RoboChart and RoboTool
Modelling, Verification and Simulation for Robotics

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

1 Introduction ........................................................................... 11

## RoboChart

2 Language Structure ................................................................. 15
   2.1 RoboChart metamodel ....................................................... 15
      2.1.1 Module .................................................................. 16
      2.1.2 Controller ................................................................. 17
      2.1.3 State Machine ............................................................. 18
      2.1.4 Type Declaration ........................................................ 20
      2.1.5 Expression ................................................................. 21
      2.1.6 Action and Statement .................................................. 25
   2.2 Timed Primitives ................................................................. 26

3 Well-formedness Conditions .................................................... 29
   3.1 Core Language ................................................................. 29
      3.1.1 Robotic Platforms ...................................................... 29
      3.1.2 Interfaces ................................................................. 30
      3.1.3 Modules ................................................................. 30
      3.1.4 Connection .............................................................. 30
      3.1.5 Controllers .............................................................. 31
      3.1.6 State Machines ......................................................... 31
      3.1.7 States ................................................................. 31
      3.1.8 Initial junctions ......................................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.9</td>
<td>Junction</td>
<td>31</td>
</tr>
<tr>
<td>3.1.10</td>
<td>Final states</td>
<td>32</td>
</tr>
<tr>
<td>3.1.11</td>
<td>Triggers</td>
<td>32</td>
</tr>
<tr>
<td>3.1.12</td>
<td>Transitions</td>
<td>32</td>
</tr>
<tr>
<td>3.1.13</td>
<td>Operations</td>
<td>32</td>
</tr>
<tr>
<td>3.1.14</td>
<td>Variables</td>
<td>32</td>
</tr>
<tr>
<td>3.1.15</td>
<td>Expressions</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Timed Language</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Timed Expressions</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Timed Statements</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Probabilistic Language</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Junctions</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Probabilistic Junctions</td>
<td>33</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Transitions</td>
<td>34</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>36</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Modules</td>
<td>36</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Controllers</td>
<td>40</td>
</tr>
<tr>
<td>4.1.3</td>
<td>State machines</td>
<td>42</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Statements</td>
<td>50</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Expressions</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Detailed Semantics: Timed Language</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1</td>
<td>State machines</td>
<td>57</td>
</tr>
<tr>
<td>4.2.2</td>
<td>States</td>
<td>70</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Timed statements</td>
<td>72</td>
</tr>
<tr>
<td>II</td>
<td>Assertions</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Assertion DSL Syntax</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Standard Assertions</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Probabilistic Assertions</td>
<td>79</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Constants</td>
<td>79</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Labels and Formulas</td>
<td>80</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Rewards</td>
<td>81</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Probabilistic Formulas</td>
<td>81</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Simulations</td>
<td>83</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Expressions</td>
<td>84</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Qualified Names</td>
<td>86</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Terminals</td>
<td>86</td>
</tr>
<tr>
<td>6</td>
<td>Assertions DSL Usage</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Standard Assertions</td>
<td>87</td>
</tr>
</tbody>
</table>
### 6.2 Probabilistic Assertions

- [6.2.1 Instructions](#) 88
- [6.2.2 Examples](#) 88

### 7 Checking core assertions 93

---

#### III Collections

8 **Collections** 97

- [8.1 Metamodel](#) 97
- [8.2 Well-formedness Conditions](#) 99
  - [8.2.1 RCCollection](#) 99
  - [8.2.2 Instantiation](#) 99
  - [8.2.3 Event](#) 99
  - [8.2.4 Trigger](#) 100
- [8.3 Semantics](#) 100

9 **Conclusions** 103

---

#### IV Appendices

A **RoboChart diagrams - an informal overview** 107

A.1 **Time primitives** 113

B **Complete Metamodel** 117

C **Mathematical Toolkit** 127

D **OCL Well-formedness Conditions** 133

- **Credits** 147
- **Bibliography** 149
- **Index of Semantic Rules** 151
- **Index of Calls to Semantic Rules** 155
- **Index** 159
<table>
<thead>
<tr>
<th></th>
<th><strong>Untimed semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Semantics of modules .................................................. 36</td>
</tr>
<tr>
<td>2</td>
<td>Hidden Module channels .................................................. 36</td>
</tr>
<tr>
<td>3</td>
<td>Memory channels .......................................................... 36</td>
</tr>
<tr>
<td>4</td>
<td>Function allEvents ....................................................... 36</td>
</tr>
<tr>
<td>5</td>
<td>Function allVariables ..................................................... 36</td>
</tr>
<tr>
<td>6</td>
<td>Function allLocalVariables .............................................. 36</td>
</tr>
<tr>
<td>7</td>
<td>Function requiredVariables ............................................. 37</td>
</tr>
<tr>
<td>8</td>
<td>Function allLocalConstants ............................................. 37</td>
</tr>
<tr>
<td>9</td>
<td>Function requiredConstants ............................................. 37</td>
</tr>
<tr>
<td>10</td>
<td>Module Memory ............................................................. 38</td>
</tr>
<tr>
<td>11</td>
<td>Constants Initialisation for Controllers and Modules ............... 38</td>
</tr>
<tr>
<td>12</td>
<td>Composition of controllers ............................................. 39</td>
</tr>
<tr>
<td>13</td>
<td>Renaming controller ....................................................... 39</td>
</tr>
<tr>
<td>14</td>
<td>Renaming controller events ............................................. 39</td>
</tr>
<tr>
<td>15</td>
<td>Buffer ............................................................................. 40</td>
</tr>
<tr>
<td>16</td>
<td>Single buffer .................................................................... 40</td>
</tr>
<tr>
<td>17</td>
<td>Semantics of controllers .................................................. 40</td>
</tr>
<tr>
<td>18</td>
<td>Controller Memory ........................................................... 41</td>
</tr>
<tr>
<td>19</td>
<td>Composition of machines .................................................. 41</td>
</tr>
<tr>
<td>20</td>
<td>Renaming state machine .................................................... 42</td>
</tr>
<tr>
<td>21</td>
<td>Renaming machine events .................................................. 42</td>
</tr>
<tr>
<td>22</td>
<td>Semantics of state machine .............................................. 42</td>
</tr>
</tbody>
</table>
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 [AD94] and Timed CSP [Sch00].

UML [UML2015] 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 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 [Ros11]. Importantly, CSP is a front end for a mathematical model that supports a number of 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 [HH98] (UTP). Use of CSP enables model checking with FDR [GABR14]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [SCJS10] and probability [ZSHQ13].

Chapter 2 describes RoboChart models, and Chapter 3 defines their well-formedness conditions. Chapter 4 presents their semantics of RoboChart in CSP. 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.
2 Language Structure .................. 15
 2.1 RoboChart metamodel
 2.2 Timed Primitives

3 Well-formedness Conditions .......... 29
 3.1 Core Language
 3.2 Timed Language
 3.3 Probabilistic Language

4 Semantics ............................. 35
 4.1 Detailed Semantics: Core Language
 4.2 Detailed Semantics: Timed Language
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 is automatically generated from a syntax definition. It includes a notion of a MachineContainer, which, as the same suggests, can include a number of state machines. As shown later in Figure 2.3, a controller, like an RCPackage, is also a MachineContainer. An interface groups variableLists, operations, and events.
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 defines the values that can be communicated via the connection, if any.
2.1 RoboChart metamodel

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.

### 2.1.2 Controller

The structure of a Controller is shown in Figure 2.3. It can be specified by a ControllerDefinition or a ControllerReference, which just names a controller defined elsewhere. A ControllerDefinition 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, and provided, required, and defined interfaces of an element. Defined interfaces of an element declare the variables 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 BasicContext that has also interfaces. A BasicContext has Variables, Operations, and Events. 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, and an initial value.

Figure 2.4 also gives the metamodel for an Operation. It has an OperationSignature, 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
Figure 2.4: Metamodel of RoboChart context for elements and operations

Figure 2.5: Metamodel of RoboChart state machines

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.
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. We can also specify start and end deadlines for a transition. Additionally, a transition may have a probability value (between 0 to 1) that means the probability of the transition being triggered.

The concrete syntax of transitions is shown in Syntax 2.1.3. The first and second expression in TriggerLabel are the start and end deadlines. The first expression in Label is the probability value and the second one is the guard condition. A trigger and a probability value are not allowed to be present in transitions at the same time.

**Syntax — Transition Label.**

Label ::= 
(TriggerLabel|'p{'Expression'}')? ('['Expression']')? ('/'Statement)?
TriggerLabel ::= ('{'Expression'}>')? Trigger ('<{'Expression'}')?

The syntax of triggers is described in Syntax 2.1.3. It consists of an optional input, output, sync or simple trigger, followed by an optional variable that record the time instant the trigger occurs, and a (potentially empty) list of clock resets. The concrete syntax of the different types of triggers is shown in Table 2.1.

**Syntax — Trigger.**

Trigger ::= (Input|Output|Sync|Simple)? ClockReset*
Input ::= Event '?' Variable
Output ::= Event '!' Expression
Sync ::= Event ',' Expression
Simple ::= Event

A trigger has an event, which can be on its own, associated with a value, or with a variable. In the
first case, we have a SimpleTrigger. Triggers whose events are associated with a value correspond to a synchronisation (SyncTrigger) or output (OutputTrigger). Finally, triggers whose events are associated with variables model input communication) (InputTrigger).

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

Table 2.1: Types of triggers.

2.1.4 Type Declaration

The metamodel for Types is given in Figure 2.6. As indicated, they include references to type declarations (TypeRef), sets (SetType), and cartesian products (ProductType). A type reference refer to type declarations, which are PrimitiveTypes, given just by their names, Enumerations, records (DataType), named definitions (NamedDefinition). Table 2.2 gives the concrete syntax for the various type constructors.

Additional types, such as functions, relations and sequences, are defined in the mathematical toolkit (Appendix C), and are implemented in the tool with the concrete syntax shown in Table 2.2.
2.1 RoboChart metamodel

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Reference</td>
<td>N</td>
<td>N is the name of a declared type.</td>
</tr>
<tr>
<td>Set Type</td>
<td>Set(T)</td>
<td>Type of sets of elements of T.</td>
</tr>
<tr>
<td>Sequence Type</td>
<td>Seq(T)</td>
<td>Type of sequences of elements of T1 to T2.</td>
</tr>
<tr>
<td>Product Type</td>
<td>T1 * T2</td>
<td>Type of pairs whose first element has type T1 and second element has type T2.</td>
</tr>
<tr>
<td>Function Type</td>
<td>T1 -&gt; T2</td>
<td>Type of functions from T1 to T2.</td>
</tr>
<tr>
<td>Relation Type</td>
<td>T1 &lt;-&gt; T2</td>
<td>Type of relations between T1 and T2.</td>
</tr>
</tbody>
</table>

Table 2.2: Concrete syntax of types.

2.1.5 Expression

Expressions include logical expressions of a first-order predicate calculus, and usual arithmetic and relational expressions. We also have a LetExpression, an IFExpression, and expressions to deal with tuples, enumerated types, function calls, and so on. The syntax boxes 2.1.5 and 2.1.5 give the concrete syntax for expressions. We omit the simple metamodel diagrams here.

```
Syntax — Expressions.
Expr ::= (?'0'..'9')+                  – integer
  | ('0'..'9')+.('0'..'9')+              – float
  | ""[""[^""\"]"]"                   – string
  | 'true' | 'false'                           – boolean
  | N      – reference
  | '<'(Expr (',' Expr)*)?'>'>          – sequence
  | '{'(Expr (',' Expr)*)?'}'          – set
  | '{' N ':' Type '}':' Expr '@' Expr '}'
  | '['Expr ',' Expr ']'              – closed interval
  | '(' Expr ',' Expr ')'             – open interval
  | N ':' N                            – enumeration constant
  | '([' (Expr (',' Expr)*)? '])'      – array access
  | Expr '[' Expr ']]'                – function application
  | Expr '(' (Expr (',' Expr)*)? ')')  – field access
  | Expr '.' N                         – continues on Syntax 2.1.5
  | ...                                – continues on Syntax 2.1.5
```
Syntax — Expressions (cont.).

Expr ::= ...
| '-' Expr  
  -- negation
| Expr '+' Expr  
  -- sum
| Expr '-' Expr  
  -- subtraction
| Expr '*' Expr  
  -- multiplication
| Expr '/' Expr  
  -- division
| Expr '%' Expr  
  -- remainder
| Expr '==>' Expr  
  -- equality
| Expr '!=' Expr!  
  -- difference
| Expr '>' Expr  
  -- greater
| Expr '>=>' Expr  
  -- greater or equal
| Expr '<' Expr  
  -- less
| Expr '<=' Expr  
  -- less or equal
| 'not' Expr  
  -- not
| Expr '/\' Expr  
  -- and
| Expr '\/' Expr  
  -- or
| Expr '=> Expr  
  -- implies
| Expr 'iff' Expr  
  -- if and only if
| 'forall' N ':' Type ']' Expr '0' Expr  
  -- universal quantification
| 'exists' N ':' Type ']' Expr '0' Expr  
  -- existential quantification
| 'exists1' N ':' Type ']' Expr '0' Expr  
  -- uniqueness quantification
| Expr '^' Expr  
  -- concatenation
| 'if' Expr 'then' Expr 'else' Expr 'end'  
  -- conditional
| 'let' N '==' Expr '0' Expr  
  -- local definition
| 'the' N ':' Type ']' Expr '0' Expr  
  -- definite description
| 'lambda' N ':' Type ']' Expr '0' Expr  
  -- lambda expression
| 'since' '(' N ')'
  -- clock expression
| 'sinceEntry' '(' N ')'
  -- state clock expression

Table 2.3 summarises primitive expression in RoboChart. Table 2.4 describes the arithmetic expressions, Table 2.5 list the expressions used to compare values, Table 2.6 covers the logical expressions, and Table 2.7 explains the remaining expressions.
### 2.1 RoboChart metamodel

<table>
<thead>
<tr>
<th>Expression</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>(0..9)+</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>(0..9)+.(0..9)+</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>&quot;...&quot;</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Reference</td>
<td>N</td>
<td>N is the name of a variable or constant.</td>
</tr>
<tr>
<td>Sequence</td>
<td>&lt;e1,e2,...&gt;</td>
<td>Sequence with values $e_i$.</td>
</tr>
<tr>
<td>Set</td>
<td>{e1,e2,...}</td>
<td>Set with values $e_i$.</td>
</tr>
<tr>
<td>Set Comprehension</td>
<td>{x:T</td>
<td>P @ e}</td>
</tr>
<tr>
<td>Interval</td>
<td>[e1,e2] or (e3,e4)</td>
<td>Closed interval between $e_1$ and $e_2$, and open interval between $e_3$ and $e_4$, or a combination of both.</td>
</tr>
<tr>
<td>Enumeration</td>
<td>E::c</td>
<td>Constant $c$ of enumeration $E$.</td>
</tr>
<tr>
<td>Tuple</td>
<td>(</td>
<td>e1,e2,...</td>
</tr>
<tr>
<td>Array</td>
<td>e[i]</td>
<td>The $i$-th element of array $e$.</td>
</tr>
<tr>
<td>Function application</td>
<td>f(e1,e2,...)</td>
<td>Apply function $f$ to parameters $e_i$.</td>
</tr>
<tr>
<td>Selection</td>
<td>e.n</td>
<td>The $n$ field of record $e$.</td>
</tr>
</tbody>
</table>

**Table 2.3: Primitive Expressions.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>$-e$</td>
<td>Arithmetical negation of expression $e$.</td>
</tr>
<tr>
<td>Sum</td>
<td>$e1 + e2$</td>
<td>Sum of $e_1$ and $e_2$.</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$e1 - e2$</td>
<td>Subtraction of $e_1$ and $e_2$.</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$e1 \ast e2$</td>
<td>Multiplication of $e_1$ by $e_2$.</td>
</tr>
<tr>
<td>Division</td>
<td>$e1 / e2$</td>
<td>Division of $e_1$ by $e_2$.</td>
</tr>
<tr>
<td>Modulo</td>
<td>$e1 % e2$</td>
<td>Remainder of dividing $e_1$ by $e_2$.</td>
</tr>
</tbody>
</table>

**Table 2.4: Arithmetic Expressions.**
### Table 2.5: Comparison Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>e1 == e2</td>
<td>True if both expressions are equal.</td>
</tr>
<tr>
<td>Different</td>
<td>e1 != e2</td>
<td>True if both expressions are different.</td>
</tr>
<tr>
<td>Greater than</td>
<td>e1 &gt; e2</td>
<td>True if e1 is greater than e2.</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>e1 &gt;= e2</td>
<td>True if e1 is greater than or equal to e2.</td>
</tr>
<tr>
<td>Less than</td>
<td>e1 &lt; e2</td>
<td>True if e1 is less than e2.</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>e1 &lt;= e2</td>
<td>True if e1 is less than or equal to e2.</td>
</tr>
</tbody>
</table>

### Table 2.6: Logical Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical not</td>
<td>not e</td>
<td>True if and only if e is false.</td>
</tr>
<tr>
<td>Logical and</td>
<td>e1 /\ e2</td>
<td>True if and only if e1 and e2 are true.</td>
</tr>
<tr>
<td>Logical or</td>
<td>e1 / e2</td>
<td>True if and only if at least one of the expressions is true.</td>
</tr>
<tr>
<td>Logical implies</td>
<td>e1 =&gt; e2</td>
<td>Equivalent to not e1 / e2.</td>
</tr>
<tr>
<td>Logical iff</td>
<td>e1 iff e2</td>
<td>Equivalent to e1=&gt;e2 /\ e2=&gt;e1.</td>
</tr>
<tr>
<td>Universal quantification</td>
<td>forall x: T</td>
<td>P @ Q</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>exists x: T</td>
<td>P @ Q</td>
</tr>
<tr>
<td>Uniqueness quantification</td>
<td>exists1 x: T</td>
<td>P @ Q</td>
</tr>
</tbody>
</table>
Table 2.7: Other Expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>(\text{e1}^{\text{e2}})</td>
<td>Concatenate sequences (\text{e1}) and (\text{e2}).</td>
</tr>
<tr>
<td>Conditional</td>
<td>(\text{if } c \text{ then } e \text{ else } f \text{ end})</td>
<td>If condition (c) is true, (\text{e1}) else (\text{e2}).</td>
</tr>
<tr>
<td>Local definition</td>
<td>(\text{let } n == e @ f)</td>
<td>Define locally (n) and use it to calculate (f).</td>
</tr>
<tr>
<td>Definite description</td>
<td>(\text{the } x : T</td>
<td>P @ e)</td>
</tr>
<tr>
<td>Lambda expression</td>
<td>(\text{lambda } x : T</td>
<td>P @ e)</td>
</tr>
</tbody>
</table>

2.1.6 Action and Statement

Similarly, the action language is very simple. Syntax 2.1.6 gives the concrete syntax of statements used to define actions in states and transitions.

Syntax — Statements.

Statement ::= ‘skip’
| N ’(’ (Expr (’,’ Expr)* )? ’)’ | – operation call |
| ‘if’ Expr | – conditional |
| ‘then’ Statement | |
| ‘else’ Statement ’end’ | |
| N ’=’ Expr | – assignment |
| N ’!’ Expr | – output event |
| N ’?’ N | – input event |
| N | – simple synchronisation |
| N ’.’ Expr | – synchronisation |
| Statement ’;’ Statement | – sequential composition |
| Statement ’{’ Expr ‘}’ | – timed statement |
| ’wait’ ‘(’ Expr ‘)’ | – wait statement |
| ’wait’ ‘(’ ‘[’ Expr ‘,’ Expr ‘]’ ‘)’ | – nondeterministic wait |
| ’#’ N | – clock reset |

Statements can be used to construct state and transition actions. The syntax of state actions is shown in Syntax 2.1.6.

Syntax — Actions.

Action ::= (’entry’ | ’during’ | ’exit’) Statement
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. A Wait and ClockReset are also statements. Finally, a transition possibly includes a deadline, and its Trigger may have a ClockReset. Table 2.9 gives the concrete syntax.
Table 2.9: Timed 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 \leq d \leq 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>

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 e to variable x, sequentially composed with a call to operation op as \texttt{x=e ; 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 \texttt{Wait(d) ; op()}. A starting deadline could constrain Wait(d), however, if we were to specify this statement as an operation \texttt{waitOp()}, then the starting deadline on \texttt{waitOp()} would be satisfied immediately, whereas this would not be the case for \texttt{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.
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 $Z^{[WD96]}$.

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.

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 and required interfaces contain only variables and operations
- **I2** Defined interfaces contain only variables and events
- **I3** Names of variables, events 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 trigger, 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 trigger, or if it is used in an INPUT, SIMPLE or SYNC 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.
### 3.1 Core Language

#### 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 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.

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

#### 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.

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

#### 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.

#### 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.
Chapter 3. Well-formedness Conditions

3.1.10 Final states

FS1 Final states cannot be the source of transitions.

3.1.11 Triggers

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

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

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

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

Tg5 A trigger 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.

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 Timed Expressions

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

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

TE3 The state \( S \) in an expression \( \text{sinceEntry}(S) \) may only reference a state within the containing expression’s state-machine.

TE4 The expressions \( \text{since}(C) \) or \( \text{sinceEntry}(S) \) may only occur in a comparison expression in which the other branch is a constant. A consequence of this restriction is that no expression can compare the value of two clocks as given by \( \text{since}(C) \) or \( \text{sinceEntry}(S) \).

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.

3.2.2 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 trigger, within the trigger’s containing state-machine.

3.3 Probabilistic Language

3.3.1 ◌ Junction

P-J1 Incoming transitions to junctions cannot have actions.

P-J2 A junction that is not initial must contain at least one incoming transition.

3.3.2 ◎ Probabilistic Junction

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

PJ1 Transitions starting in probabilistic junctions cannot have triggers. This basically just restates the condition J3.

PJ2 Every outgoing transition from a probabilistic junction must be annotated with a probability value.

PJ3 The probability values of all outgoing transitions from a probabilistic junction must sum to 1. This guarantees that it is always possible to take at least one transition out of a junction.
Chapter 3. Well-formedness Conditions

PJ4 Incoming transitions to a probabilistic junction cannot have probability values and actions.

3.3.3 Transitions

PT1 A transition cannot have a trigger and a probability value at the same time. Triggers and probability values are exclusive and cannot appear together in transitions.

PT2 The source of a transition with a probability value must be a probabilistic junction. That is, states and other junctions could not be the sources of these probabilistic transitions.

PT3 The probability value of a transition must be between 0 and 1.
For the purpose of this semantics, the functions \textit{vid}, \textit{eventId}, \textit{tid} and \textit{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 \textit{Event} contains an event \textit{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 [MRLCTW19]. The complete definition is given in the sequel.
4.1 Detailed Semantics: Core Language

4.1.1 Modules

**Rule 1. Semantics of modules**

\[
\begin{align*}
\text{Rule 1. Semantics of modules} & \\
\left[ m : \text{Module} \right]_M : \text{CSPProcess} = \\
& \left( \left( \left( \left( \left( c : \text{asyncs} \bullet \text{buffer}(c) \right) \right) \right) \right) \right) \\
& \left( \left( \left( \left( \left( \text{modMemory}(m) \right) \right) \right) \right) \right) \\
& \left( \left( \left( \left( \left( \text{composeControllers}(m, \text{ctrls}, \text{cons}) \right) \right) \right) \right) \right) \\
& \Theta_{\{\text{end}\}}\text{Skip} \setminus \{\text{end}\}
\end{align*}
\]

where

- \text{ctrls} = x : m.\text{controllers}
- \text{cons} = m.\text{connections}
- \text{asyncs} = \{ c : m.\text{connections} | c.\text{async} \land \{ c.\text{from}, c.\text{to} \} \cap \text{RoboticPlatform} = \emptyset \}
- \text{evasyncs} = \{ c : \text{asyncs} \bullet \text{eventId}(c.\text{eto}) \} \cup \{ c : \text{asyncs} \bullet \text{eventId}(c.\text{efrom}) \}

**Rule 2. Hidden Module channels**

\[
\text{hiddenModuleChannels}(m : \text{Module}) : \text{ChannelSet} = \\
\{ c : \text{asyncs} \bullet \text{eventId}(c.\text{eto}) \} \cup \{ c : \text{asyncs} \bullet \text{eventId}(c.\text{efrom}) \} \cup \text{memoryChannels}^2(m)
\]

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**

\[
\text{memoryChannels}(m : \text{Module}) : \text{ChannelSet} = \\
\{ v : \text{allLocalVariables}(m) \bullet \text{set}_{\text{vid}}(v) \} \cup \\
\{ v : \text{allLocalConstants}(m) \bullet \text{set}_{\text{vid}}(v) \} \cup \\
\{ \{ c : m.\text{controllers} | \{ v : \text{requiredVariables}(c) \bullet \text{set}_{\text{EXT}}_{\text{vid}}(v, c) \} \}
\]

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**

\[
\text{allEvents}(c : \text{Context}) : \text{Set(Variable)} = \\
\text{c.events} \cup \{ \text{i : c.interfaces} \bullet \text{i.events} \}
\]
Rule 5. Function `allVariables`

```plaintext
allVariables(c : Context) : Set(Variable) =
\{ l : c.variableList | l.modifier == 'var' • l.vars \} ∪
\{ i : c.PInterfaces • \{ l : i.variableList | l.modifier == 'var' • l.vars \} \} ∪
\{ i : c.RInterfaces • \{ l : i.variableList | l.modifier == 'var' • l.vars \} \}
```

Rule 6. Function `requiredVariables`

```plaintext
requiredVariables(c : Context) : Set(Variable) =
\{ i : c.RInterfaces • \{ l : i.variableList | l.modifier == 'var' • l.vars \} \}
```

Rule 7. Function `allLocalVariables`

```plaintext
allLocalVariables(c : Context) : Set(Variable) =
\{ l : c.variableList | l.modifier == 'var' • l.vars \} ∪
\{ i : c.PInterfaces • \{ l : i.variableList | l.modifier == 'var' • l.vars \} \}
```

Rule 8. Function `allConstants`

```plaintext
allConstants(c : Context) : Set(Variable) =
\{ l : c.variableList | l.modifier == 'const' • l.vars \} ∪
\{ i : c.PInterfaces • \{ l : i.variableList | l.modifier == 'const' • l.vars \} \} ∪
\{ i : c.RInterfaces • \{ l : i.variableList | l.modifier == 'const' • l.vars \} \}
```

Rule 9. Function `requiredConstants`

```plaintext
requiredConstants(c : Context) : Set(Variable) =
\{ i : c.RInterfaces • \{ l : i.variableList | l.modifier == 'const' • l.vars \} \}
```

Rule 10. Function `allLocalConstants`

```plaintext
allLocalConstants(c : Context) : Set(Variable) =
\{ l : c.variableList | l.modifier == 'const' • l.vars \} ∪
\{ i : c.PInterfaces • \{ l : i.variableList | l.modifier == 'const' • l.vars \} \}
```
**Rule 11. Module Memory**

\[
\text{modMemory}(m : \text{Module}) : \text{CSPProcess} = \\
\begin{align*}
\text{let } & \text{Memory}(vars) \triangleq \forall v : \text{lvars} \bullet \text{set}_\text{vid}(v)?x \rightarrow \\
& (\forall c : \text{rcontrollers}(v) \bullet \text{set}_\text{Ext}_\text{vid}(v,c):x \rightarrow \text{Skip}) \ ; \text{Memory}(\text{vars}|\text{name}(v) := x)) \\
\text{within} & \text{constInit}^1(rp) \ ; \text{Memory}(\text{varvalues}) \\
\text{where} & \\
\text{rp} & = \text{roboticPlatformDefinition}(m) \\
\text{ctrls} & = m.\text{controllers} \\
\text{lvars} & = \text{allLocalVariables}^2(rp) \\
\text{vars} & = (v : \text{lvars} \bullet \text{name}(v)) \\
\text{varvalues} & = (v : \text{lvars} \bullet \text{initial}(v)) \\
\text{rcontrollers} & = \lambda v \bullet \{ c : \text{ctrls} \mid v \in \text{requiredVariables}^2(rp) \}
\end{align*}
\]

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

**Rule 12. Constants Initialisation for Controllers and Modules**

\[
\text{constInit}(node : \text{ConnectionNode}) : \text{CSPProcess} = \\
\begin{align*}
\begin{cases}
\text{if } c.\text{initial} \neq \text{NULL} \text{ then } & \text{set}_\text{vid}(c)![[c.\text{initial}]]_\text{Expr}/ \rightarrow \text{Skip} \\
\text{else } & \text{set}_\text{vid}(c)?\text{name}(c) \rightarrow \text{Skip}
\end{cases}
\end{align*}
\]

\[
\text{where} \\
\text{consts} = \text{allConstants}^1(node)
\]

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 13. Composition of controllers
composeControllers(m : Module, ctrls : Seq(Controller), cons : Set(Connection)) : CSPProcess =
if #ctrls = 1
then

  renamingController1(m, headctrls, cons)
else

  renamingController2(m, headctrls, cons)

  composeControllers2(m, tailctrls, cons)
where

  connevts = renCtrlEvts1(m, headctrls, cons) \n  \cup {c : tailctrls \bullet renCtrlEvts2(m, c, cons)}

Rule 14. Renaming controller
renamingController(m : Module, c : Controller, cons : Set(Connection)) : CSPProcess =

  {{e : internalConns \bullet eventId(e.efrom).in <- eventId(e.efrom).out} \n  \cup {e : internalConns \bullet eventId(e.efrom).out <- eventId(e.efrom).in} \n  \cup {e : fromPlatform \bullet eventId(e.efrom) <- eventId(e.efrom)} \n  \cup {e : toPlatform \bullet eventId(e.efrom) <- eventId(e.efrom)} \n  \cup \{x \bullet requiredConstants1(c) \bullet set.vid(x) <- set.vid(x, m)}

where

  internalConns = \{x : cons \bullet \{x.from, x.to\} \subseteq Controller \land \neg x.async \land c \in \{x.from, x.to\}\}
  toPlatform = \{x : cons \bullet x.from = c \land x.to \in RoboticPlatform\}
  fromPlatform = \{x : cons \bullet x.to = c \land x.from \in RoboticPlatform\}

The controller’s set_ events for required constants are renamed to match the set_ event for the corresponding provided constants of the module (container(c)).

Rule 15. Renaming controller events
renCtrlEvts(m : Module, c : Controller, cons : Set(Connection)) : ChannelSet =

  \{(x : internalConns \bullet eventId(e.efrom)) \cup \{end\}\} \cup \{x : requiredConstants2(c) \bullet set.vid(x, m)}

where

  internalConns = \{x : cons \bullet \{x.from, x.to\} \subseteq Controller \land \neg x.async \land c \in \{x.from, x.to\}\}

Controllers that require the same constant synchronise with each other on the set_ event of the module, as well as with the module memory.
Rule 16. Buffer

\[
\text{buffer}(c : \text{Connection}) : \text{CSPProcess} = \\
\begin{cases}
\text{if } c.m \text{ then } & \text{singleBuffer}(c.efrom, c.e) \ || \ \text{singleBuffer}(c.e, c.efrom) \\
\text{else} & \text{singleBuffer}(c.efrom, c.e)
\end{cases}
\]

Rule 17. Single buffer

\[
\text{singleBuffer}(efrom : \text{Event}, eto : \text{Event}) : \text{CSPProcess} = \\
\begin{cases}
\text{if } \text{efrom.type} \neq \text{null} & \\
\quad \begin{cases}
\text{let } Buffer() \equiv \text{eventId}(efrom).out?x \rightarrow Buffer(x) \\
\quad Buffer(v) \equiv \text{eventId}(efrom).out?x \rightarrow Buffer(x) \ □ \ \text{eventId}(eto).in!v \rightarrow Buffer() \\
\quad \text{within Buffer()}
\end{cases} \\
\text{else } & \\
\quad \begin{cases}
\text{let } Buffer(false) \equiv \text{eventId}(efrom).out \rightarrow Buffer(true) \\
\quad Buffer(true) \equiv \text{eventId}(efrom).out \rightarrow Buffer(true) \ □ \ \text{eventId}(eto).in \rightarrow Buffer(false) \\
\quad \text{within Buffer(false)}
\end{cases}
\end{cases}
\]

4.1.2 Controllers

Rule 18. Semantics of controllers

\[
\begin{bmatrix}
[c : \text{ControllerDef}]_c : \text{CSPProcess} = \\
\begin{cases}
\text{let } & \\
\quad \begin{cases}
\text{for each } op : c.iOperations & \\
\quad \text{id}(op)(\{x : op.parameters \ • x.name\}) = \text{opdef}^{\text{nops}} \text{STM}
\end{cases} \\
\text{within } & \\
\quad \left( \\
\quad \begin{cases}
\text{composeMachines}^1(c, ms, cs)^{\text{nops}} \\
\quad \text{lvars} \cup \text{rvars} \cup \text{lconsts} \cup \text{rconsts}
\end{cases} \\
\quad \text{ctrlMemory}^1(c)
\right) \ \Theta \_{\text{end}} \text{Skip}
\end{cases}
\end{bmatrix}
\]

where
\[
\begin{align*}
ms &= (x : c.machines) \\
cs &= c.connections \\
opdef &= \text{findOperationDefinition}(op, c.iOperations) \\
nops &= \{ op : c.iOperations \ • \text{id}(op) \rightarrow nproc(op) \} \\
lvars &= \{ v : \text{allLocalVariables}^3(c) \ • \text{set}_\text{vid}(v) \} \\
rvars &= \{ v : \text{requiredVariables}^3(c) \ • \text{set}_\text{Ext}_\text{vid}(v) \} \\
lconsts &= \{ v : \text{allLocalConstants}^3(c) \ • \text{set}_\text{vid}(v) \} \\
rconsts &= \{ v : \text{requiredConstants}^3(c) \ • \text{set}_\text{vid}(v) \}
\end{align*}
\]

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...
Rule 19. Controller Memory

\[ \text{ctrlMemory}(c : \text{ControllerDef}) : \text{CSPProcess} = \]
\[
\begin{align*}
\text{let } & \text{Memory}(\vars) = \\
& \begin{cases} \\
\forall v : \text{iVars} \cdot \text{set}_\text{vid}(v) ? x \rightarrow \\
(\forall m : \text{rmachines}(v) \cdot \text{set}_\text{Ext}_\text{vid}(v, m) ! x \rightarrow \text{Skip}) ; \\
\text{Memory}(\vars[name(v) := x]) \end{cases} \\
\end{align*}
\]

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

where

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

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 20. Composition of machines

\[ \text{composeMachines}(c : \text{Controller}, \text{ms} : \text{Seq(StateMachineDef)}, \text{cons} : \text{Set(Connection)})^\text{nops} : \text{CSPProcess} = \]
\[
\begin{align*}
\text{if} \#\text{ms} &= 1 \\
\text{then} & \ \text{renamingMachine}^1(c, \text{headms}, \text{cons})^\text{nops} \\
\text{else} & \ \text{renamingMachine}^2(c, \text{headms}, \text{cons})^\text{nops} \\
& \{ \text{connevs} \} \\
& \text{composeMachines}^2(c, \text{tailms}, \text{cons})^\text{nops} \\
\end{align*}
\]

where

\[ \text{connevs} = \text{renStmEvts}^1(c, \text{headms}, \text{cons}) \cap \bigcup \{ m : \text{tailms} \cdot \text{renStmEvts}^2(c, m, \text{cons}) \} \]
**Rule 21. Renaming state machine**

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

\[
[\text{stm}]^{n_{\text{ops}}}_{\text{STM}} : \text{CSPProcess} = \]

\[
\begin{aligned}
\text{initialisation}^{1}(\text{stm})^{n_{\text{ops}}} & \\
\text{composeStates}^{1}(\text{stm})^{n_{\text{ops}}} & \cup \{\text{enter}, \text{entered}, \text{exit}, \text{exited}\} \\
\text{getsetChannels}^{1}(\text{stm}) & \cup \text{trigEvents}^{1}(\text{stm}) \\
\text{stmMemory}^{1}(\text{stm}) & \\
\text{renameTriggerEvents}^{1}(\text{stm}) & \\
\text{getsetLocalChannels}^{1}(\text{stm}) & \cup \{\text{internal}\} \\
\text{end} \text{Skip} & \\
\end{aligned}
\]

where

flowevts =

\[
\bigcup \{\text{SID}, \text{states}^{2}(\text{stm}) : y : \text{states}^{2}(\text{stm}) \cup \{\text{enter}, \text{entered}, \text{exit}, \text{exited}\} \}
\]

**Rule 22. Renaming machine events**

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

\[
\{x : \text{internalConns} \cup \text{eventId}(\text{efrom})\} \cup \{\text{end}\} \cup \{x : \text{requiredConstants}^{2}(m) \cup \text{set_vid}(v, c)\}
\]

where

internalConns = \{x : \text{cons} \cup \{x,\text{from}, x,\text{to}\} \subseteq \text{StateMachine} \land m \in \{x,\text{from}, x,\text{to}\}\}

toController = \{x : \text{cons} \cup x,\text{from} = m \land x,\text{to} \in \text{Controller}\}

fromController = \{x : \text{cons} \cup x,\text{to} = m \land x,\text{from} \in \text{Controller}\}

**4.1.3 State machines**

**Rule 23. Semantics of state machine**

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

\[
\begin{aligned}
\text{initialisation}^{1}(\text{stm})^{n_{\text{ops}}} & \\
\text{composeStates}^{1}(\text{stm})^{n_{\text{ops}}} & \cup \{\text{enter}, \text{entered}, \text{exit}, \text{exited}\} \\
\text{getsetChannels}^{1}(\text{stm}) & \cup \text{trigEvents}^{1}(\text{stm}) \\
\text{stmMemory}^{1}(\text{stm}) & \\
\text{renameTriggerEvents}^{1}(\text{stm}) & \\
\text{getsetLocalChannels}^{1}(\text{stm}) & \cup \{\text{internal}\} \\
\text{end} \text{Skip} & \\
\end{aligned}
\]

where

flowevts =

\[
\bigcup \{x : \text{SID}, \text{states}^{2}(\text{stm}) : y : \text{states}^{2}(\text{stm}) \cup \{\text{enter}, \text{entered}, \text{exit}, \text{exited}\} \}
\]

**Rule 24. Rename Transition Trigger Events**

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

\[
\{e : \text{allEvents}^{1}(m) ; \text{tid} : \text{TIDS} \cup \text{eventId}(e) \cup \text{eventId}(e)\}
\]
Rule 25. Function states

\[\text{states}(n : \text{NodeContainer}) : \text{Set}(\text{State}) = x.\text{nodes} \cap (\text{State} \cup \text{Final})\]

Rule 26. Initialisation

\[\text{initialisation}(n : \text{NodeContainer})^{\text{ops}} : \text{CSPProcess} = \]
\[
\begin{cases}
\text{if } js = \emptyset \text{ then } \\
\text{else let } \\
\quad \text{for each } j : js \star \text{id}(j) = \left\{ j, n, \text{true} \right\}^{\text{Skip}} \text{Skip} \text{Skip} \text{ops} \\
\quad \text{within } \\
\quad \left\{ (t : \text{Transition} \mid t.\text{source} = i), n, \text{true} \right\}^{\text{Skip}} \text{Skip} \text{Skip} \text{ops} \\
\end{cases}
\]

where
\#n.\text{nodes} > 0
\ i = \#x : n.\text{nodes} \mid x \in \text{Initial}
\ js = \text{reachableJunctions}^1(i)

Rule 27. Reachable Junctions from Node

\[\text{reachableJunctions}(n : \text{Node}) : \text{Set}(\text{Node}) =\
\{ t : \text{parent}(n).\text{transitions} \mid t.\text{target} \in \text{Junction} \star t.\text{source} \rightarrow t.\text{target} \}^+ \{ \{ n \} \}
\]

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

Rule 28. Semantics of Junctions

\[
\text{[[} j : \text{Junction}, \text{origin} : \text{NodeContainer}, \text{initial} : \text{boolean}]^{P, Q, \text{ops}} : \text{Set}(\text{Node}) =\
\begin{cases}
\bigwedge (t : \text{transitionsFrom}^1(j) \star \{ t, o, i \}^{P, Q, \text{ops}}) \\
\end{cases}
\]

Rule 29. Get and Set channels

\[\text{getsetChannels}(s : \text{StateMachineDef}) : \text{ChannelSet} =\
\begin{cases}
\{ v : \text{allVariables}^1(s) \star \text{getVid}(v) \} \cup \{ v : \text{allVariables}^2(s) \star \text{setVid}(v) \} \cup \\
\{ v : \text{allConstants}^2(s) \star \text{getVid}(v) \} \cup \{ v : \text{allConstants}^3(s) \star \text{setVid}(v) \}
\end{cases}
\]
Rule 30. Composition of states

\[
\text{composeStates}(ss : \text{seq}\text{State}, p : \text{NodeContainer})^{\text{nops}} : \text{CSPProcess} =
\]

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

where

\[
\text{cflowevts} = \text{flowEvents}^1(\text{head}\, ss, p) \cap \bigcup \{ x : \text{tail}\, ss \cdot \text{flowEvents}^2(x, p) \}
\]

Rule 31. Trigger events

\[
\text{trigEvents}(s : \text{StateMachineDef}) : \text{ChannelSet} =
\]

\[
\{ \{ t : \text{allTransitions}^1(s) \cdot \text{triggerEvent}^1(t.\text{trigger}, \text{id}(t)) \}
\]

Rule 32. Get and set local channels

\[
\text{getsetLocalChannels}(s : \text{StateMachineDef}) : \text{ChannelSet} =
\]

\[
\{ v : \text{allVariables}^3(s) \cdot \text{get}_{\text{vid}}(v) \} \cup \{ v : \text{allLocalVariables}^5(s) \cdot \text{set}_{\text{vid}}(v) \} \cup
\]

\[
\{ v : \text{allConstants}^4(s) \cdot \text{get}_{\text{vid}}(v) \} \cup \{ v : \text{allLocalConstants}^3(s) \cdot \text{set}_{\text{vid}}(v) \}
\]

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 33. Flow events

\[
\text{flowEvents}(s : \text{State}, p : \text{NodeContainer}) : \text{ChannelSet} =
\]

\[
\bigcup \{ x : \text{states}^1(p); y : \{ \text{id}(s) \} \cdot \}
\]

\[
\text{enter}.y.x, \text{entered}.y.x, \text{exit}.y.x, \text{exited}.y.x, \\
\text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y,
\}
\]

Rule 34. Semantics of states

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

This function is split in multiple rules according to the type of states.
Rule 35. Semantics of simple states

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

let

$\text{Inactive} \equiv \text{enter}\; ? o : \text{sid}s, \text{id}(s) \rightarrow \text{Activating}(o)$

$\text{Activating}(o) \equiv [[s, \text{entry}]]^{\text{nops}}_{\text{s}} ; \text{entered} \; o, \text{id}(s) \rightarrow$

$([[s, \text{during}]]^{\text{nops}}_{\text{s}} ; \text{Stop}) \Delta$

\[
\begin{align*}
\&; t : \text{transitionsFrom}^{2}(s) \bullet \text{activate} \; t, s, \text{false}] \quad \text{Inactive, Activating, ops} \\
\&; e : \text{Event} \bullet \text{if (e.type} = \text{null)} \then \text{eventId}(e) \; ? x : \text{tids} \; ? d \rightarrow \text{exit}; \text{Inactive} \\
\&; \text{else eventId}(e) \; ? x : \text{tids} \; ? d \rightarrow \text{exit}; \text{Inactive} \\
\text{for each } j : \text{reachable Junctions}^{2}(s) \bullet \text{id}(j) = [[j, s, \text{false}]] \text{Inactive, Activating, ops} \\
\end{align*}
\]

within $\text{Inactive}$

where

$\#\text{states}^{2}(s) = 0$

$\text{sid}s = \text{SIDS} \setminus \{\text{id}(s)\} \land \text{tids} = \text{TIDS} \setminus \text{TIDS}(s)$

$\text{exit} = \text{exit} \; ? \text{as} : \text{sid}s, \text{id}(s) \rightarrow [[s, \text{exit}]]^{\text{nops}}_{\text{s}} ; \text{exited}\; \text{as}, \text{id}(s) \rightarrow \text{Skip}$

Rule 36. Semantics of composite states

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

let

$\text{Inactive} \equiv \text{enter}\; ? o : \text{sid}s, \text{id}(s) \rightarrow \text{Activating}(o)$

$\text{Activating}(o) \equiv [[s, \text{entry}]]^{\text{nops}}_{\text{s}} ; \text{initialisation}^{2}(s)^{\text{nops}} ; \text{entered} \; o, \text{id}(s) \rightarrow$

$([[s, \text{during}]]^{\text{nops}}_{\text{s}} ; \text{Stop}) \Delta$

\[
\begin{align*}
\&; t : \text{transitionsFrom}^{2}(s) \bullet \text{activate} \; t, s, \text{false}] \quad \text{Inactive, Activating, ops} \\
\&; e : \text{Event} \bullet \text{if (e.type} = \text{null)} \then \text{eventId}(e) \; ? x : \text{tids} \; ? d \rightarrow \text{exit}; \text{Inactive} \\
\&; \text{else eventId}(e) \; ? x : \text{tids} \; ? d \rightarrow \text{exit}; \text{Inactive} \\
\text{for each } j : \text{reachable Junctions}^{2}(s) \bullet \text{id}(j) = [[j, s, \text{false}]] \text{Inactive, Activating, ops} \\
\end{align*}
\]

within $\text{Inactive} [[\text{flowtrigevts}]]$ $\text{composeStates}^{2}(x : \text{states}^{5}(s))^{\text{nops}} \setminus \text{flowevts}$

where

$\#\text{states}^{5}(s) = 0$

$\text{flowevts} = \bigcup \{x : \text{SIDS}, \text{states}^{7}(s) ; y : \text{states}^{5}(s) \bullet \text{enter} : x, y, \text{entered} : x, y, \text{exit} : x, y, \text{exited} : x, y\}$

$\text{flowtrigevts} = \text{flowTriggerEvents}^{1}(s)$

$\text{sid}s = \text{SIDS} \setminus \{\text{id}(s)\} \land \text{tids} = \text{TIDS} \setminus \text{TIDS}(s)$

$\text{exit} = \text{exit} \; ? \text{as} : \text{sid}s, \text{id}(s) \rightarrow \text{exitSubstates}^{2}(s) ; [[s, \text{exit}]]^{\text{nops}}_{\text{s}} ; \text{exited}\; \text{as}, \text{id}(s) \rightarrow \text{Skip}$
Rule 37. Semantics of final states

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

$$\begin{align*}
\text{enter} ? x : \text{sids.id}(s) \rightarrow \text{entered}.x.(id(s) \rightarrow \begin{cases}
\text{if (parent}(s) \in \text{StateMachine}) & \text{end} \rightarrow \text{Skip} \\
\text{else} & \text{Stop}
\end{cases}
\end{align*}$$

Rule 38. Synchronisation events between parent state and substates

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

$$\begin{align*}
\{e : \text{Event}; t : \text{TIDS} \bullet e.t\} \setminus (\text{substatesTriggers}^1(s) \cup \text{non_interrupting_reachable_transitions}) \cup \\
\bigcup \{x : \text{SIDs}_s, y : \text{states}^9(s) \bullet \{\text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y\}\}
\end{align*}$$

where

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

Rule 39. Reachable transitions

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

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

Rule 40. Triggers of substates

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

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

Rule 41. Restricted semantics of states

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

$$[s]^{\text{nops}} : \text{CSPProcess} = \begin{cases}
\text{Skip} & \text{where} \\
\text{tidsfromwithin} = \{t : \text{transitionsFrom}^6(s) \cup \text{allTransitions}^3(s) \bullet \text{id}(t)\}
\end{cases}$$

$$\text{all_other_transitions}_S = \{e : \text{Event}; t : \text{TIDS} \setminus \text{tidsfromwithin} \bullet \text{eventId}(e).tId\}$$

$$\text{all_transitions}_PS = \{e : \text{Event}; t : \text{TIDS} \bullet \text{eventId}(e).tId\}$$

$$\setminus \{t : \text{allTransitions}^2(p) \bullet \text{eventId}(t.\text{trigger}.\text{event}).t\}$$
Rule 42. Semantics of transitions

\[ [t : \text{Transition}, \text{origin} : \text{NodeContainer}, \text{initial} : \text{boolean}] P, Q, \text{ops} : \text{CSPProcess} = \]

\[
\text{if} \ \text{src} \in \text{State}
\]

\[ [t.\text{trigger}]_{\text{id}(t)} \ ; \ \text{exit}.\text{id}((\text{src})) \rightarrow \text{exitSubstates}_2(\text{src}) ; [\text{src}.\text{exit}]_{\text{ops}} ; \text{compileTarget}_1(\text{tgt}, \text{src}, \text{false}) P, Q \]

\[
\text{else if} \ \text{src} \in \text{Initial}
\]

\[ \text{internal}.\text{id}(t) \rightarrow [t.\text{action}]_{\text{ops}} ; \text{compileTarget}_2(\text{tgt}, \text{parent}((\text{src})), \text{true}) P, Q \]

\[
\text{else if} \ \text{src} \in \text{Junction}
\]

\[ \text{internal}.\text{id}(t) \rightarrow [t.\text{action}]_{\text{ops}} ; \text{compileTarget}_3(\text{tgt}, \text{origin}, \text{initial}) P, Q \]

where

\[
\text{src} = t.\text{source} \\\n\text{tgt} = t.\text{target} \]

Rule 43. Compile target

\[ \text{compileTarget}(\text{tgt} : \text{Node}, o : \text{NodeContainer}, i : \text{boolean}) P, Q : \text{CSPProcess} = \]

\[
\text{if} (\text{tgt} \in \text{State}) \text{then}
\]

\[ \text{if} (\text{tgt} = o) \text{then} \text{enter}.\text{id}(o).\text{id}(\text{tgt}) \rightarrow Q \]

\[ \text{else} \ \text{enter}.\text{id}(o).\text{id}(\text{tgt}) \rightarrow \text{entered}.\text{id}(o).\text{id}(\text{tgt}) \rightarrow \left( \begin{array}{c} \text{if} (i) \text{then} \text{Skip} \\ \text{else} P \end{array} \right) \]

\[
\text{else if} (\text{tgt} \in \text{Junction}) \text{then}
\]

\[ \text{id}(\text{tgt}) \]

Rule 44. Exit substates

\[ \text{exitSubstates}(s : \text{NodeContainer}) : \text{CSPProcess} = \]

\[
\text{if} \ #\text{states}(s) > 0 \text{then}
\]

\[ \text{exit}.\text{id}(s)_2 z : \{ x : \text{states}^1(s) : \text{id}(x) \} \rightarrow \text{exit}.\text{id}(s).z \rightarrow \text{Skip} \]

\[ \text{else} \\
\text{Skip} \]
Rule 45. State Machine Memory

\[
\text{stmMemory}(\text{stm} : \text{StateMachineDef}) : \text{CSPProcess} =
\]

\[
\text{let Memory}(\text{vars}) \triangleq
\]

\[
\begin{align*}
\text{let } & v \cdot \text{lvars} \bullet (\text{get}_\text{vid}(v) \cdot \text{name}(v) \rightarrow \text{Memory}(\text{vars})) \\
& \quad \text{let } x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]) \\
\text{let } & v \cdot \text{rvars} \bullet (\text{get}_\text{vid}(v) \cdot \text{name}(v) \rightarrow \text{Memory}(\text{vars})) \\
& \quad \text{let } x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]) \\
\text{let } & v \cdot \text{allConstants}(\text{stm}) \bullet \text{get}_\text{vid}(v) \cdot \text{name}(v) \rightarrow \text{Memory}(\text{vars}) \\
\text{let } & \cdot \text{allTransitions}(\text{stm}) \bullet \text{memoryTransition}(t) : \text{Memory}(\text{vars})
\end{align*}
\]

\[
\text{within } \text{constInitSTM}(\text{consts}, \text{stm}, \text{Memory}(\text{varvalues}))
\]

where

\[
\begin{align*}
\text{rvars} &= \text{requiredVariables}(\text{stm}) \\
\text{lvars} &= \text{allLocalVariables}(\text{stm}) \\
\text{consts} &= (v : \text{allConstants}(\text{stm}) \bullet v) \\
\text{vars} &= (v : \text{rvars} \cup \text{lvars} \circ \text{name}(v)) \cap (v : \text{consts} \circ \text{name}(v)) \\
\text{varvalues} &= (v : \text{rvars} \cup \text{lvars} \circ \text{initial}(v)) \cap (v : \text{consts} \circ \text{name}(v))
\end{align*}
\]

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 \text{get} channel. Only noninitialised constants are read. The initial value of initialised constants are passed directly to the recursive process \text{Memory}.

Rule 46. Constants Initialisation for State Machines

\[
\text{constInitSTM}(\text{cs} : \text{Seq}_1(\text{Variable}), \text{stm} : \text{StateMachineDef}, \text{P} : \text{CSPProcess}) : \text{CSPProcess} =
\]

\[
\text{buildScope}(\text{undefc}, \text{stm}, \text{let } c : \text{defs} \bullet \text{name}(c) = [[c.\text{initial}]]_{\text{Expr}^2} \text{ within } \text{P})
\]

where

\[
\begin{align*}
\text{defc} &= (x : \text{consts} \mid x.\text{initial} \neq \text{null}) \\
\text{undefc} &= (x : \text{consts} \mid x.\text{initial} = \text{null})
\end{align*}
\]
Rule 47. Build Scope

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

\[
\begin{align*}
\text{if} (\text{headcs}.\text{initial} == \text{NULL} \text{then} \\
\quad \text{set}_\text{vid}(\text{headcs}, \text{ctx})?\text{name}(\text{headcs}) \rightarrow (\text{if} \#cs == 1 \text{then} P \\
\quad \text{else buildScope}(2(\text{tailcs}, \text{ctx}, P)) \\
\quad \text{else} (\text{if} \#cs == 1 \text{then} P \\
\quad \text{else buildScope}(3(\text{tailcs}, \text{ctx}, P))))
\end{align*}
\]

Rule 48. Semantics of triggers

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

\[
\begin{align*}
\text{if} t.\text{event}.\text{type} \neq \text{null} \\
\quad \text{eventId}(t.\text{event}).\text{tid}.\text{in}?x \rightarrow \text{set}_\text{vid}(t.\text{parameter})!x \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(t.\text{event}).\text{tid}.\text{in} \rightarrow \text{Skip}
\end{align*}
\]

Rule 49. Semantics of triggers for memory

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

\[
\begin{align*}
\text{if} t.\text{event}.\text{type} \neq \text{null} \\
\quad \text{eventId}(t.\text{event}).\text{tid}.\text{in}?x \rightarrow \text{Skip} \\
\text{else} \\
\quad \text{eventId}(t.\text{event}).\text{tid}.\text{in} \rightarrow \text{Skip}
\end{align*}
\]

Rule 50. Event for transition trigger

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

\[
\text{eventId}(t.\text{event}).\text{tid}
\]

Rule 51. Memory transitions

\[
\text{memoryTransition}(t : \text{Transition}) : \text{CSPProcess} =
\]

\[
\begin{align*}
\text{if} (t.\text{condition} \neq \text{null}) \text{then} \\
\quad (\{t.\text{condition} \}^{\text{expr}}) \& \text{triggerForMemory}(1(t.\text{trigger}.\text{id}(t)) \\
\text{else} \\
\quad \text{triggerForMemory}(2(t.\text{trigger}.\text{id}(t)))
\end{align*}
\]
Chapter 4. Semantics

Rule 52. Function transitionsFrom

transitionsFrom(s : Node) : Set(Transition) =
{ t : parent(s).transitions | t.source = s • t }

Rule 53. Function allTransitions

allTransitions(s : NodeContainer) : Set(Transition) =
s.transitions ∪ ∪{ x : s.nodes | s ∈ State • allTransitions(x) }

4.1.4 Statements

Rule 54. 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 55. Read state of an expression

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

\[
\begin{align*}
\text{if} & \ (#vs = 0) \text{ then} \\
& P \\
\text{else} & \text{ get}_v(id(head vs)) ? (head vs).name \rightarrow \text{readState}(\text{tail vs}, P)
\end{align*}
\]

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 \( \llbracket - \rrbracket_{\text{StatementInContext}} \) to separate the application of \( \text{readState} \) from the core semantics of the statement given by the rule \( \llbracket - \rrbracket_{\text{Statement}} \). 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 56. Semantics of statements in context
\[ [s : \text{Statement}]^{nops} \rightarrow [\text{Statement} \rightarrow \text{CSPProcess}] = \]
readState\(^2\) (usedVariables\(^1\) (s), [s]^{nops})

Rule 57. Function usedVariables
usedVariables(s : Statement) : Set(Variable) =

\[
\begin{align*}
\text{ifs } & \in \text{Assignment} \text{ then} \\
& \quad \text{usedVariables}\(^2\) (s.right) \\
\text{else if } & \text{ifs } \in \text{Call} \text{ then} \\
& \quad \bigcup \{x : s \text{.args } \bullet \text{usedVariables}\(^3\) (x)\} \\
\text{else if } & \text{ifs } \in \text{IfStatement} \text{ then} \\
& \quad \text{usedVariables}\(^4\) (s \text{.expression}) \\
\text{else if } & \text{ifs } \in \text{SendEvent} \wedge s \text{.trigger.type } \in \{\text{SYNC}, \text{OUTPUT}\} \text{ then} \\
& \quad \text{usedVariables}\(^5\) (s \text{.trigger.value}) \\
\text{else if } & \text{ifs } \in \text{TimedStatement} \text{ then} \\
& \quad \text{usedVariables}\(^6\) (s \text{.stmt}) \cup \text{usedVariables}\(^7\) (s \text{.start}) \cup \text{usedVariables}\(^8\) (s \text{.end}) \\
\text{else} & \{} \\
\end{align*}
\]

Rule 58. Function usedVariables
usedVariables(e : Expression) : Set(Variable) =
The definition of this function is standard and omitted for now.

Rule 59. Semantics of assignment
\[ [s : \text{Assignment}]^{nops} \rightarrow [\text{Statement} \rightarrow \text{CSPProcess}] = \]
set.vid(s.left) !(s.right) \rightarrow \text{Expr} \rightarrow \text{Skip}

Rule 60. Semantics of call statement
\[ [s : \text{Call}]^{nops} \rightarrow [\text{Statement} \rightarrow \text{CSPProcess}] = \]

\[
\begin{align*}
\text{op.name} \rightarrow \text{call}(\text{op})(\{x : \text{s.args } \bullet \text{Expr}^5\});
\text{op.name} \rightarrow \text{Ret} \rightarrow \text{Skip}
\end{align*}
\]

where
\[
\text{op } = \text{s.operation}
\]
Rule 61. Semantics of if statements
\[
\text{CSPProcess} = \left\{ \begin{array}{ll}
\text{Exp} & \text{if } [s.\text{expression}] \\
\text{StatementInContext} & \text{then} [s.\text{then}] \\
\text{StatementInContext} & \text{else if } [s.\text{else}] \neq \text{null} \text{ then } [s.\text{else}] \\
\text{Skip} & \text{else}
\end{array} \right.
\]

Rule 62. Semantics of send event statements
\[
\text{CSPProcess} = \left\{ \begin{array}{ll}
\text{Skip} & \text{if } (\text{type} = \text{INPUT}) \\
& \text{then } \text{eventId}(\text{event}).\text{in} \text{?par.name} \rightarrow \text{set}_\text{vid}(\text{par}) \text{par.name} \rightarrow \text{Skip} \\
\text{Skip} & \text{if } (\text{type} = \text{OUTPUT}) \\
& \text{then } \text{eventId}(\text{event}).\text{out}! [\text{value}] \rightarrow \text{Exp} \\
\text{Skip} & \text{if } (\text{type} = \text{SIMPLE}) \\
& \text{then } \text{eventId}(\text{event}).\text{out} \rightarrow \text{Skip} \\
\text{Skip} & \text{else } \text{eventId}(\text{event}).\text{out}. [\text{value}] \rightarrow \text{Exp}
\end{array} \right.
\]

where
- type = s.trigger.type
- event = s.trigger.event
- value = s.trigger.value
- par = s.trigger.parameter

Rule 63. Semantics of sequential composition
\[
\text{CSPProcess} = \{ x : s.\text{statements} \bullet x \} \text{StatementInContext}
\]

Rule 64. Semantics of skip
\[
\text{CSPProcess} = \text{Skip}
\]

Rule 65. Semantics of actions
\[
\text{CSPProcess} = \{ a.action \} \text{StatementInContext}
\]

4.1.5 Expressions
Rule 66. Semantics of expressions
\[ [s : \text{Expression}]_{Exp} : \text{CSPExpression} = \]
This rule is split in multiple rules according to the subtype of the expression.

Rule 67. Semantics of and expression
\[ [s : \text{And}]_{Exp} : \text{CSPExpression} = \]
\[ [s.\text{left}]_{Exp} \land [s.\text{right}]_{Exp} \]

Rule 68. Semantics of array expression
\[ [s : \text{ArrayExp}]_{Exp} : \text{CSPExpression} = \]
\[ [s.\text{value}]_{Exp} (\{p : s.\text{parameters} \cdot [p]_{Exp} \}) \]

Rule 69. Semantics of boolean expression
\[ [s : \text{BooleanExp}]_{Exp} : \text{CSPExpression} = \]
\[ \text{if} (s.\text{value} = \text{TRUE}) \text{then} \text{true} \text{else} \text{false} \]

Rule 70. Semantics of call expression
\[ [s : \text{CallExp}]_{Exp} : \text{CSPExpression} = \]
\[ \text{if}(\text{name} = \text{‘size’} \land (\text{heads.\text{args} has type SetType})) \text{then} \]
\[ \text{card}([\text{heads.\text{args}]}_{Exp}) \]
\[ \text{elseif}(\text{name} = \text{‘size’} \land (\text{heads.\text{args} has type SeqType})) \text{then} \]
\[ \text{length}([\text{heads.\text{args}]}_{Exp}) \]
\[ \text{else} \]
\[ \text{name}([a : s.\text{args} \cdot [a]_{Exp}]) \]
\[ \text{where} \]
\[ \text{name} = s.\text{function.name} \]

Rule 71. Semantics of concatenation expression
\[ [s : \text{Cat}]_{Exp} : \text{CSPExpression} = \]
\[ [s.\text{left}]_{Exp} \lhd [s.\text{right}]_{Exp} \]
<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td><strong>Semantics of not equal expression</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{Different}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} \neq [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>73</td>
<td><strong>Semantics of division</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{Div}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} / [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>74</td>
<td><strong>Semantics of equality</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{Equals}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} = [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>75</td>
<td><strong>Semantics of greater or equal expression</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{GreaterOrEqual}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} \geq [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>76</td>
<td><strong>Semantics of greater than</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{GreaterThan}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} &gt; [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>77</td>
<td><strong>Semantics of if and only if expression</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{Iff}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} \leftrightarrow [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>78</td>
<td><strong>Semantics of implication</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{Implies}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$\left[ [s.\text{left}]_{\text{Expr}^{\text{left}}} \Rightarrow [s.\text{right}]_{\text{Expr}^{\text{right}}} \right]$</td>
</tr>
<tr>
<td>79</td>
<td><strong>Semantics of integer expression</strong></td>
</tr>
<tr>
<td></td>
<td>$[s : \text{IntegerExp}]_{\text{Expr}} : \text{CSPExpression} =$</td>
</tr>
<tr>
<td></td>
<td>$s.\text{value}$</td>
</tr>
</tbody>
</table>
4.1 Detailed Semantics: Core Language

**Rule 80. Semantics of less or equal expression**

\[
[[s : \text{LessOrEqual}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{32} \leq [[s.\text{right}]], \text{Expr}^{33}
\]

**Rule 81. Semantics of less than**

\[
[[s : \text{LessThan}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{34} < [[s.\text{right}]], \text{Expr}^{35}
\]

**Rule 82. Semantics of minus**

\[
[[s : \text{Minus}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{36} - [[s.\text{right}]], \text{Expr}^{37}
\]

**Rule 83. Semantics of modulus**

\[
[[s : \text{Modulus}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{38} \mod [[s.\text{right}]], \text{Expr}^{39}
\]

**Rule 84. Semantics of multiplication**

\[
[[s : \text{Mult}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{40} \times [[s.\text{right}]], \text{Expr}^{41}
\]

**Rule 85. Semantics of arithmetic negation**

\[
[[s : \text{Neg}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
-[[s.\text{exp}]], \text{Expr}^{42}
\]

**Rule 86. Semantics of logical negation**

\[
[[s : \text{Not}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
\neg [[s.\text{exp}]], \text{Expr}^{43}
\]

**Rule 87. Semantics of or expression**

\[
[[s : \text{Or}]_{\text{expr}} : \text{CSPExpression} = ]
\]

\[
[[s.\text{left}]], \text{Expr}^{44} \lor [[s.\text{right}]], \text{Expr}^{45}
\]
Chapter 4. Semantics

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 trigger 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 whose trigger is 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 \( \text{stmClocks}(\text{stm}, \text{wcs}) \), which relies on the calculation of \( \text{wcs} \), 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 \( \text{stmMemory}(\text{stm}, \text{wcs}) \) also takes \( \text{wcs} \) 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 94. Semantics of state machine**

\[
[\text{stm} : \text{StateMachineDef}](\text{stm})^{nops} : \text{TimedCSPProcess} =
\]

\[
\left( \begin{array}{c}
\text{initialisation}^3(\text{stm})^{nops} \\
\text{composeStates}^4((x : \text{stm}.\text{nodes} | x \in \text{State}).\text{stm})^{nops} \\
\{ \text{enter, exit, exited} \} \\
\text{getsetChannels}^2(\text{stm}) \cup \text{trigEvents}^2(\text{stm}) \cup \text{clockResets}^2(\text{wcs}) \cup \text{deadlineEvents}^2(\text{stm}) \\
\text{constInitSTM}^1(\text{consts}, \text{stm}, \text{stmMemory}^1(\text{stm}, \text{wcs}) \{ \text{clockMemSync} \ \text{stmClocks}^1(\text{stm}) \}) \\
\text{renameTriggerEvents}^2(\text{stm}) \\
\text{getsetLocalChannels}^2(\text{stm}) \cup \text{clockResets}^2(\text{wcs}) \cup \text{deadlineEvents}^2(\text{stm}) \cup \{ \text{internal, entered} \} \\
\Theta_{\text{end}} \text{Skip}
\end{array} \right)
\]

where

\[
\text{wcs} = \{ t : \text{allTransitions}^7(\text{stm}) | t.\text{condition} \neq \text{null} \land t \rightarrow \text{wc}(t.\text{condition}) \}
\]

\[
\text{clockMemSync} = \left( \begin{array}{c}
\{ t : \text{Transition} | t \in \text{dom} \text{wcs} \cup \text{triggerEvent}^1(t) \} \\
\cup \{ v : \text{allClockVariables}^1(\text{wcs}) \cup \text{setWC}^1(\text{vid}(v)) \}
\end{array} \right)
\]

\[
\text{flowevts} = \cup \{ x : \text{SIDS} \ \text{states}^12(\text{stm}) ; y : \text{states}^13(\text{stm}) \times \{ \text{enter} : x, y, \text{entered} : x, y, \text{exit} : x, y, \text{exited} : x, y \} \}
\]

\[
\text{consts} = \{ v : \text{allConstants}^7(\text{stm}) \cup v \}
\]
### Rule 95. Constants Initialisation for State Machines

\[
\text{constInitSTM}(cs : \text{Seq}_1(\text{Variable}), \text{stm} : \text{StateMachineDef}, P : \text{CSPProcess}) : \text{CSPProcess} = \\
\text{buildScope}^{1}(\text{undefc}, \text{stm}, \begin{align*}
&\text{let } c : \text{defs} \cdot \text{name}(c) = \left[\text{c.initial}\right] \text{Expr}^\text{M} \\
&\text{within } P
\end{align*})
\]

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

### Rule 96. Build Scope

\[
\text{buildScope}(cs : \text{Seq}_1(\text{Variable}), \text{ctx} : \text{ConnectionNode}, P : \text{CSPProcess}) : \text{CSPProcess} = \\
\begin{cases}
\text{if } (\text{headcs}).\text{initial} == \text{NULL} \text{then} \\
0 \leftrightarrow \text{set}_{\text{vid}}(\text{headcs}, \text{ctx})?\text{name}(\text{headcs}) & \begin{cases}
\text{if} \#cs == 1 \text{then} P \\
\text{else} \text{constInitSTM}^{2}(\text{tailcs}, \text{ctx}, P)
\end{cases}
\end{cases}
\]

### Clocks

Functions related to clocks are formalised in this section.

### Rule 97. allClockVariables function

\[
\text{allClockVariables}(\text{wcs} : \text{Transition} \mapsto (\text{Expression}, \text{WC})) : P \cdot \text{Variable}
\]

\[
\text{allClockVariables}(\text{wcs}) = \\
\{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom} \text{wcs} \land (e \mapsto v) \in \pi_2(\text{wcs}(t)) \cdot v \}
\]

### Rule 98. clockResets function

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

\[
\text{clockResets}(\text{stm}) = \bigcup \left\{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid \\
(t \in \text{dom} \text{wcs} \land (e \mapsto v) \in \pi_2(\text{wcs}(t)) \cdot v) \cdot \alpha_{\text{ClockReset}}^{1}(e) \right\}
\]
Rule 99. \textit{stmClocks} function

\begin{align*}
\text{stmClocks}(\text{wcs} : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} = \\
|| (t, e, v) : \{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom wcs} \land (e \rightarrow v) \in \pi_2(\text{wcs}(t)) \} \\
\text{where}
\end{align*}

\begin{align*}
\alpha_{\text{WC}}(t, e, v) &= \{ \text{triggerEvent}^2(t), \text{setWC}_{\text{vid}}(v) \} \cup \text{alphaClockReset}^2(e) \\
\text{compileWC}^1(t, e, v) &
\end{align*}

Rule 100. \textit{alphaClockReset} function

\begin{align*}
\text{alphaClockReset}(e : \text{Expression}) : \text{ChannelSet} &= \\
\text{This rule is defined by multiple rules according to the subtype of the expression:} \\
\end{align*}

Rule 101. \textit{alphaClockReset} function

\begin{align*}
\text{alphaClockReset}(e : \text{ParExp}) : \text{ChannelSet} &= \\
\text{alphaClockReset}(e) &= \text{alphaClockReset}^3(e.\text{exp})
\end{align*}

Rule 102. \textit{alphaClockReset} function

\begin{align*}
\text{alphaClockReset}(e : \text{Not}) : \text{ChannelSet} &= \\
\text{alphaClockReset}(e) &= \text{alphaClockReset}^4(e.\text{exp})
\end{align*}

Rule 103. \textit{alphaClockReset} function

\begin{align*}
\text{alphaClockReset}(e : \text{CallExp}) : \text{ChannelSet} &= \\
\text{alphaClockReset}(e) &= \text{alphaClockResetCallArgs}^1(e.\text{args})
\end{align*}

Rule 104. \textit{alphaClockResetCallArgs} function

\begin{align*}
\text{alphaClockResetCallArgs}(s : \text{seq Expression}) : \text{ChannelSet} &= \\
\text{if} \#(s) > 0 \text{then} \\
\text{alphaClockReset}^5(\text{head}(s)) \cup \text{alphaClockResetCallArgs}^2(\text{tail}(s))
\text{else} \\
\emptyset
\text{endif}
\end{align*}

Rule 105. \textit{alphaClockReset} function

\begin{align*}
\text{alphaClockReset}(e : \text{And}) : \text{ChannelSet} &= \\
\text{alphaClockReset}(e) &= \text{alphaClockReset}^6(e.\text{left}) \cup \text{alphaClockReset}^7(e.\text{right})
\end{align*}
### Rule 106. alphaClockReset function

alphaClockReset(e : Or) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^8(e.\text{left}) \cup \text{alphaClockReset}^9(e.\text{right})
\]

### Rule 107. alphaClockReset function

alphaClockReset(e : Implies) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{10}(e.\text{left}) \cup \text{alphaClockReset}^{11}(e.\text{right})
\]

### Rule 108. alphaClockReset function

alphaClockReset(e : If) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{12}(e.\text{left}) \cup \text{alphaClockReset}^{13}(e.\text{right})
\]

### Rule 109. alphaClockReset function

alphaClockReset(e : GreaterThan) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{14}(e.\text{left}) \cup \text{alphaClockReset}^{15}(e.\text{right})
\]

### Rule 110. alphaClockReset function

alphaClockReset(e : GreaterOrEqual) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{16}(e.\text{left}) \cup \text{alphaClockReset}^{17}(e.\text{right})
\]

### Rule 111. alphaClockReset function

alphaClockReset(e : LessThan) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{18}(e.\text{left}) \cup \text{alphaClockReset}^{19}(e.\text{right})
\]

### Rule 112. alphaClockReset function

alphaClockReset(e : LessOrEqual) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{20}(e.\text{left}) \cup \text{alphaClockReset}^{21}(e.\text{right})
\]

### Rule 113. alphaClockReset function

alphaClockReset(e : Equals) : ChannelSet =

\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{22}(e.\text{left}) \cup \text{alphaClockReset}^{23}(e.\text{right})
\]
4.2 Detailed Semantics: Timed Language

**Rule 114.** \textit{alphaClockReset} function
\[
\text{alphaClockReset}(e : \text{ClockExp}) : \text{ChannelSet} = \\
\text{alphaClockReset}(e) = \{ \text{clockReset.id}(e.\text{clock}) \}
\]

**Rule 115.** \textit{alphaClockReset} function
\[
\text{alphaClockReset}(e : \text{StateClockExp}) : \text{ChannelSet} = \\
\text{alphaClockReset}(e) = \{ x : \text{SIDS}_{\text{entered}}.\text{id}(x).\text{id}(e.\text{state}) \}
\]

**Rule 116.** Timed semantics of triggers
\[
[t : \text{Trigger}]^{\text{id}}_{\text{Trigger}} : \text{CSPProcess} = \\
\{ [t]^{\text{id}}_{\text{Trigger}} : (\text{||| } c : \text{t.reset} \bullet \text{ClockReset.id}(c.\text{clock}) \rightarrow \text{Skip} )
\]

Waiting Conditions

**Waiting Condition elicitation**

The following rules defined \textit{WC} used for eliciting waiting conditions.
Rule 117. wc function

\[ \text{wc}(e : \text{Expression}) : (\text{Expression}, \text{Expression} \rightarrow \text{Variable}) = \]

\[
\begin{align*}
\text{wc}(\varnothing) &= (\varnothing, \varnothing) \\
\text{wc}(\neg e) &= (\neg \text{wc}(e), \varnothing) \\
\text{wc}(f(\text{args})) &= (f(\varnothing), \varnothing) \\
\text{wc}(e_1 \land e_2) &= (\text{wc}(e_1) \land \text{wc}(e_2)) \\
\text{wc}(e_1 \lor e_2) &= (\text{wc}(e_1) \lor \text{wc}(e_2)) \\
\text{wc}(e_1 \Rightarrow e_2) &= (\text{wc}(e_1) \Rightarrow \text{wc}(e_2)) \\
\text{wc}(e_1 \iff e_2) &= (\text{wc}(e_1) \iff \text{wc}(e_2)) \\
\text{wc}(\text{since}(C) > e) &= (\{\text{since}(C) > e\} \rightarrow b) \\
\text{wc}(\text{sinceEntry}(S) > e) &= (\{\text{sinceEntry}(S) > e\} \rightarrow b) \\
\text{wc}(e > \text{since}(C)) &= (\{e > \text{since}(C)\} \rightarrow b) \\
\text{wc}(e > \text{sinceEntry}(S)) &= (\{e > \text{sinceEntry}(S)\} \rightarrow b) \\
\text{wc}(e_1 > e_2) &= (e_1 > e_2, \varnothing) \\
\text{wc}(\text{since}(C) >= e) &= (\{\text{since}(C) >= e\} \rightarrow b) \\
\text{wc}(\text{sinceEntry}(S) >= e) &= (\{\text{sinceEntry}(S) >= e\} \rightarrow b) \\
\text{wc}(e >= \text{since}(C)) &= (\{e >= \text{since}(C)\} \rightarrow b) \\
\text{wc}(e >= \text{sinceEntry}(S)) &= (\{e >= \text{sinceEntry}(S)\} \rightarrow b) \\
\text{wc}(e_1 >= e_2) &= (e_1 >= e_2, \varnothing) \\
\text{wc}(\text{since}(C) < e) &= (\{\text{since}(C) < e\} \rightarrow b) \\
\text{wc}(\text{sinceEntry}(S) < e) &= (\{\text{sinceEntry}(S) < e\} \rightarrow b) \\
\text{wc}(e < \text{since}(C)) &= (\{e < \text{since}(C)\} \rightarrow b) \\
\text{wc}(e < \text{sinceEntry}(S)) &= (\{e < \text{sinceEntry}(S)\} \rightarrow b) \\
\text{wc}(e_1 < e_2) &= (e_1 < e_2, \varnothing) \\
\text{wc}(\text{since}(C) <= e) &= (\{\text{since}(C) <= e\} \rightarrow b) \\
\text{wc}(\text{sinceEntry}(S) <= e) &= (\{\text{sinceEntry}(S) <= e\} \rightarrow b) \\
\text{wc}(e <= \text{since}(C)) &= (\{e <= \text{since}(C)\} \rightarrow b) \\
\text{wc}(e <= \text{sinceEntry}(S)) &= (\{e <= \text{sinceEntry}(S)\} \rightarrow b) \\
\text{wc}(e_1 <= e_2) &= (e_1 <= e_2, \varnothing) \\
\text{wc}(\text{since}(C) == e) &= (\{\text{since}(C) == e\} \rightarrow b) \\
\text{wc}(\text{sinceEntry}(S) == e) &= (\{\text{sinceEntry}(S) == e\} \rightarrow b) \\
\text{wc}(e == \text{since}(C)) &= (\{e == \text{since}(C)\} \rightarrow b) \\
\text{wc}(e == \text{sinceEntry}(S)) &= (\{e == \text{sinceEntry}(S)\} \rightarrow b) \\
\text{wc}(e_1 == e_2) &= (e_1 == e_2, \varnothing)
\end{align*}
\]

where

\( b \) is a fresh identifier.
Rule 118. wcArgSeq function

\[
\text{wcArgSeq}(s : \text{seq (Expression)}) : (\text{seq Expression, Expression} \rightarrow \text{Variable}) = \\
\begin{align*}
\text{if} & \#(s) > 0 \text{ then} \\
& (\langle \pi_2(\text{wcArgSeq}(\text{head}(s))) \rangle \land \pi_2(\text{wcArgSeq}(\text{tail}(s))) \\
\text{where} & \\
& \text{wcArgSeq} = \pi_1(\text{wcArgSeq}) \\
\text{else} & \\
& (\langle \rangle, \emptyset) \\
\end{align*}
\]

Waiting Condition as CSP processes

The following rules define the function \texttt{compileWC} which is used to define the CSP semantics of waiting conditions.
**Rule 119. compileWC function**

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \\
\text{compileWC}(t, \text{since}(C) >= e, v) = \\
\text{let} \\
\hspace{1em} \text{Reset} = \text{clockReset} . \text{id}(C) \rightarrow \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{Monitor} \\
\hspace{1em} \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\|\text{triggerEvent}^{2}(t)\|) \\
\hspace{1em} \triangle_{[e]} \hspace{1em} \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{RUN}(\|\text{triggerEvent}^{2}(t)\|)
\end{array} \right) \triangle \text{Reset} \\
\text{within} \\
\hspace{2em} \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{Monitor}
\]

\text{compileWC}(t, e >= \text{since}(C), v) = \\
\text{let} \\
\hspace{1em} \text{Reset} = \text{clockReset} . \text{id}(C) \rightarrow \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{Monitor} \\
\hspace{1em} \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\|\text{triggerEvent}^{3}(t)\|) \\
\hspace{1em} \triangle_{[e]} \hspace{1em} \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{RUN}(\|\text{triggerEvent}^{3}(t)\|)
\end{array} \right) \triangle \text{Reset} \\
\text{within} \\
\hspace{2em} \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{Monitor}
\]

\text{compileWC}(t, \text{sinceEntry}(S) >= e, v) = \\
\text{let} \\
\hspace{1em} \text{Reset} = \text{entered} ? \text{id}(S) \rightarrow \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{Monitor} \\
\hspace{1em} \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\|\text{triggerEvent}^{4}(t)\|) \\
\hspace{1em} \triangle_{[e]} \hspace{1em} \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{RUN}(\|\text{triggerEvent}^{4}(t)\|)
\end{array} \right) \triangle \text{Reset} \\
\text{within} \\
\hspace{2em} \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{Monitor}
\]

\text{compileWC}(t, e >= \text{sinceEntry}(S), v) = \\
\text{let} \\
\hspace{1em} \text{Reset} = \text{entered} ? \text{id}(S) \rightarrow \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{Monitor} \\
\hspace{1em} \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\|\text{triggerEvent}^{5}(t)\|) \\
\hspace{1em} \triangle_{[e]} \hspace{1em} \text{setWC}_{\text{vid}(v)}!\text{false} \rightarrow \text{RUN}(\|\text{triggerEvent}^{5}(t)\|)
\end{array} \right) \triangle \text{Reset} \\
\text{within} \\
\hspace{2em} \text{setWC}_{\text{vid}(v)}!\text{true} \rightarrow \text{Monitor}
4.2 Detailed Semantics: Timed Language

**Rule 120. 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.id}(C) \rightarrow \text{setWC.vid}(v)!false \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\left( \begin{array}{c} \\
\text{RUN}(\{ [\text{triggerEvent}^1(t)] \}) \\
\vartriangle_{\text{Expr}}^{\text{e}} + 1 \text{setWC.vid}(v)!true \rightarrow \text{RUN}(\{ [\text{triggerEvent}^2(t)] \}) \\
\end{array} \right) \quad \vartriangle \text{Reset} \\
\text{within} \\
\text{setWC.vid}(v)!false \rightarrow \text{Monitor} \\
\text{compileWC}(t, e > \text{since}(C), v) = \\
\text{let} \\
\text{Reset} = \text{clockReset.id}(C) \rightarrow \text{setWC.vid}(v)!true \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\left( \begin{array}{c} \\
\text{RUN}(\{ [\text{triggerEvent}^3(t)] \}) \\
\vartriangle_{\text{Expr}}^{\text{e}} + 1 \text{setWC.vid}(v)!false \rightarrow \text{RUN}(\{ [\text{triggerEvent}^4(t)] \}) \\
\end{array} \right) \quad \vartriangle \text{Reset} \\
\text{within} \\
\text{setWC.vid}(v)!true \rightarrow \text{Monitor} \\
\text{compileWC}(t, e > \text{sinceEntry}(S), v) = \\
\text{let} \\
\text{Reset} = \text{entered?x.id}(S) \rightarrow \text{setWC.vid}(v)!false \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\left( \begin{array}{c} \\
\text{RUN}(\{ [\text{triggerEvent}^5(t)] \}) \\
\vartriangle_{\text{Expr}}^{\text{e}} + 1 \text{setWC.vid}(v)!true \rightarrow \text{RUN}(\{ [\text{triggerEvent}^6(t)] \}) \\
\end{array} \right) \quad \vartriangle \text{Reset} \\
\text{within} \\
\text{setWC.vid}(v)!false \rightarrow \text{Monitor} \\
\text{compileWC}(t, e > \text{sinceEntry}(S), v) = \\
\text{let} \\
\text{Reset} = \text{entered?x.id}(S) \rightarrow \text{setWC.vid}(v)!true \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\left( \begin{array}{c} \\
\text{RUN}(\{ [\text{triggerEvent}^7(t)] \}) \\
\vartriangle_{\text{Expr}}^{\text{e}} + 1 \text{setWC.vid}(v)!false \rightarrow \text{RUN}(\{ [\text{triggerEvent}^8(t)] \}) \\
\end{array} \right) \quad \vartriangle \text{Reset} \\
\text{within} \\
\text{setWC.vid}(v)!true \rightarrow \text{Monitor}
**Rule 121. compileWC function**

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

\[
\text{compileWC}(t, \text{since}(C) \leq e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset} . id(C) \to \text{setWC}_v(d(v))!\text{true} \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{align*}
\left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{21}(t)]) \\
\triangle_{[\text{Exp}_C]} \text{setWC}_v(d(v))!\text{false} \to \text{RUN}([\text{triggerEvent}^{22}(t)]) \\
\end{array} \right)
\end{align*}
\]
\[
\triangle \text{Reset}
\]
\[
\text{within} \\
\text{setWC}_v(d(v))!\text{true} \to \text{Monitor}
\]
\[
\text{compileWC}(t, e <= \text{since}(C), v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset} . id(C) \to \text{setWC}_v(d(v))!\text{false} \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{align*}
\left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{21}(t)]) \\
\triangle_{[\text{Exp}_C]} \text{setWC}_v(d(v))!\text{true} \to \text{RUN}([\text{triggerEvent}^{22}(t)]) \\
\end{array} \right)
\end{align*}
\]
\[
\triangle \text{Reset}
\]
\[
\text{within} \\
\text{setWC}_v(d(v))!\text{false} \to \text{Monitor}
\]
\[
\text{compileWC}(t, e <= \text{sinceEntry}(S), v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered}?x . id(S) \to \text{setWC}_v(d(v))!\text{true} \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{align*}
\left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{23}(t)]) \\
\triangle_{[\text{Exp}_C]} \text{setWC}_v(d(v))!\text{false} \to \text{RUN}([\text{triggerEvent}^{24}(t)]) \\
\end{array} \right)
\end{align*}
\]
\[
\triangle \text{Reset}
\]
\[
\text{within} \\
\text{setWC}_v(d(v))!\text{false} \to \text{Monitor}
\]
\[
\text{compileWC}(t, e <= \text{sinceEntry}(S), v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered}?x . id(S) \to \text{setWC}_v(d(v))!\text{false} \to \text{Monitor} \\
\quad \text{Monitor} = \\
\begin{align*}
\left( \begin{array}{c}
\text{RUN}([\text{triggerEvent}^{23}(t)]) \\
\triangle_{[\text{Exp}_C]} \text{setWC}_v(d(v))!\text{true} \to \text{RUN}([\text{triggerEvent}^{24}(t)]) \\
\end{array} \right)
\end{align*}
\]
\[
\triangle \text{Reset}
\]
\[
\text{within} \\
\text{setWC}_v(d(v))!\text{false} \to \text{Monitor}
\]
Rule 122. 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{vid}(v)!\text{true} \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\quad \begin{cases} \\
\quad \text{RUN}([^\text{triggerEvent}^27(t)]) \\
\quad \text{\triangle}([e]_{\text{Expr}^27} + 1) \text{setWC}._\text{vid}(v)!\text{false} \rightarrow \text{RUN}([^\text{triggerEvent}^28(t)]) \\
\end{cases} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}._\text{vid}(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{vid}(v)!\text{false} \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\quad \begin{cases} \\
\quad \text{RUN}([^\text{triggerEvent}^29(t)]) \\
\quad \text{\triangle}([e]_{\text{Expr}^29} + 1) \text{setWC}._\text{vid}(v)!\text{true} \rightarrow \text{RUN}([^\text{triggerEvent}^30(t)]) \\
\end{cases} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}._\text{vid}(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{vid}(v)!\text{true} \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\quad \begin{cases} \\
\quad \text{RUN}([^\text{triggerEvent}^31(t)]) \\
\quad \text{\triangle}([e]_{\text{Expr}^31} + 1) \text{setWC}._\text{vid}(v)!\text{false} \rightarrow \text{RUN}([^\text{triggerEvent}^32(t)]) \\
\end{cases} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}._\text{vid}(v)!\text{false} \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{vid}(v)!\text{false} \rightarrow \text{Monitor} \\
\text{Monitor} = \\
\quad \begin{cases} \\
\quad \text{RUN}([\text{triggerEvent}^33(t)]) \\
\quad \text{\triangle}([e]_{\text{Expr}^33} + 1) \text{setWC}._\text{vid}(v)!\text{true} \rightarrow \text{RUN}([\text{triggerEvent}^34(t)]) \\
\end{cases} \triangle \text{Reset} \\
\text{within} \\
\text{setWC}._\text{vid}(v)!\text{false} \rightarrow \text{Monitor} \]
Chapter 4. Semantics

**Rule 123. compileWC function**

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \\
\text{compileWC}(t, \text{since}(C) == e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{clockReset.id}[(C)] \rightarrow \text{setWC} = \{ v \} | \text{false} \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \\
\quad \begin{cases} \\
\quad \quad \text{RUN}([\text{triggerEvent}^3(t)]) \\
\quad \quad \Delta_{\text{Exp}^7}^{[e]} \text{setWC} = \{ v \} | \text{true} \rightarrow \text{RUN}([\text{triggerEvent}^6(t)]) \\
\quad \quad \Delta_{\text{Exp}^7}^{[e]} + \text{1} \text{setWC} = \{ v \} | \text{false} \rightarrow \text{RUN}([\text{triggerEvent}^9(t)]) \\
\quad \end{cases} \\
\quad \text{within} \\
\quad \text{setWC} = \{ v \} | \text{false} \rightarrow \text{Monitor} \\
\]

**Rule 124. compileWC function**

\[
\text{compileWC}(t : \text{Transition}, e : \text{Expression}, v : \text{Variable}) : \text{TimedCSPProcess} = \\
\text{compileWC}(t, \text{sinceEntry}(C) == e, v) = \\
\text{let} \\
\quad \text{Reset} = \text{entered?x.id}[(S)] \rightarrow \text{setWC} = \{ v \} | \text{false} \rightarrow \text{Monitor} \\
\quad \text{Monitor} = \\
\quad \begin{cases} \\
\quad \quad \text{RUN}([\text{triggerEvent}^3(t)]) \\
\quad \quad \Delta_{\text{Exp}^7}^{[e]} \text{setWC} = \{ v \} | \text{true} \rightarrow \text{RUN}([\text{triggerEvent}^6(t)]) \\
\quad \quad \Delta_{\text{Exp}^7}^{[e]} + \text{1} \text{setWC} = \{ v \} | \text{false} \rightarrow \text{RUN}([\text{triggerEvent}^9(t)]) \\
\quad \end{cases} \\
\quad \text{within} \\
\quad \text{setWC} = \{ v \} | \text{false} \rightarrow \text{Monitor} \\
\]
4.2 Detailed Semantics: Timed Language

**Rule 125. triggerEvent function**

triggerEvent(t : Transition) : CSPEvent =

\[
\begin{align*}
\text{if } & t.\text{trigger} \neq \text{null} \\
\text{then} & \text{triggerEvent}(t.\text{trigger.id}(t)) \\
\text{else} & \text{internal}.\text{id}(t)
\end{align*}
\]

**Trigger deadline events**

**Rule 126. deadlineEvents function**

deadlineEvents(s : StateMachineDef) : ChannelSet =

\[
\text{deadlineEvents}(s) = \{ t : \text{allTransitions}(s) \mid t.\text{end} \neq \text{null} \land t.\text{deadline.id}(t) \}
\]

**Memory**

**Rule 127. State-machine Memory**

stmMemory(stm : StateMachineDef, wcs : Transition \to (Expression, WC)) : TimedCSPProcess =

\[
\begin{align*}
\text{let } & \text{Memory}(vars) \triangleq \\
& \begin{cases}
\vdots & \\
\text{let } v : \text{lvars} \Rightarrow (\text{get}_v(v) ! \text{name}(v) \to \text{Memory}(vars)) \quad \vdots & \\
\text{let } v : \text{rvars} \Rightarrow (\text{set}_v(v) ? x \to \text{Memory}(vars[v.name(v) := x])) \quad \vdots & \\
\text{let } v : \text{allConstants}(\text{allConstants}(\text{allConstants}(\text{allConstants}))) \Rightarrow (\text{get}_v(v) ! \text{name}(v) \to \text{Memory}(vars)) \quad \vdots & \\
\text{let } r : \text{allTransitions}(\text{allTransitions}(\text{allTransitions}(\text{allTransitions}))) \Rightarrow (\text{memoryTransition}(t, wcs) \Rightarrow \text{Memory}(vars)) \quad \vdots & \\
\text{let } v : \text{cvars} \Rightarrow (\text{setWC}_v(v) ? x \to \text{Memory}(vars[v.name(v) := x])) \quad \vdots & \\
\text{let } r : \text{allDeadlineTransitions}(\text{allDeadlineTransitions}(\text{allDeadlineTransitions}(\text{allDeadlineTransitions}))) \Rightarrow (\text{memoryDeadline}(t, wcs) \Rightarrow \text{Memory}(vars)) \quad \vdots & \\
\end{cases}
\end{align*}
\]

\[
\text{within } \text{Memory}(\text{varvalues})
\]

where

\[
\begin{align*}
rvars &= \text{requiredVariables}(\text{requiredVariables}(\text{requiredVariables}(\text{requiredVariables}))) \\
lvars &= \text{allLocalVariables}(\text{allLocalVariables}(\text{allLocalVariables}(\text{allLocalVariables}))) \\
consts &= (v : \text{allConstants}(\text{allConstants}(\text{allConstants}(\text{allConstants}))) \Rightarrow v) \\
cvars &= \text{allClockVariables}(\text{allClockVariables}(\text{allClockVariables}(\text{allClockVariables}))) \\
vars &= (v : rvars \cup lvars \cup cvars \cup \text{name}(v)) \cap (v : \text{consts} \cup \text{name}(v)) \\
\text{varvalues} &= (v : rvars \cup lvars \cup cvars \cup \text{initial}(v)) \cap (v : \text{consts} \cup \text{name}(v))
\end{align*}
\]
### Rule 128. allDeadlineTransitions function

\[
\text{allDeadlineTransitions}(s : \text{StateMachineDef}) : \text{IPTransition} =
\]

\[
\text{allDeadlineTransitions}(s) = \{ t : \text{Transitions}(s) \mid t.\text{end} \neq \text{null} \}
\]

### Rule 129. memoryTransition function

\[
\text{memoryTransition}(t : \text{Transition}, wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess}
\]

\[
\begin{cases}
\text{if}(t.\text{condition} \neq \text{null}) & \\
& \left( [\pi_1(wcs(t))]_{\text{Expr}} \right) \& \text{triggerForMemory}^2(t.\text{trigger}, \text{id}(t)) \\
\text{else} & \text{triggerForMemory}^2(t.\text{trigger}, \text{id}(t))
\end{cases}
\]

### Rule 130. Memory deadline

\[
\text{memoryDeadline}(t : \text{Transition}, wcs : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess}
\]

\[
\begin{cases}
\text{if}(t.\text{condition} \neq \text{null}) & \\
& \left( [\pi_1(wcs(t))]_{\text{Expr}} \right) \& \text{deadline.id}(t).\text{on} \rightarrow \text{Skip} \\
& \left( \neg [\pi_1(wcs(t))]_{\text{Expr}} \right) \& \text{deadline.id}(t).\text{off} \rightarrow \text{Skip} \\
\text{else} & \text{deadline.id}(t).\text{on} \rightarrow \text{Skip}
\end{cases}
\]

---

### 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 131. Semantics of states

\[
[[s : \text{State}],_S] : \text{TimedCSPProcess} =
\]

This function is split in multiple rules according to the type of states.
4.2 Detailed Semantics: Timed Language

**Rule 132. Semantics of simple states**

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

let

\[
\begin{align*}
\text{Inactive} & \triangleq \text{enter} \times : \text{sids}, \text{id}(s) \rightarrow \text{Activating}(s) \\
\text{Activating}(o) & \triangleq [s, \text{entry}]_{\text{ops}}^\text{ops} : \text{initialisation}^4(s); \text{entered}.o . \text{id}(s) \rightarrow \\
\text{exit} & \triangleq [s, \text{during}]_{\text{ops}}^\text{ops} : \text{Stop} \mid \text{triggerDeadlines}^1(s) \triangledown \\
\end{align*}
\]

\[
\begin{align*}
\text{transitionsFrom}^1(s) & \cdot [t, s, \text{false}]^\text{inactive, activating, ops}_T \\
\text{exit} & \cdot [s, \text{id}(s)]_{\text{ops}}^\text{ops} ; \text{exited}.x . \text{id}(s) \rightarrow \text{Skip}
\end{align*}
\]

within

\[
\text{Inactive}
\]

where

\[
\begin{align*}
\# \text{states}^{14}(s) & = 0 \\
\text{sids} & = \text{SIDS} \setminus \{\text{id}(s)\} \land \text{tids} = \text{TIDS} \setminus \text{tIDS}(s) \\
\text{exit} & = \text{exit} \times : \text{sids}, \text{id}(s) \rightarrow \text{exitSubstates}^3(s); [s, \text{exit}]_{\text{ops}}^\text{ops} ; \text{exited}.x . \text{id}(s) \rightarrow \text{Skip}
\end{align*}
\]

**Rule 133. Semantics of composite states**

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

let

\[
\begin{align*}
\text{Inactive} & \triangleq \text{enter} \times : \text{sids}, \text{id}(s) \rightarrow \text{Activating}(s) \\
\text{Activating}(o) & \triangleq [s, \text{entry}]_{\text{ops}}^\text{ops} : \text{initialisation}^4(s); \text{entered}.o . \text{id}(s) \rightarrow \\
\text{exit} & \triangleq [s, \text{during}]_{\text{ops}}^\text{ops} : \text{Stop} \mid \text{triggerDeadlines}^2(s) \triangledown \\
\end{align*}
\]

\[
\begin{align*}
\text{transitionsFrom}^2(s) & \cdot [t, s, \text{false}]^\text{inactive, activating, ops}_T \\
\text{exit} & \cdot [s, \text{id}(s)]_{\text{ops}}^\text{ops} ; \text{exited}.x . \text{id}(s) \rightarrow \text{Skip}
\end{align*}
\]

within

\[
(\text{Inactive} \mid \text{flowtrigevts} \triangledown \text{composeStates}^1(x : \text{states}^{15}(s), s)^{\text{ops}})
\]

where

\[
\begin{align*}
\# \text{states}^{16}(s) & > 0 \\
\text{flowtrigevts} & = \text{flowTriggerEvents}^2(s) \\
\text{sids} & = \text{SIDS} \setminus \{\text{id}(s)\} \land \text{tids} = \text{TIDS} \setminus \text{tIDS}(s) \\
\text{exit} & = \text{exit} \times : \text{sids}, \text{id}(s) \rightarrow \text{exitSubstates}^3(s); [s, \text{exit}]_{\text{ops}}^\text{ops} ; \text{exited}.x . \text{id}(s) \rightarrow \text{Skip}
\end{align*}
\]
Rule 134. Semantics of trigger deadlines

\[
\text{triggerDeadlines}(s : \text{State}) : \text{TimedCSPProcess} =
\]

\[
\begin{align*}
&\text{let} \\
&D\text{eadline}(t) \equiv \\
&(\text{readState}^\text{3} \left( \begin{array}{c}
\text{usedVariables}^0(t.\text{end}), \\
\text{deadline}.\text{id}(t) \cdot \text{on} \rightarrow \\
\text{Deadline}(t)
\end{array} \right) ; \\
&\text{within} \\
&\left\| t : \text{tDS} \cdot \text{Deadline}(t) \right\|
\end{align*}
\]

where
\[
\text{tDS} = \{ t : \text{transitionsFrom}^0(s) \mid t.\text{end} \neq \text{null} \}
\]

The composition of states is also largely unchanged when compared to the untimed Rule 30 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 \text{sinceEntry}(s).

Rule 135. Composition of states

\[
\text{composeStates}(ss : \text{seqState}, p : \text{NodeContainer}) : \text{TimedCSPProcess} =
\]

\[
\begin{align*}
&\text{if} \#ss = 1 \\
&\text{then} \\
&\text{restrictedState}^3(p, \text{headss})^{\text{nops}} \\
&\text{else} \\
&\text{restrictedState}^4(p, \text{headss})^{\text{nops}} \\
&\text{composeStates}^2(\text{tailss}, p)^{\text{nops}}
\end{align*}
\]

where
\[
\text{shflowevts} = \text{flowEvents}^3(\text{headss}, p) \cap \bigcup \{ x : \text{tailss} \cdot \text{flowEvents}^4(x, p) \}
\]

4.2.3 Timed statements

Rule 136. Semantics of statements

\[
\llbracket s : \text{Statement} \rrbracket^{\text{nops}}_{\text{Statement}} : \text{TimedCSPProcess} =
\]

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

Rule 137. Semantics of statement deadlines

\[
\llbracket s : \text{TimedStatement} \rrbracket^{\text{nops}}_{\text{Statement}} : \text{TimedCSPProcess} =
\]

\[
\llbracket s.\text{stmt} \rrbracket^{\text{nops}}_{\text{Statement}} \triangleright \llbracket s.\text{end} \rrbracket^\text{Expr^{s}}
\]
Rule 138. Semantics of wait
\[ [s : \text{Wait}]^{nops}_{\text{Statement}} : \text{TimedCSPProcess} = \]
\[ [s.\text{duration}]_{\text{Wait}} \]
where
\[ [e : \text{RangeExp}]_{\text{Wait}} = \bigcap n : [e]_{\text{Expr}} \bullet \text{Wait}(n) \]
\[ [e : \text{Expression}]_{\text{Wait}} = \text{Wait}([e]_{\text{Expr}}) \]

Rule 139. Semantics of clock reset
\[ [s : \text{ClockReset}]^{nops}_{\text{Statement}} : \text{TimedCSPProcess} = \]
\[ \text{clockReset.id}(s.\text{clock}) \rightarrow \text{Skip} \]

Rule 140. Semantics of assignment
\[ [s : \text{Assignment}]^{nops}_{\text{Statement}} : \text{CSPProcess} = \]
\[ (\text{set}_{\text{vid}}(s.\text{left})! [s.\text{right}]_{\text{Expr}} \rightarrow \text{Skip}) \triangleright 0 \]

Rule 141. Semantics of call statement
\[ [s : \text{Call}]^{nops}_{\text{Statement}} : \text{CSPProcess} = \]
\[ (\text{op.nameCall} \rightarrow \text{Skip} \triangleright 0); \text{body}; (\text{op.nameRet} \rightarrow \text{Skip}) \triangleright 0 \]
where
\[ \text{op} = s.\text{operation} \]
\[ \text{opdef} = \text{findOperationDefinition}(\text{op}, \text{ops}) \]
\[ \text{SkipAnytime} = (\text{Wait}(1); \text{SkipAnytime}) \bigcap \text{Skip} \]
\[ \text{body} = \begin{cases} 
\text{if} (\text{opdef} = \text{null}) \text{then} \\
(\text{SkipAnytime} \bigcap \text{Stop}) \\
\text{else} \\
[\text{opdef}]^{nops}_{\text{STM}} ([x : \text{s.args} \bullet [x]_{\text{Expr}}]) 
\end{cases} \]
5 Assertion DSL Syntax .................. 77
5.1 Standard Assertions
5.2 Probabilistic Assertions

6 Assertions DSL Usage ................. 87
6.1 Standard Assertions
6.2 Probabilistic Assertions

7 Checking core assertions .............. 93
5. 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.

5.1 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.
Chapter 5. Assertion DSL Syntax

### Syntax — Specification.

```plaintext
SPEC ::= N 'is' ('not')? PRED
    | N ('does' 'not' | '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.

```plaintext
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.

```plaintext
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.

```plaintext
CSP specification ::= ('timed' | untimed')? 'csp' N
                 | 'csp-begin' CSPM 'csp-end'
```
5.2 Probabilistic Assertions

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`.

### 5.2 Probabilistic Assertions

#### Syntax — Probabilistic Statements.

```plaintext
ProbStatements ::= ProbStatement*
ProbStatement ::= ConstDecl
| Constants
| Label
| Formula
| Rewards
| ProbAssertion
```

The probabilistic assertion syntax consists of various probabilistic statements:

- constant declarations (`ConstDecl` in Syntax 5.2.1),
- constant configurations (`Constants` in Syntax 5.2.1),
- labels (`Label` in Syntax 5.2.2),
- formulas (`Formula` in Syntax 5.2.2),
- rewards (`Rewards` in Syntax 5.2.3),
- probabilistic assertions (`ProbAssertion` in Syntax 5.2).

#### Syntax — Probabilistic Assertions.

```plaintext
ProbAssertion ::= 'prob' 'assertion' N ':' ProbFormula
               ('with' ('constant'| 'constants') (ConstConfigs | N))?
               ('with' 'cmdoptions' STRING)?
```

A probabilistic assertion starts with `prob`. And a name (N denotes the name category, actually it is `ID` in Syntax 5.2.8) is associated with the assertion. The body of the assertion is a probabilistic formula (`ProbFormula` in Syntax 5.2.4) 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 5.2.1) and customised PRISM command line options

#### 5.2.1 Constants

1These 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 5. Assertion DSL Syntax

**Syntax — Constant Declaration.**

\[ \text{ConstDecl ::= 'const' N ':' Type} \]

A constant variable declared with a type (\text{Type} is the RoboChart Type class, see Appendix B, but in probabilistic assertions only primitive types \text{nat}, \text{int}, \text{bool}, and \text{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.**

\[ \text{Constants ::= 'constants' N ':' ConstConfigs} \]

**Syntax — Constants Configurations.**

\[ \text{ConstConfigs ::= ConstConfig (','ConstConfig)*} \]
\[ ((',')? 'and' ConstConfig)? \]

Constants associates a name with a configuration of various constants.

**Syntax — Constant Configuration.**

\[ \text{ConstConfig ::= QualifiedNameToElement} \]
\[ (('set' 'to' | 'assigned' | 'with' 'value') Expr} \]
\[ | 'from' 'set' ConstSetExpr) \]

The configuration of each constant by \text{ConstConfig} can be a single value (\text{Expr} in Syntax 5.2.6) or a set of values from \text{ConstSetExpr}.

**Syntax — ConstSetExpr.**

\[ \text{ConstSetExpr ::= Expr ':' Expr ':' Expr} \]
\[ | '{' (Expr (',' Expr)*? '}'\]

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.

### 5.2.2 Labels and Formulas

**Syntax — Labels.**

\[ \text{Label ::= 'label' N '=' BoolExpr} \]

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.
name (see Syntax 5.2.4).

\[
\text{Syntax — Formulas.} \\
\text{Formula ::= 'label' N '=' Expr}
\]

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

### 5.2.3 Rewards

\[
\text{Syntax — Rewards.} \\
\text{Rewards ::= 'rewards' N '=' Reward+ 'endrewards'}
\]

A collection of rewards can be assigned a name and this name will be used in \( R \) operator in Syntax 5.2.4.

\[
\text{Syntax — Reward.} \\
\text{Reward ::= ('[' QualifiedNameToElement ']')? BoolExpr ':' Expr ' ;'}
\]

A reward composes a guard condition (BoolExpr), an expression, and an optional event (QualifiedToElement, in Syntax 5.2.7).

### 5.2.4 Probabilistic Formulas

\[
\text{Syntax — Probabilistic Formulas.} \\
\text{ProbFormula ::= BoolExpr} \\
| \text{StateFormula '&&' StateFormula} \\
| \text{StateFormula '||' StateFormula} \\
| \text{StateFormula '=>' StateFormula} \\
| \text{StateFormula '<==' StateFormula}
\]

A probabilistic formula could be any boolean expression, or a conjunction, or a disjunction, or an implication, or an equivalence of two state formulas.
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 For all) followed by a path formula,
- a non-probability property by a exists operator (E or Exists) followed by a path formula.

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

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

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

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

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

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

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

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

#### 5.2.5 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 5. Assertion DSL Syntax

Syntax — Simulation Method.

\[
\text{SimMethod ::= 'CI' ('at' CiMethod)? } \\
| 'ACI' ('at' CiMethod)? \\
| 'APMC' ('at' APMCMethod)? \\
| 'SPRT' ('at' SPRTMethod)?
\]

Syntax — Simulation Method.

\[
\text{CiMethod ::= ((',')? 'w' '=' Expr)? } \\
& ((',')? 'alpha' '=' alpha=Expr)? \\
& ((',')? 'n' '=' n=Expr)?
\]

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.

\[
\text{APMCMethod ::= ((',')? 'epsilon' '=' Expr)? } \\
& ((',')? 'delta' '=' Expr)? \\
& ((',')? 'n' '=' n=Expr)?
\]

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

Syntax — Simulation Method.

\[
\text{SPRTMethod ::= ((',')? 'alpha' '=' Expr)? } \\
& ((',')? 'delta' '=' Expr)?
\]

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

5.2.6 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.
5.2.7 Qualified Names

**Syntax — Qualified Name To Element.**

Qualified Name To Element ::= Named Element ('::' Named Element)*

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).

5.2.8 Terminals

**Syntax — Terminal rules.**

ID ::= ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
BOOLEAN ::= 'true' | 'false'
INT ::= ('0'..'9')*
FLOAT ::= INT'.INT
STRING ::= '"' ('\' . | !('"|')'*' '"'

```markdown
Chapter 5. Assertion DSL Syntax

5.2.7 Qualified Names

**Syntax — Qualified Name To Element.**

Qualified Name To Element ::= Named Element ('::' Named Element)*

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).

5.2.8 Terminals

**Syntax — Terminal rules.**

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

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

6.1 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.

6.2 Probabilistic Assertions

6.2.1 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 5.2.
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.

6.2.2 Examples

This section provides several probabilistic assertion examples to show how to specify probabilistic properties using our probabilistic assertion language.
Example 6.1 — 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 6.2 — 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 6.3 — 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 5.2.4 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 6.4 — Formula. A formula is just a shorthand of an expression and it is referred by putting a prefix $ in front of its name.

\[
\text{formula } f_c = \text{mod0::ctrl_ref0::c + 1}
\]

prob assertion P_stuck_loc1:

\[
\text{Prob=? [Finally mod0::ctrl_ref0::p=x & $f_c=1]} \text{ with constant C1}
\]

Example 6.5 — Rewards. The reward nbmove assigns 1 to each synchronisation over the move event that is from the state machine reference stm_ref0. Then a reward operator Reward uses this defined reward nbmove to check the average number of synchronisation over move when the variable c is equal to 0.

\[
\begin{align*}
\text{rewards nbmove} &= \\
&= \text{[mod0::ctrl_ref0::stm_ref0::move] true : 1; endrewards}
\end{align*}
\]

prob assertion R_stuck_move:

\[
\text{Reward \{nbmove\} =? [ Reachable mod0::ctrl_ref0::c=0 ]}
\]

\[
\text{with constant mod0::ctrl_ref0::batteryCharge set to 20,}
\]

\[
\text{and mod0::ctrl_ref0::chargeSpeed set to 4}
\]

Example 6.6 — Forall. This assertion checks if the robot will always finally get stuck. In particular, [ and ] are required in order to embed another path formula (Finally) within the outer path formula (Globally).

prob assertion A_stuck:

\[
\begin{align*}
\text{A [Globally [Finally "l_stuck"]]} &
\end{align*}
\]

\[
\text{with constant C1}
\]
Example 6.7 — Simulation. This example applies statistic model checking to verify the property using the CI method with supplied parameters.

```plaintext
prob assertion P_stuck_loc:
  Prob=? [Finally {mod0::ctrl_ref0::p=x} && "l_out0fPower"]
using sim with CI at alpha=0.01, n=2000, and pathlen=1000
with constants C1
```
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
about termination of the move operation.

1. Select the operation definition tool \( \text{T} \) 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 \( P_{\text{move}} \) is not deadlock free, and that the process \( P_{\text{move}} \) followed by the process \( \text{RUN}(r_{\ldots}) \) is deadlock free.
8 Collections .............................. 97
8.1 Metamodel
8.2 Well-formedness Conditions
8.3 Semantics

9 Conclusions ............................. 103
8. Collections

This chapter describes the RoboChart extensions designed to support modelling, analysis, and simulation of collections of robots. Section 8.1 describes the extensions of the metamodel of RoboChart, Section 8.2 describes the conditions that characterise well-formed RoboChart collections, and Section 8.3 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.

8.1 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 constructs, 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
Chapter 8. Collections

Figure 8.1: Metamodel for collections: RCCollection and Event

Figure 8.2: 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 8.1 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 is a broadcast event.
8.2 Well-formedness Conditions

8.2.1 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

8.2.2 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.

8.2.3 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 $\_\text{predicate}$ 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.
8.2.4 Trigger

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

### 8.3 Semantics

**Rule 142. Semantics of Collections**

\[
\begin{align*}
[c : RCCollection]_{\text{Init}} : \text{CSPProcess} = \\
\mathcal{M}^i (i) \\
\mathcal{M}^i (\{e_1, e_2 \mid (e_1, e_2) \leftarrow \text{connectedEvents}(c)\}) \\
\left( \begin{array}{c}
\mathcal{M}^i (\text{BBuffer}^1(\text{eventId}(\text{conn}.efrom), i, \text{eventId}(\text{conn}.eto), j)) \\
\mathcal{M}^i (\text{BBuffer}^2(\text{eventId}(\text{conn}.eto), j, \text{eventId}(\text{conn}.efrom), i))
\end{array} \right)
\end{align*}
\]

where

\[
\begin{align*}
\text{connectedEvents}(c : \text{Collection}) & : \mathbb{P}(\text{Event} \times \text{Event}) = \\
\{ (\text{conn} : c.\text{connections} \cdot (\text{eventId}(\text{conn}.efrom), \text{eventId}(\text{conn}.eto))) \} \\
\cup \\
\{ (\text{conn} : c.\text{connections} \cdot \text{conn}.\text{bidirect} \cdot \text{eventId}(\text{conn}.efrom), \text{eventId}(\text{conn}.eto)) \}
\end{align*}
\]

\[
\begin{align*}
\text{inds}(\text{conn} : \text{Connection}, c : \text{Collection}) & : \mathbb{P}(\text{ID} \times \text{ID}) = \\
\begin{cases}
\text{range}(\text{conn}.to, c) \times \text{range}(\text{conn}.from, c) \setminus \{(i, i)\} & \text{if } \text{conn}.\text{from} = \text{conn}.\text{to}.\text{ref}
\text{range}(\text{conn}.to, c) \times \text{range}(\text{conn}.from, c) & \text{else}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{range}(m : \text{Module}, c : \text{Collection}) & : \mathbb{P}(\text{ID} = (i : c.\text{instantiations} \cdot l.\text{module} = m).\text{range}) \\
\text{heterogeneous} & = (c.\text{from} \neq c.\text{to} \cdot \text{ref})
\end{align*}
\]
8.3 Semantics

**Rule 143. Broadcast Buffer**

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

\[
\begin{align*}
& \text{let} \\
& \quad \text{BufferEmpty} = \text{prefixIn} \rightarrow \text{BufferFull}(x) \\
& \quad \text{BufferFull}(v) = \text{prefixIn} \rightarrow \text{BufferFull}(x) \sqcup \text{prefixOut} \rightarrow \text{BufferEmpty}
\end{align*}
\]

\[
\text{within}
\]

\[
\text{BufferEmpty}
\]

where

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

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

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

**Rule 144. Semantics of triggers**

\[
[[t : \text{Trigger}]_{id} : \text{CSPProcess} =
\]

\[
\begin{align*}
& \text{if } t.\text{type} = \text{INPUT then} \\
& \quad \text{eventId}(t.\text{event}).\text{tid}?!id.?x \rightarrow \text{set}_\text{vid}(t.\text{from})?!x \rightarrow \text{set}_\text{vid}(t.\text{parameter})?!x \rightarrow \text{Skip}
\]

\[
& \text{else if } t.\text{type} = \text{SIMPLE then} \\
& \quad \text{eventId}(t.\text{event}).\text{tid}?!id \rightarrow \text{set}_\text{vid}(t.\text{from})?!x \rightarrow \text{Skip}
\]

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

where

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

**Rule 145. Semantics of send event statements**

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

\[
\begin{align*}
& \text{if } t.\text{type} = \text{OUTPUT} \lor t.\text{type} = \text{SYNC then} \\
& \quad \big| i : \{ x : \text{ID} \mid \text{[l_predicate]} \text{Expr}^{86} \big| \text{eventId}(t.\text{event}).\text{lid}?!i!\text{[l_value]} \text{Expr}^{87} \big) \rightarrow \text{Skip}
\]

\[
& \text{else if } t.\text{type} = \text{SIMPLE then} \\
& \quad \big| i : \{ x : \text{ID} \mid \text{[l_predicate]} \text{Expr}^{86} \big| \text{eventId}(t.\text{event}).\text{lid}?!i \rightarrow \text{Skip}
\]

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

where

\[
s.\text{trigger}.\text{broadcast}
\]
9. Conclusions

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 [CSW03], a process algebra that combines Z [WD96] and CSP, and includes time constructs [SCJS10]. 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 [CSW05], with a semantics given by the UTP theory in [ZSCS14].

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 [Hen96] to extend the notation, and from the UTP model of continuous variables [FTCW16] to define the semantics.
Appendices

A RoboChart diagrams - an informal overview .................................. 107
A.1 Time primitives

B Complete Metamodel ....................... 117

C Mathematical Toolkit ....................... 127

D OCL Well-formedness Conditions ....... 133
To illustrate the concepts, we present the model of a robot for chemical detection based on that in [HONHCKTT12] 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,a) takes a linear velocity lv and an angle a as parameters; it moves the vehicle forward at speed lv while turning by a degrees. The type of lv is real, and that of a 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 analysis 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 [OMG2011] 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.
Chapter A. RoboChart diagrams - an informal overview

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 2.9 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 wait\( (b) \), which waits for \( b \) time units. In the machine Movement of the chemical detector (see Figure A.5), we compose the shortRandomWalk() and changeDirection(l) calls with wait(outPeriod) and wait(evadeTime), where outPeriod and evadeTime are constants.

In the case of 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 move(l,v,a) operation, which is likely to involve just a simple assignment to actuator registers. So, the execution time of changeDirection(l) is negligible. The 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 shortRandomWalk(), although this operation is not defined, we expect it to take some time to actually effect the walk. So, 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 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 since\( (C) \), which yields the elapsed time since the most recent reset \( \#C \) of clock \( C \), and 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.

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
and angular speed, respectively. The local variable \( \text{segment} \) records how many sides of the square have been covered so far; the robot stops when it completes the square (\( \text{segment} == 4 \)). 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 \( \text{MovingForward} \) is composite. In this state, the motion is linear, unless an obstacle is detected. Linear motion is activated by calling the operation \( \text{moveForward}(\text{linear}) \) in the entry action with a constant value \( \text{linear} \) passed as a parameter.

Before \( \text{MovingForward} \) is actually entered, its entry action executes, followed by that of its substate \( \text{Observing} \), enabling the collision detection capability. Once a collision is detected, the event \( \text{collisionDetected} \) is raised by the robotic platform: the transition from \( \text{Observing} \) to the state \( \text{Collision} \) is then triggered, executing the exit action of \( \text{Observing} \) and subsequently the \( \text{avoid} \) 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 \( \text{wait}(2) \).

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 \( \text{Observing} \) 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 $\text{since}(C) == 5$. 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 $\text{sinceEntry(Turning)}$. Upon this transition, clock C is reset.

When entering the Turning state, the event $\text{stop}$ 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 $\text{stop}$ 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 $\text{stop}$ 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.

```plaintext

package robochart : robochart = "http://www.robocalc.circus/RoboChart" {
    class RCPackage {
        attribute name : String[?];
        property imports : Import[*] { composes }; 
        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];
    }
}
```
abstract class TypeDecl extends NamedElement;
class PrimitiveType extends TypeDecl;
class RecordType extends TypeDecl {
    property fields : Field[*] { composes };
}
class Field extends Member.NamedExpression;
abstract class TypedNamedElement extends NamedElement {
    property type : Type[1] { composes };
}
abstract class Member extends TypedNamedElement;
class Enumeration extends TypeDecl {
    property literals : Literal[*][1] { composes };
}
class Literal extends TypeDecl.NamedExpression {
    property types : Type[*] { composes };
}
class NameType extends TypeDecl {
    property type : Type[1] { composes };
}
abstract class Type;
class ProductType extends Type {
    property types : Type[2..*] { ordered composes };
}
class RelationType extends Type {
    property domain : Type[1] { composes };
    property range : Type[1] { composes };
}
class FunctionType extends RelationType;
class SetType extends Type {
    property domain : Type[1] { composes };
}
class SeqType extends SetType;
class TypeRef extends Type {
    property ref : TypeDecl[1];
}
class AnyType extends Type {
    attribute identifier : String[1];
}
class VariableList
```java
{
    attribute modifier : VariableModifier[1];
    property vars : Variable[*] { composes };
}
enum VariableModifier { serializable }
{
literal VAR;
literal CONST;
}
class Variable extends TypedNamedElement.NamedExpression
{
    property initial : Expression[*] { composes };
    attribute modifier : VariableModifier[1] { derived transient volatile };
}
class Event extends NamedElement
{
    property type : Type[*] { composes };
    attribute broadcast : Boolean[1];
}
class Function extends TypedNamedElement.NamedExpression
{
    property parameters : Parameter[*] { ordered composes };
    property preconditions : Expression[*] { composes };
    property postconditions : Expression[*] { composes };
}
class Parameter extends Variable;
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;
abstract class Reference;
class OperationRef extends Operation.Reference
{
    property ref : OperationDef[1];
}
class Interface extends NamedElement.BasicContext;
abstract class BasicContext
{
    property variableList : VariableList[*] { composes };
    property operations : OperationSig[*] { composes };
    property events : Event[*] { composes };
}
abstract class RoboticPlatform extends NamedElement.ConnectionNode;
class RoboticPlatformDef extends Context.RoboticPlatform;
abstract class Context extends BasicContext
class RoboticPlatformRef extends RoboticPlatform, Reference
{
    property ref : RoboticPlatformDef[1];
}
class StateMachine extends NamedElement, ConnectionNode;
class StateMachineDef extends StateMachineBody, StateMachine;
class StateMachineRef extends StateMachine, Reference
{
    property ref : StateMachineDef[1];
}
class StateMachineBody extends Context, NodeContainer
{
    property clocks : Clock[*] { composes };
}
class Clock extends NamedElement;
class State extends StateMachineBody, NodeContainer
{
    property actions : Action[*] { composes };
}
class Initial extends State;
class ProbabilisticJunction extends Junction;
class Transition extends NamedElement
{
    property source : Node[1];
    property target : Node[1];
    property start : Expression[?] { composes };
    property trigger : Trigger[?] { composes };
    property end : Expression[?] { composes };
    property condition : Expression[?] { composes };
    property action : Statement[?] { composes };
    property probability : Expression[?] { composes };
}
class Trigger
{
    property event : Event[?];
    property _from : Variable[?];
    property _predicate : Expression[?] { composes };
}
property parameter : Variable[?];
property value : Expression[?] { composes };
property time : Variable[?];
property reset : ClockReset[*] { ordered composes };
attribute _type : TriggerType[1];
}
enum TriggerType { serializable }
{
literal SIMPLE;
literal INPUT;
literal OUTPUT;
literal SYNC;
literal EMPTY;
}
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 };
  property lOperations : 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 start : Expression[?] { composes };
}
property stmt : Statement[1] { composes };  
property end : Expression[?] { 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 SendEvent extends Statement 
{  
property trigger : Trigger[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]; 
}
abstract class Expression;  
class ResultExp extends Expression;  
class ArrayExp extends Expression 
{  
property value : Expression[1] { composes };  
property parameters : Expression[*] { composes }; 
}
class ClockExp extends Expression 
{  
property clock : Clock[1]; 
}
class StateClockExp extends Expression
{
  property state : State[1];
}
abstract class BinaryExpression extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Iff extends BinaryExpression;
class Implies extends BinaryExpression;
class Or extends BinaryExpression;
abstract class QuantifierExpression extends Expression
{
  property variables : Variable[+] { composes };
  property suchthat : Expression[?] { composes };
  property predicate : Expression[1] { composes };
}
class Forall extends QuantifierExpression;
class Exists extends QuantifierExpression
{
  attribute unique : Boolean[1];
}
class LambdaExp extends Expression
{
  property variables : Variable[+] { composes };
  property suchthat : Expression[?] { composes };
  property expression : Expression[1] { composes };
}
class DefiniteDescription extends LambdaExp;
class IfExpression extends Expression
{
  property condition : Expression[1] { composes };
  property ifexp : Expression[1] { composes };
  property elseexp : Expression[1] { composes };
}
class Declaration extends NamedElement,NamedExpression
{
  property value : Expression[1] { composes };
}
class LetExpression extends Expression
{
  property declarations : Declaration[+] { composes };
  property expression : Expression[1] { composes };
}
class And extends BinaryExpression;
class Not extends Expression
{
  property exp : Expression[1] { composes };
}
class InExp extends Expression
{
    property member : Expression[1] { composes };
    property set : Expression[1] { composes };
}
class TypeExp extends Expression
{
    property type : Type[1] { composes };
}
class Equals extends BinaryExpression;
class Different extends BinaryExpression;
class GreaterThan extends BinaryExpression;
class GreaterOrEqual extends BinaryExpression;
class LessThan extends BinaryExpression;
class LessOrEqual extends BinaryExpression;
class Plus extends BinaryExpression;
class Minus extends BinaryExpression;
class Modulus extends BinaryExpression;
class Mult extends BinaryExpression;
class Div extends BinaryExpression;
class Cat extends BinaryExpression;
class Neg extends Expression
{
    property exp : Expression[1] { composes };
}
class Selection extends Expression
{
    property receiver : Expression[1] { composes };
    property member : Member[1];
}
class IntegerExp extends Expression
{
    attribute value :.ecore::EInt[1];
}
class FloatExp extends Expression
{
    attribute value :.ecore::EFloat[1];
}
class StringExp extends Expression
{
    attribute value : String[1];
}
class BooleanExp extends Expression
{
    attribute value : String[1];
}
class VarExp extends Expression
{
    property value : Variable[1];
class RefExp extends Expression {
    property ref : NamedExpression[1];
}
class ToExp extends Expression;
class FromExp extends Expression;
class IdExp extends Expression;
class AsExp extends Expression {
    property exp : Expression[1] { composes };
    property type : Type[1] { composes };
}
class IsExp extends Expression {
    property exp : Expression[1] { composes };
    property type : Type[1] { composes };
}
class EnumExp extends Expression {
    property type : Enumeration[1];
    property literal : Literal[1];
}
class ParExp extends Expression {
    property exp : Expression[1] { composes };
}
class SeqExp extends Expression {
    property values : Expression[*] { ordered composes };
}
class SetExp extends Expression {
    property values : Expression[*] { composes };
}
class SetComp extends Expression {
    property variables : Variable[*] { composes };
    property predicate : Expression[?] { composes };
    property expression : Expression[?] { composes };
}
class SetRange extends Expression {
    property start : Expression[1] { composes };
    property end : Expression[1] { composes };
}
class TupleExp extends Expression {
    property values : Expression[2..*] { ordered composes };
}
class RangeExp extends Expression
{
    attribute linterval : String[1];
    property lrange : Expression[1] { composes };
    property rrange : Expression[1] { composes };
    attribute rinterval : String[1];
}
class CallExp extends Expression
{
    property function : Expression[1] { composes };
    property args : Expression[*] { ordered composes };
}
class ElseExp extends Expression;
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 : Variable[1];
}
abstract class ConnectionNode;
abstract class NamedExpression;
class WaitingCondition
{
    property expression : Expression[?];
    property transitions : Transition[*] { ordered };
    attribute name : String[?] = '' { id };
}
class WaitingConditionRef extends Expression
{
    property ref : WaitingCondition[?];
}
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.
package set_toolkit

\[ f \text{ Union}(A: \text{Set}(?X)): \text{Set}(?X) \]
\[ \text{result} = \{x: ?X | \exists a: \text{Set}(?X) \mid a \in A \land x \in a\} \]

\[ f \text{ Inter}(A: \text{Set}(?X)): \text{Set}(?X) \]
\[ \text{result} = \{x: ?X | \forall a: \text{Set}(?X) \mid a \in A \land x \in a\} \]

\[ f \text{ notin}(m: ?X, s: \text{Set}(?X)): \text{boolean} \]
\[ \text{result} = \neg m \in s \]

\[ f \text{ diff}(s1: ?X, s2: ?X): \text{Set}(?X) \]
\[ \text{result} = \{x: ?X \mid x \in s1 \land \neg x \in s2\} \]

\[ f \text{ symetric_diff}(s1: ?X, s2: ?X): \text{Set}(?X) \]
\[ \text{result} = \{x: ?X \mid \neg (x \in s1 \iff x \in s2)\} \]

\[ f \text{ subseteq}(ss: ?X, s: ?X): \text{boolean} \]
\[ \text{result} = (\forall x: ?X \mid x \in s) \]

\[ f \text{ subset}(ss: ?X, s: ?X): \text{boolean} \]
\[ \text{result} = \text{subseteq}(ss, s) \land ss \neq s \]

\[ f \text{ union}(s1: ?X, s2: ?X): ?X \]
\[ \text{result} = \{x: ?X \mid x \in s1 \lor x \in s2\} \]

\[ f \text{ inter}(s1: ?X, s2: ?X): ?X \]
\[ \text{result} = \{x: ?X \mid x \in s1 \land x \in s2\} \]
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation_toolkit</td>
<td>`result=(p: ?X*?Y, q: ?Y*?Z</td>
</tr>
<tr>
<td><code>f_x</code> dom</td>
<td><code>(r: ?X&lt;&gt;-&gt;?X): Set(?X)</code></td>
</tr>
<tr>
<td><code>f_x</code> rimage</td>
<td><code>(r: ?X&lt;&gt;-&gt;?Y, a: Set(?X)): Set(?Y)</code></td>
</tr>
<tr>
<td><code>f_x</code> first</td>
<td><code>(p: ?X*?Y): ?X</code></td>
</tr>
<tr>
<td><code>f_x</code> second</td>
<td><code>(p: ?X*?Y): ?Y</code></td>
</tr>
<tr>
<td><code>f_x</code> ran</td>
<td><code>(r: ?X&lt;&gt;-&gt;?Y): Set(?Y)</code></td>
</tr>
<tr>
<td><code>f_x</code> id</td>
<td><code>(x: ?X&lt;&gt;-&gt;?X)</code></td>
</tr>
<tr>
<td>Imports</td>
<td><code>set_toolkit:::*</code></td>
</tr>
<tr>
<td><code>f_x</code> first</td>
<td><code>(p: ?X*?Y): ?X</code></td>
</tr>
<tr>
<td><code>f_x</code> second</td>
<td><code>(p: ?X*?Y): ?Y</code></td>
</tr>
<tr>
<td><code>f_x</code> ran</td>
<td><code>(r: ?X&lt;&gt;-&gt;?Y): Set(?Y)</code></td>
</tr>
<tr>
<td><code>f_x</code> id</td>
<td><code>(x: ?X&lt;&gt;-&gt;?X)</code></td>
</tr>
</tbody>
</table>
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 ))
Chapter D. OCL Well-formedness Conditions

-- 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() \rightarrow collect(c \mid c.interfaces \rightarrow union(c.pInterfaces)) \rightarrow includes(self)
implies self.events \rightarrow isEmpty()

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

-- I3: Names of variables, events and operations are unique
inv I3: self.variableList.vars.name
\rightarrow union(self.operations->asBag().name)
\rightarrow union(self.events->asBag().name) \rightarrow isUnique(i \mid 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()
    \rightarrow includeAll(self.moduleControllers().controllerDef().controllerRequiredVars())
and self.moduleRP().rpDef().rpProvidedOps()
    \rightarrow includeAll(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: RModule.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.oclIsKindOf(RoboticPlatform) or self.to.oclIsKindOf(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 RModule.allInstances())->exists(m | m.
    connections->includes(self))
    implies (
      (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.eto = self.efrom))
          ->forAll(c | c.bidirec))
      and (self.to.oclIsKindOf(Controller)
        implies self.to.oclAsType(Controller).controllerDef().connections
          ->select(c | (c.from = self.to and c.efrom = self.eto)
            or (c.to = self.to and c.eto = self.eto))
          ->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 RModule.allInstances())->exists(m | m.connections->includes(self))
Chapter D. OCL Well-formedness Conditions

and self.tooclIsKindOf(Controller)
) implies self.tooclAsType(Controller).controllerDef().connections
  ->select(c | (c.from = self.to and c.efrom = self.eto)
    or (c.to = self.to and c.eto = self.eto))
  ->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 RCModule.allInstances() ->exists(m | m.connections -> includes(self))
  and self.fromoclIsKindOf(Controller)
) implies self.fromoclAsType(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 | 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.tooclIsKindOf(StateMachine)
) implies self.tooclAsType(StateMachine).stmDef().ncOutputEvents() ->
  excludes(self.eto)
-- 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.fromoclIsKindOf(StateMachine)
) implies self.fromoclAsType(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.variableListvars
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 cannot 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
-- this is implicit since Initial inherits from Junction

context Junction
-- J1: A junction must contain at least one outgoing transition
inv J1: NodeContainer.allInstances().select(nc | nc.nodes->includes(self)).transitions->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().select(nc | nc.nodes->includes(self)).transitions->select(t | t.source = self and t.condition <> null).condition
-- J3: Transitions starting in junctions cannot have triggers
inv J3: NodeContainer.allInstances().select(nc | nc.nodes->includes(self)).transitions->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().select(nc | nc.nodes->includes(self)).transitions->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
   implies (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
   implies (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 I2)
   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

\[ \text{inv } \text{TSI: self.containingStateMachine().clocks->includes(self.clock)} \]

-- Auxiliary definitions

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

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

-- function to extract StateMachineDef from a StateMachine (which may be a ref)
context StateMachine
def: stmDef() : StateMachineDef =
  if selfoclIsKindOf(StateMachineDef) then
    selfoclAsType(StateMachineDef)
  else
    selfoclAsType(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 SYNC trigger.
-- or if it is used in an OUTPUT, SYNC 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())

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

-- functions on expressions to support timed expressions well-formedness
conditions

context Expression
def: parentIsTransition() : Boolean =
    if self.oclContainer().oclIsKindOf(Expression) then
        self.oclContainer().oclAsType(Expression).parentIsTransition()
    else
        Transition.allInstances()->exists(t | t.condition = self)
    endif
def: parentTransition() : Transition =
    if self.oclContainer().oclIsKindOf(Expression) then
        self.oclContainer().oclAsType(Expression).parentTransition()
    else
        self.oclContainer().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 self.oclIsKindOf(StateMachineBody) then 
    self.oclAsType(StateMachineBody) 
  else 
    self.oclContainer().oclAsType(NodeContainer).containingStateMachine() 
  endif
context Transition
def: containingStateMachine() : StateMachineBody = 
  self.oclContainer().oclAsType(NodeContainer).containingStateMachine()
context Action
def: containingStateMachine() : StateMachineBody = 
  self.oclContainer().oclAsType(State).containingStateMachine()
context Statement
def: containingStateMachine() : StateMachineBody = 
  if self.oclIsKindOf(Action) then 
    self.oclContainer().oclAsType(Action).containingStateMachine() 
  elseif self.oclIsKindOf(Transition) then 
    self.oclContainer().oclAsType(Transition).containingStateMachine() 
  else 
    self.oclContainer().oclAsType(Statement).containingStateMachine() 
  endif
context Trigger
def: containingStateMachine() : StateMachineBody = 
  if self.oclIsKindOf(Transition) then 
    self.oclContainer().oclAsType(Transition).containingStateMachine() 
  else 
    self.oclContainer().oclAsType(Statement).containingStateMachine() 
  endif
context ClockReset
def: containingStateMachine() : StateMachineBody = 
  if self.oclIsKindOf(Trigger) then 
    self.oclContainer().oclAsType(Trigger).containingStateMachine() 
  else 
    self.oclContainer().oclIsKindOf(Trigger) then 
  endif
selfoclContainer().oclAsType(Trigger).containingStateMachine()
else
  selfoclContainer().oclAsType(Statement).containingStateMachine()
endif

endpackage
Credits

\LaTeX{} style based on the The LeGrand Orange Book Template by Mathias Legrand and Vel from LaTeXTemplates.com. Licensed under CC BY-NC-SA 3.0

Cover photo by Bertrand Bouchez on Unsplash

Table of contents photo by Jack B on Unsplash

Header photo for chapter I by Markus Spiske on Unsplash

Header photo for remaining chapter by Frank Wang on Unsplash

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

- Icon made by Iconnice from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Sarfraz Shoukat from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Dario Ferrando from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Lyolya from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Google from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
- Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
Bibliography
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.

Action, 52
allClockVariables (timed), 58
allConstants, 37
allDeadlineTransitions (timed), 70
allEvents, 36
allLocalConstants, 37
allLocalVariables, 37
allTransitions, 50
allVariables, 37
alphaClockReset, 59
    And (timed), 59
    CallExp (timed), 59
    ClockExp (timed), 61
    Equals (timed), 60
    GreaterOrEqual (timed), 60
    GreaterThan (timed), 60
    Iff (timed), 60
    Implies (timed), 60
    LessOrEqual (timed), 60
    LessThan (timed), 60
    Not (timed), 59
    Or (timed), 60
    ParExp (timed), 59
    StateClockExp (timed), 61
    alphaClockResetCallArgs (timed), 59
BBuffe (collection), 101
    buffer, 40
    buildScope, 49
        timed, 58
    C (Controller), 40
    Cln (Collection, collection), 100
    clockResets (timed), 58
    compileTarget, 47
INDEX OF SEMANTIC RULES

compileWC (timed), 64–68
composeControllers, 39
composeMachines, 41
composeStates, 44, 72
constInit, 38
constInitSTM, 48
   timed, 58
ctrlMemory, 41

deadlineEvents (timed), 69

exitSubstates, 47
Expr (Expression), 53
   And Expression, 53
   Array Expression, 53
   Boolean Expression, 53
   Call Expression, 53
   Concatenation Expression, 53
   Division Expression, 54
   Equals Expression, 54
   Greater or Equal Expression, 54
   Greater Than Expression, 54
   If and Only If Expression, 54
   Implies Expression, 54
   Integer Expression, 54
   Less or Equal Expression, 55
   Less Than Expression, 55
   Minus Expression, 55
   Modulus Expression, 55
   Multiplication Expression, 55
   Negation Expression, 55
   Not Equal Expression, 54
   Not Expression, 55
   Or Expression, 55
   Parenthesised Expression, 56
   Plus Expression, 56
   Range Expression, 56
   Sequence Expression, 56
   Set Expression, 56
   Tuple Expression, 56
   timed, 70
   Composite State, 45
flowTriggerEvents, 46
flowEvents, 44
gersetChannels, 43
gersetLocalChannels, 44
hiddenModuleChannels, 36
initialisation, 43
J (Junction), 43
M (Module), 36
memoryChannels, 36
memoryDeadline (timed), 70
memoryTransition, 49
   timed, 70
modMemory, 38

reachableJunctions, 43
reachableTransitions, 46
renameTriggerEvents, 42
renamingController, 39
renamingMachine, 42
renCtrlEvts, 39
renStmEvts, 42
requiredConstants, 37
requiredVariables, 37
restrictedState, 46
S (State), 44
   timed, 70
   Composite State, 45
   timed, 71
   Final State, 46
   Simple State, 45
   timed, 71
singleBuffer, 40
Statement, 50, 50, 51
   timed, 72, 72, 73
   Call Statement, 73
   ClockReset, 73
   Wait, 73
Call Statement, 51
If Statement, 52
Send Event, 52

**collection**, 101

Sequential Composition, 52

Skip, 52

StatementInContext, 51

states, 43

STM (State Machine), 42

**timed**, 57

stmClocks (**timed**), 59

stmMemory, 48

**timed**, 69

substatesTriggers, 46

T (Transition), 47

transitionsFrom, 50

trigEvents, 44

Trigger, 49, 61

**collection**, 101

triggerDeadlines (**timed**), 72

triggerEvent, 49, 69

triggerForMemory, 49

usedVariables, 51, 51

wc (**timed**), 62

wcArgSeq (**timed**), 63
Index of Calls to Semantic Rules

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.

Action , 45\textsuperscript{1}, 45\textsuperscript{2}, 45\textsuperscript{3}, 45\textsuperscript{4}, 45\textsuperscript{5}, 45\textsuperscript{6}, 47\textsuperscript{10}, 47\textsuperscript{7}, 47\textsuperscript{8}, 47\textsuperscript{9}, 71\textsuperscript{11}, 71\textsuperscript{12}, 71\textsuperscript{13}, 71\textsuperscript{14}, 71\textsuperscript{15}, 71\textsuperscript{16}
allClockVariables (timed) , 57\textsuperscript{1}, 69\textsuperscript{2}
allConstants , 38\textsuperscript{1}, 43\textsuperscript{2}, 43\textsuperscript{3}, 44\textsuperscript{4}, 48\textsuperscript{5}, 48\textsuperscript{6}, 57\textsuperscript{7}, 69\textsuperscript{8}, 69\textsuperscript{9}
alldeadlineTransitions (timed) , 69\textsuperscript{1}
allevents , 42\textsuperscript{1}
allocalConstants , 36\textsuperscript{1}, 40\textsuperscript{2}, 44\textsuperscript{3}
allocalVariables , 36\textsuperscript{1}, 38\textsuperscript{2}, 40\textsuperscript{3}, 41\textsuperscript{4}, 44\textsuperscript{5}, 48\textsuperscript{6}, 69\textsuperscript{7}
altranstions , 44\textsuperscript{1}, 46\textsuperscript{2}, 46\textsuperscript{3}, 46\textsuperscript{4}, 48\textsuperscript{5}, 50\textsuperscript{6}, 57\textsuperscript{7}, 69\textsuperscript{8}, 69\textsuperscript{9}, 70\textsuperscript{10}
alvariables , 43\textsuperscript{1}, 43\textsuperscript{2}, 44\textsuperscript{3}
alphaClockReset (timed) , 58\textsuperscript{1}, 59\textsuperscript{2}, 59\textsuperscript{3}, 59\textsuperscript{4}, 59\textsuperscript{5}, 59\textsuperscript{6}, 59\textsuperscript{7}, 60\textsuperscript{10}, 60\textsuperscript{11}, 60\textsuperscript{12}, 60\textsuperscript{13}, 60\textsuperscript{14}, 60\textsuperscript{15}, 60\textsuperscript{16}, 60\textsuperscript{17}, 60\textsuperscript{18}, 60\textsuperscript{19}, 60\textsuperscript{20}, 60\textsuperscript{21}, 60\textsuperscript{22}, 60\textsuperscript{23}, 60\textsuperscript{8}, 60\textsuperscript{9}
alphaClockResetCallArgs (timed) , 59\textsuperscript{1}, 59\textsuperscript{2}
BBuffer , 100\textsuperscript{1}, 100\textsuperscript{2}
buffer , 36\textsuperscript{1}
buildScope , 48\textsuperscript{1}, 49\textsuperscript{2}, 49\textsuperscript{3}
buildScope (timed) , 58\textsuperscript{1}
C (Controller) , 39\textsuperscript{1}
clockResets (timed) , 57\textsuperscript{1}, 57\textsuperscript{2}
compileTarget , 47\textsuperscript{1}, 47\textsuperscript{2}, 47\textsuperscript{3}
compileWC (timed) , 59\textsuperscript{1}
composeControllers , 36\textsuperscript{1}, 39\textsuperscript{2}
composeMachines , 40\textsuperscript{1}, 41\textsuperscript{2}
composeStates , 42\textsuperscript{1}, 44\textsuperscript{2}, 45\textsuperscript{3}, 57\textsuperscript{4}
composeStates (timed) , 71\textsuperscript{1}, 72\textsuperscript{2}
constInit , 38\textsuperscript{1}, 41\textsuperscript{2}
constInitSTM , 48\textsuperscript{1}
constInitSTM (timed) , 57\textsuperscript{1}, 58\textsuperscript{2}, 58\textsuperscript{3}
ctrlMemory , 40\textsuperscript{1}
deadlineEvents (timed) , 571, 572
exitSubstates , 451, 472, 713, 714
Expr (Expression) , 381, 482, 493, 514, 515, 526, 527, 528, 5310, 5311, 5312, 5313, 5314, 5315, 5316, 5317, 5318, 5418, 5419, 5420, 5421, 5422, 5423, 5424, 5425, 5426, 5427, 5428, 5429, 5430, 5431, 5532, 5533, 5534, 5535, 5536, 5537, 5538, 5539, 5540, 5541, 5542, 5543, 5544, 5545, 5646, 5647, 5648, 5649, 5650, 5651, 5652, 5653, 5854, 6455, 6456, 6457, 6458, 6559, 6560, 6561, 6562, 6663, 6664, 6665, 6666, 6667, 6668, 6669, 6670, 6671, 6672, 6673, 6674, 6675, 6676, 6677, 6678, 7079, 7080, 7081, 7282, 7283, 7384, 7385, 10186, 10187, 10188

flowEvents , 441, 442, 723, 724
flowTriggerEvents , 451, 712

gsetChannels , 421, 572
gsetLocalChannels , 421, 572

hiddenModuleChannels , 361
initialisation , 421, 452, 573, 714

J (Junction) , 431, 452, 453, 714, 715

M (Module) , 1001
memoryChannels , 361, 362
memoryDeadline (timed) , 691
memoryTransition , 481
memoryTransition (timed) , 691
modMemory , 361

reachableJunctions , 431, 452, 453, 464, 715, 716
reachableTransitions , 461

readState , 501, 512, 723
renameTriggerEvents , 421, 572
renamingController , 391, 392
renamingMachine , 411, 412
renCtrlEvts , 391, 392
renStmEvts , 411, 412
requiredConstants , 391, 392, 403, 424, 425
requiredVariables , 361, 382, 403, 414, 415, 486, 697

restrictedState , 441, 442, 723, 724

S (State) , 461
Statement , 511, 722
StatementInContext , 521, 522, 523, 524
states , 421, 422, 443, 454, 455, 456, 457, 458, 4610, 469, 4711, 5712, 5713, 7114, 7115, 7116

STM (State machine) , 401, 422
STM (State machine, timed) , 731
stmClocks (timed) , 571
stmMemory , 421

stmMemory (timed) , 571

substatesTriggers , 461

T (Transition) , 431, 432, 433, 454, 455, 716, 717

transitionsFrom , 431, 452, 453, 464, 465, 466, 717, 718, 729

trigEvents , 421, 572, 573

Trigger , 471, 612
tiggerDeadlines (timed) , 711, 712
tiggerEvent , 441, 462, 693
tiggerEvent (timed) , 571, 592, 6410, 643, 644, 645, 646, 647, 648, 649, 6511, 6512, 6513, 6514, 6515, 6516, 6517, 6518, 6619, 6620, 6621, 6622, 6623, 6624, 6625, 6626, 6627, 6628, 6629, 6730, 6731, 6732, 6733, 6734, 6835, 6836, 6837, 6838, 6839, 6840, 6841, 6842, 6843, 6844, 6845, 6846

triggerForMemory , 491, 492, 703, 704
usedVariables, 51^1, 51^2, 51^3, 51^4, 51^5, 51^6, 51^7, 51^8, 72^9

wc (timed), 63^1
wcArgSeq (timed), 62^1, 62^2, 63^3
Index

Connection
  async, 16
  bidirec, 16
  Description, 16
efrom, 16
eto, 16
from, 16
to, 16
ConnectionNode
  Description, 16
Event
  Type, 16

MachineContainer
  Description, 15
Module
  Description, 17

RCPackage
  Description, 15
Robotic Platform
  Description, 16
  RoboticPlatformDef, 16
  RoboticPlatformRef, 16
  Well Formedness, 29