RoboChart Reference Manual

Alvaro Miyazawa    Pedro Ribeiro
Kangfeng Ye       Ana Cavalcanti    Wei Li
Jim Woodcock      Jon Timmis
# Contents

1 Introduction ............................................. 13

## RoboChart

2 Language Structure ............................................. 17
2.1 RoboChart metamodel ............................................. 17
   2.1.1 Module ............................................. 18
   2.1.2 Controller ............................................. 19
   2.1.3 State Machine ............................................. 20
   2.1.4 Type Declaration ............................................. 22
   2.1.5 Expression ............................................. 22
   2.1.6 Action and Statement ............................................. 23
2.2 Timed Primitives ............................................ 24

3 Well-formedness Conditions ............................................. 27
3.1 Core Language ............................................. 27
   3.1.1 Robotic Platforms ............................................. 27
   3.1.2 Interfaces ............................................. 28
   3.1.3 Modules ............................................. 28
   3.1.4 Connection ............................................. 28
   3.1.5 Controllers ............................................. 29
   3.1.6 State Machines ............................................. 29
   3.1.7 States ............................................. 30
   3.1.8 Initial junctions ............................................. 30
3.1.9 Junction ................................................................. 30
3.1.10 Final states .......................................................... 30
3.1.11 Communications ..................................................... 31
3.1.12 Transitions ........................................................... 31
3.1.13 Operations ......................................................... 31
3.1.14 Variables ........................................................... 31
3.1.15 Expressions ......................................................... 31
3.2 Timed Language .......................................................... 32
3.2.1 State Machines ....................................................... 32
3.2.2 Transitions .......................................................... 32
3.2.3 Timed Expressions ................................................... 32
3.2.4 Timed Statements ................................................... 32
3.3 Probabilistic Language ................................................... 33
3.3.1 Transitions .......................................................... 33
3.3.2 Probabilistic Junctions .............................................. 33
4 Semantics ................................................................. 35
4.1 Detailed Semantics: Core Language ............................... 36
4.1.1 Modules ............................................................. 36
4.1.2 Controllers ........................................................ 40
4.1.3 State machines .................................................... 42
4.1.4 Statements ........................................................ 51
4.1.5 Expressions ........................................................ 53
4.2 Detailed Semantics: Timed Language .................................. 54
4.2.1 State machines .................................................... 54
4.2.2 States ............................................................... 64
4.2.3 Timed statements .................................................. 66
5 Probabilistic Semantics ................................................... 69
5.1 Overview ............................................................... 70
5.2 Semantic Domain ..................................................... 71
5.3 Translation to PRISM .................................................. 72
5.3.1 Translation requirements ........................................ 72
5.3.2 Normal form ....................................................... 73
5.3.3 Notions for rules .................................................... 73
5.3.4 PRISM ............................................................. 73
5.3.5 Normalisation rules ............................................... 76
5.3.6 Normal form to PRISM .......................................... 79

II Assertions

6 Assertion DSL Syntax ..................................................... 99
6.1 Standard Assertions .................................................... 99
<table>
<thead>
<tr>
<th>#</th>
<th>Rule</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Semantics of modules</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>Hidden Module channels</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>Memory channels</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>Function allEvents</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Function allVariables</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>Function requiredVariables</td>
<td>37</td>
</tr>
<tr>
<td>7</td>
<td>Function allLocalVariables</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Function allConstants</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>Function requiredConstants</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>Function allLocalConstants</td>
<td>37</td>
</tr>
<tr>
<td>11</td>
<td>Function requiredOperations</td>
<td>37</td>
</tr>
<tr>
<td>12</td>
<td>Module Memory</td>
<td>38</td>
</tr>
<tr>
<td>13</td>
<td>Constants Initialisation for Controllers and Modules</td>
<td>38</td>
</tr>
<tr>
<td>14</td>
<td>Composition of controllers</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>Renaming controller</td>
<td>39</td>
</tr>
<tr>
<td>16</td>
<td>Renaming controller events</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>Buffer</td>
<td>40</td>
</tr>
<tr>
<td>18</td>
<td>Single buffer</td>
<td>40</td>
</tr>
<tr>
<td>19</td>
<td>Semantics of controllers</td>
<td>40</td>
</tr>
<tr>
<td>20</td>
<td>Controller Memory</td>
<td>41</td>
</tr>
<tr>
<td>21</td>
<td>Composition of machines</td>
<td>41</td>
</tr>
<tr>
<td>22</td>
<td>Renaming state machine</td>
<td>42</td>
</tr>
<tr>
<td>23</td>
<td>Renaming machine events</td>
<td>42</td>
</tr>
</tbody>
</table>
**Timed Semantics**

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>Semantics of state machine</td>
</tr>
<tr>
<td>68</td>
<td>Constants Initialisation for State Machines</td>
</tr>
<tr>
<td>69</td>
<td>Build Scope</td>
</tr>
<tr>
<td>70</td>
<td>allClockVariables function</td>
</tr>
<tr>
<td>71</td>
<td>clockResets function</td>
</tr>
<tr>
<td>72</td>
<td>stmClocks function</td>
</tr>
<tr>
<td>73</td>
<td>alphaClockReset function</td>
</tr>
<tr>
<td>74</td>
<td>alphaClockReset function</td>
</tr>
<tr>
<td>75</td>
<td>Semantics of transitions</td>
</tr>
<tr>
<td>76</td>
<td>Clock resets for transitions</td>
</tr>
<tr>
<td>77</td>
<td>compileWC function</td>
</tr>
<tr>
<td>78</td>
<td>triggerEvent function</td>
</tr>
<tr>
<td>79</td>
<td>deadlineEvents function</td>
</tr>
<tr>
<td>80</td>
<td>State-machine Memory</td>
</tr>
<tr>
<td>81</td>
<td>allDeadlineTransitions function</td>
</tr>
<tr>
<td>82</td>
<td>memoryTransition function</td>
</tr>
<tr>
<td>83</td>
<td>Memory deadline</td>
</tr>
<tr>
<td>84</td>
<td>Semantics of states</td>
</tr>
<tr>
<td>85</td>
<td>Semantics of simple states</td>
</tr>
<tr>
<td>86</td>
<td>Semantics of composite states</td>
</tr>
<tr>
<td>87</td>
<td>Semantics of trigger deadlines</td>
</tr>
<tr>
<td>88</td>
<td>Composition of states</td>
</tr>
<tr>
<td>89</td>
<td>Semantics of statements</td>
</tr>
<tr>
<td>90</td>
<td>Semantics of statement deadlines</td>
</tr>
<tr>
<td>91</td>
<td>Semantics of wait statement</td>
</tr>
<tr>
<td>92</td>
<td>Semantics of Wait</td>
</tr>
<tr>
<td>93</td>
<td>Semantics of Wait</td>
</tr>
<tr>
<td>94</td>
<td>Semantics of Wait</td>
</tr>
<tr>
<td>95</td>
<td>Semantics of clock reset</td>
</tr>
<tr>
<td>96</td>
<td>Semantics of assignment</td>
</tr>
<tr>
<td>97</td>
<td>Semantics of call statement</td>
</tr>
</tbody>
</table>

**Probabilistic Semantics**

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Normalisation of state machines</td>
</tr>
<tr>
<td>2</td>
<td>Normalisation of states</td>
</tr>
<tr>
<td>3</td>
<td>Normalisation of NodeContainer</td>
</tr>
<tr>
<td>4</td>
<td>Normalisation of states without outgoing transitions</td>
</tr>
<tr>
<td>5</td>
<td>Normal junctions whose incoming transitions are combinable</td>
</tr>
<tr>
<td>6</td>
<td>Combinable transition</td>
</tr>
<tr>
<td>72</td>
<td></td>
</tr>
</tbody>
</table>
1. Introduction

The current practice of programming mobile and autonomous robots does not reflect the modern outlook of their applications. Such practice is often based on standard state machines, without formal semantics, to describe the robot controller only, with time and probabilistic properties discussed in natural language. In the design stage, the state machine guides the development of a simulation, but no rigorous connection between them is established.

In this report, we present a state-machine based notation, called RoboChart, for the specification and design of robotic systems. State machines are frequently, though informally, used in presenting and explaining the patterns of behaviours of particular robotic systems. These extra constructs embed the notions of robotic platforms and their controllers; communication between controllers can be synchronous or asynchronous. Besides state machines, RoboChart includes elements to organise specifications, fostering reuse and taming complexity.

The state-machine notation is fully specified, including an action language and constructs to specify timing and probabilistic properties. Operations used in a state machine can be taken from a domain-specific API or defined by other state machines; communication between state machines inside a controller is synchronous. Operations can be given pre and postconditions.

The time primitives of RoboChart allow time budgets and deadlines to be specified for operations and events directly as part of a state machine. Constraints can be specified in association with the relative-time elapsed since the occurrence of events or the entering of states. Our time primitives are inspired by constructs of timed automata [25] and Timed CSP [28].

UML [24] state machines are popular. RoboChart, however, is customised for robotic applications,
via the extra notions of robotic platform, controller, and a specialised API. Moreover, RoboChart provides support for time and probabilistic specifications that to make it suitable for verification and automatic generation of simulations.

In this report, we formalise the semantics of the core and timed constructs of RoboChart using CSP [7]. Importantly, CSP is a front end for a mathematical model that supports a number analysis techniques such as model-checking, which provide a high degree of automation, as well as more powerful (but not automatic) verification based on interactive and theorem proving, namely, Hoare and He’s Unifying Theories of Programming [9] (UTP). Use of CSP enables model checking with FDR [30]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [6] and probability [12].

Chapter 2 describes RoboChart models, and Chapter 3 defines their well-formedness conditions. Chapter 4 presents their semantics of RoboChart in CSP. Chapter 5 presents the probabilistic semantics of RoboChart and the translation from RoboChart to PRISM. Chapter 5 describes the API available for modelling robotic systems. Chapter 6 presents a number of models specified in RoboChart. Finally, Chapter 8 concludes with a summary of the results and future work.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Language Structure</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>RoboChart metamodel</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Timed Primitives</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Well-formedness Conditions</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Core Language</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Timed Language</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Probabilistic Language</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Semantics</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Detailed Semantics: Core Language</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Detailed Semantics: Timed Language</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Probabilistic Semantics</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Semantic Domain</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Translation to PRISM</td>
<td></td>
</tr>
</tbody>
</table>
2. Language Structure

In this chapter, we first describe the metamodel of RoboChart. For an overview of the language with an example, see Appendix A.

Sections A.1 describes the features to define time properties. Finally, Section 2.1 describes the RoboChart metamodel.

2.1 RoboChart metamodel

As explained above, a model is organised in packages, with their definitions shared using an imports mechanism similar to that of Java. Figure 2.1 defines a RoboChart package RCPackage. It has an optional name, and optionally imports other packages. All elements of a model are defined in a package. So, an RCPackage can include declarations of types, interfaces, modules, robotic platforms, controllers, and state machines.

The metamodel defines a notion of a ConnectionNode, which are components that may be connected via Connections. A RCPackage may define a number of such components, including ControllerDefs, RoboticPlatformDefs, andStateMachineDefs. These are shown in Figure 2.1 and their details are presented later in Figures 2.2 to 2.5. An interface groups variable Lists, operations, events and clocks.
Chapter 2. Language Structure

2.1.1 Module

The structure of a module is detailed in Figure 2.2. It comprises a number of connection nodes and connections. ConnectionNodes are elements that can be connected, namely, platforms, controllers, and state machines. In the case of module, though, the connection nodes cannot be state machines, and this is enforced via a well-formedness condition presented in the next chapter. The RoboticPlatform can be given by a RoboticPlatformDefinition or a by a RoboticPlatformReference. The other forms of ConnectionNode are detailed in later diagrams.

Connections are between a source (from) and a target (to) node, and in a module they establish the relationship between a platform and its controllers. Connections are established via a source (efrom) and a target (eto) event. They can be asynchronous and bidirectional, as indicated by the boolean attributes async and bidirec. An event may or not have a type, which is an Expression that defines the values that can be communicated via the connection, if any.

As mentioned before, a module gives a complete account of a robotic system. It defines a robotic platform, or includes a reference to a platform defined elsewhere, to indicate the facilities available. Modules associate their robotic platforms with particular controllers to specify behaviour. RoboChart state machines are not designed to model parallel or distributed behaviours. These should be modelled at the level of controllers and modules.
2.1 RoboChart metamodel

The structure of a Controller is shown in Figure 2.3. It can be specified by a Controller Definition or a Controller Reference, which just names a controller defined elsewhere. A Controller Definition encapsulates any number of state machines and defines a Context.

The structure of a Context is detailed in Figure 2.4, but briefly it defines the variables, including constants, operations, events, clocks, and provided, required, and defined interfaces of an element. Defined interfaces of an element declare the variables, clocks and events that are used for the specification of its behaviour; they are possibly shared if several elements are used to specify that behaviour. Well formedness rules establish the valid uses of interfaces in each element.

A Context is a Basic Context that has also interfaces. A Basic Context has Variables, Operations, Events, and Clocks. Variables are grouped in variable lists, with a modifier that indicates whether they are constants or indeed variables. A Variable has a name, a type (Expression), and an initial value.

Figure 2.4 also gives the metamodel for an Operation. It has an Operation Signature, which
defines its parameters, whether it terminates and its preconditions and postconditions. If there is more than one precondition, the actual precondition of the operation is their conjunction. If there is more than one postcondition, their disjunction is the actual postcondition. An Operation can also be defined by a reference or by a StateMachineBody.

### 2.1.3 State Machine

The metamodel of RoboChart state machines is similar to that of UML state machines. Features that have been removed are parallel regions, history junctions, and interlevel transitions. Whilst the state machines are designed with sequential control in mind, they may be in parallel with other machines in the same controller and with other controllers. There is also space for parallelism in the execution of during actions.

The structure of a RoboChart state machine is shown in Figure 2.5. It can be specified by a StateMachineReference or by a StateMachineDefinition. A definition gives a name to a StateMachineBody, which, as already mentioned, describes a Context. A StateMachineBody is a NodeContainer, which is composed a number of Nodes and Transitions. A State is a Node, and can be final. A Junction is also a Node and can be initial, or a ProbabilisticJunction.

An initial node indicates where the execution of a state-machine starts, a connective node provides the means for structuring more complex path between nodes, and a final node indicates the termination of the state-machine (or of the behaviour of a state). We note that a final node is a state, as the machine can stay in a final node. An initial node, however, is actually a junction, since a machine cannot remain in the initial node. A precise terminology is that the initial state is the target of the only transition that can come out of an initial junction.

States are the main components of a state machine. A State has actions: entry, during, and exit actions, executed in particular phases of its life-cycle. A State is also a NodeContainer, since it can contain nodes and transitions supporting the hierarchical feature of state machines, where
composed states have a machine to define behaviour while in that state.

Transitions are directed connections between two nodes: a source and a target. They may be triggered by an event, guarded by a condition, and contain an action that is executed when the transition is taken and its source node has exited. We can also specify a deadline for a transition to be triggered, and associate it with a number of Clocks, that are reset when the transition is taken. Additionally, a transition may have a probability value (between 0 and 1) that captures the probability of the transition being triggered.

The concrete syntax of transitions is shown in Syntax 2.1.1. A TriggerLabel defines a trigger as a Communication, or a probability value as an Expression, but not both as they are mutually exclusive in a Transition. A TimedLabel defines zero or more ClockResets and optionally a deadline as an Expression. In Label the Expression defines the guard condition and the Statement the action of the transition.

Syntax 2.1.1 — Transition Label.
Label ::= TriggerLabel TimedLabel ('[*Expression']')? ('/Statement)?
TriggerLabel ::= (Communication['p{'Expression'}'])?
TimedLabel ::= ClockReset* ('<{'Expression'}')?
ClockReset ::= '#{'Clock

The syntax of Communications is described in Syntax 2.1.2. It consists of an input, output, sync or simple communication. The concrete syntax of the different types of communications is shown in Table 2.1.
Chapter 2. Language Structure

Syntax 2.1.2 — Communication.
Communication ::= (Input|Output|Sync|Simple)
Input ::= Event ’?’ Variable
Output ::= Event ’!’ Expression
Sync ::= Event ’.’ Expression
Simple ::= Event

A Communication has an event, which can be on its own, associated with a value, or with a variable. In the first case, we have a Communication of type SIMPLE. Communications whose events are associated with a value correspond to a synchronisation (SYNC) or output (OUTPUT). Finally, communications whose events are associated with variables model input communications (INPUT).

<table>
<thead>
<tr>
<th>Communication Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Communication (I)</td>
<td>Receives any value from the event and stores it on the variable.</td>
</tr>
<tr>
<td>Output Communication (O)</td>
<td>Sends the value of the expression through the event.</td>
</tr>
<tr>
<td>Sync Communication (Sync)</td>
<td>Synchronises on the event with the value of the expression.</td>
</tr>
<tr>
<td>Simple Communication (S)</td>
<td>Synchronises on the event.</td>
</tr>
</tbody>
</table>

Table 2.1: Types of Communications.

2.1.4 Type Declaration

The metamodel for TypeDecl is given in Figure 2.6. As indicated, a TypeDecl is a NamedExpression, which means its name can be used as an expressions. The class of TypeDecl includes PrimitiveTypes (given types in Z), NamedTypes (type aliases), RecordTypes and Enumerations. The latter defines a set of Literals.

2.1.5 Expression

For simplicity, RoboChart adopts the type system of the Z Standard [1, 15]. This provides a simple and solid basis for implementation of all tools, including model checkers and theorem provers. It
also means that we do not need to specify the type system for RoboChart separately. Accordingly, the Expression language is also based on that of Z.

We note that, however, tools may take a pragmatic approach to facilitate the use of the Z mathematical toolkit. For example, if there is difficulty in providing facilities to record and use a mathematical toolkit, it might be hardcoded in a tool. We might also provide syntactic sugar (such as notation for dealing with arrays and matrices) for readability of expressions by roboticists. It is, however, the responsibility of the tool providers to justify compatibility with the Z notation. Specifically, a well typed RoboChart model accepted by a tool should correspond in a clear way to a model that is described purely using Z and its standard mathematical toolkit.

Precisely, the definition of Expression for RoboChart is that for Z [1, page 32], excluding all productions for schema constructs, but including the notation for Predicate [1, page 31] except for conjunction definitions using newline and ";", and relation operator application, which is captured by Application in Expression.

### 2.1.6 Action and Statement

Similarly, the action language is very simple. Syntax 2.1.3 gives the concrete syntax of statements used to define actions in states and transitions.

**Syntax 2.1.3 — Statements.**

```
Statement ::= 'skip'
          | N '(': (Expr (',' Expr)*)? ')'
          | 'if' Expr 'then' Statement 'else' Statement 'end'
          | N '=' Expr
          | N '!' Expr
          | N '?' N
          | N
          | N ',' Expr
          | Statement ';'; Statement
          | Statement '<=' Expr
          | 'wait' '(' Expr ')' 'wait' '(' '[' Expr ',' Expr ']' ')' '#' N
```

Statements can be used to construct state and transition actions. The syntax of state actions is shown in Syntax 2.1.4.
Syntax 2.1.4 — Actions.

Action ::= ('entry' | 'during' | 'exit') Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>skip</td>
<td>Statement that terminates immediately.</td>
</tr>
<tr>
<td>Call</td>
<td>o(e1,e2,...)</td>
<td>Calls operation (o) with parameters (e_i).</td>
</tr>
<tr>
<td>Conditional</td>
<td>if c then S1 else S2 end</td>
<td>If (c) is true, execute (S1), otherwise execute (S2).</td>
</tr>
<tr>
<td>Assignment</td>
<td>(x = e)</td>
<td>Assign expression (e) to variable (x).</td>
</tr>
<tr>
<td>Output event</td>
<td>(ev!e)</td>
<td>Output value (e) through channel (ev).</td>
</tr>
<tr>
<td>Input event</td>
<td>(ev?x)</td>
<td>Receive value through channel (ev) and store it in variable (x).</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>(ev.e)</td>
<td>Synchronise on value (e) through event (ev).</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>(ev)</td>
<td>Synchronise on event (ev).</td>
</tr>
<tr>
<td>Sequential composition</td>
<td>(S1;S2)</td>
<td>Execute (S1), and then (S2).</td>
</tr>
</tbody>
</table>

Table 2.2: Statements.

2.2 Timed Primitives

The time primitives are described separately in Figure 2.7. They are shown in the previous section for completeness, and summarised and explained here. The timed primitives appear in the syntax of expressions, statements, and transitions. A ClockExpression since is a condition involving a clock. A StateClockExpression is a sinceEntry expression. A TimedStatement defines a deadline to terminate as an Expression. A Wait and ClockReset are also statements. The delay defined by Wait is that defined by the attribute duration as an Expression. Finally, a Transition may define a deadline as an Expression, and may have zero or more ClockResets. Table 2.3 gives the concrete syntax.
2.2 Timed Primitives

Figure 2.7: Metamodel of RoboChart time primitives

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Expression</td>
<td>since(C)</td>
<td>Expression counting elapsed time since the last reset of clock C.</td>
</tr>
<tr>
<td>State Clock Expression</td>
<td>sinceEntry(S)</td>
<td>Expression counting elapsed time since entry of state S.</td>
</tr>
<tr>
<td>Timed Statement</td>
<td>S&lt;e</td>
<td>Statement S is required to terminate within e time units.</td>
</tr>
<tr>
<td>Wait</td>
<td>wait(e)</td>
<td>Waits for e units of time.</td>
</tr>
<tr>
<td>Nondeterministic Wait</td>
<td>wait([a,b])</td>
<td>Waits nondeterministically for d units of time where a≤d≤b.</td>
</tr>
<tr>
<td>Clock Reset</td>
<td>#C</td>
<td>Resets clock C.</td>
</tr>
<tr>
<td>Trigger deadline</td>
<td>t&lt;e</td>
<td>Transition trigger t is required to take place within e units.</td>
</tr>
</tbody>
</table>

Table 2.3: Timed Primitives.

We observe that a TimedStatement defines a deadline to terminate, but not a deadline to start. The possibility to specify a deadline to start was considered, however, because statements like assignment and operation calls are immediate, only an event synchronisation or a Wait statement could introduce a delayed start. For example, consider the case where we have an assignment of expression θ to variable x, sequentially composed with a call to operation op as x:=θ ; op(), then a deadline to start would be imposed on the assignment, which is immediate, and thus would be redundant. Another scenario arises if, instead, we consider the example Wait(d) ; op(). A starting deadline could constrain Wait(d), however, if we were to specify this statement as an operation waitOp(), then the starting deadline on waitOp() would be satisfied immediately, whereas this would not be the case for Wait(d) ; op().
This section has given a diagrammatic overview of the metamodel. A textual representation that specifies all the details is presented in Appendix B.
3. Well-formedness Conditions

The metamodel presented in the previous chapters defines models that are not meaningful. A model is characterised by a module definition, and all other definitions used there, directly or indirectly. We now define a number of well-formedness conditions for a model. They encode restrictions that are necessary for an adequate semantics to be defined.

Well formedness requires well typedness. Here, however, we do not focus on this aspect, except where this is not standard for an expression or statement. The type system of RoboChart is the type system of Z[15].

We present the conditions related to each of the elements of the core language in Section 3.1. We also provide here justifications for the restrictions. We follow that with conditions on the timed language in Section 3.2.

3.1 Core Language

3.1.1 Robotic Platforms

RP1 Robotic platforms cannot require interfaces.
RP2 Defined interfaces can only have events.
RP3 The names of variables, operations, and events are unique to the platform.
RP4 Robotic platforms cannot contain clocks.

We note that variables and operations declared directly in the platform, outside an interface, are
considered as if declared in a provided interface, for the reasons already explained above. Events declared directly in the platform, on the other hand, are defined.

### 3.1.2 Interfaces

| I1 | Provided interfaces contain only variables and operations, and required interfaces contain only variables, operation and clocks. |
| I2 | Defined interfaces contain only variables, events and clocks. |
| I3 | Names of variables, events, clocks and operations are unique. |

### 3.1.3 Modules

| M1 | A module must contain exactly one robotic platform, at least one controller, and not state machines. |
| M2 | All variables and operations required by the module’s controllers must be provided by the platform. |
| M3 | Each event on the robotic platform and controllers of a module must have at most one connection to or from it within the module. |

### 3.1.4 Connection

Both modules and controllers contain connections. Their conditions restrict the types of the connected elements, the nature of the connection, and the types of the associated events, which must be the same.

| Cn1 | Connections of a module must associate only events of the robotic platform and its controllers. |
| Cn2 | Connections involving a robotic platform are always asynchronous. |
| Cn3 | Connections of a controller must associate only its events and those of its state machines. |
| Cn4 | Only events of the same type may be connected. |
| Cn5 | Bidirectional connections of a module may only involve events of a controller which are connected by bidirectional connections within the controller. |
| Cn6 | Non-bidirectional connections of a module may only connect to events of a controller which have a non-bidirectional connection from them within the controller. |
| Cn7 | Non-bidirectional connections of a module may only connect from events of a controller which have a non-bidirectional connection to them within the controller. |
| Cn8 | Non-bidirectional connections of a controller must not connect to events that a state machine uses as an output. (An event is considered to be an output if it is used in an OUTPUT or SYNC communication, or if it is used in an OUTPUT, SYNC or SIMPLE send statement.) |
| Cn9 | Non-bidirectional connections of a controller must not connect from events that a state machine uses as an input. (An event is considered to be an input if it is used in an INPUT or SIMPLE communication, or if it is used in an INPUT send statement.) |
Cn10 The to-event of a connection must be an event of its to-context and the from-event of a connection must be an event of its from-context.

Cn11 Connections between state machines are always synchronous.

### 3.1.5 Controllers

C1 A controller must contain at least one state machine.

C2 Controllers cannot provide variables or operations to other controllers.

C3 All variables required by the controller’s state machines must be provided or required by the controller.

C4 All operations required by the controller’s state machines, including the machines that define operations, must be required or defined by the controller.

C5 The names of variables, operations, and events are unique to the controller.

C6 Each event on state machines and boundary of a controller must have at most one connection to or from it within the controller.

C7 Operations must not be declared directly in a controller, but may be defined in the controller.

C8 Operations referenced in a controller must be unique, that is, there must not be two or more references to the same operation.

C9 Controllers cannot have clocks.

C10 Controllers cannot define interfaces that contain clocks.

C11 Controllers cannot require interfaces that contain clocks.

C12 The operations defined in a controller can require only operations that are either defined or required by that controller.

Variables and events declared directly in the controller are considered as part of a defined interface.

In C3, the state machines do not include those that define operations. The restriction on provision of variables for operations are catered for STM8. In C3, we cater for provision of variables by the controller.

### 3.1.6 State Machines

STM1 State machines cannot have provided interfaces

STM2 Operations in state machines can only be required, not defined.

STM3 Every state machine must have exactly one initial junction.

STM4 State machines must contain at least one state (possibly a final state).

STM5 The names of variables, operations, and events are unique to the machine.

STM6 State machines must not have operations declared directly within them.

STM7 State machines must not require interfaces containing clocks (only use them).

STM8 A state machine that requires an operation, including a machine that defines another operation, must define or require all variables required by that operation.
STM9 A state machine that requires an operation, including a machine that defines another operation, must define all events defined in that operation.

Like for controllers, variables and events declared directly, outside of an interface, in a state machine are regarded as part of a defined interface.

We note that we do not require that a machine that calls an operation $\mathcal{O}_p$ provides the operations required by $\mathcal{O}_p$. They must be provided by the controller, either by a definition, or by requiring it (from the platform). This is ensured by C4 in Section 3.1.5. This creates a lack of uniformity in the treatment of the context of a called operation. It is needed, however, since otherwise transitivity means that the machines have to provide all data (variables) for all operations that are directly or indirectly called.

### 3.1.7 States

S1 If a state has a non-empty set of nodes, then conditions 3 and 4 of state machines apply.

S2 A state has at most one of each type of action: entry, during, and exit,

S3 An operation cannot be called multiple times in parallel within the same state machine. It is enough to check that an operation called in a during action is not called by any of its descendants.

### 3.1.8 Initial junctions

IJ1 An initial junction does not have incoming transitions.

IJ2 An initial junction must have exactly one outgoing transition.

IJ3 All junction conditions apply.

### 3.1.9 Junction

J1 A junction must contain at least one outgoing transition.

J2 The guards of the transitions out of a junction must form a cover.

J3 Transitions starting in junctions cannot have triggers.

### 3.1.10 Final states

FS1 Final states cannot be the source of transitions.

FS2 Final states cannot have actions.

FS3 Final states cannot have states or transitions.
3.1 Core Language

3.1.11 Communications

Tg1 A communication of type SIMPLE has neither the parameter attribute not the value attribute set. This is a pure synchronisations and does not involve exchange of values.

Tg2 A communication of type SIMPLE must use a typeless event. This is a pure synchronisation and does not involve exchange of values.

Tg3 A communication of type INPUT must have a parameter attribute and cannot have its value attribute set.

Tg4 A communication of type OUTPUT or SYNC must have a value attribute and cannot have its parameter attribute set.

Tg5 A communication of type empty must not have its attributes event, parameter and value set.

3.1.12 Transitions

T1 The source and target of a transition must belong to the same container.

T2 If a transition has a trigger, it must be of type INPUT or SIMPLE.

3.1.13 Operations

O1 All state-machine conditions apply to operation definitions.

O2 Recursive and mutually recursive operations are not allowed.

3.1.14 Variables

V1 If the initial value of a required variable or constant of a state machine or controller is defined, it must be consistent with the value of any (complementing) variable provided by the contexts (controllers or modules) where the state machine or controller is used.

3.1.15 Expressions

E1 The variables declared in a set comprehension must not have initial values.

E2 Quantified variables in existential and universal quantifications must not have initial values.

E3 The variables quantified in a lambda expression must not have initial values.
3.2 Timed Language

3.2.1 State Machines

TSTM1 A state machine that requires an operation, including a machine that defines another operation, must define or require all clocks required in that operation. We note that a state machines that does not define an operation, but defines the behaviour of a controller, cannot requires a clock (see STM7). In this case, if it requires an operation, it must define the clocks required by that operation.

3.2.2 Transitions

TT1 A transition with a deadline must have a trigger.

3.2.3 Timed Expressions

TE1 Expressions involving since(C) and sinceEntry(S) are only permitted in transition guards.

TE2 The clock C in an expression since(C) may only reference a clock declared within the expression’s containing state-machine.

TE3 The state S in an expression sinceEntry(S) may only reference a state, and not a Final state, within the containing expression’s state-machine.

TE4 The expressions since(C) or sinceEntry(S) may only occur in a comparison expression in which the other branch is an expression that does not involve other since(C2) or sinceEntry(S2). A consequence of this restriction is that no expression can compare the value of two clocks as given by since(C) or sinceEntry(S).

TE5 An expression sinceEntry(S) may only reference a state S within the same node container as the transition whose guard uses sinceEntry(S).

Name Disambiguation in State Clock Expressions

When a state name S, referenced in a state clock expression, is ambiguous, because, for instance, there is a state and a substate with the same name in the same state machine, the fully qualified name of the state S must be used.

3.2.4 Timed Statements

TS1 A clock reset #C may only reference a clock declared within the action’s containing state-machine, or in the case of a transition, within the transition’s containing state-machine.

TS2 In a non-deterministic wait([m,n]) the condition m < n must hold.
3.3 Probabilistic Language

3.3.1 Transitions

- **PT1** The source of a transition with a probability value must be a probabilistic junction, that is, states, initial junctions, and normal junctions cannot be the sources of these probabilistic transitions.

- **PT2** The probability value of a transition must be between 0 and 1.

3.3.2 Probabilistic Junction

Probabilistic Junctions are also junctions, but with extra well-formedness conditions.

- **PJ1** There must be a probability value on every outgoing transition from a probabilistic junction.

- **PJ2** There must not be a guard on an outgoing transition from a probabilistic junction.

- **PJ3** The probability values of all outgoing transitions from a probabilistic junction must sum to 1.
4. Semantics

For the purpose of this semantics, the functions \( \text{vid}, \text{eventId}, \text{tid} \) and \( \text{id} \) calculate unique identifiers for their parameters, which are, respectively, variables, events, transitions and node containers (states and state machines). One possible implementation of such functions is to calculate the qualified name, and this is the implementation realised by RoboTool.

Additionally, in the semantics the set of events \( \text{Event} \) contains an event \text{internal}, that corresponds to the event of a triggerless transitions. In the implementation RoboTool, this is represent in the trigger by a null value, and the semantic rules have been adapter to handle it appropriately.

Finally, we assume the existence of a function that takes an expression and returns the set of variables used in that expression.

An overview of the semantics presented here, and a detailed explanation of many of the semantic definitions can be found in [5]. The complete definition is given in the sequel.
4.1 Detailed Semantics: Core Language

4.1.1 Modules

**Rule 1. Semantics of modules**

\[ [m: \text{Module}]_M: \text{CSPProcess} = \]

\[
\begin{align*}
\Theta_{\{\text{end}\}}^{\text{Skip}} \setminus \{\text{end}\}
\end{align*}
\]

\[\Theta_{\{\text{end}\}}^{\text{Skip}} \setminus \{\text{end}\}\)

\[
\begin{align*}
\Theta_{\{\text{end}\}}^{\text{Skip}} \setminus \{\text{end}\}
\end{align*}
\]

where

\[
\text{ctrls} = (x: m.\text{controllers})
\]

\[
\text{cons} = m.\text{connections}
\]

\[
\text{asyncs} = \{c: \text{cons} | c.\text{async} \land \{c.\text{from}, c.\text{to}\} \cap \text{RoboticPlatform} = \emptyset\}
\]

\[
\text{evasyncs} = \{c: \text{asyncs} \cdot \text{eventId}(c.\text{eto}) \cup \{c: \text{asyncs} \cdot \text{eventId}(c.\text{efrom})\}
\]

**Rule 2. Hidden Module channels**

\[
\text{hiddenModuleChannels}(m: \text{Module}) : \text{ChannelSet} =
\]

\[
\{c: \text{asyncs} \cdot \text{eventId}(c.\text{eto}) \cup \{c: \text{asyncs} \cdot \text{eventId}(c.\text{efrom})\} \cup \text{memoryChannels}^2(m)
\]

where

\[
\text{asyncs} = \{c: m.\text{connections} | c.\text{async} \land \{c.\text{from}, c.\text{to}\} \cap \text{RoboticPlatform} = \emptyset\}
\]

**Rule 3. Memory channels**

\[
\text{memoryChannels}(m: \text{Module}) : \text{ChannelSet} =
\]

\[
\{v: \text{allLocalVariables}^1(rp) \cdot \text{set}_\text{vid}(v)\} \cup
\{v: \text{allLocalConstants}^1(rp) \cdot \text{set}_\text{vid}(v)\} \cup
\{v: \text{requiredVariables}^1(c) \cdot \text{set}_\text{EXT}_\text{vid}(v, c)\}
\]

where

\[
\text{rp} = \text{roboticPlatformDefinition}(m)
\]

The set channels for the constants of the platform are hidden here. If the initial value is not defined, this introduces a non-determinism as all possible initial values are considered. These sets are also synchronised with the controllers to guarantee that the constant value is the same in the controllers.

**Rule 4. Function allEvents**

\[
\text{allEvents}(c: \text{Context}) : \text{Set}(\text{Event}) =
\]

\[
\text{c.events} \cup \{i: c.\text{interfaces} \cdot i.\text{events}\}
\]
Rule 5. Function allVariables

\[
\text{allVariables}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{l}: c.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{PInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \}
\]

Rule 6. Function requiredVariables

\[
\text{requiredVariables}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} 
\]

Rule 7. Function allLocalVariables

\[
\text{allLocalVariables}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{l}: c.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{PInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'var\'} \cdot \text{l}.\text{vars} \} \}
\]

Rule 8. Function allConstants

\[
\text{allConstants}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{l}: c.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{PInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} \}
\]

Rule 9. Function requiredConstants

\[
\text{requiredConstants}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} 
\]

Rule 10. Function allLocalConstants

\[
\text{allLocalConstants}(c: \text{Context}) : \text{Set(Variable)} = \\
\bigcup \{ \text{l}: c.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} \cup \\
\bigcup \{ \text{i}: c.\text{PInterfaces} \cdot \bigcup \{ \text{l}: \text{i}.\text{variableList} \mid \text{l}.\text{modifier} == \text{\'const\'} \cdot \text{l}.\text{vars} \} \}
\]

Rule 11. Function requiredOperations

\[
\text{requiredOperations}(c: \text{Context}) : \text{Set(Operation)} = \\
\bigcup \{ \text{i}: c.\text{RInterfaces} \cdot \text{i}.\text{operations} \}
\]
**Rule 12. Module Memory**

\[
\text{modMemory}(m : \text{Module}) : \text{CSPProcess} = \\
\quad \text{let } Memory(v : \text{lvars}) \cdot \text{set_vid(v)}?x \rightarrow \\
\qquad \{ c : \text{controllers(v)} \cdot \text{set.Ext_vid(v,c)} x \rightarrow \text{Skip} \}; \text{Memory}(\text{vars}|\text{name}(v) := x) \\
\text{within} \\
\quad \text{constInit}(\text{rp}) ; \text{Memory}(\text{varvalues})
\]

where

- \( \text{rp} = \text{roboticPlatformDefinition}(m) \)
- \( \text{ctrls} = m.\text{controllers} \)
- \( \text{lvars} = \text{allLocalVariables}(\text{rp}) \)
- \( \text{vars} = (v : \text{lvars} \cdot \text{name}(v)) \)
- \( \text{varvalues} = (v : \text{lvars} \cdot \text{initial}(v)) \)
- \( \text{rcontrollers} = \lambda v \cdot \{ c : \text{ctrls} \mid v \in \text{requiredVariables}(c) \} \)

For each constant, in interleaving, the module memory either sets the initial value if it is defined, or queries it.

**Rule 13. Constants Initialisation for Controllers and Modules**

\[
\text{constInit}(\text{node} : \text{ConnectionNode}) : \text{CSPProcess} = \\
\quad \| c : \text{consts} \cdot \\
\qquad \left( \begin{array}{c}
\text{if } c.\text{initial} \neq \text{NULL} \text{ then } \\
\text{set_vid(c)}!(c.\text{initial}) \downarrow \text{Expr} \rightarrow \text{Skip} \\
\text{else} \\
\text{set_vid(c)}?\text{name}(c) \rightarrow \text{Skip}
\end{array} \right)
\]

where

- \( \text{consts} = \text{allConstants}(\text{node}) \)

The function \( \text{initial} \) picks an initial value of the appropriate type for a variable. If the variable defines an initial value, this value is used.
Rule 14. Composition of controllers

\[
\text{composeControllers}(m : \text{Module}, \text{ctrls} : \text{Seq}(	ext{Controller}), \text{cons} : \text{Set}(	ext{Connection})) : \text{CSPProcess} = \\
\begin{array}{l}
\text{if} \#\text{ctrls} = 1 \\
\text{then} \\
\text{renamingController}^1(m, \text{headctrls}, \text{cons}) \\
\text{else} \\
\text{renamingController}^2(m, \text{headctrls}, \text{cons}) \\
\text{composeControllers}^2(m, \text{tailctrls}, \text{cons})
\end{array}
\]

where \(\text{connevts} = \text{renCtrlEvts}^1(m, \text{headctrls}, \text{cons}) \cap \bigcup \{c : \text{tailctrls} \cdot \text{renCtrlEvts}^2(m, c, \text{cons})\}\)

The controller’s \(\text{set}_c\) events for required constants are renamed to match the \(\text{set}_c\) event for the corresponding provided constants of the module (\(\text{container}(c)\)).

Rule 15. Renaming controller events

\[
\text{renCtrlEvts}(m : \text{Module}, c : \text{Controller}, \text{cons} : \text{Set}(	ext{Connection})) : \text{ChannelSet} = \\
\begin{array}{l}
\{c : \text{internalConns} \cdot \text{eventId}(e.eto).in \leftarrow \text{eventId}(e.efrom).out\} \\
\bigcup \{c : \text{internalConns} \cdot \text{eventId}(e.eto).out \leftarrow \text{eventId}(e.efrom).in\} \\
\bigcup \{e : \text{fromPlatform} \cdot \text{eventId}(e.efrom) \leftarrow \text{eventId}(e.eto)\} \\
\bigcup \{e : \text{toPlatform} \cdot \text{eventId}(e.eto) \leftarrow \text{eventId}(e.efrom)\} \\
\bigcup \{v : \text{requiredConstants}^2(c) \cdot \text{set}_v(\text{set}_v(v), m)\}
\end{array}
\]

where

\[
\begin{align*}
\text{internalConns} &= \{x : \text{cons} : \{x.\text{from}, x.\text{to}\} \subseteq \text{Controller} \land \neg x.\text{async} \land c \in \{x.\text{from}, x.\text{to}\}\} \\
\text{toPlatform} &= \{x : \text{cons} : x.\text{from} = c \land x.\text{to} \in \text{RoboticPlatform}\} \\
\text{fromPlatform} &= \{x : \text{cons} : x.\text{to} = c \land x.\text{from} \in \text{RoboticPlatform}\}
\end{align*}
\]

Controllers that require the same constant synchronise with each other on the \(\text{set}_c\) event of the module, as well as with the module memory.
Chapter 4. Semantics

Rule 17. Buffer

buffer(c : Connection) : CSPProcess =

if c.mult then
  singleBuffer(c.efrom, c.eto) || singleBuffer(c.eto, c.efrom)
else
  singleBuffer(c.efrom, c.eto)

Rule 18. Single buffer

singleBuffer(efrom : Event, eto : Event) : CSPProcess =

if eto.type ≠ null then
  let Buffer(()) ≡ eventid(efrom).out!x → Buffer((x))
  Buffer((v)) ≡ eventid(efrom).out!x → Buffer((x)) □ eventid(eto).in!v → Buffer(())
  within Buffer(())
else
  let Buffer(false) ≡ eventid(efrom).out → Buffer(true)
  Buffer(true) ≡ eventid(efrom).out → Buffer(true) □ eventid(eto).in → Buffer(false)
  within Buffer(false)

4.1.2 Controllers

Rule 19. Semantics of controllers

\[ c : \text{ControllerDef}, c : \text{CSPProcess} = \]

let
  for each op : c.lOperations *
  nproc(op)(\{x : op.parameters • x.name\}) = opdef.nops
  \[ \text{STM} \]
  for each op : requiredOperations(c) *
  nproc(op)(\{x : op.parameters • x.name\}) =
  o.name@Call.\{x : op.parameters • x.name\} → SKIP
  within
  \[
  \left( \left( \text{composeMachines}(c, ms, cs)^nops \right | \text{lvars} \cup \text{rvars} \cup \text{lconsts} \cup \text{rconsts} \right) \right) \Theta_{\text{env}} \text{Skip}
  \]
  where
  ms = (x : c.machines)
  cs = c.connections
  opdef = findOperationDefinition(op, c.lOperations)
  nops = \{op : c.lOperations • id(op) → nproc(op)\} \cup
  \{op : requiredOperations(c) • id(op) → nproc(op)\}
  lvars = \{v : allLocalVariables(c) • set_vid(v)\}
  rvars = \{v : requiredVariables(c) • set_Ext_vid(v)\}
  lconsts = \{v : allLocalConstants(c) • set_vid(v)\}
  rconsts = \{v : requiredConstants(c) • set_vid(v)\}
The state machine synchronise with the memory controller on the set events of all constants of the controller (required and local), but only the set events of the local constants are hidden. The set events of the required variables are later renamed and synchronised with the memory of the module.

**Rule 20. Controller Memory**

\[
\text{ctrlMemory}(c : \text{ControllerDef}) : \text{CSPProcess} =
\]

\[
\begin{align*}
\text{let } & \text{Memory}(\text{vars}) = \\
& \left( \begin{array}{l}
\begin{cases}
\Box v : \text{lvars} \cdot \text{set}_{\text{vid}}(v) ? x \\
\langle \text{m} : \text{rmachines}(v) \cdot \text{set}_{\text{Ext}_{\text{vid}}}(v, m), ! x \rightarrow \text{Skip} ; \\
\text{Memory}(\text{vars}[\text{name}(v) := x])
\end{cases}
\end{array} \\
\Box v : \text{rvars} \cdot \text{set}_{\text{Ext}_{\text{vid}}}(v) ? x \\
\langle \text{m} : \text{rmachines}(v) \cdot \text{set}_{\text{Ext}_{\text{vid}}}(v, m), ! x \rightarrow \text{Skip} ; \\
\text{Memory}(\text{vars}[\text{name}(v) := x])
\end{cases}
\right)
\end{align*}
\]

\[
\text{within } constInit^2(c) ; \text{Memory}(\text{varvalues})
\]

where

\[
\begin{align*}
\text{ms} &= c.\text{machines} \\
\text{lvars} &= \text{allLocalVariables}^4(c) \\
\text{rvars} &= \text{requiredVariables}^4(c) \\
\text{vars} &= (v : \text{rvars} \cup \text{lvars} \cdot \text{name}(v)) \\
\text{varvalues} &= (v : \text{rvars} \cup \text{lvars} \cdot \text{initial}(v)) \\
\text{rmachines} &= \lambda v \cdot \{ m : \text{ms} \mid v \in \text{requiredVariables}^5(m) \}
\end{align*}
\]

Similarly to the module memory, the controller memory initially reads the value of each constant in the controller. Both local and required constants are initialised here. The synchronisation between the controller and the module guarantee that the required constants (that are provided by the module) are initialised with the same value.

**Rule 21. Composition of machines**

\[
\text{composeMachines}(c : \text{Controller}, \text{ms} : \text{Seq(StateMachineDef)}, \text{cons} : \text{Set(Connection)})^{\text{nops}} : \\
\text{CSPProcess} =
\]

\[
\begin{align*}
\text{if } & \# \text{ms} = 1 \\
\text{then } & \text{renamingMachine}^1(c, \text{headms}, \text{cons})^{\text{nops}} \\
\text{else } & \text{renamingMachine}^2(c, \text{headms}, \text{cons})^{\text{nops}} \\
& \left[ \text{connevts} \right] \\
& \text{composeMachines}^2(c, \text{tailms}, \text{cons})^{\text{nops}}
\end{align*}
\]

where

\[
\text{connevts} = \text{renStmEvts}^1(c, \text{headms}, \text{cons}) \sqcap \bigcup \{ m : \text{tailms} \cdot \text{renStmEvts}^2(c, m, \text{cons}) \}
\]
4.1.3 State machines

**Rule 22. Renaming state machine**

\[ \text{renamingMachine}(c : \text{Controller}, m : \text{StateMachineDef}, \text{cons} : \text{Set(Connection)})^{\text{nops}} : \text{CSPProcess} = \]

\[
\begin{align*}
\text{[stm]}^{\text{nops}}_{\text{STM}} & \quad \{ e : \text{internalConns} \times \text{eventId(e.etofrom)}, \text{in} \leftarrow \text{eventId(e.etofrom)}, \text{out} \\
& \quad \cup \{ e : \text{fromController} \times \text{eventId(e.etofrom)}, \text{out} \leftarrow \text{eventId(e.etofrom)}, \text{in} \\
& \quad \cup \{ e : \text{toController} \times \text{eventId(e.etofrom)}, \text{in} \leftarrow \text{eventId(e.etofrom)} \\
& \quad \cup \{ v : \text{requiredConstants} \times (m) \times \text{set_vid}(v) \leftarrow \text{set_vid}(v,c) \}
\end{align*}
\]

\[
\text{where}
\begin{align*}
\text{internalConns} & = \{ x : \text{cons} \times \{ x \text{.from, x.to} \} \subseteq \text{StateMachine} \land m \in \{ x \text{.from, x.to} \} \\
\text{toController} & = \{ x : \text{cons} \times x \text{.from} = m \land x \text{.to} \in \text{Controller} \\
\text{fromController} & = \{ x : \text{cons} \times x \text{.to} = m \land x \text{.from} \in \text{Controller} \\
\end{align*}
\]

**Rule 23. Renaming machine events**

\[ \text{renStmEvts}(c : \text{Controller}, m : \text{StateMachineDef}, \text{cons} : \text{Set(Connection)}) : \text{ChannelSet} = \]

\[
\begin{align*}
\{ x : \text{internalConns} \times \text{eventId(x.etofrom)} \} \cup \{ \text{end} \} \cup \{ x : \text{requiredConstants} \times (m) \times \text{set_vid}(v,c) \}
\end{align*}
\]

\[
\text{where}
\begin{align*}
\text{internalConns} & = \{ x : \text{cons} \times \{ x \text{.from, x.to} \} \subseteq \text{StateMachine} \land m \in \{ x \text{.from, x.to} \} \\
\end{align*}
\]

4.1.3 State machines

**Rule 24. Semantics of state machine**

\[ [\text{stm} : \text{StateMachineDef}]^{\text{nops}}_{\text{STM}} : \text{CSPProcess} = \]

\[
\begin{align*}
\left[ \begin{array}{l}
\text{initialisation}^{\text{nops}} (\text{stm}) \\
\text{flowevts} \\
\text{composeStates}^{\text{nops}} ((\{ x : \text{stm.nodes} \times x \in \text{State} \}) \cup \{ \text{enter}, \text{entered}, \text{exit}, \text{exited} \}) \\
\Theta_{\text{end}} \text{Skip} \\
\end{array} \right] \\
\text{getsetLocalChannels}^{\text{nops}} (\text{stm}) \cup \{ \text{internal} \} \\
\end{align*}
\]

\[
\text{where}
\begin{align*}
\text{flowevts} & = \] \\
\cup \{ x : \text{SID}, \text{States}^{\text{nops}} (\text{stm}) : y : \text{States}^{\text{nops}} (\text{stm}) \times \{ \text{enter}, \text{x,y}, \text{entered}, \text{x,y}, \text{exit}, \text{x,y}, \text{exited}, \text{x,y} \} \}
\end{align*}
\]

**Rule 25. Rename Transition Trigger Events**

\[ \text{renameTriggerEvents}(\text{stm} : \text{StateMachineDef}) : \text{RenamingSet} = \]

\[
\{ e : \text{allEvents} \times (m) ; \text{tid} : \text{TIDS} \times \text{eventId(e)} \times \text{tid} \leftarrow \text{eventId(e)} \}
\]
Rule 26. Function states
\[
\text{states}(n : \text{NodeContainer}) : \text{Set(State)} = \text{x.nodes} \cap (\text{State} \cup \text{Final})
\]

Rule 27. Initialisation
\[
\text{initialisation}(n : \text{NodeContainer})^{\text{ops}} : \text{CSPProcess} = \begin{cases}
\text{if } js \neq 0 \text{ then} & [[(t : \text{Transition} \mid t.\text{source} = i), n, \text{true}]]^{\text{Skip.Skip.ops}} \\
\text{else} & \text{let} \\
& \text{for each } j : js \cdot \text{id}(j) = [[j, n, \text{true}]]^{\text{Skip.Skip.ops}} \\
& \text{within} \\
& [[(t : \text{Transition} \mid t.\text{source} = i), n, \text{true}]]^{\text{Skip.Skip.ops}} \\
\end{cases}
\]

where
\[
\#n.\text{nodes} > 0 \\
i = i x : n.\text{nodes} | x \in \text{initial} \\
js = \text{reachableJunctions}^1(i)
\]

Rule 28. Reachable Junctions from Node
\[
\text{reachableJunctions}(n : \text{Node}) : \text{Set(Node)} = \\
\{ t : \text{parent}(n).\text{transitions} \mid t.\text{target} \in \text{Junction} \cdot t.\text{source} \rightarrow t.\text{target} \}^+ \{ n \}
\]

\text{obs.}
The reachable junctions are the relational image of the transitive closure of the transitions and singleton \{n\}.

Rule 29. Semantics of Junctions
\[
[[j : \text{Junction}, \text{origin} : \text{NodeContainer}, \text{initial} : \text{boolean}]^{P,Q,\text{ops}} : \text{Set(Node)} = \\
\begin{array}{c}
\square \{ t : \text{transitionsFrom}^1(j) \cdot \{ t, o, l \}^{P,Q,\text{ops}} \\
\end{array}
\]

Rule 30. Get and Set channels
\[
\text{getsetChannels}(s : \text{StateMachineDef}) : \text{ChannelSet} = \\
\{ v : \text{allVariables}^1(s) \cdot \text{get}_{\text{vid}}(v) \} \cup \{ v : \text{allVariables}^2(s) \cdot \text{set}_{\text{vid}}(v) \} \cup \\
\{ v : \text{allConstants}^2(s) \cdot \text{get}_{\text{vid}}(v) \} \cup \{ v : \text{allConstants}^3(s) \cdot \text{set}_{\text{vid}}(v) \}
\]
Chapter 4. Semantics

**Rule 31. Composition of states**

\[
\text{composeStates}(ss: \text{seqState}, p: \text{NodeContainer})^{\text{nops}} : \text{CSPProcess} = \\
\begin{align*}
\text{if} & \#ss = 1 \\
\text{then} & \\
\text{restrictedState}^1(p, \text{headss})^{\text{nops}} \\
\text{else} & \\
\left( \text{restrictedState}^2(p, \text{headss})^{\text{nops}} \right) \setminus \text{cflowevts} \\
\end{align*}
\]

where

\[
\text{cflowevts} = \text{flowEvents}^1(\text{headss}, p) \cup \{ x : \text{tailss} \bullet \text{flowEvents}^2(x, p) \}
\]

**Rule 32. Trigger events**

\[
\text{trigEvents}(s: \text{StateMachineDef}) : \text{ChannelSet} = \\
\{ \{ t : \text{allTransitions}^1(s) \bullet \text{triggerEvent}^1(t.\text{trigger}.\text{id}(t)) \} \}
\]

**Rule 33. Get and set local channels**

\[
\text{getsetLocalChannels}(s: \text{StateMachineDef}) : \text{ChannelSet} = \\
\{ \{ v : \text{allVariables}^3(s) \bullet \text{get}_\text{vid}(v) \} \cup \{ \{ v : \text{allLocalVariables}^5(s) \bullet \text{set}_\text{vid}(v) \} \cup \\
\{ \{ v : \text{allConstants}^4(s) \bullet \text{get}_\text{vid}(v) \} \cup \{ \{ v : \text{allLocalConstants}^3(s) \bullet \text{set}_\text{vid}(v) \} \}
\]

The state machine hides all \text{get} events for both constants and variables, but only hides the \text{set} events for local constants and variables.

**Rule 34. Flow events**

\[
\text{flowEvents}(s: \text{State}, p: \text{NodeContainer}) : \text{ChannelSet} = \\
\{ \{ x : \text{states}^3(p); y : \{ \text{id}(s) \} \bullet \} \\
\text{enter}.y.x, \text{entered}.y.x, \text{exit}.y.x, \text{exited}.y.x, \\
\text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y, \\
\} \}
\]

**Rule 35. Semantics of states**

\[
\llbracket s : \text{State} \rrbracket^{\text{nops}} : \text{CSPProcess} = \\
\text{This function is split in multiple rules according to the type of states.}
\]
Rule 36. Semantics of simple states

\[
[s : \text{State}]^\text{nops} : \text{CSPProcess} = \\
\text{let} \\
\text{Inactive} \triangleq \text{enter} ? o : \text{sids.id}(s) \rightarrow \text{Activating}(o) \\
\text{Activating}(o) \triangleq [s \text{.entry}]^\text{nops} ; \text{entered}.o.id(s) \rightarrow \\
([s \text{.during}]^\text{nops} ; \text{Stop}) \Delta \text{Inactive} \text{Activating}.ops \\
\begin{cases} \\
\text{if} : \text{transitionsFrom}^2(s) \bullet [t, s, \text{false}] & \text{Inactive} \text{Activating}.ops \\
\text{else} & \\
\text{for each j : reachableJunctions}^2(s) \bullet \text{id}(j) = [j, s, \text{false}] & \text{Inactive} \text{Activating}.ops \\
\end{cases}
\]

within

Inactive

where

\#states^2(s) = 0 \\
sids = SIDS \setminus \{id(s)\} \land \text{tids} = TIDS \setminus \text{tIDS}(s) \\
exit = \text{exit} ? as : \text{sids.id}(s) \rightarrow [s \text{.exit}]^\text{nops} ; \text{exited}.as.id(s) \rightarrow \text{Skip}

Rule 37. Semantics of composite states

\[
[s : \text{State}]^\text{nops} : \text{CSPProcess} = \\
\text{let} \\
\text{Inactive} \triangleq \text{enter} ? o : \text{sids.id}(s) \rightarrow \text{Activating}(o) \\
\text{Activating}(o) \triangleq [s \text{.entry}]^\text{nops} ; \text{initialisation}^2(s)^\text{nops} ; \text{entered}.o.id(s) \rightarrow \\
([s \text{.during}]^\text{nops} ; \text{Stop}) \Delta \text{Inactive} \text{Activating}.ops \\
\begin{cases} \\
\text{if} : \text{transitionsFrom}^2(s) \bullet [t, s, \text{false}] & \text{Inactive} \text{Activating}.ops \\
\text{else} & \\
\text{for each j : reachableJunctions}^2(s) \bullet \text{id}(j) = [j, s, \text{false}] & \text{Inactive} \text{Activating}.ops \\
\end{cases}
\]

within

(Inactive \text{flowtrigger} \text{evts}) \text{composeStates}^2((x : \text{states}^2(s), s)^\text{nops}) \setminus \text{flowevts}

where

\#states^2(s) > 0 \\
\text{flowevts} = \bigcup(x : SIDS, \text{states}^2(s), y : \text{states}^2(s) \bullet [\text{enter} \cdot x \cdot y, \text{entered}.x \cdot y, \text{exit} \cdot x \cdot y, \text{exited}.x \cdot y]) \\
\text{flowtrigger} \text{evts} = \text{flowtrigger} \text{events}^2(s) \\
sids = SIDS \setminus \{id(s)\} \land \text{tids} = TIDS \setminus \text{tIDS}(s) \\
exit = \text{exit} ? as : \text{sids.id}(s) \rightarrow \text{exitedSubstates}^2(s) ; \text{[s.exit]}^\text{nops} ; \text{exited}.as.id(s) \rightarrow \text{Skip}
Rule 38. Semantics of final states

\[ [s : \text{Final}]^{\text{nops}} : \text{CSPProcess} = \]

\[
\text{enter} ? x : \text{id} s(d(s)) \rightarrow \text{entered} . x . \text{id}(s) \rightarrow \left( \begin{array}{l}
\text{if} (\text{parent}(s) \in \text{StateMachine}) \\
\text{then} \quad \text{end} \rightarrow \text{Skip} \\
\text{else} \quad \text{Stop}
\end{array} \right)
\]

Rule 39. Synchronisation events between parent state and substates

\[ \text{flowTriggerEvents} (s : \text{State}) : \text{ChannelSet} = \]

\[
(\{ e : \text{Event}; t : \text{TIDS} \bullet e . t \} \setminus (\text{substatesTriggers}^1(s) \cup \text{non_interrupting_reachable_transitions})) \cup \\
\bigcup(x . \text{SID}s . \text{states}^7(s) ; y : \text{states}^1(s) \bullet \{ \text{enter }, x . y, \text{entered}, x . y, \text{exit }, x . y, \text{exited}, x . y \})
\]

where

\[ \text{non_interrupting_reachable_transitions} = \{ t : \text{reachableTransitions}^1(s) \mid t . \text{source} \notin \text{StateMachine} \} \]

Rule 40. Reachable transitions

\[ \text{reachableTransitions}(s : \text{Node}) : \text{Set(Transition)} \]

\[
\bigcup(j : \text{reachableJunctions}^4(s) \bullet \text{transitionsFrom}^6(j)) \cup \text{transitionsFrom}^5(s)
\]

Rule 41. Triggers of substates

\[ \text{substatesTriggers}(s : \text{State}) : \text{ChannelSet} = \]

\[
\{ t : \text{allTransitions}^2(s) \bullet \text{triggerEvent}^2(t . \text{trigger}, \text{id}(t)) \}
\]

Rule 42. Restricted semantics of states

\[ \text{restrictedState}(p : \text{NodeContainer}, s : \text{State})^{\text{nops}} : \text{CSPProcess} = \]

\[
[s]^{\text{nops}} | \text{all_other_transitions}_S \setminus \text{all_transitions}_PS \} \text{Skip}
\]

where

\[ \text{tidsfromwithin} = \{ t : \text{transitionsFrom}^6(s) \cup \text{allTransitions}^3(s) \bullet \text{id}(t) \} \]

\[ \text{all_other_transitions}_S = \{ e : \text{Event}; t \text{d} : \text{TIDS} \setminus \text{tidsfromwithin} \bullet \text{eventId}(e), \text{tid} \} \]

\[ \text{all_transitions}_PS = \{ e : \text{Event}; t : \text{TIDS} \bullet \text{eventId}(e), \text{tid} \} \]

\[ \{ t : \text{allTransitions}^4(p) \bullet \text{eventId}(t . \text{trigger}, \text{event}), \text{id}(t) \} \]
Rule 43. Semantics of transitions

\[ \begin{align*}
\text{if } & \text{src } \in \text{State} \quad \text{then} \\
& [t \text{.trigger}]^t \quad \text{exit.id(src).id(src) } \rightarrow \text{exitSubstates}_2 (\text{src}) \\
& \text{exited.id(src).id(src) } \rightarrow \text{compileTarget}_1 (\text{tgtsrc}, \text{false})^PQ \\
\text{elseif } & \text{src } \in \text{Initial} \quad \text{then} \\
& \text{internal.id(t) } \rightarrow \text{compileTarget}_2 (\text{tgt}, \text{parent}(\text{src}), \text{true})^PQ \\
\text{elseif } & \text{src } \in \text{Junction} \quad \text{then} \\
& \text{internal.id(t) } \rightarrow \text{compileTarget}_3 (\text{tgt}, \text{origin}, \text{initial})^PQ \\
\text{where} \\
& \text{src } = t \text{.source} \\
& \text{tgt } = t \text{.target}
\end{align*} \]

Rule 44. Compile target

\[ \begin{align*}
\text{compileTarget}(\text{tgt}, \text{o}, \text{i})_PQ & \text{CSPProcess } = \\
& \text{if } (\text{tgt } \in \text{State}) \text{ then} \\
& \text{if } (\text{tgt } = \text{o}) \text{ then } \text{enter.id(o).id(tgt) } \rightarrow \text{Q} \\
& \text{else } \text{enter.id(o).id(tgt) } \rightarrow \text{entered.id(o).id(tgt) } \rightarrow \left( \begin{align*}
& \text{if } (\text{i}) \text{ then } \text{Skip} \\
& \text{else } P
\end{align*} \right) \\
& \text{else if } (\text{tgt } \in \text{Junction}) \text{ then} \\
& \text{id(tgt)}
\end{align*} \]

Rule 45. Exit substates

\[ \begin{align*}
\text{exitSubstates}(\text{s})_PQ & \text{CSPProcess } = \\
& \text{if } \#(\text{s}) > 0 \text{ then} \\
& \text{exit.id(s)^2z : } (x : \text{states}_1 (s) \bullet \text{id}(x)) \rightarrow \text{exited.id(s).z } \rightarrow \text{Skip} \\
& \text{else} \\
& \text{Skip}
\end{align*} \]
Chapter 4. Semantics

Rule 46. State Machine Memory

\[
\text{stmMemory}(\text{stm} : \text{StateMachineDef}) : \text{CSPProcess} = \\
\begin{align*}
& \text{let} \ Memory(\text{vars}) \triangleq \\
& \quad \begin{cases}
\text{v} : \text{lvars} \bullet (get\_vid(\text{v})!\text{name}(\text{v}) \rightarrow Memory(\text{vars})) \\
\text{v} : \text{rvars} \bullet (get\_vid(\text{v})!\text{name}(\text{v}) \rightarrow Memory(\text{vars})) \\
\text{v} : \text{allConstants}\delta(\text{stm}) \bullet \text{get}\_\text{vid}(\text{v})!\text{name}(\text{v}) \rightarrow Memory(\text{vars}) \\
\text{t} : \text{allTransitions}\delta(\text{stm}) \bullet \text{memoryTransition}\delta(\text{t}) : Memory(\text{vars}) \\
\end{cases} \\
& \quad \text{within} \ constInitSTM \left(\text{consts}, \text{stm}, Memory(\text{varvalues})\right) \\
\end{align*}
\]

where
- \text{rvars} = \text{requiredVariables}\delta(\text{stm})
- \text{lvars} = \text{allLocalVariables}\delta(\text{stm})
- \text{consts} = \{\text{v} : \text{allConstants}\delta(\text{stm}) \bullet \text{v}\}
- \text{vars} = \{\text{v} : \text{rvars} \cup \text{lvars} \bullet \text{name}(\text{v})\} \cap \{\text{v} : \text{consts} \bullet \text{name}(\text{v})\}
- \text{varvalues} = \{\text{v} : \text{rvars} \cup \text{lvars} \bullet \text{initial}(\text{v})\} \cap \{\text{v} : \text{consts} \bullet \text{name}(\text{v})\}

The state machine memory initially reads the value of all required and local constants using the function \text{constInitSTM}. These values are read in sequence and passed as a parameter to the recursive process \text{Memory}, which then offer the value of the constants through a get channel. Only noninitialised constants are read. The initial value of initialised constants are passed directly to the recursive process \text{Memory}.

Rule 47. Constants Initialisation for State Machines

\[
\text{constInitSTM}(\text{cs} : \text{Seq}_1(\text{Variable}), \text{stm} : \text{StateMachineDef}, \text{P} : \text{CSPProcess}) : \text{CSPProcess} = \\
\begin{align*}
& \text{buildScope}\delta(\text{undefc}, \text{stm}, \text{let} \ c : \text{defs} \bullet \text{name}(c) = [[\text{c}.\text{initial}]]_{\text{Expr}}^\delta \text{within} \text{P}) \\
\end{align*}
\]

where
- \text{defc} = \{\text{x} : \text{consts} | \text{x}.\text{initial} \neq \text{null}\}
- \text{undefc} = \{\text{x} \bullet \text{consts} | \text{x}.\text{initial} = \text{null}\}
Rule 48. Build Scope

\[
\text{buildScope}(cs : \text{Seq}(\text{Variable}), \text{ctx} : \text{ConnectionNode}, \text{P} : \text{CSPProcess}) : \text{CSPProcess} =
\]

\[
\begin{cases}
\text{if (headcs).initial == NULL then} \\
\quad \text{set_vid}(\text{headcs}, \text{ctx})?\text{name}(\text{headcs}) \rightarrow \\
\quad \text{else buildscope}_2^{\text{tailcs}, \text{ctx}, \text{P}}
\end{cases}
\]

Rule 49. Semantics of triggers

\[
[t : \text{Communication}]_{\text{trigger}}^\text{t} : \text{CSPProcess} =
\]

\[
\begin{cases}
\text{if } t \neq \text{null} \land t\text{.event}\text{.type} \neq \text{null} \\
\quad \text{eventid}(t\text{.event}).\text{tid}.in?x \rightarrow \text{set_vid}(t\text{.parameter})!x \rightarrow \text{Skip}
\end{cases}
\]

Rule 50. Semantics of triggers for memory

\[
\text{triggerForMemory}(t : \text{Communication}, \text{tid} : \text{TIDS}) : \text{CSPProcess} =
\]

\[
\begin{cases}
\text{if } t \neq \text{null} \land t\text{.event}\text{.type} \neq \text{null} \\
\quad \text{eventid}(t\text{.event}).\text{tid}.in?x \rightarrow \text{Skip}
\end{cases}
\]

Rule 51. Event for transition trigger

\[
\text{triggerEvent}(t : \text{Communication}, \text{tid} : \text{TIDS}) : \text{CSPEvent} =
\]

\[
\begin{cases}
\text{if } t \neq \text{null} \\
\quad \text{eventid}(t\text{.event}).\text{tid}
\end{cases}
\]

else

\[
\text{internal}\cdot\text{tid}
\]
Chapter 4. Semantics

Rule 52. Memory transitions

\[
\text{memoryTransition}(t : \text{Transition}) : \text{CSPProcess} = \\
\begin{cases}
\text{if } t.\text{trigger} \neq \text{null} \land t.\text{trigger.type} = \text{CommunicationType.SYNC} \text{ then} \\
\quad \text{if } t.\text{condition} \neq \text{null} \text{ then} \\
\quad \quad \quad \quad (\{t.\text{condition}\}_{\text{Expr}}) \land \text{eventlid}(t.\text{trigger.event}).id(t).in.\{t.\text{trigger.value}\}_{\text{Expr}}^+ \rightarrow \text{Skip} \\
\quad \quad \text{else} \\
\quad \quad \quad \quad \text{eventlid}(t.\text{trigger.event}).id(t).in.\{t.\text{trigger.value}\}_{\text{Expr}}^+ \rightarrow \text{Skip} \\
\text{else if } t.\text{trigger} \neq \text{null} \land t.\text{trigger.type} = \text{CommunicationType.INPUT} \text{ then} \\
\quad \text{if } t.\text{condition} \neq \text{null} \text{ then} \\
\quad \quad \text{let} \\
\quad \quad \quad tr = t.\text{trigger} \\
\quad \quad \quad pn = tr.\text{parameter.name} \\
\quad \quad \quad ptype = tr.\text{parameter.type} \\
\quad \quad \quad cond = t.\text{condition} \\
\quad \quad \text{within} \\
\quad \quad \quad \text{eventlid(tr.event).id(t).in?pn : \{pn | pn \leftarrow \{ptype\}_{\text{Expr}}^+ \iff \text{cond}_{\text{Expr}} \} \rightarrow \text{Skip} \\
\quad \text{else} \\
\quad \quad \quad \text{eventlid(t.trigger.event).id(t).in?t.trigger.parameter.name} \\
\text{else if } t.\text{trigger} \neq \text{null} \land t.\text{trigger.type} = \text{CommunicationType.OUTPUT} \text{ then} \\
\quad \text{if } t.\text{condition} \neq \text{null} \text{ then} \\
\quad \quad (\{t.\text{condition}\}_{\text{Expr}}^+ \land \text{eventlid}(t.\text{trigger.event}).id(t).out.\{t.\text{trigger.value}\}_{\text{Expr}}^+ \rightarrow \text{Skip} \\
\quad \text{else} \\
\quad \quad \quad \text{eventlid}(t.\text{trigger.event}).id(t).out.\{t.\text{trigger.value}\}_{\text{Expr}}^+ \rightarrow \text{Skip} \\
\text{else if } t.\text{trigger} \neq \text{null} \land t.\text{trigger.type} = \text{CommunicationType.SIMPLE} \text{ then} \\
\quad \text{if } t.\text{condition} \neq \text{null} \text{ then} \\
\quad \quad (\{t.\text{condition}\}_{\text{Expr}}^+ \land \text{eventlid}(t.\text{trigger.event}).id(t).in \rightarrow \text{Skip} \\
\quad \text{else} \\
\quad \quad \quad \text{eventlid}(t.\text{trigger.event}).id(t).in \rightarrow \text{Skip} \\
\quad \text{else \ if } t.\text{condition} \neq \text{null} \text{ then} \\
\quad \quad (\{t.\text{condition}\}_{\text{Expr}}^+ \land \text{internal}.id(t) \rightarrow \text{Skip} \\
\quad \text{else} \\
\quad \quad \quad \text{internal}.id(t) \rightarrow \text{Skip} \\
\end{cases}
\]

Rule 53. Function transitionsFrom

\[
\text{transitionsFrom}(s : \text{Node}) : \text{Set(Transition)} = \\
\{ t : \text{parent}(s).\text{transitions} \mid t.\text{source} = s \bullet t \}
\]

Rule 54. Function allTransitions

\[
\text{allTransitions}(s : \text{NodeContainer}) : \text{Set(Transition)} = \\
\{ s.\text{transitions} \cup \bigcup (x : s.\text{nodes} \mid s \in \text{State} \bullet \text{allTransitions'}(x)) \}
\]
4.1.4 Statements

Rule 55. Semantics of statements

\[ [s : \text{Statement}]^{\text{nops}}_{\text{Statement}} : \text{CSPProcess} = \]

This rule is split in multiple rules according to the subtype of the statement.

The semantics of statements, in general, has the format

\[
\text{get}_x ? x_1 \rightarrow \ldots \rightarrow \text{get}_x ? x_n \rightarrow P
\]

where the channels \( \text{get}_x \) read values from the memory and the process \( P \) models the actual statement. The input events \( \text{get}_x ? x_i \) build a context where all the state components used in the expressions of the statement are declared. The process \( P \) is then run on this context.

In order to simplify our semantic rules, we use the following function that helps in building the context.

Rule 56. Read state of an expression

\[
\text{readState}(\text{vs} : \text{seq}(\text{Variable}), P : \text{CSPProcess}) : \text{CSPProcess} =
\]

\[
\text{if}(\#\text{vs} = 0) \text{then}
P \\
\text{else}
\text{get}_\text{vid}(\text{headvs}) ? (\text{headvs}).\text{name} \rightarrow \text{readState}^1(\text{tailvs}, P)
\]

This function reads a list of state variables and executes a process in that context. The variables must be read in sequence so that the final process can be executed in the full context. The order in which the variables are read is not important because the memory is always prepared to respond to a get event.

We define the function \( [\_]^{\text{StatementInContext}} \) to separate the application of \( \text{readState} \) from the core semantics of the statement given by the rule \( [\_]^{\text{Statement}} \). We additionally use the function \( \text{usedVariables} \) that takes a statement and calculates the set of variables used by the expressions in the statement.

Rule 57. Semantics of statements in context

\[
[s : \text{Statement}]^{\text{nops}}_{\text{StatementInContext}} : \text{CSPProcess} =
\]

\[
\text{readState}^2(\text{usedVariables}^1(s), [s]^{\text{nops}}_{\text{Statement}})
\]
Rule 58. Function usedVariables

\[ \text{usedVariables}(s : \text{Statement}) : \text{Set(Variable)} = \]

\[
\begin{align*}
\text{if } s \in \text{Assignment} & \text{ then } \\
\quad \text{usedVariables}(s.\text{right}) & \text{ else if } s \in \text{Call} \text{ then } \\
\quad \bigcup \{ x : s.\text{args} \ni \text{usedVariables}(x) \} & \text{ else if } s \in \text{IfStatement} \text{ then } \\
\quad \text{usedVariables}(s.\text{expression}) & \text{ else if } s \in \text{CommunicationStmt} \land s.\text{communication.type} \in \{ \text{SYNC, OUTPUT} \} \text{ then } \\
\quad \text{usedVariables}(s.\text{communication.value}) & \text{ else if } s \in \text{TimedStatement} \text{ then } \\
\quad \text{usedVariables}(s.\text{stmt}) & \text{ else if } s \in \text{CommunicationStmt} \land s.\text{communication.type} \notin \{ \text{SYNC, OUTPUT} \} \text{ then } \\
\quad \text{usedVariables}(s.\text{start}) & \text{ else if } s \in \text{TimedStatement} \text{ then } \\
\quad \text{usedVariables}(s.\text{end}) & \text{ else }
\end{align*}
\]

\{ \}

Rule 59. Function usedVariables

\[ \text{usedVariables}(e : \text{Expression}) : \text{Set(Variable)} = \]

The definition of this function is standard and omitted for now.

Rule 60. Semantics of assignment

\[ \llbracket s : \text{Assignment} \rrbracket^{\text{ops}}_{\text{Statement}} \cdot \text{CSPProcess} = \]

\[
\begin{align*}
\text{set}_\text{vid}(s.\text{left}) & \cdot \llbracket s.\text{right} \rrbracket_{\text{Expr}^{13}} \rightarrow \text{Skip}
\end{align*}
\]

Rule 61. Semantics of call statement

\[ \llbracket s : \text{Call} \rrbracket^{\text{ops}}_{\text{Statement}} \cdot \text{CSPProcess} = \]

\[
\begin{align*}
\text{op.name} & \circ \text{ops}(\text{op})(\{ x : s.\text{args} \ni \llbracket x \rrbracket_{\text{Expr}^{15}} \}) \\
\text{where}
\end{align*}
\]

\[
\text{op} = s.\text{operation}
\]

Rule 62. Semantics of if statements

\[ \llbracket s : \text{IfStatement} \rrbracket^{\text{ops}}_{\text{Statement}} \cdot \text{CSPProcess} = \]

\[
\begin{align*}
\left( \begin{array}{c}
\text{if } \llbracket s.\text{expression} \rrbracket_{\text{Expr}^{13}} \\
\text{then } \llbracket s.\text{then} \rrbracket^{\text{ops}}_{\text{StatementInContext}} \\
\text{else if } s.\text{else} \neq \text{null} \text{ then } \llbracket s.\text{else} \rrbracket^{\text{ops}}_{\text{StatementInContext}} \\
\text{else } \text{Skip}
\end{array} \right)
\end{align*}
\]
4.1 Detailed Semantics: Core Language

**Rule 63. Semantics of communication statements**

\[
\begin{align*}
\text{if} \ (\text{type} \ = \ \text{INPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{in}\?\text{par}.\text{name} & \rightarrow \ \text{set}_{\text{vid}}(\text{par}.\text{par}.\text{name}) \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{OUTPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out}! [\text{value}]_{\text{Expr}^{16}} & \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{SIMPLE}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out} & \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(\text{event}).\text{out}. [\text{value}]_{\text{Expr}^{17}} & \rightarrow \text{Skip}
\end{align*}
\]

*where*

- \( \text{type} = \text{s.communication.type} \)
- \( \text{event} = \text{s.communication.event} \)
- \( \text{value} = \text{s.communication.value} \)
- \( \text{par} = \text{s.communication.parameter} \)

**Rule 64. Semantics of sequential composition**

\[
\begin{align*}
\text{if} \ (\text{type} \ = \ \text{INPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{in}\?\text{par}.\text{name} & \rightarrow \ \text{set}_{\text{vid}}(\text{par}.\text{par}.\text{name}) \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{OUTPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out}! [\text{value}]_{\text{Expr}^{16}} & \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{SIMPLE}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out} & \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(\text{event}).\text{out}. [\text{value}]_{\text{Expr}^{17}} & \rightarrow \text{Skip}
\end{align*}
\]

*where*

- \( \text{type} = \text{s.communication.type} \)
- \( \text{event} = \text{s.communication.event} \)
- \( \text{value} = \text{s.communication.value} \)
- \( \text{par} = \text{s.communication.parameter} \)

**Rule 65. Semantics of skip**

\[
\begin{align*}
\text{if} \ (\text{type} \ = \ \text{INPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{in}\?\text{par}.\text{name} & \rightarrow \ \text{set}_{\text{vid}}(\text{par}.\text{par}.\text{name}) \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{OUTPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out}! [\text{value}]_{\text{Expr}^{16}} & \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{SIMPLE}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out} & \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(\text{event}).\text{out}. [\text{value}]_{\text{Expr}^{17}} & \rightarrow \text{Skip}
\end{align*}
\]

*where*

- \( \text{type} = \text{s.communication.type} \)
- \( \text{event} = \text{s.communication.event} \)
- \( \text{value} = \text{s.communication.value} \)
- \( \text{par} = \text{s.communication.parameter} \)

**Rule 66. Semantics of actions**

\[
\begin{align*}
\text{if} \ (\text{type} \ = \ \text{INPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{in}\?\text{par}.\text{name} & \rightarrow \ \text{set}_{\text{vid}}(\text{par}.\text{par}.\text{name}) \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{OUTPUT}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out}! [\text{value}]_{\text{Expr}^{16}} & \rightarrow \text{Skip} \\
\text{if} \ (\text{type} \ = \ \text{SIMPLE}) \ \text{then} \\
\quad \text{eventId}(\text{event}).\text{out} & \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(\text{event}).\text{out}. [\text{value}]_{\text{Expr}^{17}} & \rightarrow \text{Skip}
\end{align*}
\]

*where*

- \( \text{type} = \text{s.communication.type} \)
- \( \text{event} = \text{s.communication.event} \)
- \( \text{value} = \text{s.communication.value} \)
- \( \text{par} = \text{s.communication.parameter} \)

4.1.5 Expressions

The translation of expressions is standard. We note, however, that the expression language of Z is much richer than that of CSP. So, in some cases, we require user input to translate the expressions. Notably, this is the case of function applications. If the function is defined in RoboChart using pre and postconditions, the user needs to provide a CSP definition.
4.2 Detailed Semantics: Timed Language

The semantics of modules and controllers is the same as the untimed semantics. Here we describe the rules of the timed semantics to accommodate the timed constructs of RoboChart, namely clocks and deadlines over triggers and actions. The untimed semantics of state machines and states is largely reused, and so we present the rules by focusing on the changes required to accommodate the timed semantics.

4.2.1 State machines

The semantics of state machines is changed to cope with clocks and transition deadlines, while the semantics of actions is changed to accommodate \( \text{Wait} \) and deadlines on actions. Clocks are not modelled explicitly, instead for each transition guarded by an expression using \( \text{since}(C) \) or \( \text{sinceEntry}(S) \), we model the timed part of such an expression explicitly using additional CSP processes. Their semantics, which is described in the sequel, is given for a state machine as \( \text{stmClocks}(stm, wcs) \), which relies on the calculation of \( wcs \), a partial function from transitions to pairs, where the first component is the guard with occurrences of \( \text{since}(C) \) and \( \text{sinceEntry}(S) \) replaced by a fresh boolean variable, and whose second component is a partial function from the original expression to the fresh boolean variable. Because an expression involving clocks can also depend on the value of other variables, the memory process \( \text{stmMemory}(stm, wcs) \) also takes \( wcs \) as a parameter. Finally, compared with the untimed semantics of a state machine, the hiding on entered events is moved to the outer composition of the memory and the states as \( \text{sinceEntry}(S) \) conditions require the clocks to observe entered events.
Rule 67. Semantics of state machine

\[
\begin{align*}
\text{stm} : \text{StateMachineDef} &\xrightarrow{\text{nops}} \text{TimedCSPProcess} = \\
&\begin{cases}
\text{initialisation}^2(\text{stm}) \xrightarrow{\text{nops}} \\
\langle \text{flowevts} \rangle \\
\text{composeStates}^4(\langle x : \text{stm}.\text{nodes} \mid x \in \text{State} \rangle, \text{stm}) \xrightarrow{\text{nops}} \\
\langle \text{getsetChannels}^2(\text{stm}) \cup \text{trigEvents}^2(\text{stm}) \cup \text{clockResets}^1(\text{wcs}) \cup \text{deadlineEvents}^1(\text{stm}) \rangle \\
\langle \text{constInitSTM}^1(\text{consts}, \text{stm}, \langle \text{stmMemory}^1(\text{stm}.\text{wcs}) \cup \text{clockMemSync} \cup \text{stmClocks}^1(\text{stm}) \rangle) \rangle \\
\langle \text{renameTriggerEvents}^2(\text{stm}) \rangle \\
\langle \text{getsetLocalChannels}^2(\text{stm}) \cup \text{clockResets}^2(\text{wcs}) \cup \text{deadlineEvents}^2(\text{stm}) \cup \{\text{\texttt{end}}, \text{\texttt{entered}}\} \rangle \\
\end{cases}
\end{align*}
\]

where
\[
\text{wcs} = \{ t : \text{allTransitions}^2(\text{stm}) \mid t.\text{condition} \neq \text{null} \land t \mapsto \text{wc}(t.\text{condition}) \}
\]

\[
\text{clockMemSync} = \begin{cases}
\langle t : \text{Transition} \mid t \in \text{dom} \text{wcs} \mapsto \text{triggerEvent}^1(t) \rangle \\
\cup \langle v : \text{allClockVariables}^1(\text{wcs}) \mapsto \text{setWC}_\text{vid}(v) \rangle
\end{cases}
\]

\[
\text{flowevts} = \bigcup \{ x : \text{SIDS} \cup \text{states}^2(\text{stm}) ; y : \text{states}^3(\text{stm}) \cup \{ \text{enter}, x, y, \text{entered}, x, y, \text{exit}, x, y, \text{exited}, x, y \} \}
\]

\[
\text{consts} = \{ v : \text{allConstants}^2(\text{stm}) \cup v \}
\]

Rule 68. Constants Initialisation for State Machines

\[
\text{constInitSTM}(\text{cs} : \text{Seq}(\text{Variable}), \text{stm} : \text{StateMachineDef}, \text{P} : \text{CSPProcess}) : \text{CSPProcess} = \\
\text{buildScope}^1(\text{undefc}, \text{stm}, \text{let} c : \text{defs} \mapsto \text{name}(c) = \text{[c.initial]}^\text{expr}^8 \text{within} \text{P})
\]

where
\[
\text{defc} = \{ x : \text{consts} \mid x.\text{initial} \neq \text{null} \}
\]

\[
\text{undefc} = \{ x : \text{consts} \mid x.\text{initial} = \text{null} \}
\]

Rule 69. Build Scope

\[
\text{buildScope}(\text{cs} : \text{Seq}(\text{Variable}), \text{ctx} : \text{ConnectionNode}, \text{P} : \text{CSPProcess}) : \text{CSPProcess} = \\
\text{if} (\text{headcs}).\text{initial} == \text{NULL} \text{then} \\
\text{0} \xleftarrow{\text{set}_\text{vid}(\text{headcs}, \text{ctx})} \text{name}(\text{headcs}) \rightarrow \text{if} \#\text{cs} == 1 \text{then} \text{P} \text{else} \text{constInitSTM}^2(\text{tailcs}, \text{ctx}, \text{P}) \text{else} \\
\text{constInitSTM}^3(\text{tailcs}, \text{ctx}, \text{P})
\]
Functions related to clocks are formalised in this section.

**Rule 70. allClockVariables function**

\[
\text{allClockVariables}(wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{PVariable}
\]

\[
\text{allClockVariables}(wcs) = \\
\{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} | t \in \text{dom} wcs \land (e \rightarrow v) \in \pi_2(wcs(t)) \bullet v \}
\]

**Rule 71. clockResets function**

\[
\text{clockResets}(wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{ChannelSet}
\]

\[
\text{clockResets}(stm) = \\
\bigcup \{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} | t \in \text{dom} wcs \land (e \rightarrow v) \in \pi_2(wcs(t)) \bullet \text{alphaClockReset}^1(e) \}
\]

**Rule 72. stmClocks function**

\[
\text{stmClocks}(wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} = \\
\| (t, e, v) : \{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} | t \in \text{dom} wcs \land (e \rightarrow v) \in \pi_2(wcs(t)) \}
\]

\[
\bullet \| [t, e, v] \| \text{compileWC}^1(t, e, v)
\]

where

\[
\alpha WC(t, e, v) = \| [\text{triggerEvent}^2(t), setWC\_vid(v)] \| \text{alphaClockReset}^2(e)
\]

The function \textit{alphaClockReset} distributes over the structure of an expression, and identifies channels for a ClockExp and StateClockExp as specified in Rules 73 and 74.

**Rule 73. alphaClockReset function**

\[
\text{alphaClockReset}(e : \text{ClockExp}) : \text{ChannelSet} = \\
\text{alphaClockReset}(e) = \{ \text{clockReset}\_id(e.\text{clock}) \}
\]

**Rule 74. alphaClockReset function**

\[
\text{alphaClockReset}(e : \text{StateClockExp}) : \text{ChannelSet} = \\
\text{alphaClockReset}(e) = \{ x : \text{SIDS}\_\text{entered}\_id(x).\text{id}(e.\text{state}) \}
\]
### Rule 75. Semantics of transitions

\[
\begin{align*}
\text{if } \text{src} \in \text{State} & : \text{tClockReset}^1(t); \text{exit} \cdot \text{id} \cdot \text{id} \rightarrow \text{exitSubstates}^3(\text{src}); \text{[\text{src} \cdot \text{exit}]} \cdot \text{nops}^5 : \\
\text{else if } \text{src} \in \text{Initial} & : \text{internal} \cdot \text{id} \rightarrow \text{tClockReset}^2(t); \text{[t.action]} \cdot \text{nops}^5 : \text{compileTarget}^4(\text{tgt}, \text{src}, \text{false})^PQ \\
\text{else if } \text{src} \in \text{Junction} & : \text{internal} \cdot \text{id} \rightarrow \text{tClockReset}^3(t); \text{[t.action]} \cdot \text{nops}^5 : \text{compileTarget}^5(\text{tgt}, \text{parent}(\text{src}), \text{true})^PQ \\
\text{where} & \\
\text{src} = t \cdot \text{source} & \\
\text{tgt} = t \cdot \text{target} & 
\end{align*}
\]

### Rule 76. Clock resets for transitions

\[
\begin{align*}
\text{tClockReset}(t : \text{Transition}) : \text{CSPProcess} = \\
\text{if } \# t \cdot \text{reset} > 0 & : c : t \cdot \text{reset} \cdot \text{ClockReset} \cdot \text{id}(c \cdot \text{clock}) \rightarrow \text{Skip} \\
\text{else} & : \text{Skip}
\end{align*}
\]

### Waiting Conditions

#### Waiting Condition elicitation

The function \( wc \) is used for eliciting waiting conditions. It distributes over the structure of an expression, and collect the abstractions of comparisons involving a since or sinceEntry expression. These abstractions are pairs of consisting of a fresh variable \( b \) and a mapping from the target expressions to \( b \).

#### Waiting Condition as CSP processes

The following rules define the function \( \text{compileWC} \) which is used to define the CSP semantics of waiting conditions.
Rule 77. compileWC function

\[ \text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \]

\[ \text{compileWC}(t, \text{since}(C) >= e, v) = \]

\[ \begin{align*}
\text{let } \quad & \\
\quad & \quad \text{Reset} = \text{clockReset.id}(C) \rightarrow \text{setWC.vid}(v)!\text{false} \rightarrow \text{Monitor} \\
\quad & \quad \text{Monitor} = \\
\quad & \quad \left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{3}(t)]) \\
\triangle_{[e]} \text{setWC.vid}(v)!\text{true} \rightarrow \text{RUN}([\text{triggerEvent}^{3}(t)])
\end{array} \right) \quad \triangle \text{Reset} \\
\end{align*} \]

within \[ \text{setWC.vid}(v)!\text{false} \rightarrow \text{Monitor} \]

\[ \text{compileWC}(t, e >= \text{since}(C), v) = \]

\[ \begin{align*}
\text{let } \\
\quad & \quad \text{Reset} = \text{clockReset.id}(C) \rightarrow \text{setWC.vid}(v)!\text{true} \rightarrow \text{Monitor} \\
\quad & \quad \text{Monitor} = \\
\quad & \quad \left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{5}(t)]) \\
\triangle_{[e]} \text{setWC.vid}(v)!\text{false} \rightarrow \text{RUN}([\text{triggerEvent}^{5}(t)])
\end{array} \right) \quad \triangle \text{Reset} \\
\end{align*} \]

within \[ \text{setWC.vid}(v)!\text{true} \rightarrow \text{Monitor} \]

\[ \text{compileWC}(t, \text{sinceEntry}(S) >= e, v) = \]

\[ \begin{align*}
\text{let } \\
\quad & \quad \text{Reset} = \text{entered?x.id}(S) \rightarrow \text{setWC.vid}(v)!\text{false} \rightarrow \text{Monitor} \\
\quad & \quad \text{Monitor} = \\
\quad & \quad \left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{7}(t)]) \\
\triangle_{[e]} \text{setWC.vid}(v)!\text{true} \rightarrow \text{RUN}([\text{triggerEvent}^{7}(t)])
\end{array} \right) \quad \triangle \text{Reset} \\
\end{align*} \]

within \[ \text{setWC.vid}(v)!\text{false} \rightarrow \text{Monitor} \]

\[ \text{compileWC}(t, e >= \text{sinceEntry}(S), v) = \]

\[ \begin{align*}
\text{let } \\
\quad & \quad \text{Reset} = \text{entered?x.id}(S) \rightarrow \text{setWC.vid}(v)!\text{true} \rightarrow \text{Monitor} \\
\quad & \quad \text{Monitor} = \\
\quad & \quad \left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{9}(t)]) \\
\triangle_{[e]} \text{setWC.vid}(v)!\text{false} \rightarrow \text{RUN}([\text{triggerEvent}^{9}(t)])
\end{array} \right) \quad \triangle \text{Reset} \\
\end{align*} \]

within \[ \text{setWC.vid}(v)!\text{true} \rightarrow \text{Monitor} \]
Rule 78. compileWC function

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \\
\text{compileWC}(t, \text{since}(C) >= e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset}.\text{id}(C) \rightarrow \text{setWC}\_\text{vid}(v)\!false \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \begin{pmatrix} \\
\text{RUN}([\text{triggerEvent}^{13}(t)\}) \\
\Delta[e]_{\text{exp}^{20}} + 1 \text{setWC}\_\text{vid}(v)\!true \rightarrow \text{RUN}([\text{triggerEvent}^{14}(t)\}) \\
\end{pmatrix} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}\_\text{vid}(v)\!false \rightarrow \text{Monitor} \\
\text{compileWC}(t, e > \text{since}(C), v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset}.\text{id}(C) \rightarrow \text{setWC}\_\text{vid}(v)\!true \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \begin{pmatrix} \\
\text{RUN}([\text{triggerEvent}^{13}(t)\}) \\
\Delta[e]_{\text{exp}^{20}} + 1 \text{setWC}\_\text{vid}(v)\!true \rightarrow \text{RUN}([\text{triggerEvent}^{14}(t)\}) \\
\end{pmatrix} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}\_\text{vid}(v)\!true \rightarrow \text{Monitor} \\
\text{compileWC}(t, \text{sinceEntry}(S) > e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered}?x.\text{id}(S) \rightarrow \text{setWC}\_\text{vid}(v)\!false \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \begin{pmatrix} \\
\text{RUN}([\text{triggerEvent}^{15}(t)\}) \\
\Delta[e]_{\text{exp}^{20}} + 1 \text{setWC}\_\text{vid}(v)\!true \rightarrow \text{RUN}([\text{triggerEvent}^{16}(t)\}) \\
\end{pmatrix} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}\_\text{vid}(v)\!false \rightarrow \text{Monitor} \\
\text{compileWC}(t, e > \text{sinceEntry}(S), v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered}?x.\text{id}(S) \rightarrow \text{setWC}\_\text{vid}(v)\!true \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \begin{pmatrix} \\
\text{RUN}([\text{triggerEvent}^{17}(t)\}) \\
\Delta[e]_{\text{exp}^{20}} + 1 \text{setWC}\_\text{vid}(v)\!false \rightarrow \text{RUN}([\text{triggerEvent}^{18}(t)\}) \\
\end{pmatrix} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}\_\text{vid}(v)\!true \rightarrow \text{Monitor} \]
Rule 79. compileWC function

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \\
\text{compileWC}(t, \text{since}(C) <= e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset.id}(C) \to \text{setWC vid}(v)\!true \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{array}{c}
\text{RUN}(\text{triggerEvent}^{19}(t)) \\
\quad \Delta_{[\text{Exp}^{27}]} \text{setWC vid}(v)\!false \to \text{RUN}(\text{triggerEvent}^{20}(t)) \\
\end{array} \\
\quad \Delta_{\text{Reset}} \\
\text{within} \\
\text{setWC vid}(v)\!true \to \text{Monitor} \\
\text{compileWC}(t, e <= \text{since}(C), v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset.id}(C) \to \text{setWC vid}(v)\!false \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{array}{c}
\text{RUN}(\text{triggerEvent}^{21}(t)) \\
\quad \Delta_{[\text{Exp}^{28}]} \text{setWC vid}(v)\!true \to \text{RUN}(\text{triggerEvent}^{22}(t)) \\
\end{array} \\
\quad \Delta_{\text{Reset}} \\
\text{within} \\
\text{setWC vid}(v)\!false \to \text{Monitor} \\
\text{compileWC}(t, e <= \text{sinceEntry}(S), v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered?x.id}(S) \to \text{setWC vid}(v)\!true \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{array}{c}
\text{RUN}(\text{triggerEvent}^{23}(t)) \\
\quad \Delta_{[\text{Exp}^{29}]} \text{setWC vid}(v)\!false \to \text{RUN}(\text{triggerEvent}^{24}(t)) \\
\end{array} \\
\quad \Delta_{\text{Reset}} \\
\text{within} \\
\text{setWC vid}(v)\!false \to \text{Monitor} \\
\text{compileWC}(t, e <= \text{sinceEntry}(S), v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered?x.id}(S) \to \text{setWC vid}(v)\!false \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{array}{c}
\text{RUN}(\text{triggerEvent}^{25}(t)) \\
\quad \Delta_{[\text{Exp}^{30}]} \text{setWC vid}(v)\!true \to \text{RUN}(\text{triggerEvent}^{26}(t)) \\
\end{array} \\
\quad \Delta_{\text{Reset}} \\
\text{within} \\
\text{setWC vid}(v)\!false \to \text{Monitor} \]
**Rule 80. compileWC function**

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} =
\]

\[
\text{compileWC}(t, \text{since}(C) < e, v) =
\]

\[
\text{let}
\]

\[
\text{Reset} = \text{clockReset}.\text{id}(C) \rightarrow \text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{Monitor}
\]

\[
\text{Monitor} = \left( \begin{array}{c}
\text{RUN}((\text{triggerEvent}^{27}(t))) \\
\triangle_{\text{Exp}^{31}}(\text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{RUN}((\text{triggerEvent}^{28}(t))))
\end{array} \right) \triangle \text{Reset}
\]

\text{within}

\[
\text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{Monitor}
\]

\[
\text{compileWC}(t, e < \text{since}(C), v) =
\]

\[
\text{let}
\]

\[
\text{Reset} = \text{clockReset}.\text{id}(C) \rightarrow \text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{Monitor}
\]

\[
\text{Monitor} = \left( \begin{array}{c}
\text{RUN}((\text{triggerEvent}^{29}(t))) \\
\triangle_{\text{Exp}^{32}}(\text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{RUN}((\text{triggerEvent}^{30}(t))))
\end{array} \right) \triangle \text{Reset}
\]

\text{within}

\[
\text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{Monitor}
\]

\[
\text{compileWC}(t, \text{sinceEntry}(S) < e, v) =
\]

\[
\text{let}
\]

\[
\text{Reset} = \text{entered}\,?\,x.\text{id}(S) \rightarrow \text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{Monitor}
\]

\[
\text{Monitor} = \left( \begin{array}{c}
\text{RUN}((\text{triggerEvent}^{31}(t))) \\
\triangle_{\text{Exp}^{33}}(\text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{RUN}((\text{triggerEvent}^{32}(t))))
\end{array} \right) \triangle \text{Reset}
\]

\text{within}

\[
\text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{Monitor}
\]

\[
\text{compileWC}(t, e < \text{sinceEntry}(S), v) =
\]

\[
\text{let}
\]

\[
\text{Reset} = \text{entered}\,?\,x.\text{id}(S) \rightarrow \text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{Monitor}
\]

\[
\text{Monitor} = \left( \begin{array}{c}
\text{RUN}((\text{triggerEvent}^{33}(t))) \\
\triangle_{\text{Exp}^{34}}(\text{setWC}._{\text{id}}(v)!\text{true} \rightarrow \text{RUN}((\text{triggerEvent}^{34}(t))))
\end{array} \right) \triangle \text{Reset}
\]

\text{within}

\[
\text{setWC}._{\text{id}}(v)!\text{false} \rightarrow \text{Monitor}
\]
Chapter 4. Semantics

Rule 81. compileWC function

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \]

\[
\text{let } \text{Reset} = \text{clockReset} . \text{id}(C) \rightarrow \text{setWC}_{\text{vid}(v)} \backslash \text{false} \rightarrow \text{Monitor} \]

\[
\begin{align*}
\text{Monitor} = & \quad \text{RUN}(\text{triggerEvent}^{35}(t)) \\
& \quad + \Delta_{\text{Exp}^{35}} [v] \text{true} \rightarrow \text{RUN}(\text{triggerEvent}^{36}(t)) \\
& \quad + \Delta_{\text{Exp}^{36}} + 1 \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{RUN}(\text{triggerEvent}^{37}(t)) \\
\end{align*}
\]

within \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{Monitor}

\[
\text{compileWC}(t, \text{e} == \text{sinceEntry}(C), v) = \\
\text{let } \text{Reset} = \text{clockReset} . \text{id}(C) \rightarrow \text{setWC}_{\text{vid}(v)} \backslash \text{false} \rightarrow \text{Monitor} \]

\[
\begin{align*}
\text{Monitor} = & \quad \text{RUN}(\text{triggerEvent}^{38}(t)) \\
& \quad + \Delta_{\text{Exp}^{38}} [v] \text{true} \rightarrow \text{RUN}(\text{triggerEvent}^{39}(t)) \\
& \quad + \Delta_{\text{Exp}^{39}} + 1 \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{RUN}(\text{triggerEvent}^{40}(t)) \\
\end{align*}
\]

within \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{Monitor}

\[
\text{compileWC}(t, \text{e} == \text{sinceEntry}(S), v) = \\
\text{let } \text{Reset} = \text{entry} ? \text{x} . \text{id}(S) \rightarrow \text{setWC}_{\text{vid}(v)} \backslash \text{false} \rightarrow \text{Monitor} \]

\[
\begin{align*}
\text{Monitor} = & \quad \text{RUN}(\text{triggerEvent}^{41}(t)) \\
& \quad + \Delta_{\text{Exp}^{41}} [v] \text{true} \rightarrow \text{RUN}(\text{triggerEvent}^{42}(t)) \\
& \quad + \Delta_{\text{Exp}^{42}} + 1 \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{RUN}(\text{triggerEvent}^{43}(t)) \\
\end{align*}
\]

within \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{Monitor}

\[
\text{compileWC}(t, \text{e} == \text{entry}(S), v) = \\
\text{let } \text{Reset} = \text{entry} ? \text{x} . \text{id}(S) \rightarrow \text{setWC}_{\text{vid}(v)} \backslash \text{false} \rightarrow \text{Monitor} \]

\[
\begin{align*}
\text{Monitor} = & \quad \text{RUN}(\text{triggerEvent}^{44}(t)) \\
& \quad + \Delta_{\text{Exp}^{44}} [v] \text{true} \rightarrow \text{RUN}(\text{triggerEvent}^{45}(t)) \\
& \quad + \Delta_{\text{Exp}^{45}} + 1 \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{RUN}(\text{triggerEvent}^{46}(t)) \\
\end{align*}
\]

within \text{setWC}_{\text{vid}(v)} \text{false} \rightarrow \text{Monitor}
4.2 Detailed Semantics: Timed Language

**Rule 83.** `triggerEvent` function

\[
\text{triggerEvent}(t : \text{Transition}) : \text{CSPEvent} = \\
\begin{cases} 
\text{if } t.\text{trigger} \neq \text{null} & \text{triggerEvent}(t.\text{trigger.id}(t)) \\
\text{else} & \text{internal.id}(t)
\end{cases}
\]

**Transition deadline events**

**Rule 84.** `deadlineEvents` function

\[
\text{deadlineEvents}(s : \text{StateMachineDef}) : \text{ChannelSet} = \\
\text{deadlineEvents}(stm) = \{ t : \text{allTransitions}(s) \mid t.\text{deadline} \neq \text{null} \Rightarrow \text{deadline.id}(t) \}
\]

**Memory**

**Rule 85.** State-machine Memory

\[
\text{stmMemory}(stm : \text{StateMachineDef}, wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} = \\
\begin{align*}
\text{let } & \text{Memory}(\text{vars}) \equiv \\
& \begin{cases} 
\forall v : \text{lvars} \Rightarrow (g_{\text{vid}}(v) \text{name}(v) \rightarrow \text{Memory}(\text{vars}) \\
\quad (s_{\text{vid}}(v) ? x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]))) \\
\end{cases} \\
& \begin{cases} 
\forall v : \text{rvars} \Rightarrow (g_{\text{vid}}(v) \text{name}(v) \rightarrow \text{Memory}(\text{vars}) \\
\quad (s_{\text{vid}}(v) ? x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x])) \\
\quad (s_{\text{Ext}}(v) ? x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]))
\end{cases} \\
& \begin{cases} 
\forall v : \text{allConstants}(\text{stm}) \Rightarrow (g_{\text{vid}}(v) \text{name}(v) \rightarrow \text{Memory}(\text{vars})) \\
\end{cases} \\
& \begin{cases} 
\forall r : \text{allTransitions}(\text{stm}) \Rightarrow (\text{memoryTransition}(t, wcs) ; \text{Memory}(\text{vars}))
\end{cases} \\
& \begin{cases} 
\forall v : \text{cvars} \Rightarrow (s_{\text{WC}}(v) ? x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]))
\end{cases} \\
& \begin{cases} 
\forall r : \text{allDeadlineTransitions}(\text{stm}) \Rightarrow (\text{memoryDeadline}(t, wcs) ; \text{Memory}(\text{vars}))
\end{cases}
\end{align*}
\]
Chapter 4. Semantics

**Rule 86.** allDeadlineTransitions function

\[
\text{allDeadlineTransitions}(s : \text{StateMachineDef}) : \mathbb{P}\text{Transition} = \\
\{ t : \text{allTransitions}^{[0]}(s) \mid t.\text{deadline} \neq \text{null} \}
\]

**Rule 87.** memoryTransition function

\[
\text{memoryTransition}(t : \text{Transition}, wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} \\
\text{if} \ (t.\text{condition} \neq \text{null}) \text{ then} \\
\left( [\pi_1 (\text{wcs}(t))]_{\text{Expr}}^{\text{Exp}} \right) \& \text{triggerForMemory}^1(t.\text{trigger}.\text{id}(t)) \\
\text{else} \\
\text{triggerForMemory}^2(t.\text{trigger}.\text{id}(t))
\]

**Rule 88.** Memory deadline

\[
\text{memoryDeadline}(t : \text{Transition}, wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} \\
\text{if} \ (t.\text{condition} \neq \text{null}) \text{ then} \\
\left( [\pi_1 (\text{wcs}(t))]_{\text{Expr}}^{\text{Exp}} \right) \& \text{deadline}.\text{id}(t).\text{on} \rightarrow \text{Skip} \\
\text{else} \\
\left( \neg [\pi_1 (\text{wcs}(t))]_{\text{Expr}}^{\text{Exp}} \right) \& \text{deadline}.\text{id}(t).\text{off} \rightarrow \text{Skip}
\]

4.2.2 States

The semantics of states is largely unchanged when compared to the untimed semantics, except that we do not hide flowtrigevts so as to be able to give semantics to sinceEntry(s), and there is an interleaving with the semantics of during action to give semantics to trigger deadlines.

**Rule 89.** Semantics of states

\[
[[s : \text{State}]]_{\text{s}} : \text{TimedCSPProcess} = \\
\text{This function is split in multiple rules according to the type of states.}
\]
4.2 Detailed Semantics: Timed Language

Rule 90. Semantics of simple states

\[ [s: \text{State}]_{\text{ops}} : \text{TimedCSPProcess} = \]

\[
\begin{align*}
\text{let} \\
\text{Inactive} & \overset{\text{enter}\, x : \text{sids}, \text{id}(s) \to \text{Activating}(x)}{=} \\
\text{Activating}(o) & \overset{[s, \text{entry}]_{\text{ops}}}{=} \text{initialisation}^4(s) : \text{entered}\, o, \text{id}(s) \to \\
(\text{[s, during]}_{\text{ops}}) & \overset{\text{Action}^\triangleleft}{=} \text{Stop} \mid \text{triggerDeadlines}^1(s) \Delta \\
& \quad \big( \begin{array}{l}
\text{\begin{cases}
\text{t} : \text{transitionsFrom}^1(s) \cdot [t, s, \text{false}]_{\text{Inactive, Activating, ops}}^\top \\
\text{e} : \text{Event} & \text{if} (e, \text{type} = \text{null}) \\
\quad \text{then} \text{eventId}(e) ? x : \text{tids} \to \text{exit} : \text{Inactive} \\
\quad \text{else} \text{eventId}(e) ? x : \text{tids} ? y \to \text{exit} : \text{Inactive}
\end{cases} \big)
\end{array} \\
\text{for each} \; j : \text{reachableJunctions}^5(s) \cdot \text{id}(j) = [j, s, \text{false}]_{\text{Inactive, Activating, ops}}^\top \\
\text{within} \; \text{Inactive} \]
\end{align*}
\]

where

\[
\begin{align*}
& \#\text{states}^4(s) = 0 \\
& \text{sids} = \text{SIDS} \setminus \{ \text{id}(s) \} \land \text{tids} = \text{TIDS} \setminus \text{tIDS}(s) \\
& \text{exit} = \text{exit}\, x : \text{sids}, \text{id}(s) \to \text{exitSubstates}^4(s) ; [s, \text{exit}]_{\text{ops}}^\top : \text{exited}\, x, \text{id}(s) \to \text{Skip}
\end{align*}
\]

Rule 91. Semantics of composite states

\[ [s: \text{State}]_{\text{ops}} : \text{TimedCSPProcess} = \]

\[
\begin{align*}
\text{let} \\
\text{Inactive} & \overset{\text{enter}\, x : \text{sids}, \text{id}(s) \to \text{Activating}(x)}{=} \\
\text{Activating}(o) & \overset{[s, \text{entry}]_{\text{ops}}}{=} \text{initialisation}^4(s) : \text{entered}\, o, \text{id}(s) \to \\
(\text{[s, during]}_{\text{ops}}) & \overset{\text{Action}^\triangleleft}{=} \text{Stop} \mid \text{triggerDeadlines}^1(s) \Delta \\
& \quad \big( \begin{array}{l}
\text{\begin{cases}
\text{t} : \text{transitionsFrom}^1(s) \cdot [t, s, \text{false}]_{\text{Inactive, Activating, ops}}^\top \\
\text{e} : \text{Event} & \text{if} (e, \text{type} = \text{null}) \\
\quad \text{then} \text{eventId}(e) ? x : \text{tids} \to \text{exit} : \text{Inactive} \\
\quad \text{else} \text{eventId}(e) ? x : \text{tids} ? y \to \text{exit} : \text{Inactive}
\end{cases} \big)
\end{array} \\
\text{for each} \; j : \text{reachableJunctions}^5(s) \cdot \text{id}(j) = [j, s, \text{false}]_{\text{Inactive, Activating, ops}}^\top \\
\text{within} \; \text{Inactive} \times \text{composeStates}^1((x : \text{states}^1(s)), s)_{\text{ops}}^\top \\
\end{align*}
\]

where

\[
\begin{align*}
& \#\text{states}^4(s) > 0 \\
& \text{flowtrigevts} = \text{flowTriggerEvents}^2(s) \\
& \text{sids} = \text{SIDS} \setminus \{ \text{id}(s) \} \land \text{tids} = \text{TIDS} \setminus \text{tIDS}(s) \\
& \text{exit} = \text{exit}\, x : \text{sids}, \text{id}(s) \to \text{exitSubstates}^4(s) ; [s, \text{exit}]_{\text{ops}}^\top : \text{exited}\, x, \text{id}(s) \to \text{Skip}
\end{align*}
\]
Chapter 4. Semantics

Rule 92. Semantics of trigger deadlines
\[
\text{triggerDeadlines}(s : \text{State}) : \text{TimedCSPProcess} = \\
\begin{align*}
\text{let} \\
\text{Deadline}(t) \equiv & \left( \begin{array}{c}
\text{deadline}.\text{id}(t).\text{on} \rightarrow \\
\text{readState}^3 \left( (\text{usedVariables}^0(t.\text{deadline}), \right. \\
(\text{deadline}.\text{id}(t).\text{off} \rightarrow \text{Skip}) \bigoplus [t.\text{deadline}]_{\text{Exp}^t} \Big) \\
\end{array} \right) \\
\text{within} \\
|| t : tDS \iff \text{Deadline}(t) \\
\text{where} \\
tDS = \{ t : \text{transitionsFrom}^9(s) \mid t.\text{deadline} \neq \text{null} \}
\end{align*}
\]

The composition of states is also largely unchanged when compared to the untimed Rule 31 except that the set \(\text{shflowevts}\) is not hidden, so as to allow a parent to observe all of its children’s flow events, and the state-machine to observe \(\text{entered}\) events required to reset an implicit clock in the case of sinceEntry(s).

Rule 93. Composition of states
\[
\text{composeStates}(ss : \text{seq}\text{State}, p : \text{NodeContainer}) : \text{TimedCSPProcess} = \\
\begin{align*}
\text{if} \# ss = 1 \text{ then} \\
\quad \text{restrictedState}^3(p, \text{headss})^\text{nops} \\
\text{else} \\
\quad \left( \begin{array}{c}
\text{restrictedState}^4(p, \text{headss})^\text{nops} \\
[\text{shflowevts}] \\
\text{composeStates}^2(\text{tailss}, p)^\text{nops}
\end{array} \right) \\
\text{where} \\
\text{shflowevts} = \text{flowEvents}^3(\text{headss}, p) \cap \bigcup \{ x : \text{tailss} \iff \text{flowEvents}^4(x, p) \}
\end{align*}
\]

4.2.3 Timed statements

Rule 94. Semantics of statements
\[
[s : \text{Statement}]^\text{nops}_{\text{Statement}} : \text{TimedCSPProcess} = \\
\text{This rule is split in multiple rules according to the subtype of the statement.}
\]

Rule 95. Semantics of statement deadlines
\[
[s : \text{TimedStatement}]^\text{nops}_{\text{Statement}} : \text{TimedCSPProcess} = \\
\begin{align*}
\quad [s.\text{stmt}]^\text{nops}_{\text{Statement}} \bigoplus [s.\text{deadline}]_{\text{Exp}^t}
\end{align*}
\]
4.2 Detailed Semantics: Timed Language

**Rule 96. Semantics of wait statement**
\[
[s : \text{Wait}]_{\text{Statement}}^{\text{nops}} : \text{TimedCSPProcess} = \\
\left[\left[\text{s.duration}\right]_{\text{Expr}}\right]_{\text{Wait}}
\]

**Rule 97. Semantics of Wait**
\[
[e : \text{Expression}]_{\text{Wait}} : \text{TimedCSPProcess} = \\
\text{This rule is split in multiple rules according to the subtype of the expression.}
\]

**Rule 98. Semantics of Wait**
\[
[e : \text{P} \cdot \text{f}]_{\text{Wait}} : \text{TimedCSPProcess} = \\
\bigcap n : @ \cdot \text{Wait}(n)
\]

**Rule 99. Semantics of Wait**
\[
[e : \text{A}]_{\text{Wait}} : \text{TimedCSPProcess} = \\
\text{Wait(e)}
\]

**Rule 100. Semantics of clock reset**
\[
[s : \text{ClockReset}]_{\text{nops}}^{\text{Statement}} : \text{TimedCSPProcess} = \\
clockReset.id(s.clock) \rightarrow \text{Skip}
\]

**Rule 101. Semantics of assignment**
\[
[s : \text{Assignment}]_{\text{nops}}^{\text{Statement}} : \text{CSPProcess} = \\
(set_{\text{vid}}(s.left)! [s.right]_{\text{Expr}}_{\text{STM}})_{\text{Expr}} \rightarrow \text{Skip} \uparrow 0
\]

where
\[
\text{op} = s.\text{operation} \\
\text{opdef} = \text{findOperationDefinition} (\text{op}, \text{ops}) \\
\text{SkipAnytime} = (\text{Wait}(1); \text{SkipAnytime}) \cap \text{Skip} \\
\text{body} = \begin{cases} \\
\text{if (opdef = null)} \text{then} \\
\text{else} \\
\left[\text{opdef}]_{\text{nops}}^{\text{STM}} \left(\{x : s.\text{args} \times [x]_{\text{Expr}}_{\text{STM}}\}\right)\right)
\end{cases}
\]

**Rule 102. Semantics of call statement**
\[
[s : \text{Call}]_{\text{nops}}^{\text{Statement}} : \text{CSPProcess} = \\
\text{(op.name Call} \rightarrow \text{Skip} \uparrow 0); \text{body}; (\text{op.name Ret} \rightarrow \text{Skip}) \uparrow 0
\]
This chapter describes and defines the probabilistic semantics of RoboChart. We give an overview of probabilistic semantics in Section 5.1, and present its UTP semantics in Section 5.2. Then in Section 5.3, we define a translation from RoboChart to PRISM [18] (the language supported by the probabilistic model checker PRISM) in order to use PRISM to analyse RoboChart models.
5.1 Overview

The long-term plan for RoboChart is to support automated verification by both model checking and theorem proving. The RoboChart semantics for its core language and its timed language presented in Chapter 4 is formalised using CSP and its dialect, tock-CSP [8]. The semantic underpinning that is UTP, but our use of CSP is primarily as a front end to enable model checking with FDR.

The semantics for the probabilistic language of RoboChart is also based on UTP, and so RoboChart has a unified semantic base. Ultimately, we aim to support its verification using the theorem prover Isabelle/UTP [26], a mechanisation of UTP in the Isabelle/HOL theorem prover [31].

To enable model checking of the probabilistic behaviour of RoboChart, we translate RoboChart models to PRISM, and then use PRISM to analyse the generated models. The translation is formalised in Section 5.3. We present the metamodel of PRISM and a set of rules to translate RoboChart models.
5.2 Semantic Domain

The probabilistic denotational semantics for RoboChart [32] is based on the weakest completion semantics [13] in UTP. The general idea is to use a forgetful function to embed standard designs into probabilistic designs. This embedding preserves program structure of a probabilistic programming language which includes abort, skip, assignment, conditional, nondeterministic choice, probabilistic choice, sequence, and recursion constructors. We are mechanising the UTP theory for probabilistic designs in Isabelle/UTP. The next step is to lift probabilistic designs into UTP’s reactive design to develop a theory of reactive probabilistic designs.
5.3 Translation to PRISM

We structure the translation from RoboChart models to PRISM models in two steps: normalisation of the RoboChart model and transformation of normalised RoboChart models to PRISM. In Section 5.3.2, we define our probabilistic normal form, namely a restricted version of RoboChart models in which all transitions between two states are probabilistic. We then define the normalisation of a RoboChart model in Section 5.3.5, and then the translation from a RoboChart model in normal form to a PRISM model in Section 5.3.6 by a set of rules. We present the notions used in rules in Section 5.3.3 and the PRISM metamodel in Section 5.3.4.

Not all features of RoboChart are currently supported. Section 5.3.1 presents our assumptions.

5.3.1 Translation requirements

We list the conditions that need to be satisfied by a RoboChart model for our technique to be applicable.

TR-TP1 *Time primitives are not used.* In DTMCs and MDPs, transitions occur in discrete-time steps. Every transition takes one unit of time. Time primitives [22] in RoboChart, however, are capable of capturing budgets and deadlines using clocks and constructs like `wait(n)`, which defines a waiting period of \( n \) units of time, and `read?x<\(n\)`, in which reading a value \( x \) through an event `read` has a deadline to take place within 2 units of time. DTMCs and MDPs do not intrinsically support clocks, time budgets and deadlines to model the corresponding RoboChart constructs.

In addition to DTMCs and MDPs, the PRISM notation also supports probabilistic timed automata (PTAs) [16, 20], which extend MDPs with the ability to model real-time behaviour through real-valued clocks [19]. The timed semantics of RoboChart, however, are based on time units [22], instead of real-valued time. The default verification method (quantitative abstraction refinement [17]) for PTAs in PRISM, therefore, cannot be used. Instead, we can use the digital clocks [21] method which uses an *integral* time model. We will extend our transformation to support PTAs in order for time primitives in RoboChart, which is part of our future work.

TR-CN1 *Connections between controllers are not asynchronous.*

TR-OP1 *Operations cannot be defined by state machines.*

TR-ST1 *States cannot have during actions.*

To cater for these constructs, the PRISM model needs to include extra modules to deal with shared variables, buffers for asynchronous communication, operation calls, and interruptions of during actions by outgoing transitions. Dealing with these extra constructs is part of our agenda for future work.
TR-TY1 Only primitive types and enumerations, sequences of these types, or sequences of sequences of these types are used.

TR-EX1 Quantification, lambda, and definite description expressions cannot be used since the PRISM notation is concrete.

PRISM supports only integer, boolean, and real numbers. Refinement techniques [2, 15] are a possible solution to deal with abstract data types and constructs in RoboChart. Support of more abstract data types and expressions is part of our plans for future work.

5.3.2 Normal form

A normalised RoboChart model satisfies the following extra conditions on transitions and junctions.

NFM-1 A state has at least one outgoing transition.
NFM-2 A transition that is from a state or a normal junction to a normal junction has an action.
NFM-3 A transition can have a trigger or an action, but not both together.

5.3.3 Notions for rules

See Table 5.1.

5.3.4 PRISM

Metamodel

See Figure 5.1 to 5.8.
Chapter 5. Probabilistic Semantics

Figure 5.2: Metamodel of PRISM types

Figure 5.3: Metamodel of PRISM expressions

Figure 5.4: Metamodel of PRISM boolean expressions
Figure 5.5: Metamodel of PRISM unary and manyary expressions

Figure 5.6: Metamodel of PRISM binary expressions

Figure 5.7: Metamodel of PRISM modules
Table 5.1: Summary of construct syntax in rules

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Junction</td>
<td>The class name from the metamodel also represents a collection of objects that have the Junction type. Similarly, State, ProbabilisticJunction, Controller, and so on. Particularly, ProbJunc is an abbreviation for ProbabilisticJunction for a compact space.</td>
</tr>
<tr>
<td>Class&lt;sub&gt;ic&lt;/sub&gt;</td>
<td>Subscripts indicate classes of different metamodels, where rc stands for RoboChart, and pr for PRISM. For example, Transition&lt;sub&gt;rc&lt;/sub&gt; and ModuleDef&lt;sub&gt;pr&lt;/sub&gt; denote the Transition in RoboChart, and the ModuleDef in PRISM.</td>
</tr>
<tr>
<td>P X</td>
<td>Power set of X.</td>
</tr>
<tr>
<td>X × Y</td>
<td>Cartesian product.</td>
</tr>
<tr>
<td>seq X</td>
<td>Finite sequences.</td>
</tr>
<tr>
<td>{ x : T</td>
<td>P • e(x) }</td>
</tr>
<tr>
<td>{ x : T</td>
<td>P }</td>
</tr>
<tr>
<td>{ x : T • e }</td>
<td>Defined as { x : T</td>
</tr>
<tr>
<td>(μ x : T</td>
<td>P)</td>
</tr>
<tr>
<td>[{ x : T • e(x) }]</td>
<td>Generalised union, where e(x) is a set expression.</td>
</tr>
<tr>
<td>p.n</td>
<td>Selection of nth element from the tuple p.</td>
</tr>
<tr>
<td>obj.v</td>
<td>Selection of value of component v from object obj.</td>
</tr>
<tr>
<td>[f₁→v₁,···]ₖ</td>
<td>Record where fᵢ is a component name and vᵢ is the value associated to that component. It represents an object of the class C with its components instantiated. If a component is not specified, it is set to null or 0 (if it is a set) by default.</td>
</tr>
<tr>
<td>R{ S }</td>
<td>Relational image of a relation R under a set S.</td>
</tr>
<tr>
<td>[[ i : T ]]ₓ</td>
<td>Transformation of i to PRISM in the context X. For example, [[···]]₆ denotes the module context.</td>
</tr>
<tr>
<td>uname(par,m)</td>
<td>This function constructs a fresh unique identifier for a new element from the supplied construct par (of type NamedElement) and the name m (a string) of the element.</td>
</tr>
<tr>
<td>uname(par,m,n)</td>
<td>Similar to uname(par,m), but this function has three parameters where both m and n are of type string.</td>
</tr>
<tr>
<td>id(n)</td>
<td>This function defines a unique identifier for an existing construct n (of type NamedElement). If n is null, it gives an empty name. One possible implementation is to use qualified names.</td>
</tr>
<tr>
<td>fname(p₁ : T₁,···) : pₙ : Tₙ → Tᵣ</td>
<td>This is a function declaration with the last parameter pₙ of type Tₙ and the result of type Tᵣ. It is commonly used to create a curried version of the function, an application of the function having all arguments except the last one.</td>
</tr>
</tbody>
</table>

5.3.5 Normalisation rules
Rule 103. Normalisation of state machines

\[
\text{stm} : \text{StateMachineDef} \mid \text{STM} : \text{StateMachineDef} = \\
\{ \text{name} \mapsto \text{stm.name}, \text{variableList} \mapsto \text{stm.variableList}, \\
\text{operations} \mapsto \text{stm.operations}, \text{events} \mapsto \text{stm.events}, \\
\text{pInterfaces} \mapsto \text{stm.pInterfaces}, \text{rInterfaces} \mapsto \text{stm.rInterfaces}, \\
\text{interfaces} \mapsto \text{stm.interfaces}, \text{transitions} \mapsto \text{transnodes.1}, \\
\text{nodes} \mapsto \text{transnodes.2} \}
\]

where

\[
\text{transnodes} = \{ \text{stm} \} \]

Rule 104. Normalisation of states

\[
\text{s} : \text{State} \mid \text{S} : \text{State} = \\
\{ \text{name} \mapsto \text{s.name}, \text{actions} \mapsto \text{s.actions}, \text{transitions} \mapsto \text{transnodes.1}, \text{nodes} \mapsto \text{transnodes.2} \}
\]

where

\[
\text{transnodes} = \{ \text{s} \} \]

Rule 105. Normalisation of NodeContainer

\[
\text{s} : \text{NodeContainer} \mid \text{CS} : P\text{Transition} \times P\text{Node} = \\
\{ \text{tp : transpjunc} \cdot \text{tp.1}, (\text{loopstr.1} \cup (\text{tp : transpjunc} \cdot \text{tp.2})) \cup \text{compstates} \cup \text{intactnodes} \}
\]

where

\[
\text{loopstr} = \text{addLoopStateTrans}^1 (\text{s})
\]

\[
\text{junctrans} = \text{combTransJunctions}^1 \left( \text{inTransCombinableJuncs}^1 (\text{s}), \cup \left( \text{transitionsOf}^1 (\text{s.transitions}) \right) \cup \text{inTransCombinableJuncs}^2 (\text{s}) \right)
\]

\[
\text{transpjunc} = \{ \text{tp : s.transitions} \cup \text{junctrans.3} \cup \text{loopstr.2} \setminus \text{junctrans.2} \cdot \text{splitTran}^1 (\text{t}) \}
\]

\[
\text{compstates} = \{ \text{n : s.nodes} \mid \text{n} \in \text{State} \land \#(\text{n.nodes}) > 0 \cdot \{ \text{n} \} \}
\]

\[
\text{intactnodes} = \{ \text{n : s.nodes} \mid \text{n} \notin \text{junctrans.1} \lor (\text{n} \in \text{State} \land \#(\text{n.nodes}) > 0) \}
\]
Rule 106. Normalisation of states without outgoing transitions

\[
\text{addLoopStateTrans}(\text{stm} : \text{NodeContainer}) : \mathbb{P}\text{State} \times \mathbb{P}\text{Transition} = \\
\begin{cases} \\
\text{if } \exists n : \text{stm} \cdot \text{nodes} \cdot n \in \text{State} \land (\forall t : \text{stm} \cdot \text{transitions} \cdot t \cdot \text{source} \neq n) \text{ then} \\
\quad \{\text{loopstate}\}, \{\text{looptran}\} \cup \text{newtrans} \\
\text{else} \\
\quad (\emptyset, \emptyset) \\
\end{cases}
\]

where

\[
\text{loopstate} = \text{newState}^1(\text{name}((\text{stm} \cdot \text{"loop"}))) \\
\text{looptran} = \text{newTransition}^1(\text{name}((\text{stm} \cdot \text{"loop_self"})), \text{loopstate}, \text{loopstate}) \\
\text{newtrans} = \bigcup \left\{ n : \text{stm} \cdot \text{nodes} \mid n \in \text{State} \land (\forall t : \text{stm} \cdot \text{transitions} \cdot t \cdot \text{source} \neq n) \right\} \\
\]
5.3 Translation to PRISM

**Rule 111. Combination of transitions of a junction**

\[
\text{combTransJunction}(j : \text{Junction}, \text{trans} : \mathbb{P}\text{Transition}) : \mathbb{P}\text{Junction} \times \mathbb{P}\text{Transition} \times \mathbb{P}\text{Transition} =
\]

\[
\begin{cases}
\text{if intransother} \neq \emptyset \text{ then } (\emptyset, \text{intranscomb}, \text{newtrans}) \text{ else } ((j), \text{trans}, \text{newtrans})
\end{cases}
\]

**where**

\[
\text{intranscomb} = \{ t : \text{trans} \mid t.\text{target} = j \land \text{isCombinableTran}^2(t) \}
\]

\[
\text{intransother} = \{ t : \text{trans} \mid t.\text{target} = j \} \setminus \text{intranscomb}
\]

\[
\text{outtrans} = \{ t : \text{trans} \mid t.\text{source} = j \}
\]

\[
\text{newtrans} = \begin{cases}
\text{if intranscomb} \neq \emptyset \\
\{ t_i : \text{intranscomb}; t_j : \text{outtrans} \mid \text{name} \mapsto \text{uname}(t_i, t_j.\text{name}), \text{source} \mapsto t_i.\text{source}, \text{target} \mapsto t_j.\text{target}, \text{trigger} \mapsto t_i.\text{trigger}, \text{condition} \mapsto t_i.\text{condition} \land t_j.\text{condition}, \text{action} \mapsto t_j.\text{action} \} & \text{Transition}
\end{cases}
\]

**Rule 112. Transition split**

\[
\text{splitTran}(t : \text{Transition}) : \mathbb{P}\text{Transition} \times \mathbb{P}\text{ProbJunc} =
\]

\[
\begin{cases}
\text{if } t.\text{trigger} \neq \text{null} \land t.\text{action} \neq \text{null} \text{ then } \\
\begin{cases}
\text{name} \mapsto \text{uname}(t, \text{"sp_1"}), \text{source} \mapsto t.\text{source}, \text{target} \mapsto \text{pj}, \\
\text{trigger} \mapsto t.\text{trigger}, \text{condition} \mapsto t.\text{condition}
\end{cases} & \begin{cases}
\text{name} \mapsto \text{uname}(t, \text{"sp_2"}), \text{source} \mapsto \text{pj}, \text{target} \mapsto t.\text{target}, \\
\text{probability} \mapsto 1.0, \text{action} \mapsto t.\text{action}
\end{cases}
\end{cases}
\]

\[
\text{else } ((t), \emptyset)
\]

**Rule 113. New State**

\[
\text{newState}(n : \text{string}) : \text{State} =
\]

\[
\begin{cases}
\text{name} \mapsto n, \text{nodes} \mapsto \emptyset, \text{transitions} \mapsto \emptyset, \text{actions} \mapsto \emptyset
\end{cases}
\]

**Rule 114. New Transition**

\[
\text{newTransition}(n : \text{string}, \text{source} : \text{Node}, \text{target} : \text{Node}) : \text{Transition} =
\]

\[
\begin{cases}
\text{name} \mapsto n, \text{source} \mapsto \text{source}, \text{target} \mapsto \text{target}, \text{trigger} \mapsto \text{null}, \\
\text{condition} \mapsto \text{null}, \text{probability} \mapsto \text{null}, \text{action} \mapsto \text{null}
\end{cases}
\]

5.3.6 Normal form to PRISM

Module
**Rule 115. Module**

\[
[M : Module], [M] : Model_{pr} = \left[\begin{array}{l}
\text{constants} \rightarrow (\{ r : ctrlrets \cdot r.1 \} \cup rprefs.1 \cup \text{exitconsts}) \\
\text{globals} \rightarrow (\{ r : ctrlrets \cdot r.2 \} \cup rprefs.2) \\
\text{modules} \rightarrow (\{ r : ctrlrets \cdot r.3 \} \cup \{ rprefs.3 \})
\end{array}\right]_{Model_{pr}}
\]

where
\[
\text{ctrlrets} = \left[\begin{array}{l}
\{ n : m.nodes | n \in \text{Controller} \}
\end{array}\right]^{C} C_{1}(
\{ \text{getVariableLists}^{1}(rp) \})
\]
\[
\text{rpoutevents} = \{ c : m.connections | c/from \in \text{RoboticPlatform} \land c/efrom/\text{type} \neq \text{null} \land c/efrom \}
\]
\[
\text{rprefs} = [\left\{ m/n : m.nodes | n \in \text{RoboticPlatform} \right\}]^{R}
\]
\[
\text{exitconsts} = \text{exitSeqCtrlConsts}() \]

**Rule 116. Exit sequence control constants**

\[
\text{exitSeqCtrlConsts}() : \mathbb{P} \text{ Constant} = \left\{ \text{inv, act}_p, \text{act}_t, \text{exited}, \text{sub}_\text{act}, \text{sub}_\text{waiting}, \text{sub}_\text{exited} \right\}
\]

where
\[
\text{inv} = \text{const int Exit\_NONE} = 0;
\]
\[
\text{act}_p = \text{const int Exit\_ACT\_Parent} = 1;
\]
\[
\text{act}_t = \text{const int Exit\_ACT\_Trans} = 2;
\]
\[
\text{exited} = \text{const int Exit\_EXITED} = 3;
\]
\[
\text{sub}_\text{act} = \text{const int Exit\_Sub\_ACT} = 4;
\]
\[
\text{sub}_\text{waiting} = \text{const int Exit\_Sub\_ACT\_Waiting} = 5;
\]
\[
\text{sub}_\text{exited} = \text{const int Exit\_Sub\_EXITED} = 6;
\]

**Robotic platforms**

**Rule 117. Robotic platforms**

\[
[\{ rp : \text{RoboticPlatformDef}, \text{outevents} : \mathbb{P} \text{ Event}, \text{opmaps} : \text{OperationSig} \leftrightarrow \text{Action} \}]_{R} : \mathbb{P} \text{ Constant} \times \mathbb{P} \text{ VarDecl} \times \text{Module} =
\]
\[
(\{ r : \text{constvars} \cdot r.1 \}, \{ r : \text{constvars} \cdot r.2 \}, \text{module})
\]

where
\[
\text{constvars} = [\left\{ \text{getVariableLists}^{1}(rp) \right\}]^{V}_{2}
\]
\[
\text{localvars} = \left\{ e : \text{outevents} \cdot \text{EVT}\_\text{id}(e) : \{ e/\text{type} \} \right\}
\]
\[
\text{eventcmds} = \left\{ \begin{array}{l}
\text{e} : \text{getEvents}^{1}(rp) * \\
\text{if} \ e/\text{outevents} \text{ then} \{ \text{v} : \{ e/\text{type} \} \cdot [\text{id}(e) \leftrightarrow \text{true} \rightarrow \text{EVT}\_\text{id}(e)' = \text{v}; \} \\
\text{else} \{ \text{id}(e) \leftrightarrow \text{true} \}
\end{array} \right\}
\]
\[
\text{opcmds} = \{ \text{act} : \text{run opmaps} \cdot \{ \text{act/name} \leftrightarrow \text{true} \leftrightarrow \text{true} \} \]
\[
\text{module} = \text{module/id}(rp), \text{vars} \rightarrow \text{localvars}, \text{commands} \rightarrow (\bigcup \text{eventcmds}) \cup \text{opcmds} \text{ endmodule}
\]

**Controllers**
5.3 Translation to PRISM

Rule 118. Controllers

\[
[\text{ctrl} : \text{ControllerDef}]_C : \text{P Constant} \times \text{P VarDecl} \times \text{P Module} \times (\text{OperationSig} \leftrightarrow \text{Action}) = \\
(\bigcup \{ r : \text{constvars} \cdot r \cdot 1 \} \cup (\bigcup \{ r : \text{stmrets} \cdot r \cdot 1 \})) \cup (\{ r : \text{constvars} \cdot r \cdot 1 \}, \{ r : \text{stmrets} \cdot r \cdot 2 \} \cdot \text{rops})
\]

where

\[
\begin{align*}
\text{constvars} &= [\ldots] \{ \text{getVariableLists}^2(\text{ctrl}) \} \\
\text{stmrets} &= \left\{ \begin{array}{l}
\text{let} \quad \text{stmoutevents} = \{ c : \text{ctrl.connections} \cup (\text{stm, stmoutevents}) \\
\quad \text{c.from} = \text{stm} \land c.efrom \neq \text{null} \}
\end{array} \right\}
\]

\[
\text{rops} = \bigcup \{ \text{stm} : \text{ctrl.machines} \cdot \text{opToActionMaps}^1(\text{stm}) \}
\]

Rule 119. Mapping from operations required by state machines to PRISM actions

\[
\text{opToActionMaps}(\text{stm} : \text{StateMachineDef}) : \text{OperationSig} \leftrightarrow \text{Action} = \\
\{ \text{op} : \text{getRequiredOperations}^3(\text{stm}) \cdot \text{op} \rightarrow \text{name} \cdot \text{name}(\text{stm, op.name}) \}_\text{Action}_V
\]

State machines

Rule 120. State machines

\[
[\text{stm} : \text{StateMachineDef, outevents} : \text{P Event}]_{\text{STM}} : \text{P Constant} \times \text{Module} = \\
(\bigcup \{ r : \text{constvars} \cdot r \cdot 1 \}) ; \\
\text{const int} \text{id}(\text{stm}) \cdot \text{LOCK}_F \cdot \text{FREE} = 0; \\
\text{stmret} \cdot 2; \\
\text{module} \text{id}(\text{stm}) \\
\quad \text{id}(\text{stm}) \cdot \text{LOCK} : [0..(\text{stmret} \cdot 1 - 1)] = \text{id}(\text{stm}) \cdot \text{LOCK}_F \cdot \text{FREE}; \\
(\bigcup \{ r : \text{constvars} \cdot r \cdot 2 \}) ; \\
\text{outeventvars}; \\
\text{stmret} \cdot 3; \\
\text{ropsvars}; \\
\text{stmret} \cdot 4; \\
\text{endmodule}
\]

where

\[
\begin{align*}
\text{constvars} &= [\ldots] \{ \text{getVariableLists}^3(\text{stm}) \} \\
\text{outeventvars} &= \{ e : \text{outevents} \cdot \text{EVT} \cdot \text{id}(e) : [\text{e.type}] \cdot t \} \\
\text{stmret} &= \bigcup \{ \text{stm, stm, 0} \cdot 1 \} \cdot S. \\
\text{ropsvars} &= \bigcup \{ \text{op} : \text{getRequiredOperations}^4(\text{stm}) \cdot \text{op} \rightarrow [\text{p.type}] \cdot t \} \\
\end{align*}
\]
Rule 121. Composite states

\[
\text{Rule 122. Exit a composite state}
\]

\[
\text{Rule 123. Entry into a composite state}
\]

\[
\text{Rule 124. Exit from a composite state}
\]
**Rule 123. Exit substates**

\[
\text{exitSubstates} \left( \begin{array}{c}
\text{states} : \mathbb{P}\text{State}, \\
\text{exit} : \text{VarDecl}_{pr}, \\
\text{scpcname} : \text{Name}, \\
\text{pcconstrs} : \mathbb{P}\text{BoolExpr}_{pr}, \\
\text{stnumber} : \text{int}
\end{array} \right) = \\
\begin{cases}
\text{if } \text{states} = \emptyset & \text{then } (\text{stnumber}, 0, 0) \\
\text{else } & (\text{othersret}.1, \text{sret}.2 \cup \text{otherret}.2, \text{sret}.3 \cup \text{otherret}.3)
\end{cases}
\]

where

\[
\begin{align*}
\text{s} & \in \text{states} \\
\text{sret} &= \text{exitSubstate}^1 (\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}) \\
\text{othersret} &= \text{exitSubstates}^2 (\text{states} \setminus \{\text{s}\}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{sret}.1)
\end{align*}
\]

**Rule 124. Exit a substate**

\[
\text{exitSubstate} (\text{s} : \text{State}, \text{exit} : \text{VarDecl}_{pr}, \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\text{BoolExpr}_{pr}, \text{stnumber} : \text{int}) = \\
\begin{cases}
\text{if } \text{isComposite}^2 (\text{s}) & \text{then } \text{exitCompSubstate}^1 (\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}) \\
\text{else } & \text{exitSimpSubstate}^1 (\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber})
\end{cases}
\]

**Rule 125. Exit a simple substate**

\[
\text{exitSimpSubstate} (\text{s} : \text{State}, \text{exit} : \text{VarDecl}_{pr}, \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\text{BoolExpr}_{pr}, \text{stnumber} : \text{int}) = \\
(\text{exitret}.1, \text{exitret}.2, \{\text{cmd1}\} \cup \text{exitret}.4)
\]

where

\[
\begin{align*}
\text{exitret} &= \left[ \text{getExitAction}^2 (\text{s}, \emptyset, \text{stnumber}, \text{scpcname}, \text{null}, \{\text{exit}\text{'name}' = \text{Exit}_\text{Sub}_\text{EXITED}\}) \right] \mathcal{A}' \\
\text{cmd1} &= \left[ \begin{array}{c}
\text{andExprs}^3 (\text{pcconstrs}) \\
\text{scpcname} = \text{id}(\text{s}) \\
\text{exit}\text{'name}' = \text{Exit}_\text{Sub}_\text{ACT}
\end{array} \right] ightarrow 1.0 : \text{exitret}.3;
\end{align*}
\]
Chapter 5. Probabilistic Semantics

Rule 126. Exit a composite substate

\[
exit\text{CompSubstate} \bigg( s : \text{State}, \text{exit} : \text{VarDecl}, \text{scpcname} : \text{Name}, \\
pconstrs : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int} \
\bigg) : \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Command} = \\
(\text{stnumber}, 0, \{\text{cmd1}, \text{cmd2}\})
\]

where
\[
\text{exitreq} = (\text{scpcname} = \text{id}(s)) \land (\text{exit.name} = \text{Exit\_Sub\_ACT})
\]
\[
\text{cmd1} = [] \left( \text{andExprs}^3(\text{pconstrs} \land \text{exitreq}) \to 1.0 : \left( \begin{array}{c}
\text{(exit.name)' = Exit\_Sub\_ACT\_Waiting} \\
\text{(\text{id}(s)\_exit' = Exit\_ACT\_Parent)}
\end{array} \right) \right)
\]
\[
\text{exitedcheck} = \left( \text{scpcname} = \text{id}(s) \land (\text{id}(s)\_exit = \text{Exit\_EXITED}) \land (\text{exit.name} = \text{Exit\_Sub\_ACT\_Waiting}) \right)
\]
\[
\text{cmd2} = [] \left( \text{andExprs}^5(\text{pconstrs} \land \text{exitedcheck}) \to 1.0 : \left( \begin{array}{c}
\text{(exit.name)' = Exit\_Sub\_EXITED} \\
\text{(\text{id}(s)\_exit' = Exit\_NONE)}
\end{array} \right) \right)
\]

Rule 127. Enter substates (with entry actions)

\[
\text{enterSubstates} \bigg( \text{states} : \mathbb{P} \text{State}, \text{stm} : \text{StateMachineDef}, \text{scpcname} : \text{Name}, \\
pconstrs : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int} \
\bigg) : \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Command} = \\
\text{if states} = 0 \text{ then} \\
(\text{stnumber}, 0, 0)
\text{else} \\
(\text{othersret}.1, \text{sret}.2 \cup \text{otherret}.2, \text{sret}.3 \cup \text{otherret}.3)
\]

where
\[
\text{s } \in \text{ states}
\]
\[
\text{sret} = \text{enterSubstate}^1(s, \text{stm}, \text{scpcname}, \text{pconstrs}, \text{stnumber})
\]
\[
\text{othersret} = \text{enterSubstates}^2(\text{states} \setminus \{s\}, \text{stm}, \text{scpcname}, \text{pconstrs}, \text{sret}.1)
\]

Rule 128. Enter a substate (with an entry action)

\[
\text{enterSubstate} \bigg( s : \text{State}, \text{stm} : \text{StateMachineDef}, \text{scpcname} : \text{Name}, \\
pconstrs : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int} \
\bigg) : \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Command} = \\
(\text{entryret}.1, \{\text{enteringconst}\} \cup \text{entryret}.2, \text{entryret}.4)
\]

where
\[
\text{enteringconst} = \text{const int} \text{id}(s)\_entering = \text{stnumber};
\]
\[
\text{extraassign} = \left( \begin{array}{c}
\text{if isComposite}^3(s) \text{ then} \\
\text{id}(s)\_scpc' = \text{id} \left( \text{getInitial}^2(s) \right)
\end{array} \right)
\]
\[
\text{else} \\
\text{id}(\text{stm})\_lock' = \text{id}(\text{stm})\_LOCK\_FREE
\]
\[
\text{entryret} = \left[ \begin{array}{c}
\text{getEntryAction}^1(s, \text{pconstrs}, \text{stnumber} + 1, \text{scpcname}, \text{enteringconst}, \\
\{\text{scpcname}' = \text{id}(s), \text{extraassign}\} \end{array} \right]
\]
5.3 Translation to PRISM

Rule 129. Constants representing states for named elements

\[
\text{constantsOfNamedElems}(\ \text{elems} : \mathbb{P} \text{NamedElement}, \text{stnumber} : \text{int}) : \text{int} \times \mathbb{P} \text{Constant} = \\
\begin{cases} 
\text{if stnumber = 0 then (stnumber, 0) else (othersret.1, \{elconst\} \cup \text{othersret.2})}
\end{cases}
\]

where

\[
\begin{align*}
\text{el} &\in \text{elems} \\
\text{elconst} &= \text{constintid}(\text{el}) = \text{stnumber}; \\
\text{othersret} &= \text{constantsOfNamedElems}(\text{elems} \cup \{\text{el}\}, \text{stnumber} + 1)
\end{align*}
\]

Rule 130. Composite substates

\[
\begin{cases}
\text{cstates : } \mathbb{P} \text{State}, \text{stm : } \text{StateMachineDef}, \text{scpcname : } \text{Name}, \text{pcconstrs : } \mathbb{P} \text{BoolExpr}, \text{trnumber : } \text{int} \\
\end{cases}
\]

\[
\begin{align*}
\text{SS} &\subseteq \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{VarDecl} \times \mathbb{P} \text{Command} = \\
\begin{cases} 
\text{if cstates = 0 then (trnumber, 0, 0, 0) else (othersret.1, csret.2 \cup \text{othersret.2, csret.3 \cup \text{othersret.3, csret.4 \cup \text{othersret.4})}}
\end{cases}
\end{align*}
\]

where

\[
\begin{align*}
\text{cs} &\in \text{cstates} \\
\text{csret} &= \begin{bmatrix} \text{cs}, \text{stm}, \text{pcconstrs} \cup \{\text{scpcname} = \text{id}(\text{cs})\}, \text{trnumber} \end{bmatrix}^{S} \\
\text{othersret} &= \begin{bmatrix} \text{cstates} \cup \{\text{cs}\}, \text{stm}, \text{scpcname}, \text{pcconstrs}, \text{csret.1} \end{bmatrix}^{S}
\end{align*}
\]

Transitions

Rule 131. Nodes (transitions)

\[
\begin{cases}
\text{nodes : } \mathbb{P} \text{Node}, \text{cs : } \text{NodeContainer}, \text{stm : } \text{StateMachineDef}, \text{exit : } \text{VarDecl}\#, \text{scpcname : } \text{Name}, \text{pcconstrs : } \mathbb{P} \text{BoolExpr}\#, \text{trnumber : } \text{int} \\
\end{cases}
\]

\[
\begin{align*}
\text{TNS} &\subseteq \text{int} \times \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Command} = \\
\begin{cases} 
\text{if nodes = 0 then (trnumber, trnumber, 0, 0) else (othersret.1, othersret.2, nret.3 \cup \text{othersret.3, nret.4 \cup \text{othersret.4})}
\end{cases}
\end{align*}
\]

where

\[
\begin{align*}
\text{n} &\in \text{othersret} \\
\text{nret} &= \begin{bmatrix} \text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber} \end{bmatrix}^{TN} \\
\text{othersret} &= \begin{bmatrix} \text{nodes} \cup \{\text{n}\}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{nret.1}, \text{nret.2} \end{bmatrix}^{TN}
\end{align*}
\]

Rule 132. Node (transitions)

\[
\begin{cases}
\text{n : } \text{Node}, \text{cs : } \text{NodeContainer}, \text{stm : } \text{StateMachineDef}, \text{exit : } \text{VarDecl}\#, \text{scpcname : } \text{Name}, \text{pcconstrs : } \mathbb{P} \text{BoolExpr}\#, \text{trnumber : } \text{int} \\
\end{cases}
\]

\[
\begin{align*}
\text{TN} &\subseteq \text{int} \times \text{int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Command} = \\
\text{This function is split in multiple rules: Rules 31, 32, 33, and 34 according to the subtype of the node.}
\end{align*}
\]
### Rule 133. Initial junction (transitions)

```plaintext
\[
\begin{aligned}
\text{n} &: \text{Initial}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{pr}, \\
\text{scpconstrs} &: \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
\text{tn} &: \text{int} \times \text{P Constant} \times \text{P Command} = \\
\text{trans} &= \{ t : \text{cs.transitions} \mid t.\text{source} = \text{n} \} \\
\text{transret} &= [\text{trans}, \text{n}, \text{stm}, \text{exit}, \text{scpconstrs}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}] \\
\text{TS} &= (\text{transret}.1, \text{transret}.2, \text{transret}.3, \text{transret}.5)
\end{aligned}
\]
```

### Rule 134. Probabilistic junction (transitions)

```plaintext
\[
\begin{aligned}
\text{n} &: \text{ProbJunc}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{pr}, \\
\text{scpconstrs} &: \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
\text{tn} &: \text{int} \times \text{int} \times \text{P Constant} \times \text{P Command} = \\
\text{trans} &= \{ t : \text{cs.transitions} \mid t.\text{source} = \text{n} \} \\
\text{transret} &= [\text{trans}, \text{n}, \text{stm}, \text{exit}, \text{scpconstrs}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}] \\
\text{cmd} &= [\{ \text{andExprs}^{6}(\text{pcconstrs}) \land (\text{scpconstr} = \text{id}(\text{n})) \} \rightarrow \text{transret}.4; \\
\text{TS} &= (\text{transret}.1, \text{transret}.2, \text{transret}.3, \{ \text{cmd} \} \cup \text{transret}.5)
\end{aligned}
\]
```

### Rule 135. Normal junction (transitions)

```plaintext
\[
\begin{aligned}
\text{n} &: \text{Junction}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{pr}, \\
\text{scpconstrs} &: \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
\text{tn} &: \text{int} \times \text{int} \times \text{P Constant} \times \text{P Command} = \\
\text{trans} &= \{ t : \text{cs.transitions} \mid t.\text{source} = \text{n} \} \\
\text{transret} &= [\text{trans}, \text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpconstrs}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}] \\
\text{TS} &= (\text{transret}.1, \text{transret}.2, \text{transret}.3, \text{transret}.5)
\end{aligned}
\]
```

### Rule 136. State (transitions)

```plaintext
\[
\begin{aligned}
\text{n} &: \text{State}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{pr}, \\
\text{scpconstrs} &: \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
\text{tn} &: \text{int} \times \text{int} \times \text{P Constant} \times \text{P Command} = \\
\text{trans} &= \{ t : \text{cs.transitions} \mid t.\text{source} = \text{n} \} \\
\text{transret} &= [\text{trans}, \text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpconstrs}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}] \\
\text{TS} &= (\text{transret}.1, \text{transret}.2, \text{transret}.3, \text{transret}.5)
\end{aligned}
\]
```
Rule 137. Transitions from a node

\[
\text{trans} : \text{Transition}, \text{n} : \text{Node}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{pr},
\text{scpcname} : \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int}.
\]

\[TS = \begin{cases}
\text{if } \text{trans} = \emptyset \text{ then } & (\text{stnumber}, \text{trnumber}, 0, 0, 0) \\
\text{else} & (\text{othersret}.1, \text{othersret}.2, \text{tret}.3 \cup \text{othersret}.3, \text{tret}.4 \cup \text{othersret}.4, \text{tret}.5 \cup \text{othersret}.5)
\end{cases}
\]

where

\[
t \in \text{trans}
\]

\[
\text{tret} = [t, \text{n}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}]
\]

\[
\text{othersret} = [\text{trans} \setminus \{t\}, \text{n}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{tret}.1, \text{tret}.2]
\]

Rule 138. Action and target of a transition

\[
\text{actionOfTran} \quad (t : \text{Transition}, \text{stm} : \text{StateMachineDef}, \text{scpcname} : \text{Name}, \text{pcconstrs} : \text{P BoolExpr}_{pr}, \text{stnumber} : \text{int})
\]

\[\text{tactionret} = \begin{cases}
\text{if } t.\text{target} \in \text{State} \land \text{hasEntryAction}^2(t.\text{target}) \text{ then } \text{id}(t.\text{target}).\text{entering} \\
\text{else } \text{id}(t.\text{target})
\end{cases}
\]

\[
\text{targetassigns} = \begin{cases}
\text{if } t.\text{target} \in \text{State} \land \neg \text{hasEntryAction}^2(t.\text{target}) \text{ then } \\
\text{if } \text{isComposite}(t.\text{target}) \text{ then } \{\text{id}(t.\text{target}) . _\text{scpc}' = \text{id}(\text{getInitial}^p(t.\text{target}))\}
\text{else } \{\text{id}(\text{stm}) . _\text{lock}' = \text{id}(\text{stm}) \text{ LOCK } \text{ FREE }\}
\text{else } \emptyset
\end{cases}
\]

\[
\text{tactionret} = \begin{cases}
t.\text{action}, \text{pcconstrs}, \text{stnumber}, \text{scpcname}, \text{null}., \\
\{\text{scpcname}' = \text{target}\} \cup \text{targetassigns}
\end{cases}
\]

This function is split in multiple rules: Rules 38, 39, 40, and 41 according to the subtype of the node n.
Chapter 5. Probabilistic Semantics

Rule 140. Transition from an initial junction

\[
\begin{align*}
\text{Rule 140. Transition from an initial junction} & \quad t : \text{Transition}, n : \text{Initial}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}, \\
& \quad \text{scpncname} : \text{Name}, \text{pcconstrs} : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
& \quad \text{where} \quad \text{locktconst} = \text{const int id}(t) = \text{trnumber}; \\
& \quad \text{tactret} = \text{actionOfTran}^1(t, \text{stm}, \text{scpncname}, \text{pcconstrs}, \text{stnumber}) \\
& \quad \text{lockassign} = \text{if } \text{tactret} \neq \emptyset \text{ then } \{\text{id(\text{stm})_lock} = \text{locktconst.name}\} \text{ else } \emptyset \\
& \quad \text{cmd1} = [[\text{andExprs}^7(\text{pcconstrs}) & (\text{scpncname} = \text{id}(n)))] \rightarrow 1.0 : \text{tactret} \cup \text{lockassign}; \\
\end{align*}
\]

Rule 141. Transition from a probabilistic junction

\[
\begin{align*}
\text{Rule 141. Transition from a probabilistic junction} & \quad t : \text{Transition}, n : \text{ProbJunc}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}, \\
& \quad \text{scpncname} : \text{Name}, \text{pcconstrs} : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
& \quad \text{where} \quad \text{tactret} = \text{actionOfTran}^2(t, \text{stm}, \text{scpncname}, \text{pcconstrs}, \text{stnumber}) \\
\end{align*}
\]

Rule 142. Transition from a normal junction

\[
\begin{align*}
\text{Rule 142. Transition from a normal junction} & \quad t : \text{Transition}, n : \text{Junction}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}, \\
& \quad \text{scpncname} : \text{Name}, \text{pcconstrs} : \mathbb{P} \text{BoolExpr}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int} \\
& \quad \text{where} \quad \text{tactret} = \text{actionOfTran}^3(t, \text{stm}, \text{scpncname}, \text{pcconstrs}, \text{stnumber}) \\
& \quad \text{scpcgrd} = \text{pcconstrs} \cup \{\text{scpncname} = \text{id}(n)\} \\
& \quad \text{condgrd} = \text{if } t.\text{condition} \neq \text{null} \text{ then } (\text{scpcgrd} \cup \{\text{[t.condition]_e}\}) \text{ else } \text{scpcgrd} \\
& \quad \text{cmd1} = [[\text{andExprs}^8(\text{condgrd})] \rightarrow 1.0 : \text{tactret} \cup \text{lockassign}; \\
\end{align*}
\]
### Rule 143. Transition from a state

\[
\begin{align*}
t &: \text{Transition}, \\
n &: \text{State}, \\
\text{stm} &: \text{StateMachineDef}, \\
\text{exit} &: \text{VarDeclPr}, \\
\text{scpcname} &: \text{Name}, \\
\text{pcconstrs} &: \mathbb{P} \text{BoolExprPr}, \\
\text{stnumber} &: \text{int}, \\
\text{trnumber} &: \text{int} \\
\end{align*}
\]

\[\text{T} = \text{stm} \times \text{stm} \times \mathbb{P} \text{Const} \times \mathbb{P} \text{Update} \times \mathbb{P} \text{Command} = \]

\[
\begin{align*}
(tactref.1, \text{trnumber} + 1, \{\text{locktconst}\} \cup \text{tactref.2}, 0, \text{tactref.4} \cup \text{exitsrcref.2} \cup \{\text{cmd1}\})
\end{align*}
\]

where

\[
\begin{align*}
\text{locktconst} &= \text{const} \times \text{int} \\
\text{scpcgrd} &= \text{pcconstrs} \cup \{\text{scpcname} = \text{id}(n)\} \\
\text{condgrd} &= \text{if } t.\text{condition} \neq \text{null} \text{ then } \left(\text{scpcgrd} \cup \{t.\text{condition}\}\right) \text{ else } \text{scpcgrd} \\
\text{lockfreegrd} &= \left(\text{id}(\text{stm})\text{.lock} = \text{id}(\text{stm})\_\text{LOCK\_FREE}\right) \\
\text{tactref} &= \text{actionOfTrans}^4(t.\text{stm}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}) \\
\text{lockgrd} &= \left(\text{id}(\text{stm})\_\text{lock} = \text{locktconst}\_\text{name}\right) \\
\text{exitsrcret} &= \left(\begin{array}{l}
\text{if } \text{isComposite}^6(n) \text{ then } \text{exitCompSrcState}^1(t.\text{scpcgrd} \cup \{\text{locktgrd}\} \cup \text{tactref.3}) \\
\text{else } \text{exitSimpSrcState}^1(t.\text{exit}, \text{scpcgrd}, \text{tactref.3})
\end{array}\right) \\
\text{trigvarassign} &= \left(\begin{array}{l}
\text{if } t.\text{trigger} \neq \text{null } \land \text{t.trigger.type} = \text{CommunicationType.INPUT} \\
\text{then } \left\{(\text{id}(t.\text{trigger.parameter}) = \text{EVT}_\_\text{id}(t.\text{trigger.event})\right)\right) \\
\text{else } 0
\end{array}\right) \\
\text{lockassign} &= \left(\begin{array}{l}
\text{if } t.\text{trigger} \neq \text{null } \text{then } \\
\left\{(\text{id}(\text{stm})\_\text{lock} = \text{locktconst}\_\text{name}\right) \\
\text{else } 0
\end{array}\right)
\end{align*}
\]

\[
\text{cmd1} = \left\{(\text{id}(t.\text{trigger.event}) \text{ andExprs}^9(\text{condgrd} \cup \text{lockfreegrd})) \rightarrow 1.0 : \text{trigvarassign} \cup \text{lockassign} \cup \text{exitsrcret.1}; \\
\text{else } (\text{id}(\text{stm})\_\text{lock}) \text{ andExprs}^{10}(\text{condgrd} \cup \text{lockfreegrd})) \rightarrow 1.0 : \text{trigvarassign} \cup \text{lockassign} \cup \text{exitsrcret.1};
\right\}
\]

### Rule 144. Exit composite source state

\[
\text{exitCompSrcState}^1(t.\text{transition}, \text{pcconstrs} : \text{BoolExprPr}, \text{curassgns} : \mathbb{P} \text{AssignmentPr})
\]

\[
\text{P Assignment} \times \mathbb{P} \text{Command} = \\
\left\{(\text{id}(t.\text{source})_\_\text{exit} = \text{Exit}\_\text{ACT}_\text{Trans} ), \text{exitcmd}\right\}
\]

where

\[
\text{exitcmd} = \left\{(\text{id}(\text{t.source})_\_\text{exit} = \text{Exit}_\text{EXITED}) \rightarrow 1.0 : \text{curassgns};
\right\}
\]

### Rule 145. Exit simple source state

\[
\text{exitSimpSrcState}^1(t.\text{transition}, \text{exit} : \text{VarDeclPr}, \text{pcconstrs} : \text{BoolExprPr}, \text{curassgns} : \mathbb{P} \text{AssignmentPr})
\]

\[
\text{P Assignment} \times \mathbb{P} \text{Command} = \\
\text{if } - \text{hasExitAction}^1(t.\text{source}) \text{ then } (\text{curassgns}, 0) \text{ else } \left\{(\text{exit}\_\text{name} = \text{Exit}_\text{Sub}\_\text{ACT} ), \text{exitcmd}\right\}
\]

where

\[
\text{exitcmd} = \left\{(\text{id}(\text{t.source})_\_\text{exit} = \text{Exit}_\text{Sub}_\text{EXITED} ) \rightarrow 1.0 : \text{curassgns};
\right\}
\]
### Rule 146. Actions

\[
\text{act : Action, pcconstrs : } P \text{BoolExpr, stnum : int, scpcname : Name, } \\
curstate : \text{Constant, assigns : } P \text{Assignment} \\
: \text{int} \times P \text{Constant} \times P \text{Assignment} \times P \text{Command} = \\
\begin{cases} 
\text{if act = null v act.action = null then} & \\
(snum, \emptyset, assigns, \emptyset) \\
\text{else} & \\
[\text{act.action, pcconstrs, snum, scpcname, curstate, assigns}]_ST \\
\end{cases}
\]

### Rule 147. Statement

\[
\text{stmt : Statement, pcconstrs : } P \text{BoolExpr, stnum : int, scpcname : Name, } \\
curstate : \text{Constant, assigns : } P \text{Assignment} \\
: \text{int} \times P \text{Constant} \times P \text{Assignment} \times P \text{Command} = \\
\text{This rule is split in multiple rules from Rule 46 to Rule 51 according to the subtype of the statement.}
\]

### Rule 148. Statement (Skip)

\[
\text{stmt : Skip, pcconstrs : } P \text{BoolExpr, stnum : int, scpcname : Name, } \\
curstate : \text{Constant, assigns : } P \text{Assignment} \\
: \text{int} \times P \text{Constant} \times P \text{Assignment} \times P \text{Command} = \\
(snum, \emptyset, assigns, \emptyset)
\]

### Rule 149. Statement (Assignment)

\[
\text{stmt : Assignment, pcconstrs : } P \text{BoolExpr, stnum : int, scpcname : Name, } \\
curstate : \text{Constant, assigns : } P \text{Assignment} \\
: \text{int} \times P \text{Constant} \times P \text{Assignment} \times P \text{Command} = \\
\begin{cases} 
\text{if curstate} \neq \text{null then} \text{stnum else stnum + 1, } & \\
\text{if curstate} \neq \text{null then} \emptyset \text{ else } \{ \text{const} \}, & \\
\text{curassgns, } \{ \text{cmd} \} & \\
\end{cases}
\]

\text{where}

\begin{align*}
\text{const} &= \text{if curstate} \neq \text{null then curstate else const int }
\text{uname (stmt, stnum)} = \text{stnum; } \\
\text{curassgns} &= \{ \text{scpcname}' = \text{const.name} \} \\
\text{cmd} &= \left( \left[ \text{andExprs} \right] \text{pcconstrs} \&\& \text{scpcname = const.name} \right) \\
& \quad \rightarrow 1.0 : \text{assigns } \cup \left[ \text{id(stmt.left)} \right]' = \left[ \text{stmt.right} \right]' \\
\end{align*}
Rule 150. Statement (CommunicationStmt)

\[
\begin{align*}
\text{stmt} &: \text{CommunicationStmt}, \text{pcconstrs} : \mathbb{P} \text{BoolExp}, \text{stnumber} : \text{int}, \text{scpcname} : \text{Name}, \\
\text{curstate} &: \text{Constant}^{\mathbb{P}}, \text{assigns} : \mathbb{P} \text{Assignment}^{\mathbb{P}}, \\
\text{cmd} &: \text{Int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Assignment} \times \mathbb{P} \text{Command}
\end{align*}
\]

\[
\begin{align*}
\text{where} \\
\text{const} &= \text{if curstate} \neq \text{null} \text{ then curstate else const } \text{int} \text{uname (stmt, stnumber)} = \text{stnumber}; \\
\text{curassgns} &= \text{assignchvar} \\
\text{curassgns} &= \text{assigninvar} \\
\text{cmd} &= \text{if stmt.communication.type} = \text{CommunicationType.OUT} \text{ then} \\
& \quad \text{else} \ 0
\end{align*}
\]

Rule 151. Statement (If Statements)

\[
\begin{align*}
\text{stmt} &: \text{IfStmt}, \text{pcconstrs} : \mathbb{P} \text{BoolExp}, \text{stnumber} : \text{int}, \text{scpcname} : \text{Name}, \\
\text{curstate} &: \text{Constant}^{\mathbb{P}}, \text{assigns} : \mathbb{P} \text{Assignment}^{\mathbb{P}}, \\
\text{cmd} &: \text{Int} \times \mathbb{P} \text{Constant} \times \mathbb{P} \text{Assignment} \times \mathbb{P} \text{Command}
\end{align*}
\]

\[
\begin{align*}
\text{where} \\
\text{const} &= \text{if curstate} \neq \text{null} \text{ then curstate else const } \text{int} \text{uname (stmt, stnumber)} = \text{stnumber}; \\
\text{curassgns} &= \text{assignchvar} \\
\text{curassgns} &= \text{assigninvar} \\
\text{cmd} &= \text{if stmt.communication.type} = \text{CommunicationType.OUT} \text{ then} \\
& \quad \text{else} \ 0
\end{align*}
\]
**Rule 152. Statement (Call)**

\[
\text{stmt : Call, pcconstrs : } \mathbb{P} \text{BoolExpr}_{pr}, \text{stnumber : int, scpname : Name,}
\text{curstate : Constant}_{pr}, \text{curupdates : } \mathbb{P} \text{Update}_{pr}, \text{assigns : } \mathbb{P} \text{Assignment}_{pr}
\]
\[
: \text{int } \times \mathbb{P} \text{Constant } \times \mathbb{P} \text{Assignment } \times \mathbb{P} \text{Command} =
\]

\[
\begin{align*}
\text{if curstate } \neq \text{ null} & \text{ then stnumber else stnumber + 1,} \\
\text{if curstate } \neq \text{ null} & \text{ then } 0 \text{ else } \{\text{const}\}, \\
\text{curassgns, \{cmd\}}
\end{align*}
\]

where

\[
\text{const } = \text{ if curstate } \neq \text{ null then curstate else const int}_{name} (\text{stmt, stnumber}) = \text{ stnumber};
\]

\[
\text{curassgns} = \{\text{scpname}' = \text{ const.name}\}
\]

\[
\text{call} = \text{name} \mapsto \text{stmt.operation.name, args} \mapsto \{a : \text{stmt.args } \cup [a]\}_{e7}
\]

\[
\text{cmd} = \left(\emptyset \text{ andExprs}_{17} \left(pcconstrs \& (\text{scpname = const.name})\right) \rightarrow \text{} 1.0 : \text{assigns } \cup \{\text{call]\right)
\]

**Rule 153. Statement (sequential composition)**

\[
\text{stmt : SeqStatement, pcconstrs : } \mathbb{P} \text{BoolExpr}_{pr}, \text{stnumber : int, scpname : Name,}
\text{curstate : Constant}_{pr}, \text{assigns : } \mathbb{P} \text{Assignment}_{pr}
\]
\[
: \text{int } \times \mathbb{P} \text{Constant } \times \mathbb{P} \text{Assignment } \times \mathbb{P} \text{Command} =
\]

\[
\left[\text{stmt.statements, pcconstrs, stnum, scpname, curstate, assigns}\right]
\]

**Rule 154. Statement (sequence of statements)**

\[
\text{sts : seq Statement, pcconstrs : } \mathbb{P} \text{BoolExpr}_{pr}, \text{stnumber : int, scpname : Name,}
\text{curstate : Constant}_{pr}, \text{assigns : } \mathbb{P} \text{Assignment}_{pr}
\]
\[
: \text{int } \times \mathbb{P} \text{Constant } \times \mathbb{P} \text{Assignment } \times \mathbb{P} \text{Command} =
\]

\[
\text{if ststs } = \langle \rangle \text{ then (stnumber, } 0, \text{ assigns, } \emptyset) \\
\text{else if } \# \text{ ststs } = 1 \text{ then }
\left[\text{headststs, pcconstrs, stnumber, scpname, curstate, assigns}\right]
\]
\[
\text{else let lres } = \left[\text{lastststs, pcconstrs, stnumber, scpname, null, assigns}\right]
\]
\[
\text{fsres } = \left[\text{frontststs, pcconstrs, lres.1, scpname, curstate, lres.3}\right]
\]
\[
\text{fsres.1, lres.2 } \cup \text{ fsres.2, fsres.3, lres.4 } \cup \text{ fsres.4}
\]

**Auxiliary functions**
### Rule 155. Get variable lists

\[
\text{getVariableLists}(c : \text{Context}) : \mathbb{P} \text{ VariableList} = (c.\text{variableList} \cup \text{vl}_p \cup \text{vl}_d)
\]

where

- \( \text{vl}_p = \{i : c.p\text{Interfaces} \bullet i.\text{variableList}\} \)
- \( \text{vl}_d = \{i : c.\text{interfaces} \bullet i.\text{variableList}\} \)

### Rule 156. Get events

\[
\text{getEvents}(c : \text{Context}) : \mathbb{P} \text{ Event} = (c.\text{events} \cup \text{ev}_d)
\]

where

- \( \text{ev}_d = \bigcup \{i : c.\text{interfaces} \bullet i.\text{events}\} \)

### Rule 157. Get required operations

\[
\text{getRequiredOperations}(c : \text{Context}) : \mathbb{P} \text{ OperationSig} = \bigcup \{i : c.r\text{Interfaces} \bullet i.\text{operations}\}
\]

### Rule 158. Get the exit action

\[
\text{getExitAction}(s : \text{State}) : \text{Action} =
\begin{align*}
\text{if} \ (\text{exitactions} = \emptyset) & \ \text{then} \\
& \null \\
\text{else} & \\
& \mu \alpha : s.\text{actions} \mid \alpha \in \text{ExitAction}
\end{align*}
\]

where

- \( \text{exitactions} = \{\alpha : s.\text{actions} \mid \alpha \in \text{ExitAction}\} \)

### Rule 159. Has the exit action

\[
\text{hasExitAction}(s : \text{State}) : \text{Boolean} =
\begin{align*}
\text{if} \ (\text{getExitAction}(s) = \null) & \ \text{then} \\
& \false \\
\text{else} & \\
& \true
\end{align*}
\]
Rule 160. Get the entry action
\[
\text{getEntryAction}(s : \text{State}) : \text{Action} = \\
\text{if } (\text{entryActions} = \emptyset) \text{ then} \\
\quad \text{null} \\
\text{else} \\
\quad \mu a : s.\text{actions} \mid a \in \text{EntryAction} \\
\text{where} \\
\quad \text{entryactions} = \{ a : s.\text{actions} \mid a \in \text{EntryAction} \}
\]

Rule 161. Has the entry action
\[
\text{hasEntryAction}(s : \text{State}) : \text{Boolean} = \\
\text{if } (\text{getEntryAction}^2(s) = \text{null}) \text{ then} \\
\quad \text{false} \\
\text{else} \\
\quad \text{true}
\]

Rule 162. Conjunction of boolean expressions
\[
\text{andExprs}(\text{exprs} : \mathbb{P} \text{BoolExpr}) : \text{BoolExpr} = \\
\text{if } (\text{exprs} = \emptyset) \text{ then} \\
\quad \text{true} \\
\text{else} \\
\quad \text{expr} \& \text{andExprs}^{\{\text{expr}\}}(\text{exprs} \setminus \{\text{expr}\}) \\
\text{where} \\
\quad \text{expr} \in \text{exprs}
\]

Rule 163. Is a state composite
\[
\text{isComposite}(s : \text{State}) : \text{boolean} = \\
\text{if } (s.\text{nodes} \neq \emptyset) \text{ then} \text{true} \quad \text{else} \quad \text{false}
\]

Rule 164. Get the initial junction of a node container
\[
\text{getInitial}(s : \text{NodeContainer}) : \text{Initial} = \\
(\mu n : s.\text{nodes} \mid n \in \text{Initial})
\]
Rule 165. Variable lists

\[
[v_1: \text{VariableList}]_{\mathcal{X}} : \mathbb{P} \text{Constant} \times \mathbb{P} \text{VarDecl} = \\
( \bigcup \text{retvars.1}, \bigcup \text{retvars.2})
\]

where

\[
\text{retvars} = \{v : v_1.\text{vars} \bullet [v]\}
\]

Rule 166. Variables

\[
[v : \text{Variable}]_{\mathcal{Y}} : \mathbb{P} \text{Constant} \times \mathbb{P} \text{VarDecl} = \\
\begin{cases}
\text{if } v.\text{modifier} = \text{"const"} & \\
\{\text{const } [v.\text{type}]_{\mathcal{E}} \text{id}(v) = [v.\text{initial}]_{\mathcal{E}} ; \emptyset & \\
\text{else if } v.\text{modifier} = \text{"var"} & \\
\emptyset \{\text{id}(v) : [v.\text{type}]_{\mathcal{E}} = [v.\text{initial}]_{\mathcal{E}} ; \emptyset & \\
\end{cases}
\]

Expressions

The translation of expressions (including types) is standard. We note, however, that the expression language of Z is much richer than that of PRISM. So, in some cases, we require user input to translate the expressions.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td><strong>Assertion DSL Syntax</strong></td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Standard Assertions</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Probabilistic Assertions</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><strong>Assertions DSL Usage</strong></td>
<td>109</td>
</tr>
<tr>
<td>7.1</td>
<td>Standard Assertions</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Probabilistic Assertions</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><strong>Checking core assertions</strong></td>
<td>115</td>
</tr>
</tbody>
</table>
6. Assertion DSL Syntax

The assertions DSL supports the specification of standard untimed and timed assertions suitable for verification with FDR (as well as CSP-M processes) and probabilistic assertions for verification with PRISM.

6.1 Standard Assertions

**Syntax — Assertions.**

```
Assertion ::= ('timed' | 'untimed')? 'assertion' N ':' SPEC
            ('in' 'the' MODEL)?
            ('with' ('constant'| 'constants') CONSTANTS)?
```

An assertion is named and marked as *timed, untimed* or both (no keyword). It contains a property specification (SPEC), and allows the specification of a model (e.g., traces model), and allows the specification of values for constants used in the specification.
### Syntax — Specification.

| SPEC ::=          | N 'is' ('not')? PRED |
|                 | N ('does' 'not' 'terminate' | 'terminates') |
|                 | N 'is' ('not')? 'reachable' 'in' N |
|                 | N REL N |
|                 | 'clock' N 'is' ('not')? 'initialised' |
| PRED ::=         | 'deadlock-free' |
|                 | 'divergence-free' |
|                 | 'deterministic' |
|                 | 'timelock-free' |
| REL ::=          | 'refines' |
|                 | 'equals' |
|                 | 'does' 'not' 'refine' |
|                 | 'is' 'not' 'equal' |

A specification can either be unary or binary. Unary assertions describe properties such as termination, state reachability, clock initialisation, deadlock freedom, divergence freedom, determinism and timelock freedom of specific RoboChart elements (i.e., state machines, controllers, and modules). Binary assertions compare two RoboChart elements via the refinement and equality relations.

### Syntax — CSP Models.

| MODEL ::=        | 'traces' 'model' |
|                 | 'failures' 'model' |
|                 | 'failures' 'divergence' 'model' |

The currently supported CSP models are the standard traces, failures and failures divergences models.

### Syntax — Constant Definitions.

| CONSTANTS ::=     | (DEF (',' DEFs)*)? |
| DEF ::=          | N ('assigned' | 'set' 'to' | 'with' 'value') Expr |

The value of constants used in the semantics of a RoboChart element, by default, defined by an initial value or through the `instantiations.csp` file. The constant definitions in an assertion can override such values using the syntax above.

### Syntax — CSP specification.

| CSP specification ::= | ('timed' | untimed')? 'csp' N |
|                       | 'csp-begin' CSPM 'csp-end' |
A CSP specification can be identified as timed, untimed or both, its name must be used to define a CSP process, which is then exported for use in assertions. The definition of the process, as well as any auxiliary definitions, is written using CSP-M between the keywords `csp-begin` and `csp-end`.

### 6.2 Probabilistic Assertions

**Syntax — Probabilistic Statements.**

\[
\text{ProbStatements} ::= \text{ProbStatement}\ast \\
\text{ProbStatement} ::= \text{ConstDecl} \\
| \text{Constants} \\
| \text{Label} \\
| \text{Formula} \\
| \text{Rewards} \\
| \text{ProbAssertion}
\]

The probabilistic assertion syntax consists of various probabilistic statements:

- constant declarations (`ConstDecl` in Syntax 6.2.1),
- constant configurations (`Constants` in Syntax 6.2.1),
- labels (`Label` in Syntax 6.2.2),
- formulas (`Formula` in Syntax 6.2.2),
- rewards (`Rewards` in Syntax 6.2.3),
- probabilistic assertions (`ProbAssertion` in Syntax 6.2).

**Syntax — Probabilistic Assertions.**

\[
\text{ProbAssertion} ::= \textiste{prob } \text{assertion } \text{N }::= \text{ProbFormula} \\
| \text{with} \ ('\text{constant}'| 'constants') (\text{ConstConfigs} \ | \text{N})? \\
| \text{with} \ 'cmdoptions' \ STRING?
\]

A probabilistic assertion starts with `prob`. And a name (\( N \) denotes the name category, actually it is \( \text{ID} \) in Syntax 6.2.8) is associated with the assertion. The body of the assertion is a probabilistic formula (\( \text{ProbFormula} \) in Syntax 6.2.4) that denotes the property to be verified. In addition, an assertion optionally has constant configurations (given in `ConstConfigs` or a reference to a constants configuration name \( N \) in Syntax 6.2.1) and customised PRISM command line options\(^1\) (given in a string).

### 6.2.1 Constants

\(^1\)These options won’t be parsed and processed. They are just passed to the PRISM command line tool to provide a flexible way for users to specify command line options.
Chapter 6. Assertion DSL Syntax

Syntax — Constant Declaration.
ConstDecl ::= 'const' N ':' Type

A constant variable declared with a type (Type is the RoboChart Type class, see Appendix B, but in probabilistic assertions only primitive types nat, int, bool, and real are allowed, which is enforced in scoping and validation rules of RoboTool) will be used in probabilistic formulas. It is referred by its simple name without qualification.

Syntax — Probabilistic Constants.
Constants ::= 'constants' N ':' ConstConfigs

Syntax — Constants Configurations.
ConstConfigs ::= ConstConfig (','ConstConfig)*
               ((',')? 'and' ConstConfig)?

Constants associates a name with a configuration of various constants.

Syntax — Constant Configuration.
ConstConfig ::= QualifiedNameToElement
              ((set 'to' | 'assigned' | 'with' 'value') Expr
              | 'from' 'set' ConstSetExpr)

The configuration of each constant by ConstConfig can be a single value (Expr in Syntax 6.2.6) or a set of values from ConstSetExpr.

Syntax — ConstSetExpr.
ConstSetExpr ::= Expr ':' Expr ':' Expr
               | '{' (Expr (',' Expr)*)? '}'

A set of values for a constant is introduced either by a range of values from a start (the first expression) to an end (the second expression) with a step (the third expression) or by a set extension.

6.2.2 Labels and Formulas

Syntax — Labels.
Label ::= 'label' N '=>' BoolExpr

Labels provide a way to identify a set of states that are of particular interest. A label has its name associated with a boolean expression and could be referred in probabilistic formulas later by its
name (see Syntax 6.2.4).

**Syntax — Formulas.**

```
Formula ::= 'label' N '=' Expr
```

Formulas define expressions for reuse. The name given in a formula definition likes a shorthand to the defined expression.

### 6.2.3 Rewards

**Syntax — Rewards.**

```
Rewards ::= 'rewards' N '=' Reward+ 'endrewards'
```

A collection of rewards can be assigned a name and this name will be used in $R$ operator in Syntax 6.2.4.

**Syntax — Reward.**

```
Reward ::= ('[' QualifiedNameToElement ']')? BoolExpr ':' Expr ';
```

A reward composes a guard condition (BoolExpr), an expression, and an optional event (QualifiedNameToElement, in Syntax 6.2.7).

### 6.2.4 Probabilistic Formulas

**Syntax — Probabilistic Formulas.**

```
ProbFormula ::= BoolExpr
  | StateFormula '&&' StateFormula
  | StateFormula '||' StateFormula
  | StateFormula '=>' StateFormula
  | StateFormula '<=>' StateFormula
```

A probabilistic formula could be any boolean expression, or a conjunction, or a disjunction, or an implication, or an equivalence of two state formulas.
Chapter 6. Assertion DSL Syntax

Syntax — State Formulas.

StateFormula ::= '"' N '"' |
| '! StateFormula |
| '(' StateFormula ')' |
| '{' ProbFormula '}' |
| ('P'|'Prob') (Bound|Query)? 
  '.logged' (UseMethod)? |
| ('R'|'Reward') ('{'N'}')? (Bound|Query)? 
  '.logged' (UseMethod)? |
| ('A'|'Forall') 'logged' 'PathFormula' |
| ('E'|'Exists') 'logged' 'PathFormula' |

A state formula could be one of the followings:

- a reference to a defined label name,
- a negation,
- simply a probabilistic formula (but enclosed between '{' and '}'),
- a probability operator (P or Prob) with an optional bound (Bound) or query (Query), followed by a path formula and an optional simulation method (UseMethod),
- a reward operator (R or Reward) with an optional bound (Bound) or query (Query), followed by a reward path formula (RPathFormula) and an optional simulation method (UseMethod),
- a non-probability property by a forall operator (A or Forall) followed by a path formula,
- a non-probability property by an exists operator (E or Exists) followed by a path formula.

Syntax — Bound.

Bound ::= ('>+' | '>=' | '<-' | '<='+) Expr

A bound is simply one of four comparison operators, followed by an expression to be compared.

Syntax — Query.

Query ::= "=?" '=' |
| 'min' "=?" '=' |
| 'max' "=?" '='

A query has three formats: the probability, the minimum probability, and the maximum probability of a formula.
6.2 Probabilistic Assertions

Path formulas could be constructed from one of six common CTL and LTL operators, and probabilistic or path formulas.

Syntax — Path Formulas.
PathFormula ::= 'Next' ProbOrPathFormula
| ProbOrPathFormula 'Until' ProbOrPathFormula
| 'Finally' (Bound)? ProbOrPathFormula
| 'Globally' (Bound)? ProbOrPathFormula
| 'Weak' 'Until' (Bound)? ProbOrPathFormula
| 'Release' (Bound)? ProbOrPathFormula

Particularly, embedded path formulas in a path formulas have to be enclosed between [ and ].

Syntax — Probabilistic or Path Formulas.
ProbOrPathFormula ::= ProbFormula
| '['PathFormula']'

Reward path formulas could specify reachability, cumulative, total and instantaneous rewards.

Syntax — Rewards Path Formulas.
RPathFormula ::= ('Reachable'|'F') ProbFormula
| ('Cumul'|'C<=') Expr
| ('Total'|'C') Expr
| 'I=' Expr

6.2.5 Simulations

Statistic model checking which is based on simulation is another way to get an approximate result of properties in addition to precise model checking.

Syntax — Use Simulation Method.
UseMethod ::= 'using' 'sim' 'with' SimMethod
| (',' 'and' 'pathlen' '=' Expr)?

There are four simulation methods: CI, ACI, APMC, and SPRT. Each of them could have an optional path length in addition to their own parameters.
Chapter 6. Assertion DSL Syntax

Syntax — Simulation Method.
SimMethod ::= 'CI' ('at' CiMethod)?
    | 'ACI' ('at' CiMethod)?
    | 'APMC' ('at' APMCMethod)?
    | 'SPRT' ('at' SPRTMethod)?

Syntax — Simulation Method.
CiMethod ::= ((',')? 'w' '=' Expr)?
              & ((',')? 'alpha' '=' alpha=Expr)?
              & ((',')? 'n' '=' n=Expr)?

Syntax — Simulation Method.
APMCMethod ::= ((',')? 'epsilon' '=' Expr)?
               & ((',')? 'delta' '=' Expr)?
               & ((',')? 'n' '=' n=Expr)?

Both CI and ACI share the same parameters: width (w), confidence level (alpha), and the number of sampling (n). Here we use & to denote both sides could be present, but no more than once. The syntax of CiMethod ensures each parameter won’t appear more than once. Actually, the validation rules implemented in RoboTool enforce that users should supply exactly two parameters.

Syntax — Simulation Method.
SPRTMethod ::= ((',')? 'alpha' '=' Expr)?
               & ((',')? 'delta' '=' Expr)?

APMC also has three parameters: approximation (epsilon), confidence level (delta), and the number of sampling (n).

Syntax — Simulation Method.

But SPRT only has two parameters: type I/II error (epsilon), and indifference (delta).

6.2.6 Expressions
Expressions defined in probabilistic assertions are different from those in RoboChart. The syntax is much simpler. The reason of simplification is because the PRISM language only supports a very small subset of expressions in RoboChart. Particularly, an expression can be a reference to a variable or constant in RoboChart via a qualified name. And furthermore, we use $ to refer to a defined formula.

Boolean expressions could be true or false, or a reference to a boolean variable or constant via a qualified name, or current statecheck (by is in), or other regular boolean expressions. The current state check expression checks if a parent state machine or composite state (the first QualifiedNameToElement) is in its direct substate or child state (the second QualifiedNameToElement) or not.
6.2.7 Qualified Names

**Syntax — Qualified Name To Element.**

```
QualifiedNameToElement ::= NamedElement (:: NamedElement)*
```

A qualified name annotated with :: provides a way to uniquely identify each instance of named elements (NamedElement) in RoboChart. Here instances for controllers and state machines denote every reference to them. For instance, there is a state machine definition (named m) in the RoboChart model and three state machine references to this definition: r1, r2, and r3. In order to refer to every element i of m, we have to use qualified names such as ...::r1::i. In particular, each qualified name shall start with a RoboChart module name, except the constants declared in probabilistic assertions (because only simple names are used for them).

6.2.8 Terminals

**Syntax — Terminal rules.**

```
ID ::= (‘a’..‘z’|‘A’..‘Z’|‘_’) (‘a’..‘z’|‘A’..‘Z’|‘_’|‘0’..‘9’)*
BOOLEAN ::= ‘true’ | ‘false’
INT ::= (‘0’..‘9’)*
FLOAT ::= INT’.’INT
STRING ::= “” (‘\’ . | !('\\|”))* ”
```
7. Assertions DSL Usage

RoboTool also provides a simple text editor for an assertion DSL, which includes syntax highlighting, auto-completion, and error feedback.

7.1 Standard Assertions

The DSL helps you write simple assertions such as deadlock freedom and refinement without requiring knowledge of the naming conventions of our semantics. More complex properties can be specified in CSP within special environments, but this requires an understanding of the structure and naming conventions of the RoboChart semantics.

1. Create a new file by right-clicking the project, and selecting [New > File].
2. Name the file with the .assertions extension, and click OK.
3. (Optional) If RoboTool has not yet been configure to find the FDR executable, select the menu item [Window > Preferences].
4. (Optional) Select the RoboChart > Analysis item, and click Browse... to select the path to the installation directory of FDR.
5. (Optional) Click OK to apply the configuration.
6. In the .assertions file, write your custom assertions. Notice that it may be necessary to use the qualified name of RoboChart elements, such as, MyController::MyStateMachine.
7. In order to verify the assertions, right-click the .assertions file, and select the [RoboTool Analysis > Run FDR] item.
8. Provided there are no errors in the assertions or models, FDR checks the assertions in the
background, and RoboTool summarises the result in the form of a report, which is automatically opened upon completion of the checks.

Alternatively, it is possible to run predefined standard assertions such as deadlock freedom and nondeterminism. These assertions are generated automatically, and must be loaded into FDR manually. The next chapter provides instructions for doing so.

7.2 Probabilistic Assertions

7.2.1 Instructions

1. Create a new file by right-clicking the project, and selecting [New > File].
2. Name the file with the .assertions extension, and click OK.
3. (Optional) If RoboTool has not yet been configure to find the PRISM executable, select the menu item [Window > Preferences].
4. (Optional) Select the RoboChart > PRISM item, and click Browse... to select the path to the installation directory of PRISM.
5. (Optional) Click OK to apply the configuration.
6. In the .assertions file, write your assertions according to the probabilistic syntax in Section 6.2.
7. (Optional) If the RoboChart model has not generated its PRISM model manually, right-click one .rct file, and select the [RoboTool > PRISM > Compile] item to generate the PRISM model.
8. In order to verify the assertions, right-click the .assertions file, and select the [RoboTool > PRISM > Run] item.
9. Provided there are no errors in the assertions or models, PRISM checks the assertions in the background, and RoboTool summarises the result in the form of a report, which is automatically opened upon completion of the checks.

7.2.2 Examples

This section provides several probabilistic assertion examples to show how to specify probabilistic properties using our probabilistic assertion language.
7.2 Probabilistic Assertions

**Example 7.1 — Constants configuration.** This example defines a constants configuration C1 that sets the constants `batteryCharge` and `chargeSpeed` from the controller reference `ctrl_ref0` (that is from the module `mod0`) to 20 and 4 respectively. Then the configuration C1 is referred in the assertion `P_deadlock`.

```plaintext
constants C1: mod0::ctrl_ref0::batteryCharge set to 20,
    and mod0::ctrl_ref0::chargeSpeed set to 4
prob assertion P_deadlock:
    !E [Finally "deadlock"] with constants C1
```

**Example 7.2 — Constant declaration.** A integer constant `x` is declared, then it is used in the assertion formula. In addition, it is configured in the constants configuration to have values from the set \{1,2,3,4,5,6\}. This assertion is equivalent to six assertions that have `x` equal to 1 to 6 separately.

```plaintext
const x: core::int
prob assertion P_stuck_loc:
    Prob=? [Finally mod0::ctrl_ref0::p=x & mod0::ctrl_ref0::c=0]
    with constant mod0::ctrl_ref0::batteryCharge set to 20,
        mod0::ctrl_ref0::chargeSpeed set to 4,
        and x from set 1:6:1
```

**Example 7.3 — Label.** This example shows two labels are defined and they are associated with boolean expressions. The second label `l_stuck` denotes the state machine is in its substate `Stuck` now. In probabilistic formulas, labels are referred via their names within " and ". Particularly, since a label reference is a `StateFormula`, in order to put it in a conjunction with other probabilistic formulas, these other formulas should be converted into state formulas as well by enclosed within { and }. See Syntax 6.2.4 for more information.

```plaintext
label l_outOfPower = mod0::ctrl_ref0::c=0
label l_stuck =
    mod0::ctrl_ref0::stm_ref0 is in mod0::ctrl_ref0::stm_ref0::Stuck
prob assertion P_stuck_loc:
    Prob=? [Finally {mod0::ctrl_ref0::p=x} && "l_outOfPower"]
    with constants C1
```
Example 7.4 — Formula. A formula is just a shorthand of an expression and it is referred by putting a prefix $\$ $ in front of its name.

```plaintext
formula f_c = mod0::ctrl_ref0::c + 1
prob assertion P_stuck_loc1:
  Prob=? [Finally mod0::ctrl_ref0::p=x & $f_c=1] with constant C1
```

Example 7.5 — Rewards. The reward $\text{nbmove}$ assigns 1 to each synchronisation over the $\text{move}$ event that is from the state machine reference $\text{stm_ref0}$. Then a reward operator $\text{Reward}$ uses this defined reward $\text{nbmove}$ to check the average number of synchronisation over $\text{move}$ when the variable $\text{c}$ is equal to 0.

```plaintext
rewards nbmove =
  [mod0::ctrl_ref0::stm_ref0::move] true : 1;
endrewards
prob assertion R_stuck_move:
  Reward {nbmove} =? [ Reachable mod0::ctrl_ref0::c=0 ]
  with constant mod0::ctrl_ref0::batteryCharge set to 20,
  and mod0::ctrl_ref0::chargeSpeed set to 4
```

Example 7.6 — Forall. This assertion checks if the robot will always finally get stuck. In particular, $[\text{and}]$ are required in order to embed another path formula (Finally) within the outer path formula (Globally).

```plaintext
prob assertion A_stuck:
  A [Globally [Finally "l_stuck"]]
  with constant C1
```
Example 7.7 — Simulation. This example applies statistic model checking to verify the property using the CI method with supplied parameters.

\[
\text{prob assertion P\_stuck\_loc:} \\
\text{Prob=? [Finally \{mod0::ctrl\_ref0::p=x\} \& \& "l\_outOfPower"]} \\
\text{using sim with CI at alpha=0.01, n=2000, and pathlen=1000} \\
\text{with constants C1}
\]
8. Checking core assertions

Along with the CSP semantics of a model, RoboTool automatically generates assertions to check standard properties such as deadlock freedom and determinism. These properties are specified in file with the suffix `_coreassertions.csp`, and can be checked by FDR.

1. The core assertions for the controller created in the previous chapter are contained in the file `mycontroller_coreassertions.csp` in the `src-gen` folder.
2. (Optional) In order to open the file in FDR directly from eclipse, select FDR as the default editor. Right-click the file, and select `[Open With > Other...]`.
3. In the Editor Selection dialog, select `External programs`.
4. Check both “Use this editor for all FILENAME files” and “Use it for all ‘*.csp’ files”.
5. Click `Browse...` to select FDR as the editor.
6. Find the FDR4 executable, and click `OK`.
7. Make sure FDR4 is selected in the Editor Selection dialog, and click `OK`.
8. The last step opens the FDR4 windows with all assertions loaded and displayed on the right-hand side panel.
9. Click the `Run All` button at the top-right corner, and wait for the checks to finish. Alternatively, click each `Check` button to run each assertion separately.
10. If any of the (positive) assertions fail, a counter example is produced. It can be viewed by clicking the `Debug` button of the assertion.

A number of the core assertions of the model created in the previous chapter fail. In particular, all determinism and deadlock freedom checks fail. This is due to the underspecification of the operation `move`, which may or may not terminate. Next, we complete our model with information...
Chapter 8. Checking core assertions

about termination of the move operation.

1. Select the operation definition tool O in the Architectural Constructs section of the palette, and click on the editor.
2. Input the operation signature and click OK.
3. Save the model, right-click the operation definition, and select [RoboChart > Toggle termination].
4. The operation definition label now indicates that the operation move terminates.
5. Reload the file mycontroller_coreassertions.csp on FDR and run the assertions.

The analysis of the updated model only fails in the verification of deadlock freedom of the move operations, which is expected as the operation terminates, and in FDR termination is not distinguished from deadlock.

In order to establish termination, we create two assertions. For example, in order to establish that the operation move terminates, we must show that the process $P_{move}$ is not deadlock free, and that the process $P_{move}$ followed by the process $\text{RUN}(\tau)$ is deadlock free.
This chapter describes the RoboChart extensions designed to support modelling, analysis and simulation of collections of robots. Section 9.1 describes the extensions of the metamodel of RoboChart, Section 9.2 describes the conditions that characterise well-formed RoboChart collections, and Section 9.3 specifies the semantics of collections based on the untimed and timed semantics in Section 4.

The contents of this chapter will be integrated into chapters 2, 3 and 4 when the extension is further validated through examples.

9.1 Metamodel

The metamodel of RoboChart is extended for collections in the following ways:

1. A new construct RCCollection is introduced to describe collections of robots modelled as Modules. Additional auxiliary construct, such as Instantiations, are also provided as part of RCCollection;
2. Events are extended to support the specification of broadcast events;
3. Triggers are extended to use broadcast events by recovering information about the source of the communication as well as by restricting the possible targets. This last feature introduces the possibility of one-to-one or one-to-many communications in a more restrictive form than broadcast (one-to-all);
4. Expressions are extended with two new types of expressions: ToExp and IdExp. They both characterise implicit parameters. The first applies only to state machines and allows the
Figure 9.1: Metamodel for collections: RCCollection and Event

Figure 9.2: Metamodel for collections: Trigger

restriction of communication patterns. The second applies to state machines, controllers and modules and provides a unique identifier for an instance of a module.

Figure 9.1 shows the part of the extensions of the metamodel. Collections are specified by RCCollections, which include:

- A VariableList specifies constant variables that can be used to instantiate the collection. For example, loose constants that bound the number of robots in the collection.
- An Instantiation describes how many instances (range) of a Module (modelling robots) are present, and assigns the instances an index.
- A ModuleRef is a placeholder for a Module and is used to specify how instances of the module can interact with instances of other modules (including instances of the same module).
- A Connection links two place holders and specifies the possible interactions between instances of the source and target place holders.

An Instantiation can include InstantiationParameters that are used to initialise constants of a module. Events are extended with a boolean attribute that is used to determine whether or not it a broadcast event.
Finally, Triggers (shown in Figure 9.2) are extended with two new attributes:

- `_from` is used to identify a variable in which to record the identifier of the source of the communication; and
- `_predicate` is used to restrict the potential targets of communication. For example, an empty predicate is equivalent to `true`, and results on the message being sent to all possible targets (determined by the set of identifiers that characterise the target event), while a predicate such as `from = v` sends the message only to the target whose identifier is recorded in the variable `v`.

### 9.2 Well-formedness Conditions

#### 9.2.1 RCCollection

- **All variables in a collection must be constants.** At the level of collections, variables are only used to instantiate constants of the modules.
- **A collection can contain any number of placeholders, but at most two of the same module.** A placeholder corresponds to any instance of the module in a collection, and since we do not allow concrete identification of instances in the diagram, the can be at most two placeholders of the same type, identifying two different, but otherwise unspecified, instances.
- **Connections between placeholders of the same module must be bidirectional.** The semantics of connections `c` between placeholders of the same module `M` is summarised by "any two different instances of `M` can interact with each other via the connection `c`". This semantics essentially equate

#### 9.2.2 Instantiation

- **The range of an instantiation must be a bounded set.** While we do allow the use of loose constants in the specification of the range, for any value that the constants can take, the set of indices must be finite.

#### 9.2.3 Event

- **Events connected in a collection must be broadcast events.** Events connected in a collection model some form of communication, which in its most general form is a broadcast. Further restriction over the patterns of communication can be modelled internally using the `_predicate` attribute of triggers.
- **Connections (at any level) involving a broadcast event in one end must link to another broadcast event.** The broadcast nature of the event is part of its type and affects the semantics of triggers, therefore the ends of the connection must be compatible.
9.2.4 Trigger

- A transition trigger must not record a value for \( \_\text{predicate} \). In the context of broadcast communications, transition triggers are interpreted as input communications, in which restriction of the targets is meaningless.

- A send event statement trigger must not record a value for \( \_\text{from} \). In the context of broadcast communications, send event statements are interpreted as output communications, in which case the source identifier obtained through the \( \_\text{from} \) attribute is redundant, as it is the identifier of the machine that contains the send event statement.

9.3 Semantics

**Rule 167. Semantics of Collections**

\[
\begin{align*}
[c : \text{RCCollection}]_{\text{Coll}} : \text{CSPProcess} &= \\
&\left\{\begin{array}{l}
\hline
\text{inst} : c.\text{instantiations} \bowtie i : \text{inst.range} \bowtie [\text{inst.module}]_M^i

\hline
\{[e_1, e_2 | (e_1, e_2) \leftarrow \text{connectedEvents}(c)]\}

\hline
\text{conn} : c.\text{connections} \bowtie (i, j) : \text{inds}(c, c) \bowtie

\begin{align*}
\text{BBuffer}^1 \left( \text{eventId}(\text{conn}.\text{efrom}), i, \text{eventId}(\text{conn}.\text{eto}), j \right)

\hline
\text{conn} : c.\text{connections} \bowtie \text{conn.bidirec} \wedge \text{heterogeneous} \bowtie (i, j) : \text{inds}(c, c) \bowtie

\begin{align*}
\text{BBuffer}^2 \left( \text{eventId}(\text{conn}.\text{eto}), j, \text{eventId}(\text{conn}.\text{efrom}), i \right)
\end{align*}
\end{align*}
\end{array}\right. \\
\right\}
\]
Rule 168. Broadcast Buffer

\[ \text{BBuffer}(e_\text{in} : \text{Event}, i : \text{ID}, e_\text{out} : \text{Event}, j : \text{ID}) : \text{CSPProcess} = \]

\[
\text{let}
\]

\[
\text{BufferEmpty} = \text{prefixIn} \rightarrow \text{BufferFull}(x)
\]

\[
\text{BufferFull}(v) = \text{prefixIn} \rightarrow \text{BufferFull}(x) \oplus \text{prefixOut} \rightarrow \text{BufferEmpty}
\]

\[
\text{within}
\]

\[
\text{BufferEmpty}
\]

\[
\text{where}
\]

\[
e_\text{in}, \text{broadcast}
\]

\[
\text{prefixIn} = \text{if} e_\text{in}.\text{type} \neq \text{null then eventId(e_\text{in})} \cdot x \text{ else eventId(e_\text{in})}
\]

\[
\text{prefixOut} = \text{if} e_\text{out}.\text{type} \neq \text{null then eventId(e_\text{out})} \cdot v \text{ else eventId(e_\text{out})}
\]

Rule 169. Semantics of triggers

\[
[[t : \text{Trigger}]]^t \text{Trigger} : \text{CSPProcess} =
\]

\[
\text{if} t.\text{type} = \text{INPUT then}
\]

\[
\text{eventId}(t.\text{event}).\text{tid} \cdot \text{if} \_ \cdot \text{id} \cdot \text{?} \cdot x \cdot \text{set_vid(t.from)} \cdot \text{if} \_ \cdot \text{set_vid(t.parameter)} \cdot !x \rightarrow \text{Skip}
\]

\[
\text{else if} t.\text{type} = \text{SIMPLE then}
\]

\[
\text{eventId}(t.\text{event}).\text{tid} \cdot \text{if} \_ \cdot \text{id} \cdot \text{?} \cdot x \cdot \text{set_vid(t.from)} \cdot !f \rightarrow \text{Skip}
\]

\[
\text{else These cases do not occur when the event is broadcast}
\]

\[
\text{where}
\]

\[
t.\text{event}.\text{broadcast}
\]

Rule 170. Semantics of send event statements

\[
[[s : \text{SendEvent}]]^s \text{Statement} : \text{CSPProcess} =
\]

\[
\text{if} t.\text{type} = \text{OUTPUT} \lor t.\text{type} = \text{SYNC then}
\]

\[
\text{||| i : \{ x : \text{ID} \ | \ [t.\_predicate]^{t.\text{predicate}}}\}_{\text{Exp}^{t.\text{predicate}}} \cdot \text{eventId}(t.\text{event}) \cdot \text{id} \cdot \text{?} \cdot t.\text{value} \cdot \text{Exp}^{t.\text{value}} \rightarrow \text{Skip}
\]

\[
\text{else if} t.\text{type} = \text{SIMPLE then}
\]

\[
\text{||| i : \{ x : \text{ID} \ | \ [t.\_predicate]^{t.\text{predicate}}}\}_{\text{Exp}^{t.\text{predicate}}} \cdot \text{eventId}(t.\text{event}) \cdot \text{id} \cdot !i \rightarrow \text{Skip}
\]

\[
\text{else These cases do not occur when the event is broadcast}
\]

\[
\text{where}
\]

\[
s.\text{trigger}.\text{broadcast}
\]
We have presented RoboChart, a diagrammatic notation for modelling of robotic systems. It is based on UML state machines, but includes the notions of robotic platform and controller, synchronous and asynchronous communications, an API of operations common to autonomous and mobile robots, a well defined action language, pre and postconditions, and time primitives. It also has a formal semantics suitable for verification. Examples of RoboChart models and their verification can be found at [www.cs.york.ac.uk/circus/RoboCalc/](http://www.cs.york.ac.uk/circus/RoboCalc/).

We have described the semantics for the core constructs of RoboChart. It uses CSP, but we envisage its extension to use Circus [3], a process algebra that combines Z [15] and CSP, and includes time constructs [6]. Use of Circus and its UTP foundation will enable use of theorem proving as well as model checking.

An approach for writing object-oriented simulations of RoboChart diagrams has also been defined. Automatic generation of simulations is possible and part of our future work. Verification of correctness of simulations will use the object-oriented version of Circus [4], with a semantics given by the UTP theory in [11].

RoboChart itself misses support for modelling the environment and the robotic platforms in model detail. It is also in our plans to take inspiration from hybrid automata [29] to extend the notation, and from the UTP model of continuous variables [27] to define the semantics.
Appendices

A  RoboChart diagrams - an informal overview .......................... 129
A.1  Time primitives

B  Complete Metamodel ..................... 139

C  Mathematical Toolkit ..................... 149

D  OCL Well-formedness Conditions ....... 155
A. RoboChart diagrams - an informal overview

To illustrate the concepts, we present the model of a robot for chemical detection based on that in [14] ¹. In our example, the robot employs a random walk and, upon detection of a chemical source, it turns on a light and drops a flag.

A robotic system is specified in RoboChart by a module, where a robotic platform is connected to one or more controllers. A robotic platform is characterised by variables, operations, and events representing its in-built facilities. For our example, the module ChemicalDetector is shown in Figure A.1, where we have a robotic platform named Vehicle and two controllers named MainController and MicroController.

Vehicle declares a number of events via named boxes on its border. The event flag is used to request an in-built flag holder to drop a flag. The events obstacle and odometer represent two sensors, one monitoring obstacles in front of the vehicle, and the other providing an estimation of the distance travelled. Finally, the event gas represents an array of in-built sensors that detect the type and intensity of gases.

An interface Operations groups operation declarations. The operation move(lv,α) takes a linear velocity lv and an angle α as parameters; it moves the vehicle forward at speed lv while turning by α degrees. The type of lv is real, and that of α is Angle, which is an enumerated type, including values left, right, front, and back for simplicity. In RoboChart, we can also define given types (uninterpreted sets), record types, and other structured types. The primitive types include numbers and strings. The operation randomWalk() carries out a random walk, and potentially does not terminate. The shortRandomWalk() operation, on the other hand, is a

¹http://tinyurl.com/hdaws7o
random walk that is guaranteed to terminate.

The operations \texttt{move}(lv,a) and \texttt{shortRandomWalk()} are defined separately, just to indicate that they terminate; \texttt{randomWalk()} is left undefined. Further elaboration of the model may include a definition for these operations via state machines, or via pre and postconditions. Such definitions would have the purpose to support reasoning, but since these are operations that we declare to be provided by the platform, they do not need to be implemented. The fact that \texttt{Vehicle} declares \texttt{Operations} as a provided (P) interface makes this clear.

The \texttt{Vehicle} behaviour is defined by the two controllers \texttt{MainController} and \texttt{MicroController} referenced in the module and defined in other diagrams. \texttt{MainController} uses \texttt{gas} to detect gases, and events \texttt{turn}, \texttt{stop} and \texttt{resume} to control the trajectory followed by the vehicle. The last three events are internal to the module and passed to \texttt{MicroController}, which implements the associated behaviours using the \texttt{move}(lv,a) operation, whilst avoiding obstacles.

\texttt{MicroController} implements obstacle avoidance using the events \texttt{obstacle} and \texttt{odometer}, implements the movement behaviors (\texttt{turn}, \texttt{stop} and \texttt{resume}) and drops a flag when a specific gas is found. The interactions between controllers and between a controller and the robotic platform are specified by arrows connecting the appropriate events. The directions of the arrows indicate the flow of information. For instance, when the \texttt{Vehicle} finds a chemical, it sends a sequence of \texttt{GasSensor} values through the \texttt{gas} event to \texttt{MainController}.

Communication with a robotic platform is always asynchronous, but communication between controllers can be synchronous or asynchronous. In our example, the communication between the controllers and the platform via the \texttt{gas}, \texttt{obstacle}, \texttt{flag}, and \texttt{odometers} events is asynchronous, as indicated by the label \texttt{async} on the arrows that represent the connections. That label is used on all connections with a robotic platform.
The connections between the controllers are all synchronous in this example. This is an abstraction, since typically controllers of a robot communicate asynchronously.

As mentioned, MainController and MicroController are defined in other diagrams, and referenced in the module ChemicalDetector. The diagrams are shown in Figures A.2 and A.3.

The behaviour of a controller is specified by one or more parallel state machines. They use variables, operations, and events that are either defined locally or required from the platform. Required interfaces identify the outer definitions that can be used. The micro-controller in Figure A.3, for instance, requires the operations provided by the platform to move the robot.

The events of a controller can be connected to those of the state machines that defines it. Communication between states machines is always synchronous, since parallelism at this level is used for convenience of modelling, rather than to indicate concrete designs.
MainController is defined by a single state machine GasAnalysis. It is referenced in the definition of MainController in Figure A.2, and defined in Figure A.4. The controller in this case just relays its events to and from the state machine.

GasAnalysis initially waits in state NoGas for an event gas communicating a value gs from the sensors. When it gets that value, it analyses it using a function call analysis(gs) to determine its nature. If there is no gas, the machine returns to the state NoGas sending a command (via an event) to the MicroController to resume the random walk. Otherwise, it moves to the state GasDetected, where it determines the intensity of the gas, using a function call intensity(gs).

If there is enough gas, indicated by an intensity greater than or equal to that of a threshold constant thr defined in the machine, it instructs the vehicle to stop using the event stop and terminates (entering the final state (F)). In this scenario, the robot found the gas. Otherwise, the machine calculates the direction of the detected gas (using the function call location(gs)) and instructs the vehicle to turn in that direction using the event turn before going to the state Reading. In that state, it reads a new value from the gas sensors for analysis.

The controller MicroController is also defined by a single state machine Movement. It also relays events to and from its state machine. It also defines an operation changeDirection(l) used by Movement when it finds an obstacle.

Movement avoids obstacles while receives events turn, resume and stop to control the movement of the vehicle. The avoidance mechanism uses the odometer event as well as a clock to detect situations where the vehicle becomes stuck. In this case, it takes special measures to leave the area before resuming it main behaviour of treating movement requests.

The strategy can be summarised as follows. When an obstacle is first detected, a clock is reset and...
the distance travelled so far is recorded before the obstacle is avoided. If, after the avoidance action, another obstacle is detected, the machine checks whether enough time has elapsed since the first obstacle, but the vehicle has not moved enough, and in this case it takes measures to get out of the area. Otherwise, it resumes its normal activity.

In most states, except while actively avoiding an obstacle, the machine can respond to requests to turn. Additionally, it may receive requests to start a random walk (using the event resume) as well as to stop the vehicle, in which case it requests a flag to be dropped and terminates. In the next section we explain in more detail the time constructs used in Movement.

RoboChart state machines are standard, but restricted and with a well defined semantics. They can have composed states, junctions, and entry, during, exit, and transition actions defined using a well defined action language. Features of UML state machines [23] deemed not essential for robotics are not included, resulting in a streamlined semantics.

Definitions of a model can be organised in packages. Like in UML, they are just containers. They do not correspond to a concept or abstraction, and so do not have an interface. An imports mechanism controls scope of the definitions. All packages that do not have a package name conceptually compose the same package. Elements defined in the unnamed package are available in all other packages. Elements defined in a package with a name can only be used if they are explicitly imported. A model is identified by one module and all the other elements there. Figures A.6 and A.7 define two packages used in our example.
In *Chemical*, we specify a data model for handling the gas sensors. This involves a number of types, some of which are just named, like `Chem`, and a number of functions acting on those types. Functions are either left undefined, like `greater`, or defined by pre and postconditions, like `intensity`. To define these conditions, RoboChart provides a simple predicate language.

In the package *Location*, we define the operation `changeDirection(l)` used in the MicroController. An API can provide a collection of such definitions organised in packages.
A.1 Time primitives

RoboChart operations take zero time, and enabled transitions take place as soon as they can be triggered. Time constraints need to be explicitly defined. In Table 2.3 we summarize the syntax of all timed constructs that can be used in the definition of state machines.

The timed budget \( b \) for an action \( A \) can be specified by sequentially composing \( A \) with the action \( \text{wait}(b) \), which waits for \( b \) time units. In the machine Movement of the chemical detector (see Figure A.5), we compose the \( \text{shortRandomWalk()} \) and \( \text{changeDirection(l)} \) calls with \( \text{wait(outPeriod)} \) and \( \text{wait(evadeTime)} \), where \( \text{outPeriod} \) and \( \text{evadeTime} \) are constants.

In the case of \( \text{changeDirection(l)} \), the software operation is very simple (see Figure A.7). It involves a condition on the value of \( l \) and a call to the \( \text{move(l,v,a)} \) operation, which is likely to involve just a simple assignment to actuator registers. So, the execution time of \( \text{changeDirection(l)} \) is negligible. The \( \text{wait(evadeTime)} \) action, in this case, represents the amount of time the software should wait for the effect of that change of direction to take place.

In the case of \( \text{shortRandomWalk()} \), although this operation is not defined, we expect it to take some time to actually effect the walk. So, \( \text{wait(outPeriod)} \) records the amount of time we expect this operation to take. More realistically, we should give a range of time here, as it is very difficult to predict the exact amount of time an operation can take. This is possible in RoboChart by specifying a (closed or open) interval of time when using the \( \text{wait} \) action. In our example, we have a deterministic budget, for simplicity.

A deadline of \( d \) time units for an action \( A \) is specified by \( A < \{d\} \), while a deadline on an event \( e \) is specified by \( e < \{d\} \). Clocks allow transitions to be guarded by constraints relative to the occurrence of clock resets and the entering of a state. For that, we can use in guards the expressions \( \text{since}(C) \), which yields the elapsed time since the most recent reset \( \#C \) of clock \( C \), and \( \text{sinceEntry}(S) \), which yields the time elapsed since entering state \( S \).

Similarly to timed automata, expressions involving clocks are restricted to comparing single timed primitive with constant expressions. We, however, allow conjunctive as well as disjunctive expressions involving more than one clock.

To further illustrate the time primitives, we consider a robot that moves at constant speed in a square pattern while avoiding obstacles. The state machine is shown in Figure A.8. We omit the simple module and controller, and operation definitions that just specify termination.

![State machine diagram](image)

When the robot is started, it transitions from the initial state, denoted by a black circle, to the state MovingForward, while resetting \( \#C \) a clock \( C \) and assigning \( 1 \) to a local variable segment. A RoboChart state machine is self-contained, in that it declares all the variables, events, and operations that it uses. In Figure A.8 two constants linear and angular are defined, to represent the linear
Chapter A. RoboChart diagrams - an informal overview

Figure A.8: Square-trajectory robot

and angular speed, respectively. The local variable segment records how many sides of the square have been covered so far; the robot stops when it completes the square (segment == 4). This is achieved by sending an event $\text{stop}$ to the platform and transitioning to the final state: a white circle. The event $\text{stop}$ is given a deadline 0, indicating that it is expected that the robotic platform is always ready to accept this event immediately.

The state MovingForward is composite. In this state, the motion is linear, unless an obstacle is detected. Linear motion is activated by calling the operation moveForward(linear) in the entry action with a constant value linear passed as a parameter.

Before MovingForward is actually entered, its entry action executes, followed by that of its substate Observing, enabling the collision detection capability. Once a collision is detected, the event collisionDetected is raised by the robotic platform: the transition from Observing to the state Collision is then triggered, executing the exit action of Observing and subsequently the avoid operation that performs the actual collision avoidance. Here we do not specify this operation, but record its budget of 2 time units by sequentially composing it with the timed primitive wait(2).

In RoboChart time elapses explicitly via budgets, unless a state has been entered and no transitions are enabled, or, every enabled transition is associated with an external event. Once the collision is resolved, a transition back to Observing is taken. Transitions are triggered once the guard is true.
and the associated event is raised, or, if there is no event or guard associated, immediately, as in this example.

The square motion pattern is achieved by limiting the linear motion to 5 time units before switching to angular motion for 2 time units, and then switching again to linear motion. Accordingly, we guard the transition from MovingForward to the state Turning with the expression since(C) == 5. Upon such a transition, the value of segment is incremented. Similarly, the angular motion is limited by guarding the transition from Turning to MovingForward using the timed primitive sinceEntry(Turning). Upon this transition, clock C is reset.

When entering the Turning state, the event stop is used to stop the robot before turning. This is an event, and so (may) require synchronisation to happen, and so, it may take time. The deadline 0, however, enforces that it must take place immediately. Since stop is actually an event of the platform (omitted here), this is simple to achieve, because the connection with the platform is asynchronous. In any case, the deadline makes the properties of the state machine independent of whether its stop event is in a synchronous or asynchronous connection.
This appendix contains the complete metamodel specified in Ecore and formatted by the tool OCLinEcore. The syntax of the representation used in this appendix is available here.

A summary of the concepts of Ecore can be found here, and a tutorial is available here.

While this metamodel leaves the class of Expressions abstract, as this class is defined in the Z Standard [1], it includes the Expression subclasses: Application, as an illustration of the embedding of Z in the metamodel of RoboChart, and the timed expressions ClockExp and StateClockExp, which are unique to RoboChart. The definition of Application depends on the notion of Template, which represents (potentially infix) operators and is only partially described here (for the full description, see [1]).

```java

package robochart : robochart = 'http://www.robocalc.circus/RoboChart' {
    class BasicPackage {
        attribute name : String[?];
        property imports : Import[*] { composes };
    }
    class RCPackage extends BasicPackage {
        property interfaces : Interface[*] { composes };
        property robots : RoboticPlatformDef[*] { composes };
        property types : TypeDecl[*] { composes };
        property machines : StateMachineDef[*] { composes };
    }
}```
property controllers : ControllerDef[*] { composes };
property modules : RCModule[*] { composes };
property operations : OperationDef[*] { composes };
property functions : Function[*] { composes };
}
class Import
{
    attribute importedNamespace : String[1];
}
abstract class NamedElement
{
    attribute name : String[1];
}

/*
 * Definition of type declarations
 */
abstract class TypeDecl extends NamedExpression;

/*
 * Primitive Types
 */
class PrimitiveType extends TypeDecl;

/*
 * Record Types
 */
class RecordType extends TypeDecl
{
    property fields : Field[*] { composes };
}
class Field extends Member.NamedExpression;
abstract class TypedNamedElement extends NamedElement
{
    property type : Expression[1] { composes };
}
abstract class Member extends TypedNamedElement;

/*
 * Enumeration Types
 */
class Enumeration extends TypeDecl
{
    property literals : Literal[*|1] { composes };
}
class Literal extends TypeDecl.NamedExpression
{
    property types : Expression[*] { composes };
}
/* 
* Named Types 
*/
class NamedType extends TypeDecl 
{
  property type : Expression[1] { composes };
}

/* 
* Variables 
*/
class VariableList 
{
  attribute modifier : VariableModifier[1];
  property vars : Variable[*] { composes };
}
enum VariableModifier { serializable }
{
  literal VAR;
  literal CONST;
}
class Variable extends TypedNamedElement,Member,Mutable 
{
  property initial : Expression[?] { composes };
  attribute modifier : VariableModifier[1] { derived transient volatile 
    };
}

/* 
* Events 
*/
class Event extends NamedElement 
{
  property type : Expression[?] { composes };
  attribute broadcast : Boolean[1];
}

/* 
* Functions 
*/
class Function extends TypedNamedElement,NamedExpression 
{
  property parameters : Parameter[*] { ordered composes };
  property preconditions : Expression[*] { composes };
  property postconditions : Expression[*] { composes };
}
class Parameter extends Variable;

/* 
* Operations 
*/
class OperationSig extends NamedElement {
    attribute terminates : Boolean[1];
    property parameters : Parameter[*] { ordered composes };  
    property preconditions : Expression[*] { composes };    
    property postconditions : Expression[*] { composes }; 
}

abstract class Operation extends NamedElement, ConnectionNode;
class OperationDef extends Operation, OperationSig,StateMachineBody;

/*
* The Reference class identifies reference constructs
*/
abstract class Reference;
class OperationRef extends Operation,Reference {
    property ref : OperationDef[1];
}

/*
* Interfaces
*/
class Interface extends NamedElement, BasicContext;
abstract class BasicContext {
    property variableList : VariableList[*] { composes }; 
    property operations : OperationSig[*] { composes }; 
    property events : Event[*] { composes }; 
    property clocks : Clock[*] { composes }; 
}

/*
* Robotic Platforms
*/
abstract class RoboticPlatform extends NamedElement, ConnectionNode;
class RoboticPlatformDef extends Context,RoboticPlatform;
abstract class Context extends BasicContext {
    property pInterfaces : Interface[*] { !unique }; 
    property rInterfaces : Interface[*] { !unique }; 
    property interfaces : Interface[*] { !unique }; 
}
class RoboticPlatformRef extends RoboticPlatform,Reference {
    property ref : RoboticPlatformDef[1];
}

/*
* State Machines
*/
abstract class StateMachine extends NamedElement, ConnectionNode;
class StateMachineDef extends StateMachineBody, StateMachine;
class StateMachineRef extends StateMachine, Reference
{
    property ref : StateMachineDef[1];
}
class StateMachineBody extends Context, NodeContainer;
class Clock extends NamedElement;
abstract class NodeContainer
{
    property nodes : Node[*] { composes };  
    property transitions : Transition[*] { composes };
}
abstract class Node extends NamedElement;
class Junction extends Node;
class Initial extends Junction;
class State extends Node, NodeContainer
{
    property actions : Action[*] { composes };
}
class Final extends State;
class ProbabilisticJunction extends Junction;
class Transition extends NamedElement
{
    property source : Node[1];
    property target : Node[1];  
    property trigger : Communication[*] { composes };
    property deadline : Expression[*] { composes };
    property condition : Expression[*] { composes };
    property action : Statement[*] { composes };
    property probability : Expression[*] { composes };
    property reset : ClockReset[*] { ordered composes };
}
class Communication
{
    property event : Event[*];
    property _from : Variable[*];
    property _predicate : Expression[*] { composes };
    property parameter : Variable[*];
    property value : Expression[*] { composes };
    attribute _type : CommunicationType[1];
}
enum CommunicationType { Serializable }
{
    literal SIMPLE;
    literal INPUT;
    literal OUTPUT;
    literal SYNC;
abstract class Action {
    property action : Statement[1] { composes };
}
class EntryAction extends Action;
class DuringAction extends Action;
class ExitAction extends Action;

abstract class Controller extends NamedElement, ConnectionNode;
class ControllerDef extends Context.Controller {
    property machines : StateMachine[*] { composes };

    /*
    * These are the operations defined or referenced locally in the
    controller
    */
    property l0Operations : Operation[*] { composes };
    property connections : Connection[*] { composes };
}
class Connection {
    property from : ConnectionNode[1];
    property to : ConnectionNode[1];
    property efrom : Event[1];
    property eto : Event[1];
    attribute async : Boolean[1];
    attribute bidirec : Boolean[1];
}
class ControllerRef extends Controller {
    property ref : ControllerDef[1];
}

class RCModule extends NamedElement {
    property connections : Connection[*] { composes };
    property nodes : ConnectionNode[*] { composes };
}
abstract class Statement;
class TimedStatement extends Statement {
    property stmt : Statement[1] { composes };
    property deadline : Expression[1] { composes };
}
class Wait extends Statement {
    property duration : Expression[1] { composes };
}
class Skip extends Statement;
class IfStmt extends Statement {
    property expression : Expression[1] { composes };
    property _'then' : Statement[1] { composes };
    property _'else' : Statement[?] { composes };
}
class Assignment extends Statement {
    property left : Assignable[1] { composes };
    property right : Expression[1] { composes };
}
class CommunicationStmt extends Statement {
    property communication : Communication[1] { composes };
}
class SeqStatement extends Statement {
    property statements : Statement[2..*] { ordered composes };
}
class ParStmt extends Statement {
    property stmt : Statement[1] { composes };
}
class Call extends Statement {
    property operation : OperationSig[1];
    property args : Expression[*] { ordered composes };
}
class ClockReset extends Statement {
    property clock : Clock[1];
}

/*
* Templates
Chapter B. Complete Metamodel

*/
abstract class Template;
class PrefixTemplate extends Template;
class PostfixTemplate extends Template;
class InfixTemplate extends Template;
class NofixTemplate extends Template;
abstract class OperatorTemplate;

/*@ Expression */
abstract class Expression;

/*@ Application includes the class of Relations of Z. */
abstract class Application extends Expression;
class PrefixApp extends Application
{
    property operator : PrefixTemplate[1];
    property arguments : Expression[+]{ ordered composes };
}
class PostfixApp extends Application
{
    property operator : PostfixTemplate[1];
    property arguments : Expression[+]{ ordered composes };
}
class InfixApp extends Application
{
    property operator : InfixTemplate[1];
    property arguments : Expression[+]{ ordered composes };
}
class NofixApp extends Application
{
    property operator : NofixTemplate[1];
    property arguments : Expression[+]{ ordered composes };
}

/*@ Timed Expression */
class ClockExp extends Expression
{
    property clock : Clock[1];
}
class StateClockExp extends Expression
{
    property state : State[1];
}
/ * * Assignable */
abstract class Assignable;
class VarSelection extends Assignable {
   property receiver : Assignable[1] { composes };
   property member : Member[1];
}
class ArrayAssignable extends Assignable {
   property value : Assignable[1] { composes };
   property parameters : Expression[+] { composes };
}
class VarRef extends Assignable {
   property name : Mutable[1];
}
abstract class ConnectionNode;
abstract class NamedExpression;
class WaitingCondition extends NamedElement {
   property expression : Expression[?];
   property transitions : Transition[*] { ordered };
}
class WaitingConditionRef extends Expression {
   property ref : WaitingCondition[?];
}
class FieldDefinition {
   property field : Field[1];
   property value : Expression[1] { composes };
}
abstract class Mutable extends NamedElement.NamedExpression;
This section presents the functions of the Z mathematical toolkit as modelled in RoboChart.

A package named core contains a number of primitive types and is imported by default in every RoboChart model. They are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural numbers</td>
<td>nat</td>
</tr>
<tr>
<td>Integers</td>
<td>int</td>
</tr>
<tr>
<td>Strings</td>
<td>string</td>
</tr>
<tr>
<td>Booleans</td>
<td>boolean</td>
</tr>
<tr>
<td>Real numbers</td>
<td>real</td>
</tr>
</tbody>
</table>

Table 3.1: Primitive types in core package.

The functions are grouped in the set, relation, function, and sequence toolkits.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>package set_toolkit</code></td>
<td></td>
</tr>
<tr>
<td><code>\text{Union}(A: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td><code>\text{Inter}(A: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td><code>\text{diff}(s1: \text{Set}(?X), s2: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td><code>\text{symetric\_diff}(s1: \text{Set}(?X), s2: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td><code>\text{subseteq}(ss: \text{Set}(?X), s: \text{Set}(?X)): \text{boolean}</code></td>
<td><code>\text{result} = \{ \forall x: ?X . x \in s \}</code></td>
</tr>
<tr>
<td><code>\text{subset}(ss: \text{Set}(?X), s: \text{Set}(?X)): \text{boolean}</code></td>
<td><code>\text{result} = \{ \text{subseteq}(ss, s) \land ss \neq s \}</code></td>
</tr>
<tr>
<td><code>\text{union}(s1: \text{Set}(?X), s2: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td><code>\text{inter}(s1: \text{Set}(?X), s2: \text{Set}(?X)): \text{Set}(?X)</code></td>
<td>`\text{result} = { x: ?X</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>f</code> isFinite(s: Set(?X)): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isTotal(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isFiniteFunction(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isFinitelInjection(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isBijection(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isTotalSurjection(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isTotalInjection(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> partitions(f: ?L &lt;= Set(?X), a: Set(?X)): boolean</td>
<td></td>
</tr>
<tr>
<td><code>f</code> isSurjection(f: ?X -&gt; ?Y): boolean</td>
<td></td>
</tr>
</tbody>
</table>
D. OCL Well-formedness Conditions

import robochart : 'robochart.ecore'
import 'http://www.eclipse.org/emf/2002/Ecore'

package robochart

-- Robotic Platforms well-formedness conditions (RoboChart reference 3.1.1)
context RoboticPlatformDef

-- We note that variables and operations declared directly in the
-- platform,
-- outside an interface, are considered as if declared in a provided
-- interface, for the reasons already explained above. Events declared
-- directly in the platform, on the other hand, are defined.
def: rpProvidedVars() : Bag(Variable) =
    self.pInterfaces.variableList.vars->union(self.variableList.vars)
def: rpProvidedOps() : Bag(OperationSig) = self.pInterfaces.operations->
    union(self.operations)
def: rpDefinedEvents() : Bag(Event) = self.interfaces.events->union(self.
    events)

-- RP1: Robotic platforms cannot require interfaces
inv RP1: self.interfaces->isEmpty()

-- RP2: Defined interfaces can only have events
inv RP2: self.interfaces->forall(i |
    i.variableList->isEmpty() and i.operations->isEmpty())

-- RP3: The names of variables, operations, and events are unique to the
-- platform
inv RP3: self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
    pInterfaces))
variableList->union(self.variableList).vars.name
->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
.operations->union(self.operations).name)
->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
.events->union(self.events).name)->isUnique(i | i)

-- Interfaces well-formedness conditions (RoboChart reference 3.1.2)
context Interface
  -- I1: Provided and required interfaces contain only variables and operations
inv I1: Context.allInstances()->collect(c | c.pInterfaces)->union(c.rInterfaces)->includes(self)
  implies self.events->isEmpty();

-- I2: Defined interfaces contain only variables and events
inv I2: Context.allInstances().interfaces->includes(self)
  implies self.operations->isEmpty();

-- I3: Names of variables, events and operations are unique
inv I3: self.variableList.vars.name
  ->union(self.operations->asBag().name)
  ->union(self.events->asBag().name)->isUnique(i | i)

-- Modules well-formedness conditions (RoboChart reference 3.1.3)
context RModule
  def: moduleControllers(): Set(Controller) =
    self.nodes->selectByKind(Controller)
def: moduleRP(): RoboticPlatform =
    self.nodes->selectByKind(RoboticPlatform)->any(true)

-- M1: A module must contain exactly one robotic platform, at least one
  controller, and not state machines
inv M1: self.nodes->selectByKind(RoboticPlatform)->one(true)
  and self.nodes->selectByKind(Controller)->exists(true)
  and self.nodes->selectByKind(StateMachine)->isEmpty();

-- M2: All variables and operations required by the module's controllers
  must be provided by the platform
inv M2: self.nodes->selectByKind(RoboticPlatform)->exists(true)
  implies (self.moduleRP().rpDef().rpProvidedVars()
    ->includesAll(self.moduleControllers().controllerDef().
      controllerRequiredVars())
  and self.moduleRP().rpDef().rpProvidedOps()
    ->includesAll(self.moduleControllers().controllerDef().
      controllerRequiredOps()));

-- M3: Each event on the robotic platform and controllers of a module
  must have at most one connection to or from it within the module
inv M3: self.nodes->selectByKind(RoboticPlatform)->exists(true)
  implies self.moduleRP().rpDef().rpDefinedEvents();
->forall(e | self.connections->select(c | c.efrom = e or c.eto = e)->size() <= 1)
and self.moduleControllers().controllerDef().controllerDefinedEvents()->
forall(e | self.connections->select(c | c.efrom = e or c.eto = e)->size() <= 1)

-- Connection well-formedness conditions (RoboChart reference 3.1.4)
context Connection
-- Cn1: Connections of a module must associate only events of the robotic
   platform and its controllers
inv Cn1: RCModule.allInstances()->select(m | m.connections->includes(self))
  ->forall(m | m.nodes->includes(self.from) and m.nodes->includes(self.to))
-- Cn2: Connections involving a robotic platform are always asynchronous
inv Cn2: (self.from.oclsIsKindOf(RoboticPlatform) or self.to.oclsIsKindOf(RoboticPlatform))
implies self.async
-- Cn3: Connections of a controller must associate only its events and
   those of its state machines
inv Cn3: ControllerDef.allInstances()->select(c | c.connections->includes(self))
  ->forall(c | c.machines->including(c)->includes(self.from)
   and c.machines->including(c)->includes(self.to))
-- Cn4: Only events of the same type may be connected
-- NOTE: this requires and equality operator on RoboChart types
-- Cn5: Bidirectional connections of a module may only involve events of
   a controller which are connected by bidirectional connections within
   the controller
inv Cn5: (self.bidirec and RCModule.allInstances())->exists(m | m.
   connections->includes(self)))
implies (self.from.oclsIsKindOf(Controller)
implies self.from.oclsAsType(Controller).controllerDef().connections
  ->select(c | (c.from = self.from and c.efrom = self.efrom)
   or (c.to = self.from and c.eto = self.efrom))
  ->forall(c | c.bidirec))
and (self.to.oclsIsKindOf(Controller)
implies self.to.oclsAsType(Controller).controllerDef().connections
  ->select(c | (c.from = self.to and c.efrom = self.eeto)
   or (c.to = self.to and c.eto = self.eeto))
  ->forall(c | c.bidirec))
-- Cn6: Non-bidirectional connections of a module may only connect to
   events of a controller which have a non-bidirectional connection from
   them within the controller
inv Cn6: (not self.bidirec
   and RCModule.allInstances())->exists(m | m.connections->includes(self))
and self.to.oclIsKindOf(Controller)
) implies self.to.oclAsType(Controller).controllerDef().connections
  -> select(c | (c.from = self.to and c.efrom = self.etoo)
  or (c.to = self.to and c.etoo = self.eeto))
  ->forall(c | not c.bidirec and c.from = self.to)
-- Cn7: Non-bidirectional connections of a module may only connect from
-- events of a controller which have a non-bidirectional connection to
-- them within the controller
inv Cn7: (not self.bidirec
    and RModule.allInstances()->exists(m | m.connections->includes(self))
and self.from.oclIsKindOf(Controller)
) implies self.from.oclAsType(Controller).controllerDef().connections
  -> select(c | (c.from = self.from and c.efrom = self.efrom)
  or (c.to = self.from and c.etoo = self.etoo))
  ->forall(c | not c.bidirec and c.to = self.from)
-- Cn8: Non-bidirectional connections of a controller must not connect to
-- events that a state machine uses as an output.
inv Cn8: (not self.bidirec
    and ControllerDef.allInstances()->exists(c | c.connections->includes(self))
and self.to.oclIsKindOf(StateMachine)
) implies self.to.oclAsType(StateMachine).stmDef().ncOutputEvents() ->
  excludes(self.etoo)
-- Cn9: Non-bidirectional connections of a controller must not connect
-- from events that a state machine uses as an input
inv Cn9: (not self.bidirec
    and ControllerDef.allInstances()->exists(c | c.connections->includes(self))
and self.from.oclIsKindOf(StateMachine)
) implies self.from.oclAsType(StateMachine).stmDef().ncInputEvents() ->
  excludes(self.efrom)

-- Controllers well-formedness conditions (RoboChart reference 3.1.5)
context ControllerDef
-- Variables and events declared directly in the controller are
-- considered
-- as part of a defined interface.
def: controllerDefinedVars() : Bag(Variable) =
  self.interfaces.variableList.vars->union(self.variableList.vars)
def: controllerDefinedEvents() : Bag(Event) = self.interfaces.events->
  union(self.events)
def: controllerRequiredVars() : Bag(Variable) = self.rInterfaces.
  variableList.vars
def: controllerRequiredOps() : Bag(OperationSig) = self.rInterfaces.
  operations

-- C1: A controller must contain at least one state machine
inv C1: self.machines ->exists(true)
-- C2: Controllers cannot provide variables or operations to other
controllers

inv C2: self.pInterfaces ->collect(i | i.variableList.vars ->union(i.operations)) ->isEmpty()

-- C3: All variables required by the controller's state-machines must be defined or required by the controller
inv C3: self.controllerRequiredVars() ->union(self.controllerDefinedVars())

-- includesAll(self.machines.stmDef().stmRequiredVars())

-- C4: All operations required by the controller's state-machines must be required or defined by the controller
inv C4: self.controllerRequiredOps() ->union(self.10Operations)

-- includesAll(self.machines.stmDef().stmRequiredOps())

-- C5: The names of variables, operations, and events are unique to the controller
-- operations declared directly in the controller are ruled out by C7
inv C5: self ->collect(c | c.interfaces ->union(c.rInterfaces) ->union(c.pInterfaces))

.variableList ->union(self.variableList).vars.name

->union(self->collect(c | c.interfaces ->union(c.rInterfaces) ->union(c.pInterfaces))

.operations ->union(self.10Operations).name

->union(self->collect(c | c.interfaces ->union(c.rInterfaces) ->union(c.pInterfaces))

.events ->union(self.events).name ->isUnique(i | i)

-- C6: Each event on state machines and boundary of a controller must have at most one connection to or from it within the controller
inv C6: self.machines ->forAll(m | m.stmDef() .stmDefinedEvents()) ->forAll(e | self.connections

->select(c | (c.from = m and c.efrom = e) or (c.to = m and c.eto = e))

->size() <= 1)

) and self.controllerDefinedEvents() ->forAll(e | self.connections

->select(c | (c.from = self and c.efrom = e) or (c.to = self and c.eto

== e)) ->size() <= 1)

-- C7: Operations must not be declared directly in a controller, but may be defined in the controller
inv C7: self.operations ->isEmpty()

-- State Machines well-formedness conditions (RoboChart reference 3.1.6)

context StateMachineDef
def: stmDefinedVars() : Bag(Variable) = self.interfaces.variableList.vars ->union(self.variableList.vars)
def: stmDefinedEvents() : Bag(Event) = self.interfaces.events ->union(self.events)
def: stmRequiredVars() : Bag(Variable) = self.rInterfaces.variableList.vars
def: stmRequiredOps() : Bag(OperationSig) = self.rInterfaces.operations

-- STM1: State machines cannot have provided interfaces
inv STM1: self.pInterfaces->isEmpty()
-- STM2: Operations in state machines can only be required, not defined
-- i.e. operations must not be declared directly in the state machine (defined interfaces can’t have operations anyway by I2)
inv STM2: self.operations->isEmpty()
-- STM3: Every state machine must have exactly one initial junction
inv STM3: self.nodes->selectByKind(Initial)->one(true)
-- STM4: State machines must contain at least one state
inv STM4: self.nodes->selectByKind(State)->exists(true)
-- STM5: The names of variables, operations, and events are unique to the machine
inv STM5: self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
   .variableList->union(self.variableList).vars.name
   ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
   .operations.name)
   ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
   .events->union(self.events).name)->isUnique(i | i)
-- STM6: State machines must not have operations declared directly within them
inv STM6: self.operations->isEmpty()

-- States well-formedness conditions (RoboChart reference 3.1.7)
context State
-- S1: If a state has a non-empty set of nodes, then conditions STM3 and STM4 apply
inv S1: self.nodes->notEmpty() implies
   self.nodes->selectByKind(Initial)->one(true)
   and self.nodes->selectByKind(State)->exists(true)
-- S2: A state has at most one of each type of action: entry, during, and exit
inv S2: self.actions->selectByKind(EntryAction)->size() <= 1
and self.actions->selectByKind(DuringAction)->size() <= 1
and self.actions->selectByKind(ExitAction)->size() <= 1

-- Initial Junctions well-formedness conditions (RoboChart reference 3.1.8)
context Initial
-- IJ1: An initial junction does not have incoming transitions
inv IJ1: NodeContainer.allInstances()->select(nc | nc.nodes->includes(self)).transitions
   ->select(t | t.target = self)->isEmpty()
-- IJ2: An initial junction must have exactly one outgoing transition
inv IJ2: NodeContainer.allInstances()->select(nc | nc.nodes->includes(self)).transitions
   ->one(t | t.source = self)
-- IJ3: All junction conditions apply
**Junction well-formedness conditions (RoboChart reference 3.1.9)**

context Junction

--- J1: A junction must contain at least one outgoing transition

inv J1: NodeContainer.allInstances() \rightarrow select(nc | nc.nodes->includes(self)).transitions
\rightarrow exists(t | t.source = self)

--- J2: The guards of the transitions out of a junction must form a cover

--- NOTE: cannot be checked in general, but does not prevent generation of semantics

--- inv J2: NodeContainer.allInstances() \rightarrow select(nc | nc.nodes->includes(self)).transitions
\rightarrow select(t | t.source = self and t.condition <> null).condition

--- J3: Transitions starting in junctions cannot have triggers

inv J3: NodeContainer.allInstances() \rightarrow select(nc | nc.nodes->includes(self)).transitions
\rightarrow select(t | t.source = self)->forall(t | t.trigger = null or t.trigger._type = TriggerType::EMPTY)

**Final states well-formedness conditions (RoboChart reference 3.1.10)**

context Final

--- FS1: Final states cannot be the source of transitions

inv FS1: NodeContainer.allInstances() \rightarrow select(nc | nc.nodes->includes(self)).transitions
\rightarrow select(t | t.source = self)->isEmpty()

**Triggers well-formedness conditions (RoboChart reference 3.1.11)**

context Trigger

--- Tg1: A trigger of type SIMPLE has neither the parameter attribute nor the value attribute set. This is a pure synchronisations and does not involve exchange of values

inv Tg1: self._type = TriggerType::SIMPLE implies (self.parameter = null and self.value = null)

--- Tg2: A trigger of type SIMPLE must use a typeless event. This is a pure synchronisations and does not involve exchange of values

inv Tg2: self._type = TriggerType::SIMPLE implies (self.event <> null and self.event.type = null)

--- Tg3: A trigger of type INPUT must have a parameter attribute and cannot have its value attribute set

inv Tg3: self._type = TriggerType::INPUT implies (self.parameter <> null and self.value = null)

--- Tg4: A trigger of type OUTPUT or SYNC must have a value attribute and cannot have its parameter attribute set

inv Tg4: (self._type = TriggerType::OUTPUT or self._type = TriggerType::SYNC) implies (self.value <> null and self.parameter = null)
Chapter D. OCL Well-formedness Conditions

-- Tg5: A trigger of type empty must not have its attributes event, parameter and value set

inv Tg5: self._type = TriggerType::EMPTY
imply (self.event = null and self.parameter = null and self.value = null)

-- Transitions well-formedness conditions (RoboChart reference 3.1.12)
context Transition
-- T1: The source and target of a transition must belong to the same container
inv T1: NodeContainer.allInstances()
-> one(nc | nc.nodes->includes(self.source) and nc.nodes->includes(self.target))

-- T2: If a transition has a trigger, it must be of type INPUT or SIMPLE
inv T2: self.trigger <> null
imply (self.trigger._type = TriggerType::INPUT
or self.trigger._type = TriggerType::SIMPLE
or self.trigger._type = TriggerType::EMPTY)

-- Operations well-formedness conditions (RoboChart reference 3.1.13)
context OperationDef
-- O1: All state-machine conditions apply to operation definitions
inv O1:
-- STM1: State machines cannot have provided interfaces
self.pInterfaces->isEmpty()
-- STM2: Operations in state machines can only be required, not defined
-- i.e. operations must not be declared directly in the state machine (defined interfaces can’t have operations anyway by T2)
and self.operations->isEmpty()
-- STM3: Every state machine must have exactly one initial junction
and self.nodes->selectByKind(Initial)->one(true)
-- STM4: State machines must contain at least one state
and self.nodes->selectByKind(State)->exists(true)
-- STM5: The names of variables, operations, and events are unique to the machine
and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
.variableList->union(self.variableList).vars->isUnique(i | i)
and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
.operations->isUnique(i | i)
and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.pInterfaces))
.events->union(self.events)->isUnique(i | i)
-- STM6: State machines must not have operations declared directly within them
and self.operations->isEmpty()
-- Variables well-formedness conditions (RoboChart reference 3.1.14)

context Variable
-- V1: If the initial value of a required variable or constant of a state machine or controller is defined, it must be consistent with the initial value of any (complementing) variable provided or required by the contexts (controllers or modules) where the state machine or controller is used.
-- NOTE: this requires expression evaluation in order to be properly defined

-- Expressions well-formedness conditions (RoboChart reference 3.1.15)
-- E1: The variables declared in a set comprehension must not have initial values
context SetComp
inv E1: self.variables->forall(v | v.initial = null)
-- E2: Quantified variables in existential and universal quantifications must not have initial values
context QuantifierExpression
inv E2: self.variables->forall(v | v.initial = null)
-- E3: The variables quantified in a lambda expression must not have initial values
context LambdaExp
inv E3: self.variables->forall(v | v.initial = null)

-- Timed Expressions well-formedness conditions (RoboChart reference 3.2.1)
-- TE1: Expressions involving since(C) and sinceEntry(S) are only permitted in transition guards
-- corresponds to CE1 and SCE1 below
-- TE2: The clock C in an expression since(C) may only reference a clock declared within the expression’s containing state-machine
-- corresponds to CE3 below
-- TE3: The state S in an expression sinceEntry(S) may only reference a state within the containing expression’s state-machine. When the name S is ambiguous, because, for instance, there is a state and a substate with the same name in the state machine, the fully qualified name of the state S must be used.
-- corresponds to SCE3 below
-- TE4: The expressions since(C) or sinceEntry(S) may only occur in a comparison expression in which the other branch is a constant
-- corresponds to CE2 and SCE2 below

-- Clock expression well-formedness conditions
context ClockExp
-- CE1: An expression since(C) may only occur as part of a transition guard
inv CE1: self.parentIsTransition()
-- CE2: An expression since(C) may only occur in a branch of a comparison expression in which the other branch is an integer or float expression

inv CE2: Expression::ComparisonExpression() -> exists(comp |
(comp.left = self
and (comp.right.oclIsKindOf(IntegerExp) or comp.right.oclIsKindOf(FloatExp)))
or (comp.right = self
and (comp.left.oclIsKindOf(IntegerExp) or comp.left.oclIsKindOf(FloatExp)))
)

-- CE3: The clock C in an expression since(C) may only reference a clock declared within the expression’s containing state-machine

inv CE3: self.parentIsTransition() implies
self.parentTransition().containingStateMachine().clocks -> includes(self.clock)

-- State Clock expression well-formedness conditions
context StateClockExp

-- SCE1: An expression sinceEntry(S) may only occur as part of a transition guard

inv SCE1: self.parentIsTransition()

-- SCE2: An expression sinceEntry(S) may only occur in a branch of a comparison expression in which the other branch is an integer or float expression

inv SCE2: ComparisonExpression() -> exists(comp |
(comp.left = self
and (comp.right.oclIsKindOf(IntegerExp) or comp.right.oclIsKindOf(FloatExp)))
or (comp.right = self
and (comp.left.oclIsKindOf(IntegerExp) or comp.left.oclIsKindOf(FloatExp)))
)

-- SCE3: The state S in an expression sinceEntry(S) may only reference a state within the containing expression’s state-machine. When the name S is ambiguous, because, for instance, there is a state and a substate with the same name in the state machine, the fully qualified name of the state S must be used.

-- the state is referenced, not named in the metamodel, so the well-formedness conditions here are not concerned with resolving the ambiguity

inv SCE3: self.parentIsTransition() implies
self.parentTransition().containingStateMachine().nestedStates() -> includes(self.state)

-- Timed Statements well-formedness conditions (RoboChart reference 3.2.2)
context ClockReset

-- TS1: A clock reset #C may only reference a clock declared within the action’s containing state-machine, or in the case of a trigger.
within the trigger's containing state-machine
inv TSI1: self.containingStateMachine().clocks->includes(self.clock)

-- Auxiliary definitions

-- function to extract RoboticPlatformDef from a RoboticPlatform (which may be a ref)
context RoboticPlatform
def: rpDef(): RoboticPlatformDef =
  if self.oclIsKindOf(RoboticPlatformDef) then
    self.oclAsType(RoboticPlatformDef)
  else
    self.oclAsType(RoboticPlatformRef).ref
  endif

-- function to extract ControllerDef from a Controller (which may be a ref)
context Controller
def: controllerDef(): ControllerDef =
  if self.oclIsKindOf(ControllerDef) then
    self.oclAsType(ControllerDef)
  else
    self.oclAsType(ControllerRef).ref
  endif

-- function to extract StateMachineDef from a StateMachine (which may be a ref)
context StateMachine
def: stmDef(): StateMachineDef =
  if self.oclIsKindOf(StateMachineDef) then
    self.oclAsType(StateMachineDef)
  else
    self.oclAsType(StateMachineRef).ref
  endif

-- functions to get input and output events of a node container
-- (An event is considered to be an output if it is used in an OUTPUT or SYN trigger.
-- or if it is used in an OUTPUT, SYN or SIMPLE send statement.)
-- (An event is considered to be an input if it is used in an INPUT or SIMPLE trigger.
-- or if it is used in an INPUT send statement.)
context NodeContainer
def: ncInputEvents(): Bag(Event) =
  self.transitions->select(t | t.trigger <> null and
    (t.trigger..type = TriggerType::INPUT or t.trigger..type = TriggerType::SIMPLE)
  ).trigger.event
  ->union(self.transitions->select(t | t.action <> null).action.
    statementInputEvents())
  ->union(self.nodes->selectByKind(NodeContainer).ncInputEvents())
  ->union(self.nodes->selectByKind(State).actions.action.)
Chapter D. OCL Well-formedness Conditions

```
statementInputEvents()

def: ncOutputEvents() : Bag(Event) =
  self.transitions->select(t | t.trigger <> null and
    (t.trigger._type = TriggerType::OUTPUT or t.trigger._type =
      TriggerType::SYNC)
  ).trigger.event
  ->union(self.transitions->select(t | t.action <> null).action.
    statementOutputEvents())
  ->union(self.nodes->selectByKind(NodeContainer).ncOutputEvents())
  ->union(self.nodes->selectByKind(State).actions.action.
    statementOutputEvents())

case Context
  def: statementInputEvents() : Bag(Event) = Bag{}
  def: statementOutputEvents() : Bag(Event) = Bag{}

  context SendEvent
    def: statementInputEvents() : Bag(Event) =
      Set{self.trigger}->select(t | t._type = TriggerType::INPUT).event
    def: statementOutputEvents() : Bag(Event) =
      Set{self.trigger}->select(t | Set{TriggerType::OUTPUT, TriggerType::
        SYNC, TriggerType::SIMPLE}->includes(t._type)).event

  context SeqStatement
    def: statementInputEvents() : Bag(Event) =
      self.statements.statementInputEvents()->asBag()
    def: statementOutputEvents() : Bag(Event) =
      self.statements.statementOutputEvents()->asBag()

  context IfStmt
    def: statementInputEvents() : Bag(Event) =
      self._'then'.statememntInputEvents()->union(self._'else'.
        statementInputEvents())
    def: statementOutputEvents() : Bag(Event) =
      self._'then'.statementOutputEvents()->union(self._'else'.
        statementOutputEvents())

-- functions on expressions to support timed expressions well-formedness
  context Expression
    def: parentIsTransition() : Boolean =
      if selfoclContainer().oclIsKindOf(Expression) then
        selfoclContainer().oclAsType(Expression).parentIsTransition()
      else
        Transition.allInstances() ->exists(t | t.condition = self)
      endif
    def: parentTransition() : Transition =
      if selfoclContainer().oclIsKindOf(Expression) then
        selfoclContainer().oclAsType(Expression).parentTransition()
      else
        selfoclContainer().oclAsType(Transition)
      endif
    def: ComparisonExpression() : Set(BinaryExpression) =
```
BinaryExpression.allInstances()->select(x | xoclIsKindOf(Equals) or xoclIsKindOf(Different)
  or xoclIsKindOf(GreaterThan) or xoclIsKindOf(LessThan)
  or xoclIsKindOf(GreaterOrEqual) or xoclIsKindOf(LessOrEqual))

-- function to obtain all states nested within a node container
context NodeContainer
  def: nestedStates() : Bag(State) =
    self.nodes->selectByKind(State)->union(self.nodes->selectByKind(State).
      nestedStates())

-- functions to find the containing state machine for various
-- constructs,
-- particularly ClockReset to define the timed statement well-
-- formedness conditions
context NodeContainer
  def: containingStateMachine() : StateMachineBody =
    if selfoclIsKindOf(StateMachineBody) then
      selfoclAsType(StateMachineBody)
    else
      selfoclContainer().oclAsType(NodeContainer).containingStateMachine()
    endif
context Transition
  def: containingStateMachine() : StateMachineBody =
    selfoclContainer().oclAsType(NodeContainer).containingStateMachine()
context Action
  def: containingStateMachine() : StateMachineBody =
    selfoclContainer().oclAsType(State).containingStateMachine()
context Statement
  def: containingStateMachine() : StateMachineBody =
    if selfoclIsKindOf(StateMachineBody) then
      selfoclIsKindOf(Action) then
        selfoclIsKindOf(Statement).containingStateMachine()
      elseif selfoclIsKindOf(Transition) then
        selfoclIsKindOf(Statement).containingStateMachine()
      else
        selfoclContainer().oclAsType(Statement).containingStateMachine()
    endif
context Trigger
  def: containingStateMachine() : StateMachineBody =
    if selfoclIsKindOf(Transition) then
      selfoclIsKindOf(Transition).containingStateMachine()
    else
      selfoclContainer().oclAsType(Statement).containingStateMachine()
    endif
context ClockReset
  def: containingStateMachine() : StateMachineBody =
    if selfoclIsKindOf(Trigger) then
      selfoclIsKindOf(Trigger).containingStateMachine()
    else
      selfoclContainer().oclIsKindOf(Action) then
        selfoclIsKindOf(Trigger).containingStateMachine()
      endif
self.oclContainer().oclAsType(Trigger).containingStateMachine()
else
self.oclContainer().oclAsType(Statement).containingStateMachine()
endif
endpackage
\textit{\LaTeX} style based on the \textit{The Legrand Orange Book Template} by Mathias Legrand and Vel from \textit{LaTeXTemplates.com}. Licensed under \textit{CC BY-NC-SA 3.0}.

Icons used in RoboTool and this report have been obtained from \textit{www.flaticon.com}. Individual credits are given below.

Icon made by Iconnice from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Sarfraz Shoukat from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Freepik from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Dario Ferrando from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Lyolya from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Freepik from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Google from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Freepik from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Freepik from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}

Icon made by Freepik from \textit{www.flaticon.com} is licensed by \textit{CC 3.0 BY}
Chapter D. OCL Well-formedness Conditions


Index of Semantic Rules

In this index you’ll find the list of semantic functions in alphabetic order, and page where they are defined. Timed versions of existig semantic rules are indexed by a `timed` item under the entry for the semantic function. Semantic functions exclusive to the timed model are identified by a `timed` annotation in parenthesis after the rule name. Rules whose names are abbreviation (e.g., S) are annotated with the full name in parenthesis.

A (Assignment)
   (probprism), 90
A (Statement)
   probprism, 90
Action, 53
actionOfTran
   probprism, 87
addLoopStateTrans
   probnorm, 78
allClockVariables (timed), 56
allConstants, 37
allDeadlineTransitions (timed), 64
allEvents, 36
allLocalConstants, 37
allLocalVariables, 37
allTransitions, 50
allVariables, 37

alphaClockReset
   ClockExp (timed), 56
   StateClockExp (timed), 56
andExprs
   probprism, 94
BBuffer (collection), 123
buffer, 40
buildScope, 49
timed, 55
C (Controller), 40
C (ControllerDef)
   probprism, 81
Cln (Collection,collection), 122
clockResets (timed), 56
combTransJunction
   probnorm, 79
INDEX OF SEMANTIC RULES

combTransJunctions
  probnorm, 78
Communication, 57
compileTarget, 47
compileWC (timed), 58–62
composeControllers, 39
composeMachines, 41
composeStates, 44, 66
constantsOfNamedElems
  probprism, 85
constInit, 38
constInitSTM, 48
timed, 55
CS (NodeContainer)
  probnorm, 77
ctrlMemory, 41
deadlineEvents (timed), 63
enterSubstate
  probprism, 84
enterSubstates
  probprism, 84
exitCompSrcState
  probprism, 89
exitCompState (State)
  probprism, 82
exitCompSubstate
  probprism, 84
exitSeqCtrlConsts
  probprism, 80
exitSimpSrcState
  probprism, 89
exitSimpSubstate
  probprism, 83
exitSubstate
  probprism, 83
exitSubstates, 47
  probprism, 83
flowEvents, 44
flowTriggerEvents, 46
getEntryAction
  probprism, 94
getEvents
  probprism, 93
getExitAction
  probprism, 93
getInitial
  probprism, 94
getRequiredOperations
  probprism, 93
getsetChannels, 43
getsetLocalChannels, 44
getVariableLists
  probprism, 93
hasEntryAction
  probprism, 94
hasExitAction
  probprism, 93
hiddenModuleChannels, 36
initialisation, 43
inTransCombinableJuncs
  probnorm, 78
isCombinableTran
  probnorm, 78
isComposite
  probprism, 94
J (Junction), 43
M (Module), 36
  probprism, 80
memoryChannels, 36
memoryDeadline (timed), 64
memoryTransition, 50
timed, 64
modMemory, 38
newState
  probrc, 79
newTransition
  probrc, 79
INDEX OF SEMANTIC RULES

opToActionMaps (StateMachineDef)
  prob prism, 81

R (RoboticPlaftorm)
  prob prism, 80
reachableJunctions, 43
reachableTransitions, 46
renameTriggerEvents, 42
renamingController, 39
renamingMachine, 42
renCtrlEvts, 39
renStmEvts, 42
requiredConstants, 37
requiredOperations, 37
requiredVariables, 37
restrictedState, 46

S (NodeContainer)
  prob prism, 82
S (State), 44
  prob norm, 77
timed, 64
  Composite State, 45
timed, 65
Final State, 46
Simple State, 45
timed, 65
singleBuffer, 40
splitTran
  prob norm, 79
SS
  prob prism, 85
ST (Call)
  (prob prism), 92
ST (IfStmt)
  (prob prism), 91
ST (SendEvent)
  (prob prism), 91
ST (SeqStatement)
  (prob prism), 92
ST (Skip)
  (prob prism), 90
ST (Statement)
  (prob prism), 90
Statement, 51, 51, 52
timed, 66, 66, 67
  Call Statement, 67
  ClockReset, 67
  Wait, 67
  Call Statement, 52
  CommunicationStmt, 53
  If Statement, 52
  Send Event
collection, 123
  Sequential Composition, 53
  Skip, 53
StatementInContext, 51
states, 43
STM (State Machine), 42
timed, 55
STM (StateMachineDef)
  prob norm, 77
  prob prism, 81
stmClocks (timed), 56
stmMemory, 48
timed, 63
M Statement)
  (prob prism), 92
substatesTriggers, 46

T (Initial)
  prob prism, 88
T (Junction)
  prob prism, 88
T (Node)
  prob prism, 87
T (ProbJunc)
  prob prism, 88
T (State)
  prob prism, 89
T (Transition), 47, 57
TN (Initial)
  prob prism, 86
<table>
<thead>
<tr>
<th>TN (Junction)</th>
<th>probprism, 86</th>
</tr>
</thead>
<tbody>
<tr>
<td>TN (Node)</td>
<td>probprism, 85</td>
</tr>
<tr>
<td>TN (ProbJunc)</td>
<td>probprism, 86</td>
</tr>
<tr>
<td>TN (State)</td>
<td>probprism, 86</td>
</tr>
<tr>
<td>TNS</td>
<td>probprism, 85</td>
</tr>
<tr>
<td>transitionsFrom, 50</td>
<td></td>
</tr>
<tr>
<td>transitionsOf</td>
<td>probnorm, 78</td>
</tr>
<tr>
<td>trigEvents, 44</td>
<td></td>
</tr>
<tr>
<td>Trigger, 49</td>
<td>collection, 123</td>
</tr>
<tr>
<td>triggerDeadlines (timed), 66</td>
<td></td>
</tr>
<tr>
<td>triggerEvent, 49, 63</td>
<td></td>
</tr>
<tr>
<td>triggerForMemory, 49</td>
<td></td>
</tr>
<tr>
<td>TS (Node)</td>
<td>probprism, 87</td>
</tr>
<tr>
<td>usedVariables, 52, 52</td>
<td></td>
</tr>
<tr>
<td>V (Variable)</td>
<td>probprism, 95</td>
</tr>
<tr>
<td>VL (VariableList)</td>
<td>probprism, 95</td>
</tr>
<tr>
<td>Wait, 67, 67, 67</td>
<td></td>
</tr>
</tbody>
</table>
In this index you’ll find the location of call to the semantic rules. For each call of a semantic function, the page number superscripted with the usage index is provided. The index of the call is unique with respect to the semantic function, and also shown superscripted in the call location.

A (Actions,prob prism) , 82^1, 83^2, 84^3, 87^4
Action , 45^1, 45^2, 45^3, 45^4, 45^5, 45^6, 47^10, 47^7, 47^8, 47^9, 57^11, 57^12, 57^13, 57^14, 65^15, 65^16, 65^17, 65^18, 65^19, 65^{20}
actionOfTran (.prob prism) , 88^1, 88^2, 88^3, 89^4
addLoopStateTrans (probnorm) , 77^1
allClockVariables (timed) , 55^1, 63^2
allConstants , 38^1, 43^2, 43^3, 44^4, 48^5, 48^6, 55^7, 63^8, 63^9
allDeadlineTransitions (timed) , 63^1
allEvents , 42^1
allLocalConstants , 36^1, 40^2, 44^3
allLocalVariables , 36^1, 38^2, 40^3, 41^4, 44^5, 48^6, 63^7
allTransitions , 44^1, 46^2, 46^3, 46^4, 48^5, 50^6, 55^7, 63^8, 63^9, 64^{10}
allVariables , 43^1, 43^2, 44^3

alphaClockReset (timed) , 56^1, 56^2
andExprs (prob prism) , 82^1, 82^2, 83^3, 84^4, 84^5, 86^6, 88^7, 88^8, 89^10, 89^{11}, 89^{12}, 89^9, 90^{13}, 91^{14}, 91^{15}, 91^{16}, 92^{17}, 94^{18}
BBuffer , 122^1, 122^2
buffer , 36^1
buildScope , 48^1, 49^2, 49^3
buildScope (timed) , 55^1
C (Controller) , 39^1
C (Controller,prob prism) , 80^1
clockResets (timed) , 55^1, 55^2
combTransJunction (probnorm) , 78^1
combTransJunctions (probnorm) , 77^1, 78^2
compileTarget , 47^1, 47^2, 47^3, 57^4, 57^5, 57^6
compileWC (timed) , 56^1
composeControllers , 36^1, 39^2
composeMachines , 401, 412
composeStates , 421, 442, 453, 554
composeStates (timed) , 651, 662
constantsOfNamedElems (.probprism) , 821, 852
constInit , 381, 412
constInitSTM , 481
constInitSTM (timed) , 551, 552, 553
CS (NodeContainer,probnorm) , 771, 772
ctrlMemory , 401
deadlineEvents (timed) , 551, 552
e (Expression,probprism) , 881, 892, 903, 914, 915, 916, 927, 956, 959
enterSubstate (.probprism) , 841
enterSubstates (.probprism) , 821, 842
exitCompSrcState (.probprism) , 891
exitCompState (.probprism) , 821
exitCompSubstate (.probprism) , 831
exitSeqCtrlConsts (.probprism) , 801
exitSimpSrcState (.probprism) , 891
exitSimpSubstate (.probprism) , 831
exitSubstate (.probprism) , 831
exitSubstates , 451, 472, 573, 654, 655
exitSubstates (Expression) , 381, 482, 5010, 5011, 5012, 503, 504, 505, 506, 507, 508, 509, 5213, 5214, 5215, 5316, 5317, 5518, 5819, 5820, 5821, 5822, 5923, 5924, 5925, 5926, 6027, 6028, 6029, 6030, 6031, 6132, 6133, 6134, 6235, 6236, 6237, 6238, 6239, 6240, 6241, 6242, 6443, 6444, 6445, 6446, 6647, 6748, 6749, 12350, 12351, 12352
flowEvents , 441, 442, 663, 664
flowTriggerEvents , 451, 652
getExitAction (probprism) , 821, 832, 933
getInitial (probprism) , 821, 842, 873
getRequiredOperations
(Operations,probprism) , 811, 812
getsSetChannels , 421, 552
getsSetLocalChannels , 421, 552
getVariableLists (VariableList,probprism) , 801, 812, 813
hasEntryAction (probprism) , 821, 872, 873
hasExitAction (probprism) , 891
hiddenModuleChannels , 361
initialisation , 421, 452, 553, 654
inTransCombinableJuncs (probnorm) , 771, 772
isCombinableJuncs (probnorm) , 781, 792
isComposite (probprism) , 821, 832, 843, 894
J (Junction) , 431, 452, 453, 654, 655
M (Module) , 1221
memoryChannels , 361, 362
memoryDeadline (timed) , 631
memoryTransition , 481
memoryTransition (timed) , 631
modMemory , 361
newState (.probrc) , 781
newTransition (.probrc) , 781, 782
opToActionMaps (State machine,probprism) , 811
R (RoboticPlatform,probprism) , 801
reachableJunctions , 431, 452, 453, 461, 655, 656
reachableTransitions , 461
readState , 511, 512, 663
renameTriggerEvents , 421, 552
renamingController , 391, 392
renamingMachine , 411, 412
renCtrlEvts , 391, 392
INDEX OF CALLS TO SEMANTIC RULES
Index

Connection
  async, 18
  bidirec, 18
  Description, 18
  efrom, 18
  eto, 18
  from, 18
  to, 18
ConnectionNode
  Description, 17, 18
Event

Expression, 18

Module
  Description, 18

RCPackage
  Description, 17

Robotic Platform
  Description, 18
  RoboticPlatformDef, 18
  RoboticPlatformRef, 18
  Well Formedness, 27