
eqn get_as(sp, [], lt) = [];
  as_h(sp, s, lt) → get_as(sp, s |> ls, lt) = [s] ++ get_as(sp, ls, lt);
  !as_h(sp, s, lt) → get_as(sp, s |> ls, lt) = get_as(sp, ls, lt);

map as_h : Spec # State # List(Transition) → Bool;
var sp : Spec;
  s : State;
  t : Transition;
  lt : List(Transition);
eqn as_h(sp, s, []) = false;
  css(sp, s, tra_s(t)) → as_h(sp, s, t |> lt) = true;
  !css(sp, s, tra_s(t)) → as_h(sp, s, t |> lt) = as_h(sp, s, lt);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% initiate
map initiate : Spec # List(State) → List(State);
var sp : Spec;
  ls : List(State);
eqn initiate(sp, ls) = get_unique(initiate_helper(sp, ls, ls));

% is_cs_initiated
map is_cs_initiated : Spec # List(State) # State → Bool;
var sp : Spec;
  s, s′ : State;
  ls : List(State);
eqn is_cs_initiated(sp, [], s′) = false;
  (s in sm_cr(sp)(s′)) → is_cs_initiated(sp, s |> ls, s′) = true;
  !(s in sm_cr(sp)(s′)) → is_cs_initiated(sp, s |> ls, s′) = is_cs_initiated(sp, ls, s′);
```

```
map initiate_helper : Spec # List(State) # List(State) → List(State);
var sp : Spec;
  s : State;
  ls1, ls2 : List(State);
eqn initiate_helper(sp, [], ls2) = [];
  is_ss(sp, s) → initiate_helper(sp, s |> ls1, ls2) = [s] ++ initiate_helper(sp, ls1, ls2);
  is_cs(sp, s) && is_cs_initiated(sp, ls2, s) →
      initiate_helper(sp, s |> ls1, ls2) = [s] ++ initiate_helper(sp, ls1, ls2);
  is_cs(sp, s) && !is_cs_initiated(sp, ls2, s) → initiate_helper(sp, s |> ls1, ls2) =
      [s] ++ get_entry_children(sp, s) ++ initiate_helper(sp, ls1 ++ get_entry_children(sp, s), ls2);
  is_ps(sp, s) → initiate_helper(sp, s |> ls1, ls2) =
      [s] ++ get_entry_children(sp, s) ++ initiate_helper(sp, ls1 ++ get_entry_children(sp, s), ls2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Execution State Update
map esu : Spec # List(State) # List(Transition) → List(State);
var sp : Spec;
  ex : List(State);
  t : List(Transition);
eqn esu(sp, ex, t) =
      initiate(sp, get_unique(list_minus_state(ex, get_as(sp, ex, t)) ++ get_et(sp, t)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map cts : Spec # List(State) → Bool;
var sp : Spec;
  lst : List(State);
eqn cts(sp, lst) = cts_h(sp, lst, lst);

map cts_h : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  lst, lst' : List(State);
  s : State;
eqn cts_h(sp, [], lst') = false;
  cts_hh(sp, s, lst') → cts_h(sp, s |> lst, lst') = true;
  !cts_hh(sp, s, lst') → cts_h(sp, s |> lst, lst') = cts_h(sp, lst, lst');

map cts_hh : Spec # State # List(State) → Bool;
var sp : Spec;
  s, s' : State;
  lst : List(State);
eqn cts_hh(sp, s, []) = false;
  css(sp, s, s') → cts_hh(sp, s, s' |> lst) = true;
  !css(sp, s, s') → cts_hh(sp, s, s' |> lst) = cts_hh(sp, s, lst);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map get_targets : List(Transition) → List(State);
var t : Transition;
  lt : List(Transition);
eqn get_targets([]) = [];
    get_targets(t |> lt) = [tra_s(t)] ++ get_targets(lt);

map enabled : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  ex : List(State);
  e : OnEvent;
eqn enabled(sp, ex, e) = is_tr_def_ev(sp, ex, e) && !cuo(sp, ex, e) &&
      !cts(sp, get_targets(get_prio_tr_event(sp, ex, e)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


act FAIL;
  ev_print_job;
  ev_finish_job;
  ev_finish_color;
  ev_finish_scaling;
  ev_resolve_error;
  ev_reset_error;
  ev_color_error;
  ev_submit_job;
  val_non_empty_states;
  val_one_entry_root_state;
  val_cs_one_entry_child;
  val_ps_entry_children;
  val_ss_no_children;
  val_cs_atleast_one_child;
  val_ps_atleast_two_children;
  val_transition;
  val_child_rel_irrefl;
  val_child_rel_1_parent;
  val_child_rel_acyclic;

proc SM(sp : Spec, ex : List(State), valid : List(Nat)) =
  (valid == []) → (
    (enabled(sp, ex, ev_print_job)
      → ev_print_job.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_print_job)), valid)
      <> ev_print_job.F())
    + (enabled(sp, ex, ev_finish_job)
      → ev_finish_job.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_finish_job)), valid)
      <> ev_finish_job.F())
    + (enabled(sp, ex, ev_finish_color)
      → ev_finish_color.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_finish_color)), valid)
      <> ev_finish_color.F())
    + (enabled(sp, ex, ev_finish_scaling)
      → ev_finish_scaling.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_finish_scaling)), valid)
      <> ev_finish_scaling.F())
    + (enabled(sp, ex, ev_resolve_error)
      → ev_resolve_error.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_resolve_error)), valid)
      <> ev_resolve_error.F())
    + (enabled(sp, ex, ev_reset_error)
      → ev_reset_error.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_reset_error)), valid)
      <> ev_reset_error.F())
    + (enabled(sp, ex, ev_color_error)
      → ev_color_error.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_color_error)), valid)
      <> ev_color_error.F())
    + (enabled(sp, ex, ev_submit_job)
      → ev_submit_job.SM(sp, esu(sp, ex, get_prio_tr_event(sp, ex, ev_submit_job)), valid)
      <> ev_submit_job.F())
  ) <> (
    (head(valid) == 1) → val_non_empty_states.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 2) → val_one_entry_root_state.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 3) → val_cs_one_entry_child.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 4) → val_ss_no_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 5) → val_transition.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 6) → val_ps_entry_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 7) → val_ps_atleast_two_children.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 8) → val_child_rel_1_parent.SM(sp, ex, tail(valid) ++ [0])
    + (head(valid) == 9) → val_child_rel_acyclic.SM(sp, ex, tail(valid) ++ [0])
  );

proc F = FAIL.F();

init SM(smmt_spec, init_state(smmt_spec), is_well_defined(smmt_spec))
```

Appendix C

Complete mCRL2 Specification

```
% mCRL2 Specification representing SMMT Specification printer (Namespace: cpp.jordi.examples.printer)

% mCRL2 Representation of the SMMT Specification

sort
  CustomType = struct a;
  OnEvent = struct ev_print_job | ev_finish_job | ev_finish_color | ev_finish_scaling |
                    ev_resolve_error | ev_reset_error | ev_color_error | ev_submit_job;
  DoEvent = struct do_event;
  State = struct st_idle | st_error | st_unresolved | st_resolved | st_printing | st_preparing_job |
                  st_color_correction | st_pre_cc | st_post_cc | st_scaling | st_pre_scaling |
                  st_post_scaling | st_printing_job | st_INTERNAL;
  Pair = struct p(p_ex : List(State), p_d : List(DoEvent));
  OEH = struct oeh(doEvents : List(DoEvent), target : State);
  Transition = struct tra(onEvent : OnEvent, doEvents : List(DoEvent), target : State);
  LState = struct states(ss_ss : List(State), ss_cs : List(State), ss_ps : List(State),
              ss_js : List(State));
  Spec = struct sm(
    OnEvents : List(OnEvent),
    DoEvents : List(DoEvent),
    States : LState,
    EntryStates : List(State),
    Children : State → List(State),
    Descendants : State → List(State),
    EntryDescendants : State → List(State),
    Transitions: State → List(Transition),
    Joins: State → List(State),
    CondEntryHandlers : State → List(Transition),
    OtherEntryHandlers : State → List(OEH),
    ExitHandlers : State → List(DoEvent));

map child_relation : State → List(State);
eqn child_relation(st_idle) = [];
  child_relation(st_error) = [st_unresolved, st_resolved];
  child_relation(st_unresolved) = [];
  child_relation(st_resolved) = [];
  child_relation(st_printing) = [st_preparing_job, st_printing_job];
  child_relation(st_preparing_job) = [st_color_correction, st_scaling];
  child_relation(st_color_correction) = [st_pre_cc, st_post_cc];
  child_relation(st_pre_cc) = [];
  child_relation(st_post_cc) = [];
  child_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
  child_relation(st_pre_scaling) = [];
  child_relation(st_post_scaling) = [];
  child_relation(st_printing_job) = [];
  child_relation(st_INTERNAL) = [];

map desc_relation : State → List(State);
eqn desc_relation(st_idle) = [];
  desc_relation(st_error) = [st_unresolved, st_resolved];
  desc_relation(st_unresolved) = [];
  desc_relation(st_resolved) = [];
  desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_post_cc,
                          st_scaling, st_pre_scaling, st_post_scaling, st_printing_job];
  desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_post_cc, st_scaling,
                            st_pre_scaling, st_post_scaling];
  desc_relation(st_color_correction) = [st_pre_cc, st_post_cc];
  desc_relation(st_pre_cc) = [];
  desc_relation(st_post_cc) = [];
  desc_relation(st_scaling) = [st_pre_scaling, st_post_scaling];
  desc_relation(st_pre_scaling) = [];
  desc_relation(st_post_scaling) = [];
  desc_relation(st_printing_job) = [];
  desc_relation(st_INTERNAL) = [];

map entry_desc_relation : State → List(State);
eqn entry_desc_relation(st_idle) = [];
  entry_desc_relation(st_error) = [st_unresolved];
  entry_desc_relation(st_unresolved) = [];
  entry_desc_relation(st_resolved) = [];
  entry_desc_relation(st_printing) = [st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
                                      st_pre_scaling];
  entry_desc_relation(st_preparing_job) = [st_color_correction, st_pre_cc, st_scaling,
                                           st_pre_scaling];
  entry_desc_relation(st_color_correction) = [st_pre_cc];
  entry_desc_relation(st_pre_cc) = [];
  entry_desc_relation(st_post_cc) = [];
  entry_desc_relation(st_scaling) = [st_pre_scaling];
  entry_desc_relation(st_pre_scaling) = [];
  entry_desc_relation(st_post_scaling) = [];
  entry_desc_relation(st_printing_job) = [];
  entry_desc_relation(st_INTERNAL) = [];

map transition_relation : State → List(Transition);
eqn transition_relation(st_idle) = [tra(ev_submit_job, [], st_printing)];
  transition_relation(st_error) = [tra(ev_reset_error, [], st_idle)];
  transition_relation(st_unresolved) = [tra(ev_resolve_error, [], st_resolved)];
  transition_relation(st_resolved) = [];
  transition_relation(st_printing) = [];
  transition_relation(st_preparing_job) = [];
  transition_relation(st_color_correction) = [tra(ev_color_error, [], st_error)];
  transition_relation(st_pre_cc) = [tra(ev_finish_color, [], st_post_cc),
                                    tra(ev_finish_scaling, [], st_error)];
  transition_relation(st_post_cc) = [tra(ev_print_job, [], st_printing_job)];
  transition_relation(st_scaling) = [];
  transition_relation(st_pre_scaling) = [tra(ev_finish_scaling, [], st_post_scaling)];
  transition_relation(st_post_scaling) = [tra(ev_print_job, [], st_printing_job)];
  transition_relation(st_printing_job) = [tra(ev_finish_job, [], st_idle)];
  transition_relation(st_INTERNAL) = [];

map join_relation : State → List(State);
eqn join_relation(st_idle) = [];
  join_relation(st_error) = [];
  join_relation(st_unresolved) = [];
  join_relation(st_resolved) = [];
  join_relation(st_printing) = [];
  join_relation(st_preparing_job) = [];
  join_relation(st_color_correction) = [];
  join_relation(st_pre_cc) = [];
  join_relation(st_post_cc) = [];
  join_relation(st_scaling) = [];
  join_relation(st_pre_scaling) = [];
  join_relation(st_post_scaling) = [];
  join_relation(st_printing_job) = [];
  join_relation(st_INTERNAL) = [];

map cond_entry_handler : State → List(Transition);
eqn cond_entry_handler(st_idle) = [];
  cond_entry_handler(st_error) = [];
  cond_entry_handler(st_unresolved) = [];
  cond_entry_handler(st_resolved) = [];
  cond_entry_handler(st_printing) = [];
  cond_entry_handler(st_preparing_job) = [];
  cond_entry_handler(st_color_correction) = [];
  cond_entry_handler(st_pre_cc) = [];
  cond_entry_handler(st_post_cc) = [];
  cond_entry_handler(st_scaling) = [];
  cond_entry_handler(st_pre_scaling) = [];
  cond_entry_handler(st_post_scaling) = [];
  cond_entry_handler(st_printing_job) = [];
  cond_entry_handler(st_INTERNAL) = [];

map other_entry_handler : State → List(OEH);
eqn other_entry_handler(st_idle) = [];
  other_entry_handler(st_error) = [];
  other_entry_handler(st_unresolved) = [];
  other_entry_handler(st_resolved) = [];
  other_entry_handler(st_printing) = [];
  other_entry_handler(st_preparing_job) = [];
  other_entry_handler(st_color_correction) = [];
  other_entry_handler(st_pre_cc) = [];
  other_entry_handler(st_post_cc) = [];
  other_entry_handler(st_scaling) = [];
  other_entry_handler(st_pre_scaling) = [];
  other_entry_handler(st_post_scaling) = [];
  other_entry_handler(st_printing_job) = [];
  other_entry_handler(st_INTERNAL) = [];

map exit_handler : State → List(DoEvent);
eqn exit_handler(st_idle) = [];
  exit_handler(st_error) = [];
  exit_handler(st_unresolved) = [];
  exit_handler(st_resolved) = [];
  exit_handler(st_printing) = [];
  exit_handler(st_preparing_job) = [];
  exit_handler(st_color_correction) = [];
  exit_handler(st_pre_cc) = [];
  exit_handler(st_post_cc) = [];
  exit_handler(st_scaling) = [];
  exit_handler(st_pre_scaling) = [];
  exit_handler(st_post_scaling) = [];
  exit_handler(st_printing_job) = [];
  exit_handler(st_INTERNAL) = [];

map smmt_spec : Spec;
eqn smmt_spec = sm(
  [ev_print_job, ev_finish_job, ev_finish_color, ev_finish_scaling, ev_resolve_error, ev_reset_error,
      ev_color_error, ev_submit_job],
  [do_event],
  states(
    [st_idle, st_unresolved, st_resolved, st_pre_cc, st_post_cc, st_pre_scaling, st_post_scaling,
      st_printing_job],
    [st_error, st_printing, st_color_correction, st_scaling],
    [st_preparing_job],
    []
  ),
  [st_idle, st_unresolved, st_preparing_job, st_color_correction, st_pre_cc, st_scaling,
      st_pre_scaling],
  child_relation,
  desc_relation,
  entry_desc_relation,
  transition_relation,
  join_relation,
  cond_entry_handler,
  other_entry_handler,
  exit_handler
);

map ss_s : LState → List(State);
  ss_c : LState → List(State);
  ss_p : LState → List(State);
  ss_j : LState → List(State);
var ss, cs, ps, js : List(State);
eqn ss_s(states(ss, cs, ps, js)) = ss;
  ss_c(states(ss, cs, ps, js)) = cs;
  ss_p(states(ss, cs, ps, js)) = ps;
  ss_j(states(ss, cs, ps, js)) = js;

map sm_o : Spec → List(OnEvent);
  sm_d : Spec → List(DoEvent);
  sm_s : Spec → List(State);
  sm_ss : Spec → LState;
  sm_es : Spec → List(State);
  sm_cr : Spec → State → List(State);
  sm_dr : Spec → State → List(State);
  sm_edr : Spec → State → List(State);
  sm_tr : Spec → State → List(Transition);
  sm_jr : Spec → State → List(State);
  sm_ce : Spec → State → List(Transition);
  sm_oe : Spec → State → List(OEH);
  sm_ex : Spec → State → List(DoEvent);
var lo : List(OnEvent);
  ld : List(DoEvent);
  s : LState;
  ls2: List(State);
  cr : State → List(State);
  dr : State → List(State);
  edr : State → List(State);
  tr : State → List(Transition);
  jr : State → List(State);
  c : State → List(Transition);
  o : State → List(OEH);
  ex : State → List(DoEvent);
eqn sm_o(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = lo;
    sm_d(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = ld;
  sm_s(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = ss_s(s) ++ ss_c(s) ++ ss_p(s) ++ ss_j(s);
  sm_ss(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = s;
  sm_es(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = ls2;
  sm_cr(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = cr;
  sm_dr(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = dr;
  sm_edr(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = edr;
  sm_tr(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = tr;
  sm_jr(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = jr;
  sm_ce(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = c;
  sm_oe(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = o;
  sm_ex(sm(lo, ld, s, ls2, cr, dr, edr, tr, jr, c, o, ex)) = ex;

map tra_o : Transition → OnEvent;
  tra_d : Transition → List(DoEvent);
  tra_s : Transition → State;
var e : OnEvent;
  s : State;
  ld : List(DoEvent);
eqn tra_o(tra(e, ld, s)) = e;
  tra_d(tra(e, ld, s)) = ld;
  tra_s(tra(e, ld, s)) = s;

map oeh_d : OEH → List(DoEvent);
  oeh_s : OEH → State;
var ld : List(DoEvent);
  s' : State;
eqn oeh_d(oeh(ld, s')) = ld;
  oeh_s(oeh(ld, s')) = s';

map pair_ex : Pair → List(State);
  pair_de : Pair → List(DoEvent);
var ls : List(State);
  ld : List(DoEvent);
eqn pair_ex(p(ls, ld)) = ls;
    pair_de(p(ls, ld)) = ld;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Validation Checks

map is_well_defined : Spec → List(Nat);
var sp : Spec;
eqn is_well_defined(sp) = val_non_empty_states(sp)
             ++ val_one_entry_root_state(sp)
             ++ val_cs_one_entry_child(sp)
             ++ val_ps_entry_children(sp)
             ++ val_js_not_entry_state(sp)
             ++ val_ss_no_children(sp)
             ++ val_ps_atleast_two_children(sp)
             ++ val_js_no_children(sp)
             ++ val_js_child_of_ps(sp)
             ++ val_transition(sp)
             ++ val_only_joins_js(sp)
             ++ val_join_relation(sp)
             ++ val_cond_entry_handler(sp)
             ++ val_other_entry_handler(sp)
             ++ val_child_rel_1_parent(sp)
             ++ val_child_rel_acyclic(sp);

% Validation Check 1
map val_non_empty_states : Spec → List(Nat);
var sp : Spec;
eqn val_non_empty_states(sp) = if(sm_s(sp) == [], [1], []);

% Validation Check 2
map val_one_entry_root_state : Spec → List(Nat);
var sp : Spec;
eqn val_one_entry_root_state(sp) = if(exists s : State . is_entry_root_state(sp, s) &&
      !(exists s' : State . s != s' && is_entry_root_state(sp, s')), [], [2]);
```

```
% Validation Check 3
map val_cs_one_entry_child : Spec → List(Nat);
var sp : Spec;
eqn val_cs_one_entry_child(sp) = if(forall s' : State . s' in ss_c(sm_ss(sp))  => (
    exists s : State . s in sm_s(sp) && s in sm_cr(sp)(s') && s in sm_es(sp)
      && !(exists r : State . r in sm_s(sp) && r in sm_cr(sp)(s') && r in sm_es(sp) && s != r)
  ), [], [3]);

% Validation Check 4
map val_ps_entry_children : Spec → List(Nat);
var sp : Spec;
eqn val_ps_entry_children(sp) = if(forall s, s' : State . (s' in ss_p(sm_ss(sp)) && s in sm_s(sp) &&
      !(s in ss_j(sm_ss(sp))) && s in sm_cr(sp)(s')) => s in sm_es(sp), [], [4]);

% Validation Check 5
map val_js_not_entry_state : Spec → List(Nat);
var sp : Spec;
eqn val_js_not_entry_state(sp) = if(list_intersect(ss_j(sm_ss(sp)), sm_es(sp)) == [], [], [5]);

% Validation Check 6
map val_ss_no_children : Spec → List(Nat);
var sp : Spec;
eqn val_ss_no_children(sp) = if(forall s' : State . s' in ss_s(sm_ss(sp)) =>
      !(exists s : State . s in sm_s(sp) && s in sm_cr(sp)(s')), [], [6]);

% Validation Check 7
map val_ps_atleast_two_children : Spec → List(Nat);
var sp : Spec;
eqn val_ps_atleast_two_children(sp) = if(forall s'' : State . s'' in ss_p(sm_ss(sp)) => (
    exists s, s' : State . s in sm_s(sp) && s' in sm_s(sp) && s != s' && s in sm_cr(sp)(s'') &&
      s' in sm_cr(sp)(s'')
  ), [], [7]);

% Validation Check 8
map val_js_no_children : Spec → List(Nat);
var sp : Spec;
eqn val_js_no_children(sp) = if(forall s' : State . s' in ss_j(sm_ss(sp)) =>
      !(exists s : State . s in sm_s(sp) && s in sm_cr(sp)(s')), [], [8]);

% Validation Check 9
map val_js_child_of_ps : Spec → List(Nat);
var sp : Spec;
eqn val_js_child_of_ps(sp) = if(forall s : State . s in ss_j(sm_ss(sp)) =>
      (exists s' : State . s' in ss_p(sm_ss(sp)) && s in sm_cr(sp)(s')), [], [9]);

% Validation Check 10
map val_transition : Spec → List(Nat);
var sp : Spec;
eqn val_transition(sp) = if(forall s : State . s in sm_s(sp) =>
      (forall e : OnEvent . e in sm_o(sp) => (tr_count(sm_tr(sp)(s), e) < 2)), [], [10]);

map tr_count : List(Transition) # OnEvent → Nat;
var e, e' : OnEvent;
  D : List(DoEvent);
  s' : State;
  T : List(Transition);
eqn tr_count([], e') = 0;
  (e == e') → tr_count(tra(e, D, s') |> T, e') = 1 + tr_count(T, e');
  (e != e') → tr_count(tra(e, D, s') |> T, e') = tr_count(T, e');

% Validation Check 11
map val_only_joins_js : Spec → List(Nat);
var sp : Spec;
eqn val_only_joins_js(sp) = if(forall s : State . s in sm_s(sp) =>
      ((s in ss_j(sm_ss(sp))) == (# sm_jr(sp)(s) > 0)), [], [11]);

% Validation Check 12
map val_join_relation : Spec → List(Nat);
var sp : Spec;
eqn val_join_relation(sp) = if(forall s, s' : State . (s in ss_j(sm_ss(sp)) &&
      s' in ss_p(sm_ss(sp)) && s in sm_cr(sp)(s')) => (
          forall s'' : State . s'' in sm_jr(sp)(s) => s'' in sm_dr(sp)(s')
  ), [], [12]);

% Validation Check 13
map val_cond_entry_handler : Spec → List(Nat);
var sp : Spec;
eqn val_cond_entry_handler(sp) = if(forall s : State . s in sm_s(sp) =>
      (forall c : Transition . c in sm_ce(sp)(s) => (tra_d(c) == [] => s != tra_s(c))), [], [13]);
```

```
% Validation Check 14
map val_other_entry_handler : Spec → List(Nat);
var sp : Spec;
eqn val_other_entry_handler(sp) = if(forall s : State . s in sm_s(sp) =>
      (forall o : OEH . o in sm_oe(sp)(s) => (oeh_d(o) == [] => s != oeh_s(o))), [], [14]);

% Validation Check 15
map val_child_rel_1_parent : Spec → List(Nat);
var sp : Spec;
eqn val_child_rel_1_parent(sp) = if(forall x1, x2, x3 : State .
      (x1 in sm_cr(sp)(x2) && x1 in sm_cr(sp)(x3)) => (x2 == x3), [], [15]);

% Validation Check 16
map val_child_rel_acyclic : Spec → List(Nat);
var sp : Spec;
eqn val_child_rel_acyclic(sp) = if(forall s, s' : State .
      !(is_desc_of(sp, s, s') && is_desc_of(sp, s', s)), [], [16]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Definitions specifying the semantics in mCRL2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% List Interaction

map subset : List(State) # List(State) → Bool;
var s : State;
  ls, ls' : List(State);
eqn subset([], ls') = true;
  !(s in ls') → subset(s |> ls, ls') = false;
  (s in ls') → subset(s |> ls, ls') = subset(ls, ls');

map get_unique : List(State) → List(State);
var S : List(State);
eqn get_unique(S) = get_unique_helper(S, []);

map get_unique_helper : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn get_unique_helper([], ls2) = [];
  !(s in ls2) → get_unique_helper(s |> ls1, ls2) = [s] ++ get_unique_helper(ls1, ls2 ++ [s]);
  (s in ls2) → get_unique_helper(s |> ls1, ls2) = get_unique_helper(ls1, ls2);

map list_minus_state : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn list_minus_state([], ls2) = [];
  !(s in ls2) → list_minus_state(s |> ls1, ls2) = [s] ++ list_minus_state(ls1, ls2);
  (s in ls2) → list_minus_state(s |> ls1, ls2) = list_minus_state(ls1, ls2);

map list_minus_doevent : List(DoEvent) # DoEvent → List(DoEvent);
var d, d' : DoEvent;
  ld : List(DoEvent);
eqn list_minus_doevent([], d') = [];
  d != d' → list_minus_doevent(d |> ld, d') = [d] ++ list_minus_doevent(ld, d');
  d == d' → list_minus_doevent(d |> ld, d') = list_minus_doevent(ld, d');

map ft : List(Transition) → List(Transition);
var t : Transition;
  lt : List(Transition);
eqn ft([]) = [];
  (tra_s(t) != st_INTERNAL) → ft(t |> lt) = [t] ++ ft(lt);
  (tra_s(t) == st_INTERNAL) → ft(t |> lt) = ft(lt);

map sort_states : List(State) # List(State) → List(State);
var s : State;
  ls, ex : List(State);
eqn sort_states(ex, []) = [];
  s in ex → sort_states(ex, s |> ls) = [s] ++ sort_states(ex, ls);
  !(s in ex) → sort_states(ex, s |> ls) = sort_states(ex, ls);

map list_union : List(State) # List(State) → List(State);
var ls, ls' : List(State);
  s : State;
eqn list_union(ls, ls') = get_unique(ls ++ ls');

map list_intersect : List(State) # List(State) → List(State);
```

```
var ls, ls' : List(State);
  s : State;
eqn list_intersect([], ls') = [];
  (s in ls') → list_intersect(s |> ls, ls') = [s] ++ list_intersect(ls, ls');
  !(s in ls') → list_intersect(s |> ls, ls') = list_intersect(ls, ls');

map overlap : List(State) # List(State) → Bool;
var s : State;
  ls, ls' : List(State);
eqn overlap([], ls') = false;
  (s in ls') → overlap(s |> ls, ls') = true;
  !(s in ls') → overlap(s |> ls, ls') = overlap(ls, ls');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% IsSimpleState

map is_ss : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_ss(sp, s) = s in ss_s(sm_ss(sp));

%% IsCompositeState
map is_cs : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_cs(sp, s) = s in ss_c(sm_ss(sp));

%% IsParallelState
map is_ps : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_ps(sp, s) = s in ss_p(sm_ss(sp));

%% IsJointState
map is_js : Spec # State → Bool;
var sp : Spec;
    s : State;
eqn is_js(sp, s) = s in ss_j(sm_ss(sp));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% IsRootState
map is_root_state : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_root_state(sp, s) = exists s' : State . s' in sm_s(sp) &&
      !(exists s'' : State . s in sm_cr(sp)(s''));

%% IsEntryRootState
map is_entry_root_state : Spec # State → Bool;
var sp : Spec;
  s : State;
eqn is_entry_root_state(sp, s) = is_root_state(sp, s) && s in sm_es(sp);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% IsDescendantOf
map is_desc_of : Spec # State # State → Bool;
var sp : Spec;
    s, s' : State;
eqn is_desc_of(sp, s, s') = s in sm_dr(sp)(s');

%% IsEntryChildOf
map is_entry_child_of : Spec # State # State → Bool;
var sp : Spec;
  s, s' : State;
eqn is_entry_child_of(sp, s, s') = s in sm_cr(sp)(s') && s in sm_es(sp);

%% IsEntryDescendantOf
map is_entry_descendant_of : Spec # State # State → Bool;
var sp : Spec;
  s, s' : State;
eqn is_entry_descendant_of(sp, s, s') = s in sm_edr(sp)(s');

%% GetEntryChildren
map get_entry_children : Spec # State → List(State);
var sp : Spec;
  s : State;
```

```
eqn get_entry_children(sp, s) = get_entry_children_helper(sm_cr(sp)(s), sm_es(sp));

map get_entry_children_helper : List(State) # List(State) → List(State);
var s : State;
  ls1, ls2 : List(State);
eqn get_entry_children_helper([], ls2) = [];
  (s in ls2) →
      get_entry_children_helper(s |> ls1, ls2) = [s] ++ get_entry_children_helper(ls1, ls2);
  !(s in ls2) →
      get_entry_children_helper(s |> ls1, ls2) = get_entry_children_helper(ls1, ls2);

%% getAncestors
map get_ancestors : Spec # State → List(State);
var sp : Spec;
  s : State;
eqn get_ancestors(sp, s) = get_ancestors_helper(sp, sm_s(sp), s);

map get_ancestors_helper : Spec # List(State) # State → List(State);
var sp : Spec;
  s, s' : State;
  S : List(State);
eqn get_ancestors_helper(sp, [], s') = [];
  (is_desc_of(sp, s', s)) →
      get_ancestors_helper(sp, s |> S, s') = [s] ++ get_ancestors_helper(sp, S, s');
  !(is_desc_of(sp, s', s)) →
      get_ancestors_helper(sp, s |> S, s') = get_ancestors_helper(sp, S, s');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% InitState
map init_state : Spec → Pair;
var sp : Spec;
eqn init_state(sp) = get_init_state_pair(sp, jsu(sp, init_state_helper(sp, sm_s(sp))));

map get_init_state_pair : Spec # List(State) → Pair;
var sp : Spec;
  ls : List(State);
eqn get_init_state_pair(sp, ls) = p(ls, get_do_events_entry_state_init(sp, ls));

%% Init State Helper
map init_state_helper : Spec # List(State) → List(State);
var sp : Spec;
  s : State;
  ls : List(State);
eqn init_state_helper(sp, []) = [];
  (is_entry_root_state(sp, s) || (exists s' : State . is_entry_root_state(sp, s') &&
      is_entry_descendant_of(sp, s, s'))) →
          init_state_helper(sp, s |> ls) = [s] ++ init_state_helper(sp, ls);
  !(is_entry_root_state(sp, s) || (exists s' : State . is_entry_root_state(sp, s') &&
      is_entry_descendant_of(sp, s, s'))) →
          init_state_helper(sp, s |> ls) = init_state_helper(sp, ls);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% HasOutgoingTransitionForEvent
map has_out_tr_for_event : Spec # State # OnEvent → Bool;
var sp : Spec;
  s : State;
  e : OnEvent;
eqn has_out_tr_for_event(sp, s, e) =
      exists s' : State . exists ld : List(DoEvent) . tra(e, ld, s') in sm_tr(sp)(s);

%% isTransitionDefinedForEvent
map is_tr_def_ev : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  s : State;
  ex : List(State);
  e : OnEvent;
eqn is_tr_def_ev(sp, ex, e) = exists s : State . s in ex && has_out_tr_for_event(sp, s, e);
    is_tr_def_ev(sp, [], e) = false;
  has_out_tr_for_event(sp, s, e) → is_tr_def_ev(sp, s |> ex, e) = true;
  !has_out_tr_for_event(sp, s, e) → is_tr_def_ev(sp, s |> ex, e) = is_tr_def_ev(sp, ex, e);

%% GetTransition
map get_transition : Spec # State # OnEvent → Transition;
var sp : Spec;
  s : State;
  e : OnEvent;
eqn get_transition(sp, s, e) = get_transition_helper(sm_tr(sp)(s), e);
```

```
map get_transition_helper : List(Transition) # OnEvent → Transition;
var T : List(Transition);
  t : Transition;
  e : OnEvent;
eqn tra_o(t) == e → get_transition_helper(t |> T, e) = t;
  tra_o(t) != e → get_transition_helper(t |> T, e) = get_transition_helper(T, e);

%% GetPrioritisedTransitionsEvent
map get_prio_tr_event : Spec # List(State) # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  ex : List(State);
  e : OnEvent;
eqn !is_tr_def_ev(sp, ex, e) → get_prio_tr_event(sp, ex, e) = [];
  is_tr_def_ev(sp, ex, e) → get_prio_tr_event(sp, ex, e) = get_prio_tr_event_helper(sp, ex, ex, e);

map get_prio_tr_event_helper : Spec # List(State) # List(State) # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  ex : List(State);
  ss : List(State);
  e : OnEvent;
eqn get_prio_tr_event_helper(sp, ex, [], e) = [];
  (has_out_tr_for_event(sp, s, e) && !(exists s' : State . s' in ex && is_desc_of(sp, s', s)
      && has_out_tr_for_event(sp, s', e))) → get_prio_tr_event_helper(sp, ex, s |> ss, e) =
          [get_transition(sp, s, e)] ++ get_prio_tr_event_helper(sp, ex, ss, e);
  !(has_out_tr_for_event(sp, s, e) && !(exists s' : State . s' in ex && is_desc_of(sp, s', s)
      && has_out_tr_for_event(sp, s', e))) → get_prio_tr_event_helper(sp, ex, s |> ss, e) =
          get_prio_tr_event_helper(sp, ex, ss, e);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map css : Spec # State # State → Bool;
var sp : Spec;
  s, s' : State;
eqn css(sp, s, s') = cs_rs(sp, sm_s(sp), s, s') || cs_sr(sp, sm_s(sp), s, s');

map cs_rs : Spec # List(State) # State # State → Bool;
var sp : Spec;
  s, s', r : State;
  ls : List(State);
eqn cs_rs(sp, [], s, s') = true;
  (is_root_state(sp, r) && s in sr(sp, r) && s' in sr(sp, r)) →
      cs_rs(sp, r |> ls, s, s') = false;
  !(is_root_state(sp, r) && s in sr(sp, r) && s' in sr(sp, r)) →
      cs_rs(sp, r |> ls, s, s') = cs_rs(sp, ls, s, s');

map cs_sr : Spec # List(State) # State # State → Bool;
var sp : Spec;
  c, s, s' : State;
  ls : List(State);
eqn cs_sr(sp, [], s, s') = false;
  (is_cs(sp, c) && is_desc_of(sp, s, c) && is_desc_of(sp, s', c)) →
      cs_sr(sp, c |> ls, s, s') = cs_sr(sp, ls, s, s') || cs_sr_h(sp, sm_cr(sp)(c), c, s, s');
  !(is_cs(sp, c) && is_desc_of(sp, s, c) && is_desc_of(sp, s', c)) →
      cs_sr(sp, c |> ls, s, s') = cs_sr(sp, ls, s, s');

map cs_sr_h : Spec # List(State) # State # State # State → Bool;
var sp : Spec;
  ls : List(State);
  c, c', s, s' : State;
eqn cs_sr_h(sp, [], c', s, s') = false;
  (s in sr(sp, c)) → cs_sr_h(sp, c |> ls, c', s, s') = cs_sr_hh(sp, sm_cr(sp)(c'), c, s, s');
  !(s in sr(sp, c)) → cs_sr_h(sp, c |> ls, c', s, s') = cs_sr_h(sp, ls, c', s, s');

map cs_sr_hh : Spec # List(State) # State # State # State → Bool;
var sp : Spec;
  ls : List(State);
  c, c', s, s' : State;
eqn cs_sr_hh(sp, [], c', s, s') = false;
  (c != c' && s' in sr(sp, c)) → cs_sr_hh(sp, c |> ls, c', s, s') = true;
  !(c != c' && s' in sr(sp, c)) → cs_sr_hh(sp, c |> ls, c', s, s') = cs_sr_hh(sp, ls, c', s, s');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map sr : Spec # State → List(State);
var sp : Spec;
  s  : State;
eqn sr(sp, s) = [s] ++ sm_dr(sp)(s);

```

```
map cuo : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  ex : List(State);
  e : OnEvent;
eqn cuo(sp, ex, e) = cuoh(sp, ex, ex, e);

map cuoh : Spec # List(State) # List(State) # OnEvent → Bool;
var sp : Spec;
  ex, ex' : List(State);
  s : State;
  e : OnEvent;
eqn cuoh(sp, ex, [], e) = false;
  (has_out_tr_for_event(sp, s, e) && cuo_ps_rg(sp, ex, ps_anc(sp, get_ancestors(sp, s)), s, e)) →
      cuoh(sp, ex, s |> ex', e) = true;
  !(has_out_tr_for_event(sp, s, e) && cuo_ps_rg(sp, ex, ps_anc(sp, get_ancestors(sp, s)), s, e)) →
      cuoh(sp, ex, s |> ex', e) = cuoh(sp, ex, ex', e);

map ps_anc : Spec # List(State) → List(State);
var sp : Spec;
  ls : List(State);
  s : State;
eqn ps_anc(sp, []) = [];
  is_ps(sp, s) → ps_anc(sp, s |> ls) = [s] ++ ps_anc(sp, ls);
  !is_ps(sp, s) → ps_anc(sp, s |> ls) = ps_anc(sp, ls);

map cuo_ps_rg : Spec # List(State) # List(State) # State # OnEvent → Bool;
var sp : Spec;
  ex, ps : List(State);
  s, s' : State;
  e : OnEvent;
eqn cuo_ps_rg(sp, ex, [], s', e) = false;
  cuo_ps_one(sp, ex, sm_cr(sp)(s), e) →
      cuo_ps_rg(sp, ex, s |> ps, s', e) = true;
  !cuo_ps_one(sp, ex, sm_cr(sp)(s), e) →
      cuo_ps_rg(sp, ex, s |> ps, s', e) = cuo_ps_rg(sp, ex, ps, s', e);

map cuo_ps_one : Spec # List(State) # List(State) # OnEvent → Bool;
var sp : Spec;
  ex, cr : List(State);
  s, s' : State;
  e : OnEvent;
eqn cuo_ps_one(sp, ex, [], e) = false;
  !is_js(sp, s) && unhandled_subregion(sp, ex, sr(sp, s), e, false) →
      cuo_ps_one(sp, ex, s |> cr, e) = true;
  is_js(sp, s) || !unhandled_subregion(sp, ex, sr(sp, s), e, false) →
      cuo_ps_one(sp, ex, s |> cr, e) = cuo_ps_one(sp, ex, cr, e);

map unhandled_subregion : Spec # List(State) # List(State) # OnEvent # Bool → Bool;
var sp : Spec;
  ex, st : List(State);
  s : State;
  e : OnEvent;
  b : Bool;
eqn unhandled_subregion(sp, ex, [], e, b) = b;
  (has_out_tr_for_event(sp, s, e) && (s in ex)) →
      unhandled_subregion(sp, ex, s |> st, e, b) = false;
  (has_out_tr_for_event(sp, s, e) && !(s in ex)) →
      unhandled_subregion(sp, ex, s |> st, e, b) = unhandled_subregion(sp, ex, st, e, true);
  !has_out_tr_for_event(sp, s, e) →
      unhandled_subregion(sp, ex, s |> st, e, b) = unhandled_subregion(sp, ex, st, e, b);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map is_exec_state : Spec # List(State) → Bool;
var sp : Spec;
  ls : List(State);
eqn is_exec_state(sp, ls) = ies_one_root_state(sp, ls, false) && ies_cs(sp, ls, ls) &&
      ies_ps(sp, ls, ls) && ies_js(sp, ls, ls) && ies_parent(sp, ls, ls);

map ies_one_root_state : Spec # List(State) # Bool → Bool;
var sp : Spec;
  s : State;
  ls : List(State);
  b : Bool;
eqn ies_one_root_state(sp, [], b) = b;
  is_root_state(sp, s) && !b →
      ies_one_root_state(sp, s |> ls, b) = ies_one_root_state(sp, ls, true);
  is_root_state(sp, s) && b →
      ies_one_root_state(sp, s |> ls, b) = false;
  !is_root_state(sp, s) →
```

```
    ies_one_root_state(sp, s |> ls, b) = ies_one_root_state(sp, ls, b);

map ies_cs : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  s : State;
  ls, ex : List(State);
eqn ies_cs(sp, [], ex) = true;
  (!is_cs(sp, s) || (# list_intersect(sm_cr(sp)(s), ex) == 1)) →
      ies_cs(sp, s |> ls, ex) = ies_cs(sp, ls, ex);
  (is_cs(sp, s) && (# list_intersect(sm_cr(sp)(s), ex) != 1)) →
      ies_cs(sp, s |> ls, ex) = false;

map remove_js : Spec # List(State) → List(State);
var sp : Spec;
  s : State;
  ls : List(State);
eqn remove_js(sp, []) = [];
  is_js(sp, s) → remove_js(sp, s |> ls) = remove_js(sp, ls);
  !is_js(sp, s) → remove_js(sp, s |> ls) = [s] ++ remove_js(sp, ls);

map ies_ps : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  s : State;
  ls, ex : List(State);
eqn ies_ps(sp, [], ex) = true;
  (is_ps(sp, s) && !subset(remove_js(sp, sm_cr(sp)(s)), ex)) →
      ies_ps(sp, s |> ls, ex) = false;
  !(is_ps(sp, s) && !subset(remove_js(sp, sm_cr(sp)(s)), ex)) →
      ies_ps(sp, s |> ls, ex) = ies_ps(sp, ls, ex);

map ies_js : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  s : State;
  ls, ex : List(State);
eqn ies_js(sp, [], ex) = true;
  (is_js(sp, s) && !subset(sm_jr(sp)(s), ex)) →
      ies_js(sp, s |> ls, ex) = false;
  !(is_js(sp, s) && !subset(sm_jr(sp)(s), ex)) →
      ies_js(sp, s |> ls, ex) = ies_js(sp, ls, ex);

map ies_parent : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  s : State;
  ls, ex : List(State);
eqn ies_parent(sp, [], ex) = true;
  (is_root_state(sp, s) || contains_parent(sp, ex, s)) →
      ies_parent(sp, s |> ls, ex) = ies_parent(sp, ls, ex);
  !(is_root_state(sp, s) || contains_parent(sp, ex, s)) →
      ies_parent(sp, s |> ls, ex) = false;

map contains_parent : Spec # List(State) # State → Bool;
var sp : Spec;
  ls : List(State);
  s, s' : State;
eqn contains_parent(sp, [], s') = false;
  s' in sm_cr(sp)(s) → contains_parent(sp, s |> ls, s') = true;
  !(s' in sm_cr(sp)(s)) → contains_parent(sp, s |> ls, s') = contains_parent(sp, ls, s');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% getEnteredTargets
map get_et : Spec # List(Transition) → List(State);
var sp : Spec;
  e : OnEvent;
  s' : State;
  ld : List(DoEvent);
  T : List(Transition);
eqn get_et(sp, []) = [];
  get_et(sp, tra(e, ld, s') |> T) = get_unique([s'] ++ get_ancestors(sp, s') ++ get_et(sp, T));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map get_xs : Spec # List(State) # List(Transition) → List(State);
var sp : Spec;
  s : State;
  ls : List(State);
  lt : List(Transition);
eqn get_xs(sp, [], lt) = [];
  xs_h(sp, s, lt) → get_xs(sp, s |> ls, lt) = [s] ++ get_xs(sp, ls, lt);
  !xs_h(sp, s, lt) → get_xs(sp, s |> ls, lt) = get_xs(sp, ls, lt);
```

```

map xs_h : Spec # State # List(Transition) → Bool;
var sp : Spec;
  s : State;
  t : Transition;
  lt : List(Transition);
eqn xs_h(sp, s, []) = false;
  css(sp, s, tra_s(t)) → xs_h(sp, s, t |> lt) = true;
  !css(sp, s, tra_s(t)) → xs_h(sp, s, t |> lt) = xs_h(sp, s, lt);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% initiate
map initiate : Spec # List(State) → List(State);
var sp : Spec;
  ls : List(State);
eqn initiate(sp, ls) = get_unique(initiate_helper(sp, ls, ls));

%% is_cs_initiated
map is_cs_initiated : Spec # List(State) # State → Bool;
var sp : Spec;
  s, s' : State;
  ls : List(State);
eqn is_cs_initiated(sp, [], s') = false;
  (s in sm_cr(sp)(s')) → is_cs_initiated(sp, s |> ls, s') = true;
  !(s in sm_cr(sp)(s')) → is_cs_initiated(sp, s |> ls, s') = is_cs_initiated(sp, ls, s');

map initiate_helper : Spec # List(State) # List(State) → List(State);
var sp : Spec;
  s : State;
  ls1, ls2 : List(State);
eqn initiate_helper(sp, [], ls2) = [];
  s == st_INTERNAL →
      initiate_helper(sp, s |> ls1, ls2) = initiate_helper(sp, ls1, ls2);
  is_ss(sp, s) || is_js(sp, s) →
      initiate_helper(sp, s |> ls1, ls2) = [s] ++ initiate_helper(sp, ls1, ls2);
  is_cs(sp, s) && is_cs_initiated(sp, ls2, s) →
      initiate_helper(sp, s |> ls1, ls2) = [s] ++ initiate_helper(sp, ls1, ls2);
  is_cs(sp, s) && !is_cs_initiated(sp, ls2, s) →
      initiate_helper(sp, s |> ls1, ls2) = [s] ++ get_entry_children(sp, s) ++
          initiate_helper(sp, ls1 ++ get_entry_children(sp, s), ls2);
  is_ps(sp, s) →
      initiate_helper(sp, s |> ls1, ls2) = [s] ++ get_entry_children(sp, s) ++
          initiate_helper(sp, ls1 ++ get_entry_children(sp, s), ls2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map jsu : Spec # List(State) → List(State);
var sp : Spec;
  s : State;
  ex : List(State);
eqn jsu(sp, ex) = jsu_helper(sp, remove_js(sp, ex), sm_s(sp), sm_s(sp), false);

map jsu_helper : Spec # List(State) # List(State) # List(State) # Bool → List(State);
var sp : Spec;
  ex, ls, ls' : List(State);
  s : State;
  b : Bool;
eqn jsu_helper(sp, ex, [], ls', true) = jsu_helper(sp, ex, ls', ls', false);
  jsu_helper(sp, ex, [], ls', false) = ex;
  (is_js(sp, s) && !(s in ex) && subset(sm_jr(sp)(s), ex)) →
      jsu_helper(sp, ex, s |> ls, ls', b) = jsu_helper(sp, ex ++ [s], ls, ls', true);
  !(is_js(sp, s) && !(s in ex) && subset(sm_jr(sp)(s), ex)) →
      jsu_helper(sp, ex, s |> ls, ls', b) = jsu_helper(sp, ex, ls, ls', b);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Execution State Update
map esu : Spec # List(State) # List(Transition) → List(State);
var sp : Spec;
  ex : List(State);
  t : List(Transition);
eqn esu(sp, ex, t) = sort_states(jsu(sp, initiate(sp,
      get_unique(list_minus_state(ex, get_xs(sp, ex, ft(t))) ++ get_et(sp, ft(t))))), sm_s(sp));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map get_do_events : Spec # List(State) # List(State) # List(Transition) # OnEvent → List(DoEvent);
var sp : Spec;
  ent, exi : List(State);
  lt : List(Transition);
  e : OnEvent;
eqn get_do_events(sp, ent, exi, lt, e) = get_do_events_exit(sp, exi) ++
      get_do_events_transitions(lt) ++ get_do_events_entry(sp, ent, e);

map get_do_events_transitions : List(Transition) → List(DoEvent);
var o : OnEvent;
  ld : List(DoEvent);
  s : State;
  lt : List(Transition);
eqn get_do_events_transitions([]) = [];
  get_do_events_transitions(tra(o, ld, s) |> lt) = ld ++ get_do_events_transitions(lt);

map get_do_events_exit : Spec # List(State) → List(DoEvent);
var sp : Spec;
  ls : List(State);
  s : State;
eqn get_do_events_exit(sp, []) = [];
  get_do_events_exit(sp, s |> ls) = sm_ex(sp)(s) ++ get_do_events_exit(sp, ls);

map get_do_events_entry : Spec # List(State) # OnEvent → List(DoEvent);
var sp : Spec;
  ent : List(State);
  s : State;
  e : OnEvent;
eqn get_do_events_entry(sp, [], e) = [];
  get_do_events_entry(sp, s |> ent, e) =
      get_do_events_entry_state(e, sm_ce(sp)(s), sm_oe(sp)(s)) ++ get_do_events_entry(sp, ent, e);

map get_do_events_entry_state : OnEvent # List(Transition) # List(OEH) → List(DoEvent);
var lt : List(Transition);
  e, e' : OnEvent;
  ld : List(DoEvent);
  s' : State;
  lo : List(OEH);
eqn get_do_events_entry_state(e, [], []) = [];
  (e == e') → get_do_events_entry_state(e, tra(e', ld, s') |> lt, lo) =
      ld ++ get_do_events_entry_state(e, lt, []);
  (e != e') → get_do_events_entry_state(e, tra(e', ld, s') |> lt, lo) =
      get_do_events_entry_state(e, lt, lo);
  get_do_events_entry_state(e, [], oeh(ld, s') |> lo) = ld ++ get_do_events_entry_state(e, [], lo);

map get_do_events_entry_state_init : Spec # List(State) → List(DoEvent);
var sp : Spec;
  ls : List(State);
  s : State;
eqn get_do_events_entry_state_init(sp, []) = [];
  get_do_events_entry_state_init(sp, s |> ls) =
      get_do_from_oeh(sm_oe(sp)(s)) ++ get_do_events_entry_state_init(sp, ls);

map get_do_from_oeh : List(OEH) → List(DoEvent);
var lo : List(OEH);
  o : OEH;
eqn get_do_from_oeh([]) = [];
  get_do_from_oeh(o |> lo) = oeh_d(o) ++ get_do_from_oeh(lo);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map entry_tr : Spec # List(State) # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  ex : List(State);
  e : OnEvent;
eqn entry_tr(sp, [], e) = [];
  entry_tr(sp , s |> ex, e) = ehth_other(sp, s, e) ++ ehth_cond(sp, s, e) ++ entry_tr(sp , ex, e);

map ehth_other : Spec # State # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  e : OnEvent;
eqn ehth_other(sp, s, e) = ehth_other_helper(sm_oe(sp)(s), s, e);

map ehth_other_helper : List(OEH) # State # OnEvent → List(Transition);
var ld : List(DoEvent);
  s, s' : State;
  lo : List(OEH);
  e : OnEvent;
eqn ehth_other_helper([], s, e) = [];
  (s != s' && ld == []) →
      ehth_other_helper(oeh(ld, s') |> lo, s, e) = [tra(e, [], s')] ++ ehth_other_helper(lo, s, e);
  (s == s' || ld != []) →
      ehth_other_helper(oeh(ld, s') |> lo, s, e) = ehth_other_helper(lo, s, e);

map ehth_cond : Spec # State # OnEvent → List(Transition);
var sp : Spec;
  s : State;
  e : OnEvent;
eqn ehth_cond(sp, s, e) = ehth_cond_helper(sm_ce(sp)(s), s, e);

map ehth_cond_helper : List(Transition) # State # OnEvent → List(Transition);
var ld : List(DoEvent);
  s, s' : State;
  lt : List(Transition);
  e, e' : OnEvent;
eqn ehth_cond_helper([], s, e') = [];
  (ld == [] && e == e' && s != s') → ehth_cond_helper(tra(e, ld, s') |> lt, s, e') =
      [tra(e, [], s')] ++ ehth_cond_helper(lt, s, e');
  !(ld == [] && e == e' && s != s') → ehth_cond_helper(tra(e, ld, s') |> lt, s, e') =
      ehth_cond_helper(lt, s, e');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map get_targets : List(Transition) → List(State);
var t : Transition;
  lt : List(Transition);
eqn get_targets([]) = [];
    (tra_s(t) == st_INTERNAL) → get_targets(t |> lt) = get_targets(lt);
    (tra_s(t) != st_INTERNAL) → get_targets(t |> lt) = [tra_s(t)] ++ get_targets(lt);

% Transition Handler
map tr_handler : Spec # List(State) # OnEvent → Pair;
var sp : Spec;
  ex : List(State);
  e : OnEvent;
eqn tr_handler(sp, ex, e) = tr_handler_helper_a(sp, ex, get_prio_tr_event(sp, ex, e), [], e);

map tr_handler_helper_a : Spec # List(State) # List(Transition) # List(DoEvent) # OnEvent → Pair;
var sp : Spec;
  ex : List(State);
  lt : List(Transition);
  ld : List(DoEvent);
  e : OnEvent;
eqn (lt == []) → tr_handler_helper_a(sp, ex, lt, ld, e) =
      p(ex, ld);
  (lt != []) → tr_handler_helper_a(sp, ex, lt, ld, e) =
      tr_handler_helper_b(sp, ex, esu(sp, ex, lt), lt, ld, e);

map tr_handler_helper_b : Spec # List(State) # List(State) # List(Transition) # List(DoEvent) #
      OnEvent → Pair;
var sp : Spec;
  ex, ex' : List(State);
  lt : List(Transition);
  ld : List(DoEvent);
  e : OnEvent;
eqn tr_handler_helper_b(sp, ex, ex', lt, ld, e) = tr_handler(sp, ex', lt, ld, e,
      list_union(list_intersect(ex', get_jr(sp, get_targets(lt))),
          list_union(get_targets(lt), list_minus_state(ex', ex))), list_minus_state(ex, ex'));

map get_jr : Spec # List(State) → List(State);
var sp : Spec;
  ent : List(State);
eqn get_jr(sp, ent) = get_jr_helper(sp, ss_j(sm_ss(sp)), ent);

map get_jr_helper : Spec # List(State) # List(State) → List(State);
var sp : Spec;
  js, ent : List(State);
  s : State;
eqn get_jr_helper(sp, [], ent) = [];
  overlap(sm_jr(sp)(s), ent) → get_jr_helper(sp, s |> js, ent) = [s] ++ get_jr_helper(sp, js, ent);
  !overlap(sm_jr(sp)(s), ent) → get_jr_helper(sp, s |> js, ent) = get_jr_helper(sp, js, ent);

map tr_handler : Spec # List(State) # List(Transition) # List(DoEvent) # OnEvent # List(State) #
      List(State) → Pair;
var sp : Spec;
  ex', ent, exi : List(State);
  lt : List(Transition);
  ld : List(DoEvent);
  e : OnEvent;
eqn tr_handler(sp, ex', lt, ld, e, ent, exi) = tr_handler_helper_a(sp, ex', entry_tr(sp, ent, e),
      ld ++ get_do_events(sp, ent, exi, lt, e), e);
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map cts : Spec # List(State) → Bool;
var sp : Spec;
  lst : List(State);
eqn cts(sp, lst) = cts_h(sp, lst, lst);

map cts_h : Spec # List(State) # List(State) → Bool;
var sp : Spec;
  lst, lst' : List(State);
  s : State;
eqn cts_h(sp, [], lst') = false;
  cts_hh(sp, s, lst') → cts_h(sp, s |> lst, lst') = true;
  !cts_hh(sp, s, lst') → cts_h(sp, s |> lst, lst') = cts_h(sp, lst, lst');

map cts_hh : Spec # State # List(State) → Bool;
var sp : Spec;
  s, s' : State;
  lst : List(State);
eqn cts_hh(sp, s, []) = false;
  css(sp, s, s') → cts_hh(sp, s, s' |> lst) = true;
  !css(sp, s, s') → cts_hh(sp, s, s' |> lst) = cts_hh(sp, s, lst);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map enabled : Spec # List(State) # OnEvent → Bool;
var sp : Spec;
  ex : List(State);
  e : OnEvent;
eqn enabled(sp, ex, e) = is_tr_def_ev(sp, ex, e) && !cuo(sp, ex, e) &&
      !cts(sp, get_targets(get_prio_tr_event(sp, ex, e)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


act FAIL;
  ev_print_job;
  ev_finish_job;
  ev_finish_color;
  ev_finish_scaling;
  ev_resolve_error;
  ev_reset_error;
  ev_color_error;
  ev_submit_job;
  val_non_empty_states;
  val_one_entry_root_state;
  val_cs_one_entry_child;
  val_ps_entry_children;
  val_js_not_entry_state;
  val_ss_no_children;
  val_cs_atleast_one_child;
  val_ps_atleast_two_children;
  val_js_no_children;
  val_js_child_of_ps;
  val_transition;
  val_only_joins_js;
  val_join_relation;
  val_cond_entry_handler;
  val_other_entry_handler;
  val_child_rel_irrefl;
  val_child_rel_1_parent;
  val_child_rel_acyclic;

proc SM(sp : Spec, pair : Pair, valid : List(Nat)) =
  (valid == []) → (
    (pair_de(pair) == []) → (
      (enabled(sp, pair_ex(pair), ev_print_job)
          → ev_print_job.SM(sp, tr_handler(sp, pair_ex(pair), ev_print_job), valid)
          <> ev_print_job.F())
      + (enabled(sp, pair_ex(pair), ev_finish_job)
          → ev_finish_job.SM(sp, tr_handler(sp, pair_ex(pair), ev_finish_job), valid)
          <> ev_finish_job.F())
      + (enabled(sp, pair_ex(pair), ev_finish_color)
          → ev_finish_color.SM(sp, tr_handler(sp, pair_ex(pair), ev_finish_color), valid)
          <> ev_finish_color.F())
      + (enabled(sp, pair_ex(pair), ev_finish_scaling)
          → ev_finish_scaling.SM(sp, tr_handler(sp, pair_ex(pair), ev_finish_scaling), valid)
          <> ev_finish_scaling.F())
      + (enabled(sp, pair_ex(pair), ev_resolve_error)
          → ev_resolve_error.SM(sp, tr_handler(sp, pair_ex(pair), ev_resolve_error), valid)
          <> ev_resolve_error.F())
      + (enabled(sp, pair_ex(pair), ev_reset_error)
          → ev_reset_error.SM(sp, tr_handler(sp, pair_ex(pair), ev_reset_error), valid)
          <> ev_reset_error.F())
      + (enabled(sp, pair_ex(pair), ev_color_error)
          → ev_color_error.SM(sp, tr_handler(sp, pair_ex(pair), ev_color_error), valid)
          <> ev_color_error.F())
      + (enabled(sp, pair_ex(pair), ev_submit_job)
          → ev_submit_job.SM(sp, tr_handler(sp, pair_ex(pair), ev_submit_job), valid)
          <> ev_submit_job.F())
    )
  ) <> (
    (head(valid) == 1) → val_non_empty_states.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 2) → val_one_entry_root_state.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 3) → val_cs_one_entry_child.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 4) → val_ps_entry_children.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 5) → val_js_not_entry_state.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 6) → val_ss_no_children.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 7) → val_cs_atleast_one_child.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 8) → val_ps_atleast_two_children.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 9) → val_js_no_children.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 10) → val_js_child_of_ps.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 11) → val_transition.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 12) → val_only_joins_js.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 13) → val_join_relation.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 14) → val_cond_entry_handler.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 15) → val_other_entry_handler.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 16) → val_child_rel_irrefl.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 17) → val_child_rel_1_parent.SM(sp, pair, tail(valid) ++ [0])
    + (head(valid) == 18) → val_child_rel_acyclic.SM(sp, pair, tail(valid) ++ [0])
  );

proc F = FAIL.F();

init SM(smmt_spec, init_state(smmt_spec), is_well_defined(smmt_spec));
```