Abstract

In this report, we provide a reference manual for a UML-like notation called RoboChart, designed specifically for modelling autonomous and mobile robots, and including timed and probabilistic primitives. We describe the syntax of RoboChart and its extensions, as well as the well-formedness conditions and semantics of the language constructs. Additionally, usage of the language is discussed via an application programming interface (API) and examples.
## Contents

1 Introduction 10

2 Syntax 12
  2.1 RoboChart metamodel 12

3 Well-formedness Conditions 22
  3.1 Core Language 22
    3.1.1 Robotic Platforms 22
    3.1.2 Interfaces 23
    3.1.3 Modules 23
    3.1.4 Connection 23
    3.1.5 Controllers 23
    3.1.6 State Machines 24
    3.1.7 States 24
    3.1.8 Initial junctions 25
    3.1.9 Junction 25
    3.1.10 Final states 25
    3.1.11 Transitions 25
    3.1.12 Operations 25
  3.2 Timed Language 25
    3.2.1 Timed Expressions 25
    3.2.2 Timed Statements 26

4 Semantics 27
  4.1 Detailed Semantics: Core Language 28
List of Figures

2.1 Metamodel of RoboChart packages ........................................ 13
2.2 Metamodel of RoboChart modules ......................................... 14
2.3 Metamodel of RoboChart controllers ..................................... 14
2.4 Metamodel of RoboChart context for elements and operations ..... 15
2.5 Metamodel of RoboChart state machines ................................. 16
2.6 Concrete syntax of transitions .............................................. 16
2.7 Metamodel of RoboChart triggers ........................................ 17
2.8 Concrete syntax of triggers ............................................... 17
2.9 Metamodel of RoboChart types .......................................... 18
2.10 Metamodel of RoboChart time primitives ............................. 21
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Semantics of modules</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Memory channels</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Function allVariables</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Function requiredVariables</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Function allLocalVariables</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Module Memory</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Composition of controllers</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Renaming controller</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>Renaming controller events</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>Buffer</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>Semantics of controllers</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>Controller Memory</td>
<td>34</td>
</tr>
<tr>
<td>13</td>
<td>Composition of machines</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Renaming state machine</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>Renaming machine events</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>Semantics of state machine</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>Function substates</td>
<td>36</td>
</tr>
<tr>
<td>18</td>
<td>Initialisation</td>
<td>37</td>
</tr>
<tr>
<td>19</td>
<td>Get and Set channels</td>
<td>37</td>
</tr>
<tr>
<td>20</td>
<td>Composition of states</td>
<td>37</td>
</tr>
<tr>
<td>21</td>
<td>Trigger events</td>
<td>38</td>
</tr>
<tr>
<td>22</td>
<td>Get and set local channels</td>
<td>38</td>
</tr>
<tr>
<td>23</td>
<td>Flow events</td>
<td>38</td>
</tr>
<tr>
<td>24</td>
<td>Semantics of states</td>
<td>38</td>
</tr>
<tr>
<td>25</td>
<td>Semantics of simple states</td>
<td>39</td>
</tr>
<tr>
<td>26</td>
<td>Semantics of composite states</td>
<td>40</td>
</tr>
<tr>
<td>27</td>
<td>Semantics of final states</td>
<td>40</td>
</tr>
<tr>
<td>28</td>
<td>Synchronisation events between parent state and substates</td>
<td>41</td>
</tr>
<tr>
<td>29</td>
<td>Triggers of substates</td>
<td>41</td>
</tr>
<tr>
<td>30</td>
<td>Restricted semantics of states</td>
<td>41</td>
</tr>
<tr>
<td>31</td>
<td>Semantics of transitions</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>32</td>
<td>Compile target</td>
<td>42</td>
</tr>
<tr>
<td>33</td>
<td>Exit substates</td>
<td>43</td>
</tr>
<tr>
<td>34</td>
<td>State-machine Memory</td>
<td>43</td>
</tr>
<tr>
<td>35</td>
<td>Semantics of triggers</td>
<td>44</td>
</tr>
<tr>
<td>36</td>
<td>Event for transition trigger</td>
<td>44</td>
</tr>
<tr>
<td>37</td>
<td>Memory transitions</td>
<td>44</td>
</tr>
<tr>
<td>38</td>
<td>Function transitionsFrom</td>
<td>44</td>
</tr>
<tr>
<td>39</td>
<td>Function allTransitions</td>
<td>45</td>
</tr>
<tr>
<td>40</td>
<td>Function allSTMTransitions</td>
<td>45</td>
</tr>
<tr>
<td>41</td>
<td>Semantics of statements</td>
<td>45</td>
</tr>
<tr>
<td>42</td>
<td>Read state of an expression</td>
<td>46</td>
</tr>
<tr>
<td>43</td>
<td>Semantics of statements in context</td>
<td>46</td>
</tr>
<tr>
<td>44</td>
<td>Function usedVariables</td>
<td>47</td>
</tr>
<tr>
<td>45</td>
<td>Semantics of assignment</td>
<td>47</td>
</tr>
<tr>
<td>46</td>
<td>Semantics of call statement</td>
<td>48</td>
</tr>
<tr>
<td>47</td>
<td>Semantics of if statements</td>
<td>48</td>
</tr>
<tr>
<td>48</td>
<td>Semantics of send event statements</td>
<td>49</td>
</tr>
<tr>
<td>49</td>
<td>Semantics of sequential composition</td>
<td>49</td>
</tr>
<tr>
<td>50</td>
<td>Semantics of skip</td>
<td>49</td>
</tr>
<tr>
<td>51</td>
<td>Semantics of actions</td>
<td>50</td>
</tr>
<tr>
<td>52</td>
<td>Semantics of expressions</td>
<td>50</td>
</tr>
<tr>
<td>53</td>
<td>Semantics of and expression</td>
<td>50</td>
</tr>
<tr>
<td>54</td>
<td>Semantics of array expression</td>
<td>51</td>
</tr>
<tr>
<td>55</td>
<td>Semantics of boolean expression</td>
<td>51</td>
</tr>
<tr>
<td>56</td>
<td>Semantics of call expression</td>
<td>52</td>
</tr>
<tr>
<td>57</td>
<td>Semantics of concatenation expression</td>
<td>52</td>
</tr>
<tr>
<td>58</td>
<td>Semantics of not equal expression</td>
<td>52</td>
</tr>
<tr>
<td>59</td>
<td>Semantics of division</td>
<td>53</td>
</tr>
<tr>
<td>60</td>
<td>Semantics of equality</td>
<td>53</td>
</tr>
<tr>
<td>61</td>
<td>Semantics of greater or equal expression</td>
<td>53</td>
</tr>
<tr>
<td>62</td>
<td>Semantics of greater than</td>
<td>53</td>
</tr>
<tr>
<td>63</td>
<td>Semantics of if and only if expression</td>
<td>54</td>
</tr>
<tr>
<td>64</td>
<td>Semantics of implication</td>
<td>54</td>
</tr>
<tr>
<td>65</td>
<td>Semantics of integer expression</td>
<td>54</td>
</tr>
<tr>
<td>66</td>
<td>Semantics of less or equal expression</td>
<td>54</td>
</tr>
<tr>
<td>67</td>
<td>Semantics of less than</td>
<td>55</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>Semantics of minus ........................................ 55</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>Semantics of modulus ........................................ 55</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>Semantics of multiplication ................................. 55</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Semantics of arithmetic negation ............................. 56</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Semantics of logical negation ................................. 56</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>Semantics of or expression .................................. 56</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Semantics of parenthesised expression ....................... 56</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>Semantics of plus ............................................ 57</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>Semantics of range expression ................................ 57</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>Semantics of sequence expression ............................ 57</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>Semantics of set expression .................................. 57</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>Semantics of tuple expression ................................ 57</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>Timed Semantics .................................................. 58</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>Semantics of state machine ..................................... 59</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>allClockVariables function ..................................... 59</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>clockResets function ............................................ 60</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>stmClocks function ............................................... 60</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>alphaClockReset function ....................................... 60</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>alphaClockReset function ....................................... 61</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>alphaClockReset function ....................................... 61</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>alphaClockResetCallArgs function .............................. 61</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>alphaClockReset function ....................................... 61</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>alphaClockReset function ....................................... 62</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>alphaClockReset function ....................................... 62</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>alphaClockReset function ....................................... 62</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>alphaClockReset function ....................................... 62</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>alphaClockReset function ....................................... 63</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>alphaClockReset function ....................................... 63</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>alphaClockReset function ....................................... 63</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>alphaClockReset function ....................................... 63</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>alphaClockReset function ....................................... 64</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>alphaClockReset function ....................................... 64</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>wc function ...................................................... 64</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>wc function (ParExp) ........................................... 65</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>102</td>
<td>wc function (Not)</td>
<td>65</td>
</tr>
<tr>
<td>103</td>
<td>wc function (CallExp)</td>
<td>65</td>
</tr>
<tr>
<td>104</td>
<td>wcArgSeq function</td>
<td>66</td>
</tr>
<tr>
<td>105</td>
<td>wc function (And)</td>
<td>66</td>
</tr>
<tr>
<td>106</td>
<td>wc function (Or)</td>
<td>67</td>
</tr>
<tr>
<td>107</td>
<td>wc function (Implies)</td>
<td>67</td>
</tr>
<tr>
<td>108</td>
<td>wc function (Iff)</td>
<td>67</td>
</tr>
<tr>
<td>109</td>
<td>wc function (GreaterThan)</td>
<td>68</td>
</tr>
<tr>
<td>110</td>
<td>wc function (GreaterOrEqual)</td>
<td>68</td>
</tr>
<tr>
<td>111</td>
<td>wc function (LessThan)</td>
<td>69</td>
</tr>
<tr>
<td>112</td>
<td>wc function (LessOrEqual)</td>
<td>69</td>
</tr>
<tr>
<td>113</td>
<td>compileWC function</td>
<td>70</td>
</tr>
<tr>
<td>114</td>
<td>getClockReset function</td>
<td>70</td>
</tr>
<tr>
<td>115</td>
<td>getClockReset function</td>
<td>71</td>
</tr>
<tr>
<td>116</td>
<td>getClockReset function</td>
<td>71</td>
</tr>
<tr>
<td>117</td>
<td>getClockReset function</td>
<td>71</td>
</tr>
<tr>
<td>118</td>
<td>compileWC function</td>
<td>72</td>
</tr>
<tr>
<td>119</td>
<td>compileWC function</td>
<td>73</td>
</tr>
<tr>
<td>120</td>
<td>compileWC function</td>
<td>74</td>
</tr>
<tr>
<td>121</td>
<td>compileWC function</td>
<td>75</td>
</tr>
<tr>
<td>122</td>
<td>compileWC function</td>
<td>76</td>
</tr>
<tr>
<td>123</td>
<td>deadlineEvents function</td>
<td>77</td>
</tr>
<tr>
<td>124</td>
<td>triggerEvent function</td>
<td>77</td>
</tr>
<tr>
<td>125</td>
<td>State-machine Memory</td>
<td>78</td>
</tr>
<tr>
<td>126</td>
<td>allTriggerDeadlineTransitions function</td>
<td>79</td>
</tr>
<tr>
<td>127</td>
<td>memoryTransition function</td>
<td>79</td>
</tr>
<tr>
<td>128</td>
<td>Memory deadline</td>
<td>79</td>
</tr>
<tr>
<td>129</td>
<td>Timed semantics of states</td>
<td>80</td>
</tr>
<tr>
<td>130</td>
<td>Timed semantics of simple states</td>
<td>81</td>
</tr>
<tr>
<td>131</td>
<td>Timed semantics of composite states</td>
<td>82</td>
</tr>
<tr>
<td>132</td>
<td>Semantics of trigger deadlines</td>
<td>83</td>
</tr>
<tr>
<td>133</td>
<td>Timed composition of states</td>
<td>83</td>
</tr>
<tr>
<td>134</td>
<td>Semantics of timed statements</td>
<td>84</td>
</tr>
<tr>
<td>135</td>
<td>Function usedVariables for timed semantics</td>
<td>84</td>
</tr>
<tr>
<td>136</td>
<td>Semantics of Deadlines</td>
<td>84</td>
</tr>
<tr>
<td>137</td>
<td>Semantics of Wait</td>
<td>84</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

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

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

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

The time primitives of RoboChart allow time budgets and deadlines to be specified for operations and events directly as part of a state machine. Constraints can be specified in association with the relative-time elapsed since the occurrence of events or the entering of states. Our time primitives are inspired by constructs of timed automata [1] and Timed CSP [10].

UML [8] state machines are popular. RoboChart, however, is customised for robotic applications, via the extra notions of robotic platform, controller, and a specialised API. Moreover, RoboChart provides support for time and probabilistic specifications that to make it suitable for verification and automatic generation of simulations.

In this report, we formalise the semantics of the core and timed constructs of RoboChart using CSP [9]. Importantly, CSP is a front end for a mathematical model that supports a number
analysis techniques such as model-checking, which provide a high degree of automation, as well as more powerful (but not automatic) verification based on interactive and theorem proving, namely, Hoare and He’s Unifying Theories of Programming [7] (UTP). Use of CSP enables model checking with FDR [5]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [11] and probability [14].

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.
Chapter 2

Syntax

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

Sections ?? 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 variable lists, operations, and events.

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) events. 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.

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.

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 2.2: Metamodel of RoboChart modules

Figure 2.3: Metamodel of RoboChart controllers

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 can also be defined by a reference or by a StateMachineBody.

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.
Figure 2.5: Metamodel of RoboChart state machines

(({Expression}>>)? Trigger (<{Expression}>)? ([Expression])? (/Statement)?

Figure 2.6: Concrete syntax of transitions

The concrete syntax of transitions is shown in Figure 2.6. The first and second expression are the start and end deadlines. The syntax of triggers is described in Figure 2.8. The third expression is the transition conditions and the statement is the transition action.

A Trigger is defined in Figure 2.7. It has an event. It can be on its own, in which case we have a SimpleTrigger. It, however, can also provide a synchronisation value (SyncTrigger) or an output value (OutputTrigger), or yet take an input in a parameter variable (InputTrigger).

The concrete syntax of triggers is shown in Figure 2.8. 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.
Figure 2.7: Metamodel of RoboChart triggers

\[(\text{Input}|\text{Output}|\text{Sync}|\text{Simple})?\ (\@\text{Variable})?\ \text{ClockReset}\]

Figure 2.8: Concrete syntax of triggers

Table 2.1: Concrete syntax of triggers.

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Trigger (I)</td>
<td>Event ? Variable</td>
<td>Receives any value from the event and stores it on the variable.</td>
</tr>
<tr>
<td>Output Trigger (O)</td>
<td>Event ! Expression</td>
<td>Sends the value of the expression through the event.</td>
</tr>
<tr>
<td>Sync Trigger (Sync)</td>
<td>Event . Expression</td>
<td>Synchronises on the event with the value of the expression.</td>
</tr>
<tr>
<td>Simple Trigger (S)</td>
<td>Event</td>
<td>Synchronises on the event.</td>
</tr>
</tbody>
</table>
Table 2.2: Concrete syntax of types

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesised Type</td>
<td>(T)</td>
<td></td>
</tr>
<tr>
<td>Type Reference</td>
<td>N</td>
<td>N is the name of a declared type.</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>

The metamodel for Types is given in Figure 2.9. As indicated, they include PrimitiveTypes, given just by their names, Enumerations, records (DataType), sets, relations, functions, and sequences (SetType, RelationType, and so on). A TypeDeclaration gives a name to type to be used in TypeReferences. Table 2.2 gives the concrete syntax for the various type constructors.

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. Table 2.3 gives the concrete syntax for expressions. We omit the simple metamodel diagrams here.

Similarly, the action language is very simple. Table 2.5 gives the concrete syntax of statements used to define actions in states and transitions.

The time primitives are described separately in Figure 2.10. They are basically some extra expressions and statements. A ClockExpression since is a condition involving a clock. A StateClockExpression is a sinceEntry expression. A TimedStatement defines a deadline. A Wait
<table>
<thead>
<tr>
<th>Element</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>Quoted values</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 e1.</td>
</tr>
<tr>
<td>Set</td>
<td>{e1,e2,...}</td>
<td>Set with values e1.</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 e1 and e2, and open interval between e3 and e4, or a combination of both.</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>Selection</td>
<td>e.n</td>
<td>The n field of record e.</td>
</tr>
<tr>
<td>Negation</td>
<td>-e</td>
<td>Arithmetical negation of expression e.</td>
</tr>
<tr>
<td>Concatenation</td>
<td>e1^e2</td>
<td>Concatenate sequences e1 and e2.</td>
</tr>
<tr>
<td>Modulo</td>
<td>e1 % e2</td>
<td>Remainder of dividing e1 by e2.</td>
</tr>
<tr>
<td>Division</td>
<td>e1 / e2</td>
<td>Division of e1 by e2.</td>
</tr>
<tr>
<td>Multiplication</td>
<td>e1 * e2</td>
<td>Multiplication of e1 by e2.</td>
</tr>
<tr>
<td>Sum</td>
<td>e1 + e2</td>
<td>Sum of e1 and e2.</td>
</tr>
<tr>
<td>Subtraction</td>
<td>e1 - e2</td>
<td>Subtraction of e1 and e2.</td>
</tr>
<tr>
<td>Conditional</td>
<td>if c then e else f end</td>
<td>If condition c is true, e1 else e2.</td>
</tr>
<tr>
<td>Local definition</td>
<td>let n == e @ f</td>
<td>Define locally n and use it to calculate f.</td>
</tr>
<tr>
<td>Definite description</td>
<td>the x: T</td>
<td>P @ e</td>
</tr>
<tr>
<td>Lambda expression</td>
<td>lambda x: T</td>
<td>P @ e</td>
</tr>
<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 2.3: Concrete syntax of expressions (1)
Table 2.4: Concrete syntax of expressions (2)

<table>
<thead>
<tr>
<th>Element</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 quantifier</td>
<td>forall x: T</td>
<td>P @ Q</td>
</tr>
<tr>
<td>Existential quantifier</td>
<td>exists x: T</td>
<td>P @ Q</td>
</tr>
<tr>
<td>Uniqueness Existential quantifier</td>
<td>exists1 x: T</td>
<td>P @ e</td>
</tr>
</tbody>
</table>

Table 2.5: Concrete syntax of statements

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>skip</td>
<td>Statement that terminates immediately.</td>
</tr>
<tr>
<td>Call</td>
<td>o(e1,e2,...)</td>
<td>Calls operation o with parameters e1.</td>
</tr>
<tr>
<td>Conditional</td>
<td>if c then S1 else S2 end</td>
<td>If c is true, execute S1, otherwise execute S2.</td>
</tr>
<tr>
<td>Assignment</td>
<td>x = e</td>
<td>Assign expression e to variable x.</td>
</tr>
<tr>
<td>Output event</td>
<td>ev!e</td>
<td>Output value e through channel ev.</td>
</tr>
<tr>
<td>Input event</td>
<td>ev?x</td>
<td>Receive value through channel ev and store it in variable x.</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>ev.e</td>
<td>Synchronise on value e through channel ev.</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>ev</td>
<td>Synchronise on event ev.</td>
</tr>
<tr>
<td>Sequential composition</td>
<td>S1;S2</td>
<td>Execute S1, and then S2.</td>
</tr>
</tbody>
</table>
Table 2.6: Concrete syntax of time primitives

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

and ClockReset are also statements. Finally, a Transition possibly includes a deadline, and its Trigger may have a ClockReset. Table 2.6 gives the concrete syntax.

This section has given a diagrammatic overview of the metamodel. A textual representation that specifies all the details is presented in Appendix A.
Well-formedness Conditions

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

Well formedness requires well typedness. Here, however, we do not focus on this aspect, except where this is not standard for an expression or statement. The type system of RoboChart is the type system of Z[12].

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. Finally, in Section ??, we provide a formalisation of all conditions using the Z notation of axiomatic descriptions [12].

3.1 Core Language

3.1.1 Robotic Platforms

- Robotic platforms cannot require interfaces.
- Defined interfaces can only have events
- 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

• Provided and required interfaces contain only variables and operations

• Defined interfaces contain only variables and events

• Names of variables, events and operations are unique.

3.1.3 Modules

• A module must contain exactly one robotic platform and at least one controller.

• All variables and operations required by the module’s controllers must be provided by the platform.

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.

• Connections of a module must associate only events of the robotic platform and its controllers.

• Connections involving a robotic platform are always asynchronous.

• Connections of a controller must associate only its events and those of its state machines.

• Only events of the same type may be connected.

3.1.5 Controllers

• A controller must contain at least one state machine.

• Controllers cannot provide variables or operations to other controllers.

• Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.
• All variables required by the controller’s state-machines must be provided or required by the controller.

• All operations required by the controller’s state-machines must be required or defined by the controller.

• Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.

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

• All operations required by the controller’s state-machines must be required or defined by the controller.

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

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

3.1.6 State Machines

• State machines cannot have provided interfaces

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

• Every state machine must have exactly one initial junction.

• State machines must contain at least one state.

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

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

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

• A state has at most one of each type of action: entry, during, and exit,
3.1.8 Initial junctions

- An initial junction does not have incoming transitions.
- An initial junction must have exactly one outgoing transition.
- All junction conditions apply.

3.1.9 Junction

- A junction must contain at least one outgoing transition.
- The guards of the transitions out of a junction must form a cover.
- Transitions starting in junctions cannot have triggers.

3.1.10 Final states

- Final states cannot be the source of transitions.

3.1.11 Transitions

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

3.1.12 Operations

- All state-machine conditions apply to operation definitions.

3.2 Timed Language

3.2.1 Timed Expressions

- Expressions involving since(C) and sinceEntry(S) are only permitted in transition guards.
- The clock C in an expression since(C) may only reference a clock declared within the expression’s containing state-machine.
• The state $S$ in an expression $\text{sinceEntry}(S)$ 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 expressions $\text{since}(C)$ or $\text{sinceEntry}(S)$ may only be compared with a constant expression, and only when using one of the following operators: $>$, $<$, $\geq$, $\leq$, $==$. A consequence of this restriction is that no expression can compare the value of two clocks as given by $\text{since}(C)$ or $\text{sinceEntry}(S)$.

3.2.2 Timed Statements

• 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.
Semantics

For the purpose of this semantics, the functions $vid$, $eventId$, $tid$ and $id$ 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 $Event$ contains an event $internal$, 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.
4.1 Detailed Semantics: Core Language

4.1.1 Modules

Rule 1. Semantics of modules
\[ [m : \text{Module}]_{\mathcal{M}} : \text{CSPProcess} = \]

\[
\Theta_{\text{end } \text{Skip}} \left\{ \begin{aligned}
\| \ c & : \text{asyncs} \bullet \text{buffer}^1(c) \\
\| \text{evasyncs} & \\
\| \text{modMemory}^1(m) & \\
\| \text{memoryChannels}^1(m) & \\
\| \text{composeControllers}^1(m, rp, c.trls, cons) & \\
\{ \text{evasyncs} \cup \text{memoryChannels}^2(m) \}
\end{aligned} \right. \\
\{ \text{end} \}
\]

where
\[ \begin{aligned}
\text{rp} &= m.\text{roboticPlatform} \\
\text{c.trls} &= m.\text{controllers} \\
\text{c.cons} &= m.\text{connections} \\
\text{asyncs} &= \{ c : \text{cons} \mid c.\text{async} \land \{c.\text{from}, c.\text{to}\} \not\subseteq \text{RP} \} \\
\text{evasyncs} &= \{ c : \text{asyncs} \bullet \text{eventId}(c.\text{eto}, c.\text{to}) \} \cup \\
& \quad \{ c : \text{asyncs} \bullet \text{eventId}(c.\text{efrom}, c.\text{from}) \}
\end{aligned} \]

Rule 2. Memory channels
\[
\text{memoryChannels}(m : \text{Module}) : \text{ChannelSet} =
\]

\[
\{ v : \text{allLocalVariables}^1(rp) \bullet \text{set} \_ \text{vid}(v) \} \cup \\
\{ c : \text{c.trls} \bullet \{ v : \text{requiredVariables}^1(c) \bullet \text{set} \_ \text{EXT} \_ \text{vid}(v) \} \}
\]
Rule 3. Function allVariables

\[ \text{allVariables}(c : \text{Context}) : \text{Set} (\text{Variable}) = \]

\[ \bigcup \{ l : c.\text{variableList} \bullet l.\text{vars} \} \cup \]
\[ \bigcup \{ i : c.\text{PInterfaces} \bullet \bigcup \{ l : i.\text{variableList} \bullet l.\text{vars} \} \} \cup \]
\[ \bigcup \{ i : c.\text{RInterfaces} \bullet \bigcup \{ l : i.\text{variableList} \bullet l.\text{vars} \} \} \]

Rule 4. Function requiredVariables

\[ \text{requiredVariables}(c : \text{Context}) : \text{Set} (\text{Variable}) = \]

\[ \bigcup \{ i : c.\text{RInterfaces} \bullet \bigcup \{ l : i.\text{variableList} \bullet l.\text{vars} \} \} \]

Rule 5. Function allLocalVariables

\[ \text{allLocalVariables}(c : \text{Context}) : \text{Set} (\text{Variable}) = \]

\[ \bigcup \{ l : c.\text{variableList} \bullet l.\text{vars} \} \cup \bigcup \{ i : c.\text{PInterfaces} \bullet \bigcup \{ l : i.\text{variableList} \bullet l.\text{vars} \} \} \]
The function \textit{initial} picks an initial value of the appropriate type for a variable. If the variable defines an initial value, this value is used.
Rule 7. Composition of controllers

\[
\text{composeControllers}(m : \text{Module}, rp : \text{RoboticPlatform},
\text{ctrls} : \text{Seq}(\text{Controller}), cons : \text{Set}(\text{Connection})) : \text{CSPProcess} =
\]

\[
\begin{align*}
\text{if } \#\text{ctrls} = 1 \\
\text{then} \\
\text{renamingController}^1(\text{head ctrls}, cons) \\
\text{else} \\
\text{renamingController}^2(\text{head ctrls}, cons) \\
\left\langle \text{connevts} \right\rangle \\
\text{composeControllers}^2(m, rp, \text{tail ctrls}, cons)
\end{align*}
\]

where

\[
\text{connevts} = \text{renCtrlEvts}^1(\text{head ctrls}, cons) \cap \\
\bigcup \{c : \text{tail ctrls} \cdot \text{renCtrlEvts}^2(c, cons)\}
\]

Rule 8. Renaming controller

\[
\text{renamingController}(c : \text{Controller}, cons : \text{Set}(\text{Connection})) : \text{CSPProcess} =
\]

\[
\left\lfloor \begin{array}{c}
\{x : \text{internalConns} \cup \text{fromPlatform} \cdot \text{eventId}(e.eto) \leftrightarrow \text{eventId}(e.efrom)\} \\
\cup \{x : \text{toPlatform} \cdot \text{eventId}(e.efrom) \leftrightarrow \text{eventId}(e.eto)\}
\end{array} \right\rfloor
\]

where

\[
\begin{align*}
\text{internalConns} &= \{x : \text{cons} \cdot \{x.\text{from}, x.\text{to}\} \subseteq \text{Controller} \land \neg x.\text{async} \land c \in \{x.\text{from}, x.\text{to}\}\} \\
\text{toPlatform} &= \{x : \text{cons} \cdot x.\text{from} = c \land x.\text{to} \in \text{RoboticPlatform}\} \\
\text{fromPlatform} &= \{x : \text{cons} \cdot x.\text{to} = c \land x.\text{from} \in \text{RoboticPlatform}\}
\end{align*}
\]
Rule 9. Renaming controller events
renCtrlEvts(c : Controller, cons : Set(Connection)) : ChannelSet =

\{x : internalConns \cdot eventId(e.efrom)\} \cup \{end\}
where
internalConns = \{x : cons \cdot \{x.from, x.to\} \subseteq Controller \land \neg x.async \land c \in \{x.from, x.to\}\}

Rule 10. Buffer
buffer(c : Connection) : CSPProcess =

if c.efrom.type \neq null
then
let Buffer(\langle \rangle) \triangleq eventId(c.efrom) ? x \rightarrow Buffer(\langle x \rangle)
Buffer(\langle v \rangle) \triangleq eventId(c.efrom) ? x \rightarrow Buffer(\langle x \rangle) \parallel eventId(c.eto)!v \rightarrow Buffer(\langle \rangle)
within Buffer(\langle \rangle)
else
let Buffer(false) \triangleq eventId(c.efrom) \rightarrow Buffer(true)
Buffer(true) \triangleq eventId(c.efrom) \rightarrow Buffer(true) \parallel eventId(c.eto) \rightarrow Buffer(false)
within Buffer(false)
4.1.2 Controllers

Rule 11. Semantics of controllers
\[ [c : \text{ControllerDef}] C : \text{CSProcess} = \]

\[
\left( \left( \begin{array}{c}
\text{composeMachines}^1(c, ms, cs) \\
[lvars \cup rvars]
\end{array} \right) \right) \setminus \left( \begin{array}{c}
\text{ctrlMemory}^1(c) \\
[\text{hvars} \cup \text{rvars}]
\end{array} \right) \Theta_{\text{end}} \text{Skip}
\]

where
- \( ms = c.\text{machines} \)
- \( cs = c.\text{connections} \)
- \( lvars = \{ v : \text{allLocalVariables}^3(c) \cdot \text{set}_\text{vid}(v, c) \} \)
- \( rvars = \{ v : \text{requiredVariables}^3(c) \cdot \text{set}_\text{Ext}_\text{vid}(v, c) \} \)
- \( \text{hvars} = \{ v : \text{allLocalVariables}^4(c) \cdot \text{set}_\text{vid}(v, c) \} \cup \{ v : \text{allLocalVariables}^5(c) \cdot \text{get}_\text{vid}(v, c) \} \)
Rule 12. Controller Memory

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

\[
\begin{align*}
\text{let } & \text{Memory}(\text{vars}) \triangleq \\
& \begin{cases}
\Box v : \text{lvars} \bullet \text{set_vid}(v, c)?x \rightarrow \\
& \langle m : \text{rmachines}(v) \bullet \text{set_Ext_vid}(v, m)!x \rightarrow \text{Skip} ; \\
& \text{Memory}(\text{vars}[\text{name}(v) := x]) \rangle \\
\text{let } & \text{Memory}(\text{varvalues}) \\
\text{where } \\
& \text{ms} = c.\text{machines} \\
& \text{lvars} = \text{allLocalVariables}^6(c) \\
& \text{rvars} = \text{requiredVariables}^4(c) \\
& \text{vars} = \langle v : \text{rvars} \cup \text{lvars} \bullet \text{name}(v) \rangle \\
& \text{varvalues} = \langle v : \text{rvars} \cup \text{lvars} \bullet \text{initial}(v) \rangle \\
& \text{rmachines} = \lambda v \bullet \{ m : \text{ms} | v \in \text{requiredVariables}^5(m) \} \\
\end{cases}
\end{align*}
\]
Rule 13. Composition of machines

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

\[
\begin{align*}
    \text{if } & \#ms = 1 \\
    \text{then} & \quad \text{renamingMachine}^1(\text{head ms, cons}) \\
    \text{else} & \quad \text{renamingMachine}^2(\text{head ms, cons}) \\
    & \quad \|\text{connevts}\| \\
    & \quad \text{composeMachines}^2(c, \text{tail ms, cons}) \\
\end{align*}
\]

where

\[
\text{connevts} = \text{renStmEvts}^1(\text{head ms, cons, c}) \cap U\{m : \text{tail ms} \cdot \text{renStmEvts}^2(m, \text{cons, c})\}
\]

Rule 14. Renaming state machine

\[ \text{renamingMachine}(m : \text{StateMachine}, cons : \text{Set}(\text{Connection})) : \text{CSPProcess} = \]

\[
\begin{align*}
    [m]_{\text{STM}} \left[ \{x : \text{internalConns} \cup \text{fromController} \circ \text{eventId}^{(\text{eto})} \leftarrow \text{eventId}^{(\text{efrom})}\} \cup \{x : \text{toController} \circ \text{eventId}^{(\text{efrom})} \leftarrow \text{eventId}^{(\text{eto})}\} \right] \\
\end{align*}
\]

where

\[
\text{internalConns} = \{x : \text{cons} \circ \{x.\text{from}, x.\text{to}\} \subseteq \text{StateMachine} \land m \in \{x.\text{from}, x.\text{to}\}\}
\]

\[
\text{toController} = \{x : \text{cons} \circ x.\text{from} = m \land x.\text{to} \in \text{Controller}\}
\]

\[
\text{fromController} = \{x : \text{cons} \circ x.\text{to} = m \land x.\text{from} \in \text{Controller}\}
\]
Rule 15. Renaming machine events

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

\[\{x : \text{internalConns} \cdot \text{eventId}(e.\text{from})\} \cup \{\text{end}\}\]

where

\[\text{internalConns} = \{x : \text{cons} \cdot \{x.\text{from}, x.\text{to}\} \subseteq \text{StateMachine} \land m \in \{x.\text{from}, x.\text{to}\}\}\]

4.1.3 State machines

Rule 16. Semantics of state machine

\[\text{[stm} : \text{StateMachineDef}]_{\text{STMA}} : \text{CSPProcess} =\]

\[
\left(\begin{array}{l}
\begin{array}{l}
\text{initialisation}^1(\text{stm}) \\ \text{flowevts}
\end{array} \\
\begin{array}{l}
\text{composeStates}^1(x : \text{stm.nodes} | x \in \text{State}, \text{stm}) \\
\{\text{enter}, \text{entered}, \text{exit}, \text{exited}\}
\end{array} \\
\begin{array}{l}
\text{getsetChannels}^1(\text{stm}) \cup \text{trigEvents}^1(\text{stm})
\end{array} \\
\text{stmMemory}^1(\text{stm})
\end{array} \right) \\
\text{getsetLocalChannels}^1(\text{stm}) \cup \{\text{internal}\}
\]

where

\[\text{flowevts} = \]

\[\bigcup\{x : \text{SIDS} \setminus \text{substates}^1(\text{stm}); y : \text{substates}^2(\text{stm}) \cdot \{\text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y\}\}\]

Rule 17. Function substates

\[\text{substates}(n : \text{NodeContainer}) : \text{Set} (\text{State}) =\]

\[x.\text{nodes} \cap (\text{State} \cup \text{Final})\]
Rule 18. Initialisation

initialisation(n : NodeContainer) : CSPProcess =

\[
\text{if } (#\text{n.nodes} > 0) \text{then}
\]
\[
[\text{it} : \text{Transition} \mid \text{t.source} = (ix : \text{n.nodes} \mid x \in \text{Initial}), n, \text{true}]^{\text{Skip, Skip}}
\]
\[
\text{else } \text{Skip}
\]

Rule 19. Get and Set channels

gsetChannels(s : StateMachineDef) : ChannelSet =

\[
\{v : \text{allVariables}^1(s) \cdot \text{get.vid}(v)\} \cup \{v : \text{allVariables}^2(s) \cdot \text{set.vid}(v)\}
\]

Rule 20. Composition of states

composeStates(ss : seq State, p : NodeContainer) : CSPProcess =

\[
\text{if } \#\text{ss} = 1 \\
\quad \text{then}
\]
\[
\text{restrictedState}^1(p, \text{head ss})
\]
\[
\text{else}
\]
\[
\left(\text{restrictedState}^2(p, \text{head ss}) \parallel \text{showevts}\right) \setminus \text{shflowevts}
\]

where

\[
\text{shflowevts} = \text{flowEvents}^1(\text{head ss}, p) \cap \bigcup \{x : \text{tail ss} \cdot \text{flowEvents}^2(x, p)\}
\]
Rule 21. 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 22. Get and set local channels
\[
\text{getsetLocalChannels}(s: \text{StateMachineDef}) : \text{ChannelSet} = \\
\{ v : \text{allLocalVariables}^7(s) \cdot \text{get}_{\text{vid}}(v) \} \cup \{ v : \text{allLocalVariables}^8(s) \cdot \text{set}_{\text{vid}}(v) \} \\
\cup \{ v : \text{requiredVariables}^6(s) \cdot \text{get}_{\text{vid}}(v) \}
\]

Rule 23. Flow events
\[
\text{flowEvents}(s: \text{State}, p: \text{NodeContainer}) : \text{ChannelSet} = \\
\bigcup \{ x : \text{substates}^3(p) ; y : \{ \text{id}(s) \} \cdot \{} \\
\quad \text{enter}.y.x, \text{entered}.y.x, \text{exit}.y.x, \text{exited}.y.x, \\
\quad \text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y, \\
\quad \} \}
\]

Rule 24. Semantics of states
\[
[s: \text{State}]_S : \text{CSPProcess} = \\
\text{This function is split in multiple rules according to the type of states.}
\]
Rule 25. Semantics of simple states
\[ s : \text{State} \] \[ S : \text{CSPProcess} = \]

\[
\text{let } \\
\text{Inactive} \triangleq \text{enter}^? x : \text{sids}! \text{id}(s) \rightarrow \text{Activating}(x) \\
\text{Activating}(o) \triangleq \left[ s.\text{entry} \right] : \text{initialisation}^2(s); \text{entered}! \text{id}(s) \rightarrow \\
\left( \left[ s.\text{during} \right] : \text{Stop} \right) \triangle \\
\begin{cases} \\
\text{t} : \text{transitionsFrom}^1(s) \cdot \left[ t, s, \text{false} \right]^\text{Inactive,Activating}_{T^2} \\
\text{e} : \text{Event} \cdot \text{if (e.type == null)} \\
\text{then eventId}(e)^? x : \text{tids} \rightarrow \text{exit}; \text{Inactive} \\
\text{else eventId}(e)^? x : \text{tids}? y \rightarrow \text{exit}; \text{Inactive} \\
\end{cases}
\]

within \text{Inactive} \\

where \#

\text{substates}^4(s) = 0 \\
\text{flowtrigevts} = \text{flowTriggerEvents}^1(s) \\
\text{sids} = SIDS \setminus \{ \text{id}(s) \} \\
\text{exit} = \text{exit}^? x : \text{sids}! \text{id}(s) \rightarrow \text{exitSubstates}^1(s) ; \left[ s.\text{exit} \right] : \text{exited}! \text{id}(s) \rightarrow \text{Skip} \\
\text{tids} = TIDS \setminus \text{tIDS}(s)
Rule 26. Semantics of composite states

\[ s : \text{State} \]
\[ S : \text{CSPProcess} \]

let

\[ \text{Inactive} \triangleq \text{Enter}^{2} \circ \text{sid} \circ \text{id} \rightarrow \text{Activating}^{2} \circ \text{x} \]
\[ \text{Activating}^{2} \circ \text{o} \rightarrow \text{initialisation}^{2} \circ \text{sid} \circ \text{id} \rightarrow \text{Stop} \]△

\(\begin{align*}
\square t & \circ \text{transitionsFrom}^{2} \circ \text{sid} \circ \text{id} \rightarrow \text{_inactive}^{2} \circ \text{Activating}^{2} \\
\square e \circ \text{Event} & \circ \text{if} (\text{e.type} == \text{null}) \rightarrow \text{eventId} \circ \text{sid} \circ \text{id} \rightarrow \text{inactive}^{2} \\
& \text{else} \text{eventId} \circ \text{sid} \circ \text{id} \rightarrow \text{inactive}^{2} \\
\end{align*}\)

within

\( (\text{inactive} \| \text{flowTriggerEvents} \| \text{composeStates}^{2} \circ \text{substates}^{2} \circ \text{sid} \circ \text{id}) \setminus \text{flowTriggerEvents} \)

where

\#\text{substates}^{2} > 0

\text{flowTriggerEvents} = \text{flowTriggerEvents}^{2} \circ \text{sid} \circ \text{id}

\text{sid} = \text{SID} \setminus \{ \text{id} \circ \text{sid} \}

\text{exit} = \text{exit} \circ \text{sid} \circ \text{id} \rightarrow \text{exitSubstates}^{2} \circ \text{sid} \circ \text{id} \rightarrow \text{exit} \circ \text{sid} \circ \text{id} \rightarrow \text{Skip}

\text{tid} = \text{TID} \setminus \text{tID} \circ \text{sid} \circ \text{id}

Rule 27. Semantics of final states

\[ s : \text{Final} \]
\[ S : \text{CSPProcess} \]

\[ \text{Enter} \circ \text{sid} \circ \text{id} \rightarrow \text{entered} \circ \text{sid} \circ \text{id} \rightarrow \text{end} \rightarrow \text{Skip} \]

\[ \text{else Stop} \]
Rule 28. Synchronisation events between parent state and substates

\[
\text{flowTriggerEvents}(s : \text{State}) : \text{ChannelSet} = \\
\{ \{ e : \text{Event}; t : TIDS \cdot e.t \} \setminus \text{substatesTriggers}(s) \} \cup \\
\{ \{ x : \text{SID}s \setminus \text{substates}(s) \cdot \text{enter}x.y, \text{entered}x.y, \text{exit}x.y, \text{exited}x.y \} \}
\]

Rule 29. Triggers of substates

\[
\text{substatesTriggers}(s : \text{State}) : \text{ChannelSet} = \\
\{ t : \text{allTransitions}(s) \cdot \text{triggerEvent}(t.\text{trigger}, \text{id}(t)) \}
\]

Rule 30. Restricted semantics of states

\[
\text{restrictedState}(p : \text{NodeContainer}, s : \text{State}) : \text{CSPProcess} = \\
[ [ s ] \cdot \{ \text{all_other_transitions}_S \setminus \text{all_transitions}_PS \} ] \text{Skip} \\
\text{where} \\
tidsfromwithin = \{ t : \text{transitionsFrom}(s) \cup \text{allTransitions}(s) \cdot \text{id}(t) \} \\
\text{all_other_transitions}_S = \{ e : \text{Event}; \text{tid} : \text{TIDS} \setminus \text{tidsfromwithin} \cdot \text{eventId}(e).\text{tid} \} \\
\text{all_transitions}_PS = \{ e : \text{Event}; \text{tid} : \text{TIDS} \cdot \text{eventId}(e).\text{tid} \} \\
\setminus \{ t : \text{allTransitions}(p) \cdot \text{eventId}(t.\text{trigger.event}).\text{id}(t) \} 
\]
Rule 31. Semantics of transitions

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

\[
\begin{align*}
\text{if src} & \in \text{State} & \left[ t.\text{trigger} \right]_{\text{Trigger}}^{t(t)} & ; \left[ \text{exit}!\text{id}(\text{src})!\text{id}(\text{src}) \rightarrow \text{exitSubstates}^3(\text{src}) ; \left[ \text{src.exit} \right] \right]_{\text{Action}}^{\text{src.exit}} ; \\
\text{else if src} & \in \text{Initial} & \left[ t.\text{trigger} \right]_{\text{Trigger}}^{t(t)} & ; \left[ t.\text{action} \right]_{\text{Action}}^{t(t)} ; \text{compileTarget}^1(\text{tgt}, \text{src}, \text{false}, P, Q) \\
\text{else if src} & \in \text{Junction} & \left[ t.\text{trigger} \right]_{\text{Trigger}}^{t(t)} & ; \left[ t.\text{action} \right]_{\text{Action}}^{t(t)} ; \text{compileTarget}^3(\text{tgt}, \text{origin}, \text{initial}, P, Q) \\
\text{where} & & \text{src} = t.\text{source} & \\
& & \text{tgt} = t.\text{target} & \\
\end{align*}
\]

Rule 32. Compile target

\[
\begin{align*}
\text{compileTarget}(\text{tgt} : \text{Node}, \text{o} : \text{NodeContainer}, \text{i} : \text{boolean}, P : \text{CSPProcess}, Q : \text{CSPProcess}) : \\
\text{CSPProcess} = \\
\text{if (tgt} \in \text{State}) & \text{then} \\
\text{if (tgt} = \text{o}) & \text{then enter}!\text{id}(\text{o})!\text{id}(\text{tgt}) \rightarrow Q \\
\text{else enter}!\text{id}(\text{o})!\text{id}(\text{tgt}) & \rightarrow \text{entered}!\text{id}(\text{o})!\text{id}(\text{tgt}) \rightarrow \left( \text{if } (\text{i}) \text{ then Skip } \right) \\
\text{else if (tgt} \in \text{Junction}) & \text{then} \\
\square & \{ t : \text{transitionsFrom}^4(\text{tgt}) \bullet [t, o, i]_{T}^{P,Q} \}
\end{align*}
\]
Rule 33. Exit substates

\[
\text{exitSubstates}(s : \text{NodeContainer}) : \text{CSPProcess} = \\
\text{exit}!id(s)?z : \{x : \text{substates}\{s\} \cdot id(x)\} \rightarrow \text{exited}!id(s)!z \rightarrow \text{Skip}
\]

Rule 34. State-machine Memory

\[
\text{stmMemory}(s : \text{StateMachine}) : \text{CSPProcess} = \\
\text{let } Memory(v) \Rightarrow \\
\begin{array}{l}
\quad \forall v : lvars \cdot \left( get\_vid(v,s)!name(v) \rightarrow Memory(v) \right) \\
\quad \forall v : rvars \cdot \left( \left( \left( set\_vid(v,s)?x \rightarrow Memory(v[\text{name}(v) := x]) \right) \right) \right) \\
\end{array}
\]

\[
\text{within } Memory(\text{varvalues}) \\
\text{where } \\
\begin{array}{l}
\quad rvars = \text{requiredVariables}(s) \\
\quad lvars = \text{allLocalVariables}(s) \\
\quad vars = \langle v : rvars \cup lvars \cdot \text{name}(v) \rangle \\
\quad \text{varvalues} = \langle v : rvars \cup lvars \cdot \text{initial}(v) \rangle
\end{array}
\]
Rule 35. Semantics of triggers
\[
[t : \text{Trigger}, \text{tid} : \text{CSPProcess}] =
\]
\[
\begin{align*}
\text{if } t_{\text{event}.\text{type}} \neq \text{null} & \quad \text{id}(t_{\text{event}}) \text{?}x\quad \rightarrow \text{set}_\text{vid}(t_{\text{parameter}})!x\quad \rightarrow \text{Skip} \\
\text{else} & \quad \text{id}(t_{\text{event}}) \text{.tid} \rightarrow \text{Skip}
\end{align*}
\]

Rule 36. Event for transition trigger
\[
\text{triggerEvent}(t : \text{Trigger}, \text{tid} : \text{TIDS}) : \text{CSPEvent} =
\]
\[
\text{id}(t_{\text{event}}) \text{.tid}
\]

Rule 37. Memory transitions
\[
\text{memoryTransition}(t : \text{Transition}, P : \text{CSPProcess}) : \text{CSPProcess} =
\]
\[
\begin{align*}
\text{if } (t_{\text{condition}} \neq \text{null}) & \quad \text{then} \\
\left(\left[\text{t.condition}\right]^{\text{Expr}^\prime}\right) \& \left[\text{t.trigger}\right]^{\text{id}(t) \text{.CSPProcess}^\text{Trigger}} & \quad P \\
\text{else} & \quad \left[\text{t.trigger}\right]^{\text{id}(t) \text{.CSPProcess}^\text{Trigger}} & \quad P
\end{align*}
\]

Rule 38. Function transitionsFrom
\[
\text{transitionsFrom}(s : \text{State}) : \text{Set} (\text{Transition}) =
\]
\[
\{ t : \text{parent}(s).\text{transitions} \mid t_{\text{source}} = s \circ t \}
Rule 39. Function allTransitions

\[
\text{allTransitions}(s : \text{State}) : \text{Set(Transition)} = \\
\text{s.transitions} \cup \bigcup \{x : \text{s.nodes} \mid s \in \text{State} \cdot \text{allTransitions}^6(x)\}
\]

Rule 40. Function allSTMTransitions

\[
\text{allSTMTransitions}(s : \text{StateMachineDef}) : \text{Set(Transition)} = \\
\text{s.transitions} \cup \bigcup \{x : \text{s.nodes} \mid s \in \text{State} \cdot \text{allTransitions}^7(x)\}
\]

4.1.4 Statements

Rule 41. Semantics of statements

\[
[s : \text{Statement}] \text{Statement} : \text{CSPProcess} = \\
\text{This rule is split in multiple rules according to the subtype of the statement.}
\]

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 get\_x\_i read values from the memory and the process P models the actual statement. The input events get\_x\_i?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 42. Read state of an expression

\[
\text{readState}(\text{vs} : \text{seq(Variable)}, P : \text{CSPProcess}) : \text{CSPProcess} = \\
\begin{align*}
&\text{if} (\#\text{vs} = 0) \text{ then} \\
&P \\
&\text{else} \\
&\text{get}_{\text{vid}}(\text{head vs})?((\text{head vs}).\text{name} \rightarrow \text{readState}^1(\text{tail vs}, P))
\end{align*}
\]

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

Rule 43. Semantics of statements in context

\[
[\text{s} : \text{Statement}]_{\text{StatementInContext}} : \text{CSPProcess} = \\
\begin{align*}
&\text{readState}^2(\text{usedVariables}^1(\text{s}), [\text{s}]^\_\text{Statement}^1)
\end{align*}
\]
Rule 44. Function usedVariables

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

\[
\text{if } s \in \text{Assignment} \text{ then } \\
\text{usedV}(s.\text{right}) \\
\text{else if } s \in \text{Call} \text{ then } \\
\bigcup \{ x : s.\text{args} \bullet \text{usedV}(x) \} \\
\text{else if } s \in \text{IfStatement} \text{ then } \\
\text{usedV}(s.\text{expression}) \\
\text{else if } s \in \text{SendEvent} \land s.\text{trigger.type} \in \{ \text{SYNC, OUTPUT} \} \text{ then } \\
\text{usedV}(s.\text{trigger.value}) \\
\text{else} \\
\{\}
\]

Rule 45. Semantics of assignment

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

\[
\text{set_vid}(s.\text{left})!s.\text{right} \xrightarrow{\text{Expr}} \text{Skip}
\]
Rule 46. Semantics of call statement
\[
[s : \text{Call}]_{\text{Statement}} : \text{CSPProcess} = \]
\[
op_{\text{name}} \text{Call} \rightarrow \text{body}; \quad \nop_{\text{name}} \text{Ret} \rightarrow \text{Skip}
\]
where
\[
op = s_{\text{operation}}
\]
\[
op_{\text{def}} = \text{findOperationDefinition}(\op)
\]
\[
\text{body} = \begin{cases}
\begin{aligned}
\text{if } (\op_{\text{def}} = \text{null}) & \text{ then } \\
(\text{Skip } \sqcap \text{ Stop}) & \\
\text{else } & \\
[\op_{\text{def}}] \quad \{ \{ x : s_{\text{args}} \cdot [x] \} \} & \\
\end{aligned}
\end{cases}
\]

Rule 47. Semantics of if statements
\[
[s : \text{IfStatement}]_{\text{Statement}} : \text{CSPProcess} = \]
\[
\begin{cases}
\begin{aligned}
\text{if } [s_{\text{expression}}]_{\text{Expr}^3} & \text{ then } [s_{\text{then}}]_{\text{StatementInContext}^1} \\
\text{else if } (s_{\text{else}} \neq \text{null}) & \text{ then } [s_{\text{else}}]_{\text{StatementInContext}^2} \quad \text{else } \text{Skip}
\end{aligned}
\end{cases}
\]
Rule 48. Semantics of send event statements

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

\[
\begin{align*}
\text{if} (\text{type} = \text{INPUT}) & \text{ then} & \\
\hspace{1em} \text{eventId}(\text{event})?\text{par.name} & \rightarrow \text{set.vid} (\text{par})!\text{par.name} & \rightarrow \text{Skip} & \\
\text{if} (\text{type} = \text{OUTPUT}) & \text{ then} & \\
\hspace{1em} \text{eventId}(\text{event})!\text{value} & \rightarrow \text{Skip} & \\
\text{if} (\text{type} = \text{SIMPLE}) & \text{ then} & \\
\hspace{1em} \text{eventId}(\text{event}) & \rightarrow \text{Skip} & \\
\text{else} & & \\
\hspace{1em} \text{eventId}(\text{event}).\text{value} & \rightarrow \text{Skip} & \\
\end{align*}
\]

\text{where}

\begin{align*}
\text{type} & = \text{s.trigger.type} \\
\text{event} & = \text{s.trigger.event} \\
\text{value} & = \text{s.trigger.value} \\
\text{par} & = \text{s.trigger.parameter}
\end{align*}

Rule 49. Semantics of sequential composition

\[ s : \text{SeqStatement} \rightarrow \text{Statement} : \text{CSPProcess} = \]

\[
\{ x : \text{s.statements} \bullet [x] \} \rightarrow \text{StatementInContext}^2
\]

Rule 50. Semantics of skip

\[ s : \text{Skip} \rightarrow \text{Statement} : \text{CSPProcess} = \]

\text{Skip}
Rule 51. Semantics of actions

\[ a : \text{Action} \, \text{Action} : \text{CSPProcess} = \]

\[ [a.\text{action}] \text{StatementInContext}^4 \]

4.1.5 Expressions

Rule 52. Semantics of expressions

\[ s : \text{Expression} \, \text{Expr} : \text{CSPExpression} = \]

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

Rule 53. Semantics of and expression

\[ s : \text{And} \, \text{Expr} : \text{CSPExpression} = \]

\[ [s.\text{left}] \, \text{Expr}^7 \wedge [s.\text{right}] \, \text{Expr}^8 \]
Rule 54. Semantics of array expression
\[ s : \text{ArrayExp} \]_{Expr} : \text{CSPExpression} =
\[
\text{if } (s.\text{value} = \text{TRUE}) \text{ then } \text{true} \text{ else } \text{false}
\]
Rule 56. Semantics of call expression

\[ [s : \text{CallExp}]_{\text{Expr}} : \text{CSPExpression} = \]

\[
\begin{align*}
\text{if } (\text{name} = \text{'size'} \land (\text{head.s.\text{args has type SetType}))) \text{then} \\
\quad \text{card}([\text{head.s.\text{args}]}_{\text{Expr}^{14}}) \\
\text{else if } (\text{name} = \text{'size'} \land (\text{head.s.\text{args has type SeqType}))) \text{then} \\
\quad \text{length}([\text{head.s.\text{args}]}_{\text{Expr}^{12}}) \\
\text{else} \\
\quad \text{name}([\{a : \text{s.\text{args} • a} \}_a]_{\text{Expr}^{13}})
\end{align*}
\]

where
\[
\text{name} = \text{s.function.name}
\]

Rule 57. Semantics of concatenation expression

\[ [s : \text{Cat}]_{\text{Expr}} : \text{CSPExpression} = \]

\[
\quad [\text{s.left}]_{\text{Expr}^{14}} \land [\text{s.right}]_{\text{Expr}^{15}}
\]

Rule 58. Semantics of not equal expression

\[ [s : \text{Different}]_{\text{Expr}} : \text{CSPExpression} = \]

\[
\quad [\text{s.left}]_{\text{Expr}^{16}} \neq [\text{s.right}]_{\text{Expr}^{17}}
\]

52
Rule 59. Semantics of division

\[
[s : \text{Div}]_{E_{xpr} : \text{CSPExpression}} = [s.\text{left}]_{E_{xpr}^{18}} / [s.\text{right}]_{E_{xpr}^{19}}
\]

Rule 60. Semantics of equality

\[
[s : \text{Equals}]_{E_{xpr} : \text{CSPExpression}} = [s.\text{left}]_{E_{xpr}^{20}} = [s.\text{right}]_{E_{xpr}^{21}}
\]

Rule 61. Semantics of greater or equal expression

\[
[s : \text{GreaterOrEqual}]_{E_{xpr} : \text{CSPExpression}} = [s.\text{left}]_{E_{xpr}^{22}} \geq [s.\text{right}]_{E_{xpr}^{23}}
\]

Rule 62. Semantics of greater than

\[
[s : \text{GreaterThan}]_{E_{xpr} : \text{CSPExpression}} = [s.\text{left}]_{E_{xpr}^{24}} > [s.\text{right}]_{E_{xpr}^{25}}
\]
Rule 63. Semantics of if and only if expression

\[
[s : \text{Iff}]_{Expr} : \text{CSPExpression} = \\
\begin{align*}
[s.\text{left}]_{Expr^{26}} & \Leftrightarrow [s.\text{right}]_{Expr^{27}}
\end{align*}
\]

Rule 64. Semantics of implication

\[
[s : \text{Implies}]_{Expr} : \text{CSPExpression} = \\
\begin{align*}
[s.\text{left}]_{Expr^{28}} & \Rightarrow [s.\text{right}]_{Expr^{29}}
\end{align*}
\]

Rule 65. Semantics of integer expression

\[
[s : \text{IntegerExp}]_{Expr} : \text{CSPExpression} = \\
\begin{align*}
\text{s.value}
\end{align*}
\]

Rule 66. Semantics of less or equal expression

\[
[s : \text{LessOrEqual}]_{Expr} : \text{CSPExpression} = \\
\begin{align*}
[s.\text{left}]_{Expr^{30}} & \leq [s.\text{right}]_{Expr^{31}}
\end{align*}
\]
Rule 67. Semantics of less than
\[ [s : \text{LessThan}]_{\text{Expr}} : \text{CSPExpression} = \]
\[ [\text{Expr}^{32}]_{\text{Expr}} < [\text{Expr}^{33}]_{\text{Expr}} \]

Rule 68. Semantics of minus
\[ [s : \text{Minus}]_{\text{Expr}} : \text{CSPExpression} = \]
\[ [\text{Expr}^{34}]_{\text{Expr}} - [\text{Expr}^{35}]_{\text{Expr}} \]

Rule 69. Semantics of modulus
\[ [s : \text{Modulus}]_{\text{Expr}} : \text{CSPExpression} = \]
\[ [\text{Expr}^{36}]_{\text{Expr}} \mod [\text{Expr}^{37}]_{\text{Expr}} \]

Rule 70. Semantics of multiplication
\[ [s : \text{Mult}]_{\text{Expr}} : \text{CSPExpression} = \]
\[ [\text{Expr}^{38}]_{\text{Expr}} \times [\text{Expr}^{39}]_{\text{Expr}} \]
Rule 71. Semantics of arithmetic negation
\[ [s : \text{Neg}] \_E_{\text{Exp}} : \text{CSPExpression} = \]
\[ -[s.\text{exp}] \]

Rule 72. Semantics of logical negation
\[ [s : \text{Not}] \_E_{\text{Exp}} : \text{CSPExpression} = \]
\[ \neg [s.\text{exp}] \]

Rule 73. Semantics of or expression
\[ [s : \text{Or}] \_E_{\text{Exp}} : \text{CSPExpression} = \]
\[ [s.\text{left}] \lor [s.\text{right}] \]

Rule 74. Semantics of parenthesised expression
\[ [s : \text{ParExp}] \_E_{\text{Exp}} : \text{CSPExpression} = \]
\[ ([s.\text{exp}]) \]

where
Rule 75. Semantics of plus
\[ [s : \text{Plus}]_{\text{Expr}} : \text{CSPExpression} = \]
\[
[s.\text{left}]_{\text{Expr}^{45}} + [s.\text{right}]_{\text{Expr}^{46}}
\]

Rule 76. Semantics of range expression
\[ [s : \text{RangeExp}]_{\text{Expr}} : \text{CSPExpression} = \]
\[
\{ x : \mathbb{N} \mid [s.\text{lrangle}]_{\text{Expr}^{47}} \text{ rel1 } x \text{ rel2 } [s.\text{rrange}]_{\text{Expr}^{48}} \}
\]
where
\[
\text{rel1} = \text{if } (e.\text{interval} = /) \text{ then } \leq \text{ else } < \\
\text{rel2} = \text{if } (e.\text{interval} = /) \text{ then } \geq \text{ else } >
\]

Rule 77. Semantics of sequence expression
\[ [s : \text{SeqExp}]_{\text{Expr}} : \text{CSPExpression} = \]
\[
\langle \{ x : s.\text{values} \bullet [x] \}_{\text{Expr}^{49}} \rangle
\]

Rule 78. Semantics of set expression
\[ [s : \text{SetExp}]_{\text{Expr}} : \text{CSPExpression} = \]
\[
\{ \{ x : s.\text{values} \bullet [x] \}_{\text{Expr}^{50}} \}
\]
Rule 79. Semantics of tuple expression

\[
\text{Expr} : \text{CSPExpression} = (\{ x : s.values \cdot [x]_\text{Expr} \})
\]

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 \textit{Wait} and deadlines on actions. Clocks are not modelled explicitly; instead for each transition whose trigger is guarded by an expression using \textit{since(C)} or \textit{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 \textit{stmClocks(stm, wcs)}, which relies on the calculation of \textit{wcs}, a partial function form transitions to pairs, where the first component is the guard with occurrences of \textit{since(C)} and \textit{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 \textit{stmMemory(stm, wcs)} also takes \textit{wcs} as a parameter. Finally, compared with the untimed semantics of a state machine, the hiding on \textit{entered} events is moved to the outer composition of the memory and the states as \textit{sinceEntry(S)} conditions require the clocks to observe \textit{entered} events.
Rule 80. Semantics of state machine

\[ stm : \text{StateMachineDef} \mid \text{STM} = \]
\[
\begin{array}{l}
\left( \text{initialisation}^4(\text{stm}) \right) \\
\{ \text{flowevts} \} \\
\text{composeStates}^4(\langle x : \text{stm.nodes} \mid x \in \text{State} \rangle, \text{stm}) \\
\setminus \langle \text{enter, exit, exited} \rangle \\
\{ \text{getsetChannels}^2(\text{stm}) \cup \text{trigEvents}^2(\text{stm}) \cup \text{clockResets}^1(\text{wcs}) \cup \text{deadlineEvents}^2(\text{stm}) \} \\
\left( \text{stmMemory}^1(\text{stm}, \text{wcs}) \parallel \text{clockMemSync} \parallel \text{stmClocks}^1(\text{stm}) \right) \\
\setminus (\text{clockMemSync} \cup \text{trigEvents}^3(\text{stm})) \\
\{ \text{getsetLocalChannels}^2(\text{stm}) \cup \text{clockResets}^2(\text{stm}) \cup \text{deadlineEvents}^2(\text{stm}) \cup \{ \text{internal\_ entered} \} \} \\
\Theta(\text{end\_ Skip}) \\
\end{array}
\]

where

\[
\text{wcs} = \{ t : \text{allSTMTransitions}^1(\text{stm}) \mid \text{t.condition} \neq \text{null} \land \text{t} \mapsto \text{wc}(\text{t.condition}) \}
\]

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

\[
\text{flowevts} = \\
\bigcup \{ x : \text{SIDS} \setminus \text{substates}^9(\text{stm}) ; y : \text{substates}^{10}(\text{stm}) \cup \{ \text{enter}\_x.\_y, \text{entered}\_x.\_y, \text{exit}\_x.\_y, \text{exited}\_x.\_y \} \}
\]

Clocks

Functions related to clocks are formalised in this section.

Rule 81. allClockVariables function

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

\[
\text{allClockVariables}(\text{wcs}) = \\
\{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom wcs} \land (e \mapsto v) \in \pi_2(\text{wcs}(t)) \land v \}
\]
Rule 82. clockResets function

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

\[
\text{clockResets}(\text{stm}) = \bigcup \left\{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom} \text{wcs} \land (e \mapsto v) \in \pi_2(\text{wcs}(t)) \land \alpha\text{ClockReset}^1(e) \right\}
\]

Rule 83. stmClocks function

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

\[
\| (t, e, v) : \{ t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom} \text{wcs} \land (e \mapsto v) \in \pi_2(\text{wcs}(t)) \}, \| \alpha\text{WC}(t, e, v) \| \text{compileWC}(t, e, v)
\]

where

\[
\alpha\text{WC}(t, e, v) = \{ \text{triggerEvent}^2(t), \text{setWC}_{\text{vid}}(v) \} \cup \alpha\text{ClockReset}^2(e)
\]

Rule 84. alphaClockReset function

\[
\text{alphaClockReset}(e : \text{Expression}) : \text{ChannelSet} =
\]

This rule is defined by multiple rules according to the subtype of the expression:

\(85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97\).

Rule 85. alphaClockReset function

\[
\text{alphaClockReset}(e : \text{ParExp}) : \text{ChannelSet} =
\]

\[
\text{alphaClockReset}(e) = \alpha\text{ClockReset}^3(e.\text{exp})
\]
Rule 86. alphaClockReset function
alphaClockReset(e : Not) : ChannelSet =

\[\text{alphaClockReset}(e) = \text{alphaClockReset}^4(e.\exp)\]

Rule 87. alphaClockReset function
alphaClockReset(e : CallExp) : ChannelSet =

\[\text{alphaClockReset}(e) = \text{alphaClockResetCallArgs}^1(e.\text{args})\]

Rule 88. alphaClockResetCallArgs function
alphaClockResetCallArgs(s : seq Expression) : ChannelSet =

\[
\begin{align*}
\text{if } \#(s) > 0 \text{ then } \\
\quad \text{alphaClockReset}^5(\text{head}(s)) \cup \text{alphaClockResetCallArgs}^2(\text{tail}(s)) \\
\text{else } \\
\quad \emptyset \\
\text{endif}
\end{align*}
\]

Rule 89. alphaClockReset function
alphaClockReset(e : And) : ChannelSet =

\[\text{alphaClockReset}(e) = \text{alphaClockReset}^6(e.\text{left}) \cup \text{alphaClockReset}^7(e.\text{right})\]
Rule 90. alphaClockReset function
alphaClockReset(e : Or) : ChannelSet =

alphaClockReset(e) = alphaClockReset^8(e.left) \cup alphaClockReset^9(e.right)

Rule 91. alphaClockReset function
alphaClockReset(e : Implies) : ChannelSet =

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

Rule 92. alphaClockReset function
alphaClockReset(e : Iff) : ChannelSet =

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

Rule 93. alphaClockReset function
alphaClockReset(e : GreaterThan) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{14}(e.left) \cup alphaClockReset^{15}(e.right)
Rule 94. \( \text{alphaClockReset} \) function
\[
\text{alphaClockReset}(e : \text{GreaterOrEqual}) : \text{ChannelSet} =
\]
\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{16}(e.\text{left}) \cup \text{alphaClockReset}^{17}(e.\text{right})
\]

Rule 95. \( \text{alphaClockReset} \) function
\[
\text{alphaClockReset}(e : \text{LessThan}) : \text{ChannelSet} =
\]
\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{18}(e.\text{left}) \cup \text{alphaClockReset}^{19}(e.\text{right})
\]

Rule 96. \( \text{alphaClockReset} \) function
\[
\text{alphaClockReset}(e : \text{LessOrEqual}) : \text{ChannelSet} =
\]
\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{20}(e.\text{left}) \cup \text{alphaClockReset}^{21}(e.\text{right})
\]

Rule 97. \( \text{alphaClockReset} \) function
\[
\text{alphaClockReset}(e : \text{Equals}) : \text{ChannelSet} =
\]
\[
\text{alphaClockReset}(e) = \text{alphaClockReset}^{22}(e.\text{left}) \cup \text{alphaClockReset}^{23}(e.\text{right})
\]
Rule 98. alphaClockReset function
alphaClockReset(e : ClockExp) : ChannelSet =

\[
\text{alphaClockReset}(e) = \{ \text{clockReset}.id(e.\text{clock}) \}
\]

Rule 99. alphaClockReset function
alphaClockReset(e : StateClockExp) : ChannelSet =

\[
\text{alphaClockReset}(e) = \{ x : \text{SIDS} \mid \text{entered}.id(x).id(e.\text{state}) \}
\]

Waiting Conditions

Waiting Condition elicitation  The following rules defined wc used for eliciting waiting conditions.

Rule 100. wc function
wc(e : Expression) : (Expression, Expression \rightarrow Variable) =

This rule is split into multiple rules according to the expression subtype: (101, 102, 103, 105, 106, 107, 108, 109, 110, 111, 112, 113).
In every other case this function is defined by \( wc(e) = (e, \emptyset) \)
Rule 101. wc function (ParExp)

\[ wc(e : ParExp) : (Expression, Expression \rightarrow Variable) = \]

\[ wc(e) = (e_{new}, \pi_2(wc_{result})) \]
where

\[ wc_{result} = wc^1(e, exp) \]
\[ e_{new} = e \oplus (exp \mapsto \pi_1(wc_{result})) \]

Rule 102. wc function (Not)

\[ wc(e : Not) : (Expression, Expression \rightarrow Variable) = \]

\[ wc(e) = (e_{new}, \pi_2(wc_{result})) \]
where

\[ wc_{result} = wc^2(e, exp) \]
\[ e_{new} = e \oplus (exp \mapsto \pi_1(wc_{result})) \]

Rule 103. wc function (CallExp)

\[ wc(e : CallExp) : (Expression, Expression \rightarrow Variable) = \]

\[ wc(e) = (e_{new}, \pi_2(wc_{result})) \]
where

\[ wc_{result} = wc_{ArgSeq}^1(e, args) \]
\[ e_{new} = e \oplus (args \mapsto \pi_1(wc_{result})) \]
Rule 104. wcArgSeq function

\[ \text{wcArgSeq}(s : \text{seq(Expression)}) : (\text{seq Expression, Expression} \to \text{Variable}) = \]

\[
\text{if } \#(s) > 0 \text{ then}
\]
\[
(\langle \pi_1(\text{wc}_\text{head}) \rangle \bigcap \pi_1(\text{wc}_\text{tail}), \pi_2(\text{wc}_\text{head}) \cup \pi_2(\text{wc}_\text{tail}))
\]
\[
\text{where}
\]
\[
\text{wc}_\text{head} = \text{wc}_3(\text{head}(s))
\]
\[
\text{wc}_\text{tail} = \text{wcArgSeq}^2(\text{tail}(s))
\]
\[
\text{else}
\]
\[
(\langle \rangle, \emptyset)
\]
\[
\text{endif}
\]

Rule 105. wc function (And)

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

\[
\text{wc}(e) = (e_{\text{new}}, \pi_2(\text{wc}_{\text{left}}) \cup \pi_2(\text{wc}_{\text{right}}))
\]
\[
\text{where}
\]
\[
\text{wc}_{\text{left}} = \text{wc}_4(e.\text{left})
\]
\[
\text{wc}_{\text{right}} = \text{wc}_5(e.\text{right})
\]
\[
e_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(\text{wc}_{\text{left}}), \text{right} \mapsto \pi_1(\text{wc}_{\text{right}}))
\]
Rule 106. wc function (Or)

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

\[ wc(\text{e}) = (\text{e}_{\text{new}}, \pi_2(wc_{\text{left}}) \cup \pi_2(wc_{\text{right}})) \]

where

\[ wc_{\text{left}} = wc^6(\text{e}_{\text{left}}) \]

\[ wc_{\text{right}} = wc^7(\text{e}_{\text{right}}) \]

\[ e_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(wc_{\text{left}}), \text{right} \mapsto \pi_1(wc_{\text{right}})) \]

Rule 107. wc function (Implies)

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

\[ wc(\text{e}) = (\text{e}_{\text{new}}, \pi_2(wc_{\text{left}}) \cup \pi_2(wc_{\text{right}})) \]

where

\[ wc_{\text{left}} = wc^8(\text{e}_{\text{left}}) \]

\[ wc_{\text{right}} = wc^9(\text{e}_{\text{right}}) \]

\[ e_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(wc_{\text{left}}), \text{right} \mapsto \pi_1(wc_{\text{right}})) \]

Rule 108. wc function (Iff)

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

\[ wc(\text{e}) = (\text{e}_{\text{new}}, \pi_2(wc_{\text{left}}) \cup \pi_2(wc_{\text{right}})) \]

where

\[ wc_{\text{left}} = wc^{10}(\text{e}_{\text{left}}) \]

\[ wc_{\text{right}} = wc^{11}(\text{e}_{\text{right}}) \]

\[ e_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(wc_{\text{left}}), \text{right} \mapsto \pi_1(wc_{\text{right}})) \]
Rule 109. wc function (GreaterThan)

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

\[
\begin{align*}
\text{if} \quad & \left( e\!.\text{left} \in \text{ClockExp} \lor e\!.\text{right} \in \text{ClockExp} \\
& \lor \quad e\!.\text{left} \in \text{StateClockExp} \lor e\!.\text{right} \in \text{StateClockExp} \right) \\
\text{then} \quad & (b, \{e \mapsto b\}) \\
\text{else} \quad & (e, \emptyset) \\
\text{endif}
\end{align*}
\]

where
\[b\] is a fresh identifier

Rule 110. wc function (GreaterOrEqual)

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

\[
\begin{align*}
\text{if} \quad & \left( e\!.\text{left} \in \text{ClockExp} \lor e\!.\text{right} \in \text{ClockExp} \\
& \lor \quad e\!.\text{left} \in \text{StateClockExp} \lor e\!.\text{right} \in \text{StateClockExp} \right) \\
\text{then} \quad & (b, \{e \mapsto b\}) \\
\text{else} \quad & (e, \emptyset) \\
\text{endif}
\end{align*}
\]

where
\[b\] is a fresh identifier
Rule 111. wc function (LessThan)

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

\[
\begin{align*}
& \text{if } \left( e.\text{left} \in \text{ClockExp} \lor e.\text{right} \in \text{ClockExp} \right) \lor \\
& \quad \left( e.\text{left} \in \text{StateClockExp} \lor e.\text{right} \in \text{StateClockExp} \right) \\
& \quad \text{then } \left( b, \{ e \mapsto b \} \right) \\
& \text{else } \\
& \quad (e, \emptyset) \\
& \text{endif}
\end{align*}
\]

where

\( b \) is a fresh identifier

Rule 112. wc function (LessOrEqual)

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

\[
\begin{align*}
& \text{if } \left( e.\text{left} \in \text{ClockExp} \lor e.\text{right} \in \text{ClockExp} \right) \lor \\
& \quad \left( e.\text{left} \in \text{StateClockExp} \lor e.\text{right} \in \text{StateClockExp} \right) \\
& \quad \text{then } \left( b, \{ e \mapsto b \} \right) \\
& \text{else } \\
& \quad (e, \emptyset) \\
& \text{endif}
\end{align*}
\]

where

\( b \) is a fresh identifier
Rule 113. \textit{wc} function (Equals)

\[
\text{wc}(e : \text{Equals}) : (\text{Expression}, \text{Expression} \rightarrow \text{Variable}) = \\
\begin{cases}
  \text{if} \left( \begin{array}{l}
e.\text{left} \in \text{ClockExp} \lor e.\text{right} \in \text{ClockExp} \\
          \lor 
          e.\text{left} \in \text{StateClockExp} \lor e.\text{right} \in \text{StateClockExp}
\end{array} \right) \\
  \text{then} \\
  (b, \{e \mapsto b\}) \\
  \text{else} \\
  (e, \emptyset)
\end{cases}
\]

where 
\(b\) is a fresh identifier

Waiting Condition as CSP processes The following rules define the function \textit{compileWC} which is used to define the CSP semantics of waiting conditions.

Rule 114. \textit{compileWC} function

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

This rule is split into multiple rules (118, 119, 120, 121 and 122) according to the expression subtype.

Rule 115. \textit{getClockReset} function

\[
\text{getClockReset}(e : \text{Expression}) : \text{TimedCSPProcess} =
\]

This rule is split into two rules 116 and 117 according to the subtype.
Rule 116. getClockReset function

gETCHCKReset(e : StateClockExp) : TimedCSPProcess =

\ entered?x?Id(e.state) \rightarrow Skip

Rule 117. getClockReset function

getClockReset(e : ClockExp) : TimedCSPProcess =

clockReset.id(e.clock) \rightarrow Skip
Rule 118. compileWC function
compileWC(t : Transition, e : GreaterThanOrEqual, v : Variable) : TimedCSPProcess =

if (e.left ∈ ClockExp ∨ e.left ∈ StateClockExp) then
  let
    Reset = getClockReset1(e.left); setWC_vide(v)false → Monitor
    Monitor = \(\begin{cases}
    \text{RUN}\{\text{triggerEvent}^3(t)}\} \quad \Delta \text{Reset} \\
    \text{setWC_vide(v)}true \rightarrow \text{RUN}\{\text{triggerEvent}^4(t)}\} \quad \text{within}
  \end{cases}\)

else if (e.right ∈ ClockExp ∨ e.right ∈ StateClockExp) then
  let
    Reset = getClockReset2(e.right); setWC_vide(v)true → Monitor
    Monitor = \(\begin{cases}
    \text{RUN}\{\text{triggerEvent}^5(t)}\} \quad \Delta \text{Reset} \\
    \text{setWC_vide(v)}false \rightarrow \text{RUN}\{\text{triggerEvent}^6(t)}\} \quad \text{within}
  \end{cases}\)

endif
Rule 119. compileWC function

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

\[
\begin{align*}
\text{if } (e.\text{left} & \in \text{ClockExp} \lor e.\text{left} \in \text{StateClockExp}) \text{ then} \\
\text{let} & \\
\text{Reset} &= \text{getClockReset}^3(e.\text{left}); \text{ setWC}._\text{vid}(v)!\text{false} \rightarrow \text{Monitor} \\
\text{Monitor} &= \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^7(t)\}) \\
\triangle (e.\text{right} + 1) \text{ setWC}._\text{vid}(v)!\text{true} \rightarrow \text{RUN}(\{\text{triggerEvent}^9(t)\})
\end{array} \right) \quad \text{\(\Delta \text{ Reset}\)} \\
\text{within} & \\
\text{setWC}._\text{vid}(v)!\text{false} \rightarrow \text{Monitor}
\end{align*}
\]

\[
\text{else if } (e.\text{right} \in \text{ClockExp} \lor e.\text{right} \in \text{StateClockExp}) \text{ then} \\
\text{let} & \\
\text{Reset} &= \text{getClockReset}^4(e.\text{right}); \text{ setWC}._\text{vid}(v)!\text{true} \rightarrow \text{Monitor} \\
\text{Monitor} &= \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^9(t)\}) \\
\triangle (e.\text{left} + 1) \text{ setWC}._\text{vid}(v)!\text{false} \rightarrow \text{RUN}(\{\text{triggerEvent}^{10}(t)\})
\end{array} \right) \quad \text{\(\Delta \text{ Reset}\)} \\
\text{within} & \\
\text{setWC}._\text{vid}(v)!\text{true} \rightarrow \text{Monitor}
\end{align*}
\]

\[\text{endif}\]
Rule 120. compileWC function

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

\[
\begin{align*}
\text{if } (e.\text{left} \in \text{ClockExp} \lor e.\text{left} \in \text{StateClockExp}) \text{ then} \\
\quad \text{let} \\
\quad \quad \text{Reset} = \text{getClockReset}^5(e.\text{left}); \ \text{setWC}_{\text{vid}}(v)\!\! \text{true} \rightarrow \text{Monitor} \\
\quad \quad \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^{11}(t)\}) \\
\Delta_{[e.\text{right}]_{\text{Exp}^56}} \text{setWC}_{\text{vid}}(v)\!\! \text{false} \rightarrow \text{RUN}(\{\text{triggerEvent}^{12}(t)\}) \\
\end{array} \right) \triangle \text{Reset} \\
\quad \text{within} \\
\quad \text{setWC}_{\text{vid}}(v)\!\! \text{true} \rightarrow \text{Monitor} \\
\text{else if } (e.\text{right} \in \text{ClockExp} \lor e.\text{right} \in \text{StateClockExp}) \text{ then} \\
\quad \text{let} \\
\quad \quad \text{Reset} = \text{getClockReset}^6(e.\text{right}); \ \text{setWC}_{\text{vid}}(v)\!\! \text{false} \rightarrow \text{Monitor} \\
\quad \quad \text{Monitor} = \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^{13}(t)\}) \\
\Delta_{[e.\text{left}]_{\text{Exp}^57}} \text{setWC}_{\text{vid}}(v)\!\! \text{true} \rightarrow \text{RUN}(\{\text{triggerEvent}^{14}(t)\}) \\
\end{array} \right) \triangle \text{Reset} \\
\quad \text{within} \\
\quad \text{setWC}_{\text{vid}}(v)\!\! \text{false} \rightarrow \text{Monitor} \\
\text{endif}
\end{align*}
\]
Rule 121. compileWC function

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

\[
\begin{align*}
\text{if } (e.\text{left} \in \text{ClockExp} \lor e.\text{left} \in \text{StateClockExp}) & \text{ then} \\
\text{let } & \\
\text{Reset = } get\text{ClockReset}^7(e.\text{left}); \text{ setWC}_v(\text{true}) & \text{ → Monitor} \\
\text{Monitor = } & \\
\left( \begin{array}{c}
\text{RUN}\left(\{\text{triggerEvent}^{15}(t)\}\right) \\
\triangle\left(\left[\text{Exp}^{58}\right]_{\text{E}_0} + 1\right)\text{ setWC}_v(\text{false}) & \text{ → RUN}\left(\{\text{triggerEvent}^{16}(t)\}\right)
\end{array} \right) \triangle\text{Reset} \\
\text{within } & \\
\text{setWC}_v(\text{true}) & \text{ → Monitor} \\
\text{else if } (e.\text{right} \in \text{ClockExp} \lor e.\text{right} \in \text{StateClockExp}) & \text{ then} \\
\text{let } & \\
\text{Reset = } get\text{ClockReset}^8(e.\text{right}); \text{ setWC}_v(\text{false}) & \text{ → Monitor} \\
\text{Monitor = } & \\
\left( \begin{array}{c}
\text{RUN}\left(\{\text{triggerEvent}^{17}(t)\}\right) \\
\triangle\left(\left[\text{Exp}^{59}\right]_{\text{E}_0} + 1\right)\text{ setWC}_v(\text{true}) & \text{ → RUN}\left(\{\text{triggerEvent}^{18}(t)\}\right)
\end{array} \right) \triangle\text{Reset} \\
\text{within } & \\
\text{setWC}_v(\text{false}) & \text{ → Monitor} \\
\text{endif}
\end{align*}
\]
Rule 122. compileWC function

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

if \((e.\text{left} \in \text{ClockExp} \lor e.\text{left} \in \text{StateClockExp})\) then

let

\[
\begin{align*}
\text{Reset} &= \text{getClockReset}^9(e.\text{left}); \; \text{setWC vid(v)!false} \rightarrow \text{Monitor} \\
\text{Monitor} &= \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^{19}(t)\}) \\
\Delta_{e.\text{right}} \text{setWC vid(v)!true} \rightarrow \text{RUN}(\{\text{triggerEvent}^{20}(t)\}) \\
\Delta_{e.\text{right}} (e.\text{right} + 1) \text{setWC vid(v)!false} \rightarrow \text{RUN}(\{\text{triggerEvent}^{21}(t)\}) \\
\end{array} \right) \quad \triangle \text{Reset}
\end{align*}
\]

within

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

else if \((e.\text{right} \in \text{ClockExp} \lor e.\text{right} \in \text{StateClockExp})\) then

let

\[
\begin{align*}
\text{Reset} &= \text{getClockReset}^{10}(e.\text{right}); \; \text{setWC vid(v)!false} \rightarrow \text{Monitor} \\
\text{Monitor} &= \left( \begin{array}{c}
\text{RUN}(\{\text{triggerEvent}^{22}(t)\}) \\
\Delta_{e.\text{left}} \text{setWC vid(v)!true} \rightarrow \text{RUN}(\{\text{triggerEvent}^{23}(t)\}) \\
\Delta_{e} (e + 1) \text{setWC vid(v)!false} \rightarrow \text{RUN}(\{\text{triggerEvent}^{24}(t)\}) \\
\end{array} \right) \quad \triangle \text{Reset}
\end{align*}
\]

within

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

endif
Trigger Deadlines

Rule 123. deadlineEvents function
deadlineEvents(s : StateMachineDef) : ChannelSet =

deadlineEvents(stm) = \{ t : allSTMTransitions^2(s) | t.end \neq null \land \text{deadline} \cdot id(t) \}

Rule 124. triggerEvent function
triggerEvent(t : Transition) : CSPEvent =

if t.trigger \neq null
    triggerEvent^3(t.trigger.id(t))
else
    internal__id(t)
Rule 125. State-machine Memory

\[
\text{stmMemory}(s : \text{StateMachine}, \text{wcs} : \text{Transition} \rightarrow \langle \text{Expression}, \text{WC} \rangle) : \text{TimedCSPProcess} =
\]

\[
\text{let } \text{Memory} (\text{vars}) \triangleq
\begin{cases}
\quad \forall v : \text{lvars} \bullet \left( \begin{array}{l}
\text{get}_{\text{vid}}(v, s)\text{name}(v) \rightarrow \text{Memory}(\text{vars}) \\
\quad \quad \text{set}_{\text{vid}}(v, s)\text{?}x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x])
\end{array} \right) \\
\quad \forall v : \text{rvars} \bullet \\
\quad \quad \left( \begin{array}{l}
\text{get}_{\text{vid}}(v, s)\text{name}(v) \rightarrow \text{Memory}(\text{vars}) \\
\quad \quad \quad \text{set}_{\text{vid}}(v, s)\text{?}x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x]) \\
\quad \quad \quad \text{set}_{\text{Ext}_{\text{vid}}}(v, s)\text{?}x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x])
\end{array} \right) \\
\quad \forall v : \text{cvars} \bullet \text{set}_{\text{WC}_{\text{vid}}}(v)\text{?}x \rightarrow \text{Memory}(\text{vars}[\text{name}(v) := x])
\end{cases}
\]

\[
\text{within}
\]

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

where

\[
\begin{align*}
\text{rvars} &= \text{requiredVariables}^3(s) \\
\text{lvars} &= \text{allLocalVariables}^{10}(s) \\
\text{cvars} &= \text{allClockVariables}^2(\text{wcs}) \\
\text{vars} &= \{ v : \text{rvars} \cup \text{lvars} \cup \text{cvars} \bullet \text{name}(v) \} \\
\text{varvalues} &= \{ v : \text{rvars} \cup \text{lvars} \cup \text{cvars} \bullet \text{initial}(v) \}
\end{align*}
\]
Rule 126. allTriggerDeadlineTransitions function

\[
\text{allTriggerDeadlineTransitions}(s : \text{StateMachineDef}) : \mathcal{P} \text{Transition} =
\]

\[
\text{allTriggerDeadlineTransitions}(s) = \{ t : \text{allSTMTransitions}^A(s) | t.\text{end} \neq \text{null} \}
\]

Rule 127. memoryTransition function

memoryTransition(t : Transition, wcs : Transition \rightarrow (Expression, WC)) : TimedCSPProcess

if (t.condition \neq \text{null}) then

\[
(\left\llbracket \pi_1(\text{wcs}(t)) \right\rrbracket^{\text{Expr}^{64}}_E) \& \left\llbracket \text{t.trigger}^{\text{id}(t)}_T \right\rrbracket^{\text{Trigger}^6}_T
\]

else

\[
\left\llbracket \text{t.trigger}^{\text{id}(t)}_T \right\rrbracket^{\text{Trigger}^7}_T
\]

Rule 128. Memory deadline

memoryDeadline(t : Transition, wcs : Transition \rightarrow (Expression, WC)) : TimedCSPProcess

if (t.condition \neq \text{null}) then

\[
(\left\llbracket \pi_1(\text{wcs}(t)) \right\rrbracket^{\text{Expr}^{65}}_E) \& \text{deadline}!^{\text{id}(t)}_{\text{on}} \rightarrow \text{Skip}
\]

\[
(\neg \left\llbracket \pi_1(\text{wcs}(t)) \right\rrbracket^{\text{Expr}^{66}}_E) \& \text{deadline}!^{\text{id}(t)}_{\text{off}} \rightarrow \text{Skip}
\]

else

\[
\text{deadline}!^{\text{id}(t)}_{\text{on}} \rightarrow \text{Skip}
\]
4.2.2 States

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

Rule 129. Timed semantics of states

\[ s : \text{State} \] \[ \overline{S} : \text{TimedCSPProcess} = \]

This function is split in multiple rules according to the type of states.
Rule 130. Timed semantics of simple states

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

\[
\text{let } \quad \text{Inactive} \triangleq \text{enter}？x : \text{sids}!id(s) \rightarrow \text{Activating}(x) \\
\text{Activating}(o) \triangleq [s.\text{entry}] \quad \text{initialisation}^5(s) ; \text{entered}!o!id(s) \rightarrow \\
(\text{triggerDeadlines}^1(s) || [s.\text{during}] \quad \text{Action}^{11}; \text{Stop}) \triangle \\
\begin{align*}
&\square t : \text{transitionsFrom}^5(s) \bullet [t,s,\text{false}]^\text{Inactive,Activating} \\
&\square e : \text{Event} \quad \text{if } (e.\text{type} == \text{null}) \\
&\quad \text{then } \text{eventId}(e)？x : \text{tids} \rightarrow \text{exit; Inactive} \\
&\quad \text{else } \text{eventId}(e)？x : \text{tids}?y \rightarrow \text{exit: Inactive} \\
\end{align*}
\]

within

\text{Inactive}

where

\[
\begin{align*}
#\text{substates}^{11}(s) &= 0 \\
\text{sids} &= SIDS \setminus \{id(s)\} \\
\text{exit} &= \text{exit}'?x : \text{sids}!id(s) \rightarrow \text{exitSubstates}^4(s) ; [s.\text{exit}] \quad \text{Action}^{12} \\
\text{tids} &= TIDS \setminus tIDS(s) \\
\end{align*}
\]
Rule 131. Timed semantics of composite states

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

\[
\text{let Inactive} \triangleq \text{enter}?x : \text{sid}!\text{id}(s) \rightarrow \text{Activating}(x)
\]

\[
\text{Activating}(o) \triangleq [s.\text{entry}]_\text{Action}^6 : \text{initialisation}^0(s); \text{entered}!o!\text{id}(s) \rightarrow
\]

\[
(\text{triggerDeadlines}^2(s) || [s.\text{during}]_\text{Action}^8 ; \text{Stop}) \bigtriangleup
\]

\[
(\bigcirc t : \text{transitionsFrom}^5(s) \bullet [t,s,\text{false}]_T^6 \\text{Inactive,Activating})
\]

\[
\bigcirc e : \text{Event} \bullet \begin{cases} 
\text{if}(e.\text{type} == \text{null}) \\
\text{then eventId}(e)?x : \text{tids} \rightarrow \text{exit}; \text{Inactive} \\
\text{else eventId}(e)?x : \text{tids}?y \rightarrow \text{exit}; \text{Inactive} 
\end{cases}
\]

within

\[(\text{Inactive} \mid \text{flowTriggerEvts} \mid \text{composeStates}^1(\text{substates}^{12}(s), s))\]

where

\[
\#\text{substates}^{13}(s) > 0
\]

flowTriggerEvts = \text{flowTriggerEvents}^3(s)

\[
sids = SIDS \setminus \{\text{id}(s)\}
\]

\[
\text{exit} = \text{exit}!x : \text{sid}!\text{id}(s) \rightarrow [s.\text{exit}]_\text{Action}^6 ; \text{exitSubstates}^5(s); \text{exitSubstates}^5(s) \rightarrow \text{Skip}
\]

\[
tids = TIDS \setminus tIDS(s)
\]
Rule 132. Semantics of trigger deadlines

\[
\text{triggerDeadlines}(s : \text{State}) : \text{TimedCSPProcess} = \\
||| t : tDS \bullet \left( \mu X \bullet \left( \begin{array}{c}
\text{deadline}!id(t)!on \rightarrow \\
\text{readState}^3 \left( \begin{array}{c}
\text{usedV}(t.\text{end}), \\
\text{deadline}!id(t)!off \rightarrow \text{Skip} \quad \text{Expr}^{67}
\end{array} \right) \right) \right) ; X
\right)
\]

where

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

The composition of states is also largely unchanged when compared to the untimed Rule 20 except that the set of flow events is not hidden, so as to allow a parent to observe all of its children’s flow events, and the state-machine to observe entered events required to reset an implicit clock in the case of sinceEntry(s).

Rule 133. Timed composition of states

\[
\text{composeStates}(ss : \text{seq State}, p : \text{NodeContainer}) : \text{TimedCSPProcess} = \\
\text{if} \#ss = 1 \\
\text{then} \\
\text{restrictedState}^3(p, \text{head ss}) \\
\text{else} \\
\left( \text{restrictedState}^4(p, \text{head ss}) \\
|| \text{shflowevts} \\
\text{composeStates}^2(tail ss, p) \right)
\]

where

\[
\text{shflowevts} = \text{flowEvents}^3(\text{head ss}, p) \cap \bigcup \{ x : \text{tail ss} \bullet \text{flowEvents}^4(x, p) \}
\]
4.2.3 Timed statements

Rule 134. Semantics of timed statements

\[ [s : \text{Statement}]_{\text{Statement}} : \text{TimedCSPProcess} = \]

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

Rule 135. Function usedVariables for timed semantics

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

\[
\begin{aligned}
\text{if } s \in \text{TimedStatement} & \text{ then } \\
\text{usedV}(s.\text{end}) \cup \text{usedVariables}(s.\text{stmt}) \\
\text{else if } s \in \text{Wait} & \text{ then } \\
\text{usedV}(s.\text{duration}) \\
\text{else} & \\
\text{usedVariables}^2(s)
\end{aligned}
\]

Rule 136. Semantics of Deadlines

\[ [s : \text{TimedStatement}]_{\text{Statement}} : \text{TimedCSPProcess} = \]

\[
\begin{aligned}
[s.\text{stmt}]_{\text{Statement}} & \triangleright [s.\text{end}]_{\text{Expr}}^{68}
\end{aligned}
\]

Rule 137. Semantics of Wait

\[ [s : \text{Wait}]_{\text{Statement}} : \text{TimedCSPProcess} = \]

\[ \text{Wait}([s.\text{duration}]_{\text{Expr}}^{69}) \]

84
Rule 138. Semantics of Clock Reset

\[ [s: \text{ClockReset}] \text{Statement} : \text{TimedCSPProcess} = \]

\[ \text{clockReset}!d(s.\text{clock}) \rightarrow \text{Skip} \]
Chapter 5

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/.

We have described the semantics for the core constructs of RoboChart. It uses CSP, but we envisage its extension to use Circus [2], a process algebra that combines Z [12] and CSP, and includes time constructs [11]. 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 [3], with a semantics given by the UTP theory in [13].

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 [6] to extend the notation, and from the UTP model of continuous variables [4] to define the semantics.
Our core notation is a state-machine based language with specific components that provide for sequential behaviours and parallelism in a restricted manner. Essentially, state-machines are intended to specify sequential behaviours, whilst parallelism is modelled by controllers. The top-level components of a RoboChart specification is a module, which represents a single robot recording assumptions about the hardware as well as the controlling software.

A.1 Robotic Platforms

A robotic platform is characterised by variables, events, and operations representing in-built facilities of the hardware. It represents the observables interactions between the robot, its environment and controller.

abstract class RoboticPlatform extends ConnectionNode;
class RoboticPlatformDef extends NamedElement,Context,RoboticPlatform;
class RoboticPlatformRef extends NamedElement,RoboticPlatform {
    property ref : RoboticPlatformDef[?];
}
abstract class NamedElement {
    attribute name : String[?];
}
abstract class BasicContext {
    property variableList : VariableList[*|1] { ordered composes };
    property operations : Operation[*|1] { ordered composes };
    property events : Event[*|1] { ordered composes };
}
abstract class Context extends BasicContext {
    property pInterfaces : Interface[*|1] { !unique };

property rInterfaces : Interface[*|1] { !unique };  
property interfaces : Interface[*|1] { !unique };  
}

A.2 Interfaces

An interface encapsulates events and variables declarations as well as operation signatures. Interfaces are used to record information about the assumptions a component makes, and what assumptions can be made about a component.

```plaintext
class Interface extends NamedElement, BasicContext;
```

A.3 Variables

An variable can be declared in interfaces, robotic platforms, controllers and state-machines. Variable declarations in interfaces are used to validate a model with respect to assumptions about definition and usage of variables in the context of a module (composition of controllers and robotic platforms). Variables declared in controllers and robotic platforms are shared among its associated elements (state-machines and controllers, respectively), but are ultimately only used by state-machines. State-machines can themselves declare variables that are local to the state-machine. Variables are typed and may declare an initial value.

```plaintext
class VariableList {
    attribute modifier : String[?];
    property vars : Variable[*|1] { ordered composes };  
}

class Variable extends NamedExpression {
    attribute name : String[?];
    property type : Type[?] { composes };  
    property initial : Expression[?] { composes };  
    attribute modifier : String[?] { derived transient volatile };  
}  
```
A.4 Constants

A constant is similar to a variable except that its value cannot be changed. Besides its usage as a meaningful name or abbreviation for otherwise complex or meaningless values, it can also be used without a concrete value to indicate a loose values that is fixed but unspecified.

Variable with modifier == 'con'

A.5 Events

An event are the main form of interaction between a state-machine and its environment, be it other state-machines, controllers or the robotic platform. Events can be typed or untyped, where typed events carry values, and untyped events model a simple interaction where no extra information can be inferred except that two parallel components interacted. Whilst events are not explicitly divided between input and output events, their roles are exclusive (events cannot be used as both inputs and outputs) and are determined from the connections in the model. The connections between events (not the events themselves) determine if the communication takes place synchronously or asynchronously.

class Event extends NamedElement {
    property type : Type[?] { composes };
    attribute broadcast : Boolean[?];
}

A.6 Required, Provided and Defined Interfaces

A required interface specifies the assumptions a state-machine or controller makes about the environment, the robotic platform and other controllers. It is used to declare abstract controllers and state-machines that do not depend on specific platforms, only on specific operations and variables. It is worth mentioning that required interfaces can be used to specify assumptions about the kind of state variables are available in a robotic platform. This allows for instance the specification of movement operations independently of the particular platform.
based solely on the assumption that a potential target platform supports changing linear and angular speed by setting specific variables.

**Interfaces of Context**

A provided interface specifies what variables and operations a robotic platform provides. It is used mainly to validate the well-formedness of controllers and modules by guaranteeing that the assumptions made (through required interfaces) are actually satisfied by some component in the composition.

**Interfaces of Context**

A defined interface is used to declare variables and events in an element. It has the same effect of declaring each variable and event separately. It is helpful to complement required interfaces containing only variables.

**Interfaces of Context**

### A.7 State-Machines

A state-machine is the construct dedicated for the specification of sequential behaviours. It contains a number of nodes that represent steps (stable or not) of the behaviour and transitions that describe when and how control is transferred from one node to another.

```java
abstract class StateMachine extends ConnectionNode;
abstract class ConnectionNode;
class StateMachineDef extends NamedElement,StateMachineBody,StateMachine;
class StateMachineRef extends Node,StateMachine {
    property ref : StateMachineDef[?];
}
class StateMachineBody extends Context,NodeContainer {
    property clocks : Clock[*|1] { ordered composes };
```
A.8 States

A state is one of the main components of a state-machine. It describes a stable configuration of the state-machine and has three distinctive phases in its life-cycle: entering, executing and exiting. Each of these phases has an associated action: entry, during and exit actions.

States are divided between simple and composite states. Simple states can only contain the actions mentioned above, whilst composite states can themselves contain states as well as transitions and other nodes.

```java
class State extends Node, NodeContainer {
   property actions : Action[*|1] { ordered composes };
}
```

A.9 Final states

A final state represents the completion of the internal behaviours of a state-machine or composite state. While the meaning of a final state is the same in both cases, state-machines and composite states react differently to reaching a final state. While a composite state rests in the final state and waits for one of its transitions (or the parents transitions) to be executed, the state-machine terminates as soon as the final state is reached.

```java
class Final extends State;
```
A.10 Junction node

A junction node represents an unstable configuration of the state-machine. Unlike in a (stable) state, the state-machine cannot rest and execute other behaviours (actions, substates, etc) while in a junction node. At this point, all it can do is follow one of the outgoing transitions.

In order to guarantee that execution can progress once in a junction node, two well-formedness condition are defined. The first requires that there are event triggers in the outgoing transitions, and the second requires that the outgoing transitions form a cover, that is, the conjunction of all their guards is equivalent to true. Notice that we do not require them to be disjoint as the selection of outgoing transition may be non-deterministic.

class Junction extends Node;

A.11 Initial nodes

An initial node represents an entry point of a state-machine or composite state. It indicate where the state-machine or composite state must start executing to enter its substates.

The main validation rule related to initial nodes is that any state-machine or composite state (state with one or more subnodes) must have exactly one initial node.

class Initial extends Junction;

A.12 Transitions

A transition defines one possible path between nodes in a state-machine or composite state. It contains source and target nodes as well as, optionally, a trigger in the form of an event, a boolean condition, and an action. The transition can only be executed if the event in the trigger is available and the condition is true.
The transition action is executed after the source state (if it exists) is exited, but before the target state (if it exists) is entered. Notice that source and target states are not necessarily available. For example transitions between junction nodes have neither.

class Transition extends NamedElement {
    property source : Node[?];
    property target : Node[?];
    property start : Expression[?] { composes };
    property trigger : Trigger[?] { composes };
    property end : Expression[?] { composes };
    property condition : Expression[?] { composes };
    property action : Statement[?] { composes };
}
class Trigger {
    property time : Variable[?];
    property reset : ClockReset[*|1] { ordered composes };
    property event : Event[?];
}
class InputTrigger extends Trigger {
    attribute parameter : String[?];
}
class OutputTrigger extends Trigger {
    property value : Expression[?] { composes };
}
class SyncTrigger extends Trigger {
    property value : Expression[?] { composes };
}
class SimpleTrigger extends Trigger;

A.13 Controllers

A controller models a collection of potentially parallel cooperating state machines; it can be used, for instance, to encapsulate specific well-defined functionalities that are implemented by multiple state machines. Controllers are the elements of RoboChart that interact
directly with robotic platform.

abstract class Controller extends ConnectionNode;
class ControllerDef extends NamedElement,Context,Controller,MachineContainer {
    property connections : Connection[*|1] { ordered composes };
}
abstract class MachineContainer {
    property machines : StateMachine[*|1] { ordered composes };
}
class ControllerRef extends NamedElement,Controller {
    property ref : ControllerDef[?];
}

A.14 Connection

A connection is a link between events within (and on the boundary) of state machines, controllers, and robotic platforms. These connections are used to specify the interactions between state machines in a controller, and between controllers in a module. They are used also to specify relays between the state machines and the containing controller, and between controllers and robotic platforms in a module.

class Connection {
    property from : ConnectionNode[?];
    property efrom : Event[?];
    property to : ConnectionNode[?];
    property eto : Event[?];
    attribute async : Boolean[?];
    attribute mult : Boolean[?];
}
A.15 Modules

class Module extends NamedElement {
    property connections : Connection[*|1] { ordered composes };
    property nodes : ConnectionNode[*|1] { ordered composes };
}

A.16 Statements

abstract class Statement;
class TimedStatement extends Statement {
    property start : Expression[?] { composes };
    property stmt : Statement[?] { composes };
    property end : Expression[?] { composes };
}
class Wait extends Statement {
    property duration : Expression[?] { composes };
}
class Skip extends Statement;
class IfStmt extends Statement {
    property expression : Expression[?] { composes };
    property _'then' : Statement[?] { composes };
    property _'else' : Statement[?] { composes };
}
class Assignment extends Statement {
    property left : Assignable[?] { composes };
    property right : Expression[?] { composes };
}
class SendEvent extends Statement {
    property trigger : Trigger[?] { composes };
}
class SeqStatement extends Statement {
    property statements : Statement[*|1] { ordered composes };
}
class ParStmt extends Statement {
    property stmt : Statement[?] { composes };
}
class Call extends Statement {
    property operation : Operation[?];
    property args : Expression[*|1] { ordered composes };
}

A.17 Expressions

abstract class Expression;
class ResultExp extends Expression;
class ArrayExp extends Expression {
    property value : Expression[?] { composes };
    property parameters : Expression[*|1] { composes };
}
class ClockExp extends Expression {
    property clock : Clock[?];
}
class StateClockExp extends Expression {
    property state : State[?];
}
class Iff extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Implies extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Or extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Forall extends Expression {

property variables : Variable[*|1] { ordered composes };
property suchthat : Expression[*] { composes };
property predicate : Expression[*] { composes };
}
class Exists extends Expression {
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[*] { composes };
  property predicate : Expression[*] { composes };
  attribute unique : Boolean[*];
}
class LambdaExp extends Expression {
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[*] { composes };
  property expression : Expression[*] { composes };
}
class DefiniteDescription extends Expression {
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[*] { composes };
  property expression : Expression[*] { composes };
}
class IfExpression extends Expression {
  property condition : Expression[*] { composes };
  property ifexp : Expression[*] { composes };
  property elseexp : Expression[*] { composes };
}
class Declaration {
  attribute name : String[*];
  property value : Expression[*] { composes };
}
class LetExpression extends Expression {
  property declarations : Declaration[*|1] { ordered composes };
  property expression : Expression[*] { composes };
}
class And extends Expression {
  property left : Expression[*] { composes };
  property right : Expression[*] { composes };
}
class Not extends Expression {
    property exp : Expression[] { composes };
}

class InExp extends Expression {
    property member : Expression[] { composes };
    property set : Expression[] { composes };
}

class TypeExp extends Expression {
    property type : Type[] { composes };
}

class Equals extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class Different extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class GreaterThan extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class GreaterOrEqual extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class LessThan extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class LessOrEqual extends Expression {
    property left : Expression[] { composes };
    property right : Expression[] { composes };
}

class Plus extends Expression {
property left : Expression[?] { composes };
property right : Expression[?] { composes };
}
class Minus extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Modulus extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Mult extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Div extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Cat extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Neg extends Expression {
    property exp : Expression[?] { composes };
}
class Selection extends Expression {
    property receiver : Expression[?] { composes };
    property member : Member[?];
}
class IntegerExp extends Expression {
    attribute value : ecore::EInt[?];
}
class FloatExp extends Expression {
    attribute value : ecore::EFloat[?];
}
class StringExp extends Expression {
    attribute value : String[?];
}
class BooleanExp extends Expression {
    attribute value : String[?];
}
class VarExp extends Expression {
    property value : Variable[?];
}
class RefExp extends Expression {
    property ref : NamedExpression[?];
}
class EnumExp extends Expression {
    property type : Enumeration[?];
    property constant : Constant[?];
}
class ParExp extends Expression {
    property exp : Expression[?] { composes };
}
class SeqExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}
class SetExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}
class SetComp extends Expression {
    property variables : Variable[*|1] { ordered composes };
    property predicate : Expression[?] { composes };
    property expression : Expression[?] { composes };
}
class SetRange extends Expression {
    property start : Expression[?] { composes };
    property end : Expression[?] { composes };
}
class TupleExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}
class RangeExp extends Expression {
    attribute linterval : String[?];
    property lrange : Expression[?] { composes };
    property rrange : Expression[?] { composes };
    attribute rinterval : String[?];
}
class CallExp extends Expression {
    property function : Function[?];
    property args : Expression[1|1] { ordered composes };
}
class ElseExp extends Expression;

A.18 Type Declaration

class TypeDecl extends NamedElement;

A.19 Primitive Types

T A primitive type ....

class PrimitiveType extends TypeDecl;

A.20 Datatypes

D A datatype ....

class DataType extends TypeDecl{
    property fields : Field[*|1] { ordered composes };
}
class Field extends Member, NamedExpression;

A.21 Enumeration

class Enumeration extends TypeDecl {
    property constants : Constant[*|1] { ordered composes };
}
abstract class NamedExpression;
class Constant extends NamedExpression {
    attribute name : String[?];
}

A.22 Type Constructors

abstract class Type;
class AnyType extends Type {
    attribute identifier : String[?];
}
class ProductType extends Type {
    property types : Type[*|1] { ordered composes };
}
class FunctionType extends Type {
    property domain : Type[?] { composes };
    property range : Type[?] { composes };
}
class RelationType extends Type {
    property domain : Type[?] { composes };
    property range : Type[?] { composes };
}
class SetType extends Type {

property domain : Type[?] { composes }; 
}
class SeqType extends Type {
    property domain : Type[?] { composes }; 
}
class TypeRef extends Type {
    property ref : TypeDecl[?]; 
}
Complete Metamodel: Timed Language

B.1 Clock

A clock ...

```java
class Instant {
    property instant : Clock[?];
}
class Clock {
    attribute type : String[?];
    attribute name : String[?];
}
```

B.2 Timed Statements

```java
class Statement {
    property start : Expression[?] { composes };
    property stmt : Statement[?] { composes };
    property end : Expression[?] { composes };
}
class Wait extends Statement {
    property duration : Expression[?] { composes };
}
```
B.3 Timed Expressions

class ClockExp extends Expression {
    property instant : Clock[?];
}
class StateClockExp extends Expression {
    property state : State[?];
}

B.4 Timed Triggers

class Trigger {
    property time : Variable[?];
    property instant : Instant[*|1] { ordered composes };
    property event : Event[?];
}

class Transition extends NamedElement {
    property source : Node[?];
    property target : Node[?];
    property start : Expression[?] { composes };
    property trigger : Trigger[?] { composes };
    property end : Expression[?] { composes };
    property condition : Expression[?] { composes };
    property action : Statement[?] { composes };
}
Appendix C

Credits

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

aturas

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

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 Revicon 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


Index of Semantic Rules

In this index you’ll find the list of semantic functions in alphabetic order, and page where they are defined. Timed versions of existing 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, 50
allClockVariables (timed), 59
allLocalVariables, 29
allSTMTransitions, 45
allTransitions, 45
allTriggerDeadlineTransitions (timed), 79
allVariables, 29
alphaClockReset, 60
  And (timed), 61
  CallExp (timed), 61
  ClockExp (timed), 64
  Equals (timed), 63
  GreaterOrEqual (timed), 63
  GreaterThan (timed), 62
  If (timed), 62
  Implies (timed), 62
  LessOrEqual (timed), 63
  LessThan (timed), 63
  Not (timed), 61
  Or (timed), 62
  ParExp (timed), 60
  StateClockExp (timed), 64
  alphaClockResetCallArgs (timed), 61

buffer, 32

C (Controller), 33
clockResets (timed), 60
compileTarget, 42
compileWC (timed), 70, 72, 73, 74, 75, 76
composeControllers, 31
composeMachines, 35
composeStates, 37, 83
ctrlMemory, 34
deadlineEvents (timed), 77
exitSubstates, 43
Expr (Expression), 50
  And Expression, 50
  Array Expression, 51
  Boolean Expression, 51
  Call Expression, 52
  Concatenation Expression, 52
  Division Expression, 53
  Equals Expression, 53
  Greater or Equal Expression, 53
  Greater Than Expression, 53
  If and Only If Expression, 54
  Implies Expression, 54
  Integer Expression, 54
  Less or Equal Expression, 54
  Less Than Expression, 55
  Minus Expression, 55
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 , $39^1, 39^2, 39^3, 40^4, 40^5, 40^6$, $42^1, 42^2, 42^8, 42^9, 81^1, 81^2$, $81^3, 82^1, 82^4, 82^5$, allClockVariables (timed) , $59^1, 78^2$

allLocalVariables , $28^1, 30^2, 33^1, 33^2$, $33^3, 34^6, 38^7, 38^8, 43^9, 78^{10}$

allSTMTransitions , $59^1, 77^2, 78^3, 79^4$

allTransitions , $38^1, 41^2, 41^3, 41^4, 43^5$, $46^5, 46^7$

allTriggerDeadlineTransitions (timed) , $78^1$

allVariables , $37^1, 37^2$

alphaClockReset (timed) , $60^1, 60^2, 60^3$, $61^4, 61^5, 61^6, 61^7, 62^{10}, 62^{11}$, $62^{12}, 62^{13}, 62^{14}, 62^{15}, 62^8, 62^9$, $63^{16}, 63^{17}, 63^{18}, 63^{19}, 63^{20}$, $63^{21}, 63^{22}, 63^{23}$

alphaClockResetCallArgs (timed) , $61^1$, $61^2$

buffer , $28^1$

C (Controller) , $31^1$

clockResets (timed) , $59^1, 59^2$

compileTarget , $42^1, 42^2, 42^3$

composeControllers , $28^1, 31^2$

composeMachines , $33^1, 35^2$

composeStates , $36^1, 37^2, 40^3, 59^4$

composeStates (timed) , $82^1, 83^2$

ctrlMemory , $33^1$

deadlineEvents (timed) , $59^1, 59^2$

exitSubstates , $39^1, 40^2, 42^3, 81^4, 82^5$

Expr (Expression) , $44^1, 47^2, 48^3, 48^4$, $49^5, 49^6, 50^7, 50^8, 51^{10}, 51^9$, $52^{11}, 52^{12}, 52^{13}, 52^{14}, 52^{15}$, $52^{16}, 52^{17}, 53^{18}, 53^{19}, 53^{20}$, $53^{21}, 53^{22}, 53^{23}, 53^{24}, 53^{25}$, $54^{26}, 54^{27}, 54^{28}, 54^{29}, 54^{30}$, $54^{31}, 55^{32}, 55^{33}, 55^{34}, 55^{35}$, $55^{36}, 55^{37}, 55^{38}, 55^{39}, 56^{40}$, $56^{41}, 56^{42}, 56^{43}, 56^{44}, 57^{45}$, $57^{46}, 57^{47}, 57^{48}, 57^{49}, 57^{50}$, $58^{51}, 72^{52}, 72^{53}, 73^{54}, 73^{55}$, $74^{56}, 74^{57}, 75^{58}, 75^{59}, 76^{60}$, $76^{61}, 76^{62}, 76^{63}, 79^{64}, 79^{65}$, $79^{66}, 83^{67}, 84^{68}, 84^{69}$

flowEvents , $37^1, 37^2, 83^{3}, 83^{4}$

flowTriggerEvents , $39^1, 40^2, 82^3$

getchannelReset (timed) , $72^1, 72^2, 73^3$, $73^4, 74^5, 74^6, 75^7, 75^8, 75^{10}, 76^9$

getchannels , $36^1, 59^2$

getsetlocalChannels , $36^1, 59^2$

initialisation , $36^1, 39^2, 40^3, 59^4, 81^5, 82^6$

memoryChannels , $28^1, 28^2$
memoryDeadline (timed), 78
memoryTransition, 43
memoryTransition (timed), 78
modMemory, 28

readState, 46, 46^2, 83
renamingController, 31^1, 31^2
renamingMachine, 35^1, 35^2
renCtrlEvts, 31^1, 31^2
renStmEvts, 35^1, 35^2
requiredVariables, 28^1, 30^2, 33^3, 34^4, 34^5, 38^6, 43^7, 78^8
restrictedState, 37^1, 37^2, 83^3, 83^4

S (State), 41
Statement, 46, 84^2
StatementInContext, 48^1, 48^2, 49^3, 50^4
STM (State machine), 35^1, 48^2
stmClocks (timed), 59^1
stmMemory, 36^1
stmMemory (timed), 59^1

substates, 36^1, 36^2, 38^3, 39^4, 40^5, 40^6, 41^7, 43^8, 59^10, 59^9, 81^11, 82^12, 82^13
substatesTriggers, 41^1

T (Transition), 37^1, 39^2, 40^3, 42^4, 81^5, 82^6
transitionsFrom, 39^1, 40^2, 41^3, 42^4, 81^5, 82^6, 83^7
trigEvents, 36^1, 59^2, 59^3
Trigger, 42^1, 42^2, 42^3, 44^4, 44^5, 79^6, 79^7
triggerDeadlines (timed), 81^1, 82^2
triggerEvent, 38^1, 41^2, 77^3
triggerEvent (timed), 59^1, 60^2, 73^3, 72^4, 72^5, 72^6, 73^10, 73^8, 73^9, 74^11, 74^12, 74^13, 74^14, 75^15, 75^16, 75^17, 75^18, 76^19, 76^20, 76^21, 76^22, 76^23, 76^24

usedVariables, 46^1, 84^2

wc (timed), 65^1, 65^2, 66^3, 66^4, 66^5, 67^10, 67^11, 67^6, 67^7, 67^8, 67^9
wcArgSeq (timed), 65^1, 66^2
Index

Connection
    async, 13
    bidirec, 13
    Description, 12
    efrom, 13
    eto, 13
    from, 12
to, 12
ConnectionNode
    Description, 12
Event
    Type, 13

MachineContainer
    Description, 12
Module
    Description, 13
RCPackage
    Description, 12
Robotic Platform
    Description, 12
    Metamodel, 87
    RoboticPlatformDef, 12
    RoboticPlatformRef, 12
    Well Formedness, 22