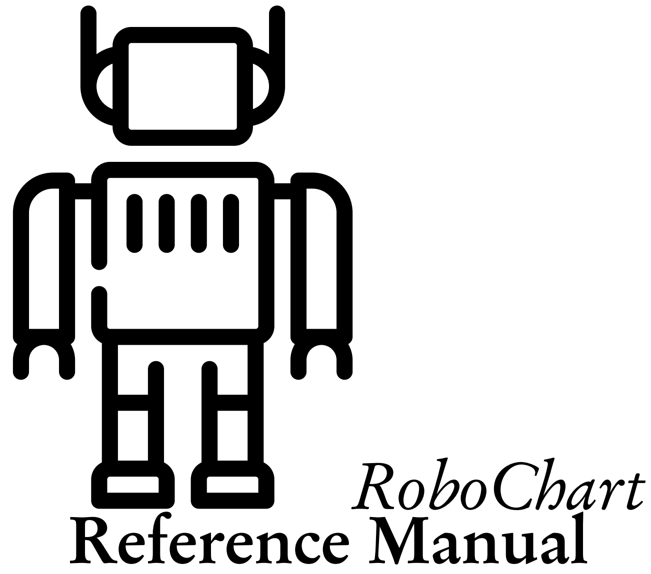


RoboChart Reference Manual

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

May 31, 2019

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 . . . 24

3.1.6 State Machines . . . 25

3.1.7 States . . . 25

3.1.8 Initial junctions . . . 25

3.1.9 Junction . . . 25

3.1.10 Final states . . . 26

3.1.11 Triggers . . . 26

3.1.12 Transitions . . . 26

3.1.13 Operations . . . 26

3.1.14 Variables . . . 26

3.2 Timed Language . . . 27

3.2.1 Timed Expressions . . . 27

3.2.2 Timed Statements . . . 27

4 Semantics 28

4.1 Detailed Semantics: Core Language . . . 29

4.1.1 Modules 29
4.1.2 Controllers 35
4.1.3 State machines 38
4.1.4 Statements 50
4.1.5 Expressions 54
4.2 Detailed Semantics: Timed Language 62
4.2.1 State machines 62
Clocks 64
Waiting Conditions 69
Trigger deadline events 78
Memory 79
4.2.2 States 80
4.2.3 Timed statements 84

5 Collections 87
5.1 Metamodel 87
5.2 Well-formedness Conditions 89
5.2.1 RCCollection 89
5.2.2 Instantiation 90
5.2.3 Event 90
5.2.4 Trigger 90
5.3 Semantics 91

6 Conclusions 94

A RoboChart diagrams - an informal overview 95
A.1 Time primitives 100

B Complete Metamodel 105

C Mathematical Toolkit 118

D Credits 122

Index of Semantic Rules 126

Index of Calls to Semantic Rules 129

List of Figures

2.1 Metamodel of RoboChart packages 13
2.2 Metamodel of RoboChart modules 13
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 15
2.6 Concrete syntax of transitions 16
2.7 Metamodel of RoboChart triggers 16
2.8 Concrete syntax of triggers 17
2.9 Metamodel of RoboChart types 19
2.10 Metamodel of RoboChart time primitives 20

5.1 Metamodel for collections: RCCollection and Event 88
5.2 Metamodel for collections: Trigger 89

A.1 Chemical Detector 96
A.2 MainController 97
A.3 MicroController 98
A.4 GasAnalysis state machine 98
A.5 Movement state machine 100
A.6 Chemical package 101
A.7 Location package 101
A.8 Square-trajectory robot 103

List of rules

Untimed semantics **29**

1 Semantics of modules . . . 29
2 Hidden Module channels . . . 29
3 Memory channels . . . 30
4 Function allEvents . . . 30
5 Function allVariables . . . 30
6 Function requiredVariables . . . 30
7 Function allLocalVariables . . . 31
8 Function allConstants . . . 31
9 Function requiredConstants . . . 31
10 Function allLocalConstants . . . 31
11 Module Memory . . . 32
12 Constants Initialisation for Controllers and Modules . . . 32
13 Composition of controllers . . . 33
14 Renaming controller . . . 33
15 Renaming controller events . . . 34
16 Buffer . . . 34
17 Single buffer . . . 35
18 Semantics of controllers . . . 35
19 Controller Memory . . . 36
20 Composition of machines . . . 37
21 Renaming state machine . . . 37
22 Renaming machine events . . . 38
23 Semantics of state machine . . . 38
24 Rename Transition Trigger Events . . . 39
25 Function states . . . 39
26 Initialisation . . . 39
27 Get and Set channels . . . 39
28 Composition of states . . . 40
29 Trigger events . . . 40
30 Get and set local channels . . . 40
31 Flow events . . . 41

32 Semantics of states . . . 41
33 Semantics of simple states . . . 42
34 Semantics of composite states . . . 43
35 Semantics of final states . . . 43
36 Synchronisation events between parent state and substates . . . 44
37 Triggers of substates . . . 44
38 Restricted semantics of states . . . 44
39 Semantics of transitions . . . 45
40 Compile target . . . 45
41 Exit substates . . . 46
42 State Machine Memory . . . 47
43 Constants Initialisation for State Machines . . . 48
44 Build Scope . . . 48
45 Semantics of triggers . . . 48
46 Semantics of triggers for memory . . . 49
47 Event for transition trigger . . . 49
48 Memory transitions . . . 49
49 Function transitionsFrom . . . 49
50 Function allTransitions . . . 50
51 Semantics of statements . . . 50
52 Read state of an expression . . . 50
53 Semantics of statements in context . . . 51
54 Function usedVariables . . . 51
55 Function usedVariables . . . 52
56 Semantics of assignment . . . 52
57 Semantics of call statement . . . 52
58 Semantics of if statements . . . 53
59 Semantics of send event statements . . . 53
60 Semantics of sequential composition . . . 54
61 Semantics of skip . . . 54
62 Semantics of actions . . . 54
63 Semantics of expressions . . . 54
64 Semantics of and expression . . . 55
65 Semantics of array expression . . . 55
66 Semantics of boolean expression . . . 55
67 Semantics of call expression . . . 56

68 Semantics of concatenation expression . . . 56
69 Semantics of not equal expression . . . 56
70 Semantics of division . . . 57
71 Semantics of equality . . . 57
72 Semantics of greater or equal expression . . . 57
73 Semantics of greater than . . . 57
74 Semantics of if and only if expression . . . 58
75 Semantics of implication . . . 58
76 Semantics of integer expression . . . 58
77 Semantics of less or equal expression . . . 58
78 Semantics of less than . . . 59
79 Semantics of minus . . . 59
80 Semantics of modulus . . . 59
81 Semantics of multiplication . . . 59
82 Semantics of arithmetic negation . . . 60
83 Semantics of logical negation . . . 60
84 Semantics of or expression . . . 60
85 Semantics of parenthesised expression . . . 60
86 Semantics of plus . . . 61
87 Semantics of range expression . . . 61
88 Semantics of sequence expression . . . 61
89 Semantics of set expression . . . 61
90 Semantics of tuple expression . . . 62

Timed Semantics **62**
91 Semantics of state machine . . . 63
92 Constants Initialisation for State Machines . . . 64
93 Build Scope . . . 64
94 allClockVariables function . . . 65
95 clockResets function . . . 65
96 stmClocks function . . . 65
97 alphaClockReset function . . . 65
98 alphaClockReset function . . . 66
99 alphaClockReset function . . . 66
100 alphaClockReset function . . . 66
101 alphaClockResetCallArgs function . . . 66

102 alphaClockReset function 67
103 alphaClockReset function 67
104 alphaClockReset function 67
105 alphaClockReset function 67
106 alphaClockReset function 68
107 alphaClockReset function 68
108 alphaClockReset function 68
109 alphaClockReset function 68
110 alphaClockReset function 69
111 alphaClockReset function 69
112 alphaClockReset function 69
113 Timed semantics of triggers 69
114 wc function 70
115 wcArgSeq function 71
116 compileWC function 72
116 compileWC function 73
116 compileWC function 74
116 compileWC function 75
116 compileWC function 76
116 compileWC function 77
117 triggerEvent function 77
118 deadlineEvents function 78
119 State-machine Memory 79
120 allDeadlineTransitions function 80
121 memoryTransition function 80
122 Memory deadline 80
123 Semantics of states 81
124 Semantics of simple states 81
125 Semantics of composite states 82
126 Semantics of trigger deadlines 83
127 Composition of states 84
128 Semantics of statements 84
129 Semantics of statement deadlines 84
130 Semantics of wait 85
131 Semantics of clock reset 85
132 Semantics of assignment 85

133 Semantics of call statement 86
1 Semantics of Collections 91
2 Broadcast Buffer 92
3 Semantics of triggers 92
4 Semantics of send event statements 93

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 [13].

UML [11] 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 [12]. 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 [8] (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 [14] and probability [17].

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 A.

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

2.1 RoboChart metamodel

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

The metamodel is automatically generated from a syntax definition. It includes a notion of a MachineContainer, which, as the same suggests, can include a number of state machines. As shown later in Figure 2.3, a controller, like an RCPackage, is also a MachineContainer. An interface groups variableLists, operations, and events.

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

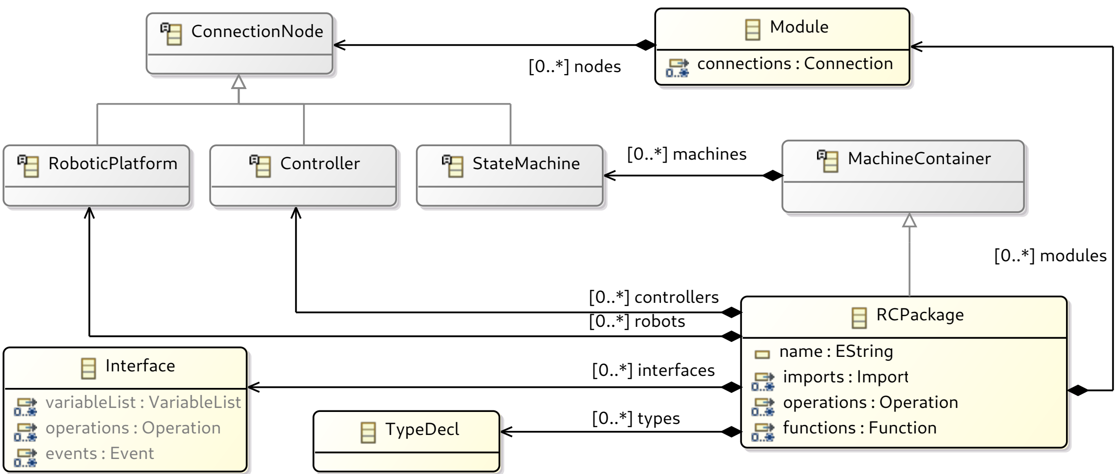


Figure 2.1: Metamodel of RoboChart packages

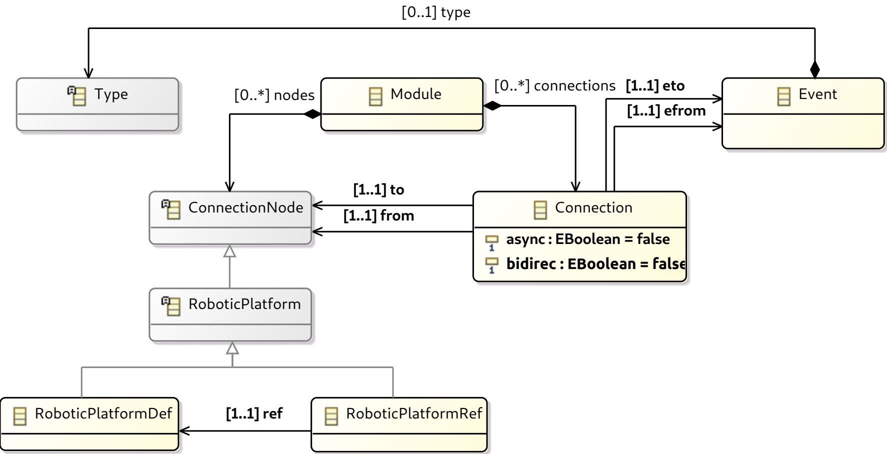


Figure 2.2: Metamodel of RoboChart modules

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.

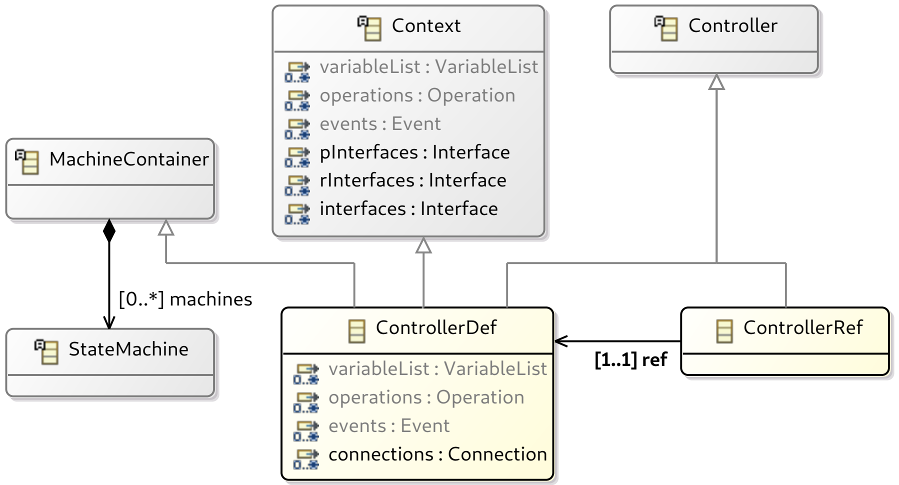


Figure 2.3: Metamodel of RoboChart controllers

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

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

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

Figure 2.4 also gives the metamodel for an Operation. It has an OperationSignature, which defines its parameters, whether it terminates and its preconditions and postconditions. If there is more than one precondition, the actual precondition of the operation is their conjunction. If there is more than one postcondition, their disjunction is the actual postcondition. An Operation 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

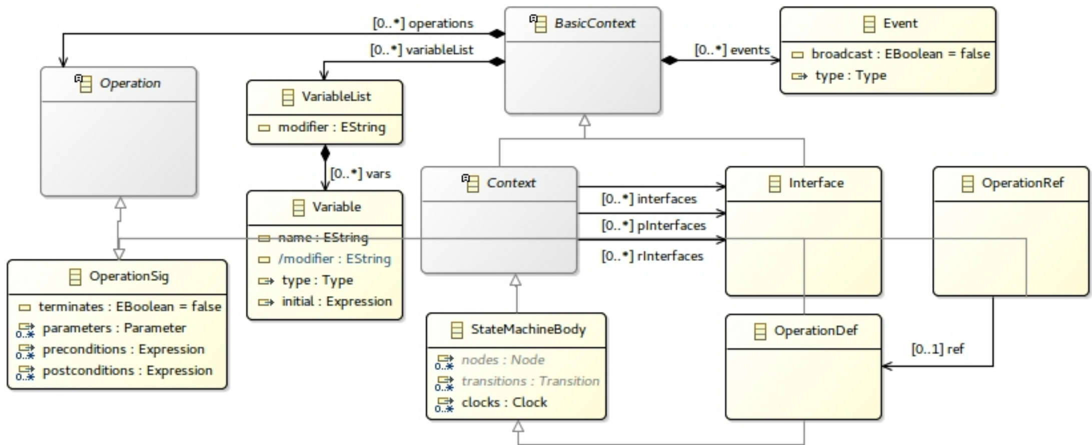


Figure 2.4: Metamodel of RoboChart context for elements and operations

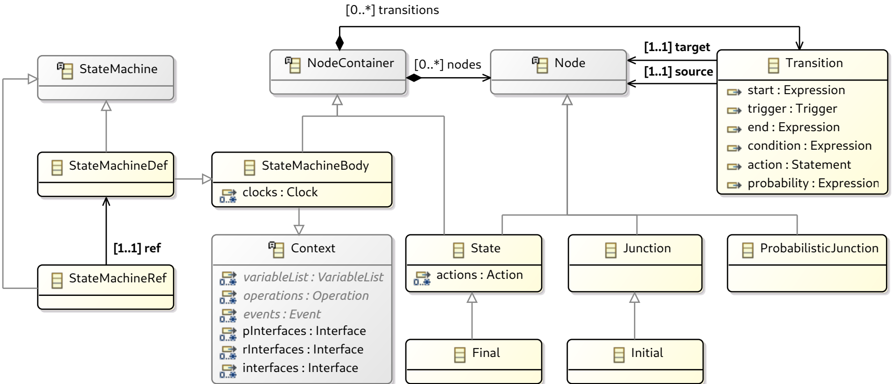


Figure 2.5: Metamodel of RoboChart state machines

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 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

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

Figure 2.6: Concrete syntax of transitions

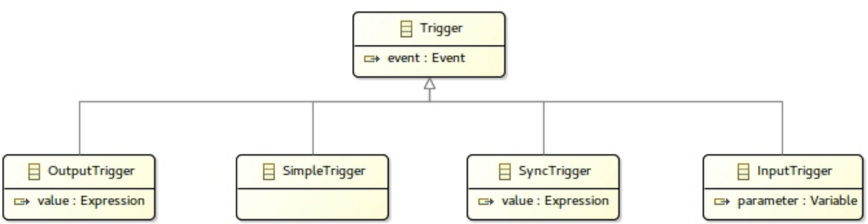


Figure 2.7: Metamodel of RoboChart triggers

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 an end deadline for a transition.

The concrete syntax of transitions is shown in Figure 2.6. The first expression is an end deadline. The syntax of triggers is described in Figure 2.8. The second 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 and a (possibly empty) list of clock resets. The concrete syntax of the different types of triggers is shown in Table 2.1.

The metamodel for Types is given in Figure 2.9. As indicated, they include references to type declarations (TypeRef), sets (SetType), and cartesian products (ProductType). A type refer-

```
(Input|Output|Sync|Simple)? ClockReset*
```

Figure 2.8: Concrete syntax of triggers

Element	**Concrete Syntax**	**Comment**
Input Trigger (I)	`Event ? Variable`	Receives any value from the event and stores it on the variable.
Output Trigger (O)	`Event ! Expression`	Sends the value of the expression through the event.
Sync Trigger (Sync)	`Event . Expression`	Synchronises on the event with the value of the expression.
Simple Trigger (S)	`Event`	Synchronises on the event.

Table 2.1: Concrete syntax of triggers.

ence refers to type declarations, which are PrimitiveTypes, given just by their names, Enumerations, records (DataType), named definitions (NamedDefinition). Table 2.2 gives the concrete syntax for the various type constructors.

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

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.

Element	Concrete Syntax	Comment
Parenthesised Type	`(T)`	
Type Reference	`N`	`N` is the name of a declared type.
Set Type	`Set(T)`	Type of sets of elements of `T`.
Sequence Type	`Seq(T)`	Type of sequences of elements of `T1` to `T2`.
Product Type	`T1 * T2`	Type of pairs whose first element has type `T1` and second element has type `T2`.
Function Type	`T1 -> T2`	Type of functions from `T1` to `T2`.
Relation Type	`T1 <-> T2`	Type of relations between `T1` and `T2`.

Table 2.2: Concrete syntax of types

<table>
<tr><th>Element</th><th>Concrete Syntax</th><th>Comment</th></tr>
<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>"..."</td><td>Quoted values</td></tr>
<tr><td>Boolean</td><td>true | false</td><td></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><e1,e2,...></td><td>Sequence with values ei.</td></tr>
<tr><td>Set</td><td>{e1,e2,...}</td><td>Set with values ei.</td></tr>
<tr><td>Set Comprehension</td><td>{x:T | P @ e}</td><td>Set containing values e, calculated from elements of type T, for which the predicate P holds.</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>Enumeration</td><td>E::c</td><td>Constant c of enuneration E.</td></tr>
<tr><td>Tuple</td><td>(|e1,e2,...|)</td><td>Tuple containing elements ei.</td></tr>
<tr><td>Array</td><td>e[i]</td><td>The i-th element of array e.</td></tr>
<tr><td>Function application</td><td>f(e1,e2,...)</td><td>Apply function f to parameters ei.</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 | P @ e</td><td>The value e calculated based on the unique x for which P holds.</td></tr>
<tr><td>Lambda expression</td><td>lambda x: T | P @ e</td><td>The anonymous function that takes values of type T for which P holds, to values e calculated based on x.</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 > e2</td><td>True if e1 is greater than e2.</td></tr>
<tr><td>Greater than or equal to</td><td>e1 >= e2</td><td>True if e1 is greater than or equal to e2.</td></tr>
<tr><td>Less than</td><td>e1 < e2</td><td>True if e1 is less than e2.</td></tr>
<tr><td>Less than or equal to</td><td>e1 <= e2</td><td>True if e1 is less than or equal to e2.</td></tr>
</table>

Table 2.3: Concrete syntax of expressions (1)

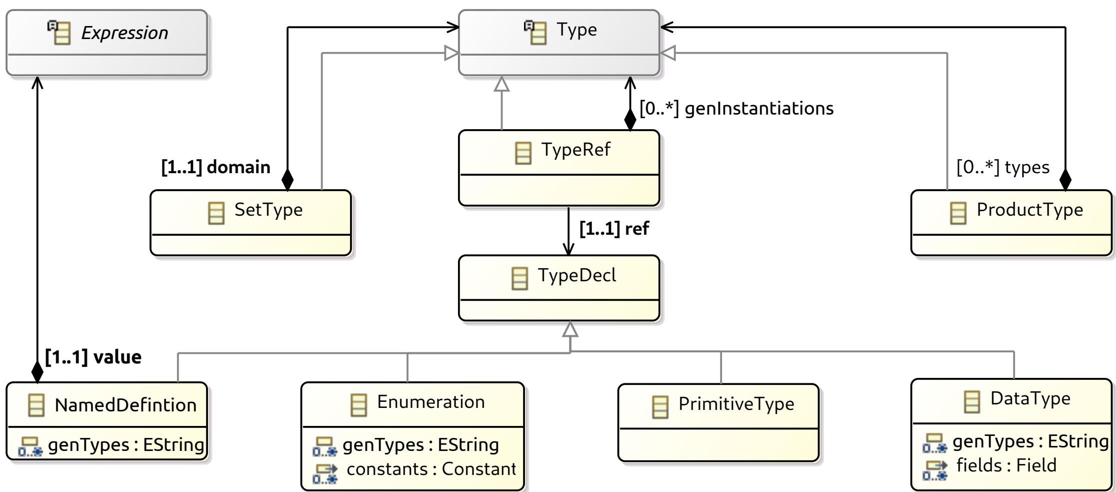


Figure 2.9: Metamodel of RoboChart types

Element	Concrete Syntax	Comment
Logical not	`not e`	True if and only if `e` is false.
Logical and	`e1 /\ e2`	True if and only if `e1` and `e2` are true.
Logical or	`e1 \/ e2`	True if and only if at least one of the expressions is true.
Logical implies	`e1 => e2`	Equivalent to `!not e1 \/ e2`.
Logical iff	`e1 iff e2`	Equivalent to `e1 => e2 /\ e2 => e1`
Universal quantifier	`forall x: T | P @ Q`	True if and only if for all elements of `T`, if `P` is true, then `Q` is true.
Existential quantifier	`exists x: T | P @ Q`	True if and only if there is an element of `T`, for which `P` is true and `Q` is true.
Uniqueness Existential quantifier	`exists1 x: T | P @ Q`	True if and only if there is a unique element of `T`, for which `P` and `Q` are true.

Table 2.4: Concrete syntax of expressions (2)

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

Table 2.5: Concrete syntax of statements

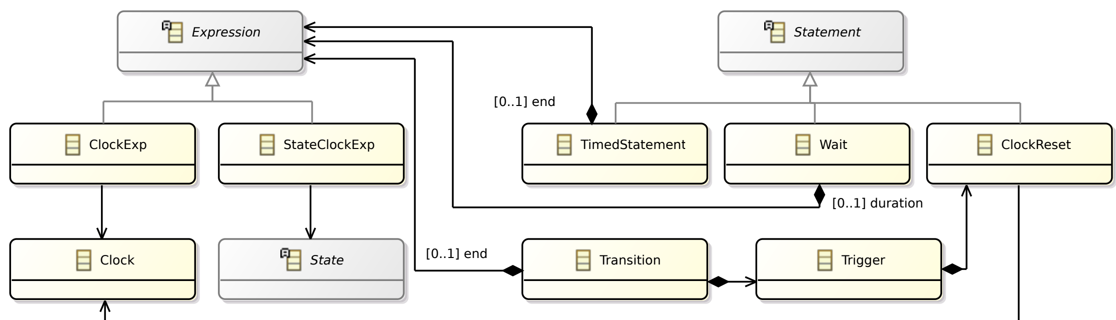


Figure 2.10: Metamodel of RoboChart time primitives

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 an expression involving a clock that yields the time elapsed since the last reset of a clock. A StateClockExpression is a sinceEntry expression. A TimedStatement defines a deadline. A Wait and ClockReset are also statements. Finally, a Transition possibly includes a deadline, and its Trigger may have a ClockReset. Table 2.6 gives the concrete syntax.

We observe that a TimedStatement defines a deadline to terminate, but not a deadline to start. The possibility to specify a deadline to start was considered, however, because statements like assignment and operation calls are immediate, only an event synchronisation or a Wait statement could introduce a delayed start. For example, consider the case where we have an

Element	Concrete Syntax	Comment
Clock Expression	`since(C)`	Expression counting elapsed time since the last reset of clock C.
State Clock Expression	`sinceEntry(S)`	Expression counting elapsed time since entry of state S.
Timed Statement	`S<{e}`	Statement S is required to terminate within e time units.
Wait	`wait(e)`	Waits for e units of time.
Nondeterministic Wait	`wait([a,b])`	Waits nondeterminstically for d units of time where $a \leq d \leq b$.
Clock Reset	`#C`	Resets clock C.
Transition Trigger deadline	`t<{e}`	Transition trigger t is required to take place within e units.

Table 2.6: Concrete syntax of time primitives

assignment of expression e to variable x, sequentially composed with a call to operation op as x=e ; op(), then a deadline to start would be imposed on the assignment, which is immediate, and thus would be redundant. Another scenario arises if, instead, we consider the example Wait(d) ; op(). A starting deadline could constrain Wait(d), however, if we were to specify this statement as an operation waitOp(), then the starting deadline on waitOp() would be satisfied immediately, whereas this would not be the case for Wait(d) ; op().

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

Chapter 3

Well-formedness Conditions

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

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

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

3.1 Core Language

3.1.1 Robotic Platforms

RP1 Robotic platforms cannot require interfaces.

RP2 Defined interfaces can only have events

RP3 The names of variables, operations, and events are unique to the platform.

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

3.1.2 Interfaces

I1 Provided and required interfaces contain only variables and operations

I2 Defined interfaces contain only variables and events

I3 Names of variables, events and operations are unique.

3.1.3 Modules

M1 A module must contain exactly one robotic platform, at least one controller, and not state machines.

M2 All variables and operations required by the module's controllers must be provided by the platform.

M3 Each event on the robotic platform and controllers of a module must have at most one connection to or from it within the module.

3.1.4 Connection

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

Cn1 Connections of a module must associate only events of the robotic platform and its controllers.

Cn2 Connections involving a robotic platform are always asynchronous.

Cn3 Connections of a controller must associate only its events and those of its state machines.

Cn4 Only events of the same type may be connected.

Cn5 Bidirectional connections of a module may only involve events of a controller which are connected by bidirectional connections within the controller.

Cn6 Non-bidirectional connections of a module may only connect to events of a controller which have a non-bidirectional connection from them within the controller.

Cn7 Non-bidirectional connections of a module may only connect from events of a controller which have a non-bidirectional connection to them within the controller.

Cn8 Non-bidirectional connections of a controller must not connect to events that a state machine uses as an output. (An event is considered to be an output if it is used in an OUPUT or SYNC trigger, or if it is used in an OUTPUT, SYNC or SIMPLE send statement.)

Cn9 Non-bidirectional connections of a controller must not connect from events that a state machine uses as an input. (An event is considered to be an input if it is used in an INPUT, SYNC or SIMPLE trigger, or if it is used in an INPUT or SYNC send statement.)

3.1.5 Controllers

C1 A controller must contain at least one state machine.

C2 Controllers cannot provide variables or operations to other controllers.

C3 Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.

C4 All variables required by the controller's state-machines must be provided or required by the controller.

C5 All operations required by the controller's state-machines must be required or defined by the controller.

C6 Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.

C7 All variables required by the controller's state-machines must be provided or required by the controller.

C8 All operations required by the controller's state-machines must be required or defined by the controller.

C9 The names of variables, operations, and events are unique to the controller.

C10 Each event on state machines and boundary of a controller must have at most one connection to or from it within the controller.

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

3.1.6 State Machines

STM1 State machines cannot have provided interfaces

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

STM3 Every state machine must have exactly one initial junction.

STM4 State machines must contain at least one state.

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

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

3.1.7 States

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

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

3.1.8 Initial junctions

IJ1 An initial junction does not have incoming transitions.

IJ2 An initial junction must have exactly one outgoing transition.

IJ3 All junction conditions apply.

3.1.9 Junction

J1 A junction must contain at least one outgoing transition.

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

J3 Transitions starting in junctions cannot have triggers.

3.1.10 Ⓕ Final states

FS1 Final states cannot be the source of transitions.

3.1.11 Triggers

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

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

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

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

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

3.1.12 → Transitions

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

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

3.1.13 O Operations

O1 All state-machine conditions apply to operation definitions.

3.1.14 X Variables

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

3.2 Timed Language

3.2.1 Timed Expressions

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

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

TE3 The state S in an expression sinceEntry(S) may only reference a state 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.

TE4 The expressions since(C) or sinceEntry(S) may only be compared with a constant expression, and only when using one of the following operators: $>$, $<$, $>=$, $<=$, $==$. A consequence of this restriction is that no expression can compare the value of two clocks as given by since(C) or sinceEntry(S).

3.2.2 Timed Statements

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

Chapter 4

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.

An overview of the semantics presented here, and a detailed explanation of many of the semantic definitions can be found in [9]. The complete definition is given in the sequel.

4.1 Detailed Semantics: Core Language

4.1.1 Modules

Rule 1. Semantics of modules

$[\![\mathsf{m} : \mathsf{Module}]\!]_{\mathcal{M}} : \mathsf{CSPProcess} =$

$$\left(\left(\left(\left(\begin{array}{l} ||| \; c : \mathsf{asyncs} \bullet \mathsf{buffer}^1(\mathsf{c}) \\ \quad [\![\mathsf{evasyncs}]\!] \\ \left(\begin{array}{l} \mathsf{modMemory}^1(\mathsf{m}) \\ \quad [\![\mathsf{memoryChannels}^1(\mathsf{m})]\!] \\ \mathsf{composeControllers}^1(\mathsf{m}, \mathsf{ctrls}, \mathsf{cons}) \end{array}\right) \end{array}\right)\right) \setminus \mathsf{hiddenModuleChannels}^1(\mathsf{m}) \right) \Theta_{\{end\}} Skip \right) \setminus \{end\}$$

where

$\mathsf{ctrls} = \langle \mathsf{x} : \mathsf{m.controllers} \rangle$

$\mathsf{cons} = \mathsf{m.connections}$

$\mathsf{asyncs} = \{\mathsf{c} : \mathsf{cons} \mid \mathsf{c.async} \wedge \{\mathsf{c.from}, \mathsf{c.to}\} \cap \mathsf{RoboticPlatform} = \emptyset\}$

$\mathsf{evasyncs} = \{\mathsf{c} : \mathsf{asyncs} \bullet \mathsf{eventId}(\mathsf{c.eto})\} \cup \{\mathsf{c} : \mathsf{asyncs} \bullet \mathsf{eventId}(\mathsf{c.efrom})\}$

Rule 2. Hidden Module channels

$\mathsf{hiddenModuleChannels}(\mathsf{m} : \mathsf{Module}) : \mathsf{ChannelSet} =$

$$\{\mathsf{c} : \mathsf{asyncs} \bullet \mathsf{eventId}(\mathsf{c.eto})\} \cup \{\mathsf{c} : \mathsf{asyncs} \bullet \mathsf{eventId}(\mathsf{c.efrom})\} \cup \mathsf{memoryChannels}^2(\mathsf{m})$$

where

$\mathsf{asyncs} = \{\mathsf{c} : \mathsf{m.connections} \mid \mathsf{c.async} \wedge \{\mathsf{c.from}, \mathsf{c.to}\} \cap \mathsf{RoboticPlatform} = \emptyset\}$

Rule 3. Memory channels

$\mathsf{memoryChannels}(m : \mathsf{Module}) : \mathsf{ChannelSet} =$

$$\{v : \mathsf{allLocalVariables}^1(m) \bullet \mathit{set}_\mathsf{vid}(v)\} \cup$$
$$\{v : \mathsf{allLocalConstants}^1(m) \bullet \mathit{set}_\mathsf{vid}(v)\} \cup$$
$$\bigcup\{c : m.\mathsf{controllers} \bullet \{v : \mathsf{requiredVariables}^1(c) \bullet \mathit{set}_\mathit{EXT}_\mathsf{vid}(v, c)\}\}$$

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

Rule 4. Function allEvents

$\mathsf{allEvents}(c : \mathsf{Context}) : \mathsf{Set}(\mathsf{Variable}) =$

$$c.\mathsf{events} \cup \bigcup\{i : c.\mathsf{interfaces} \bullet i.\mathsf{events}\}$$

Rule 5. Function allVariables

$\mathsf{allVariables}(c : \mathsf{Context}) : \mathsf{Set}(\mathsf{Variable}) =$

$$\bigcup\{l : c.\mathsf{variableList} \mid l.\mathsf{modifier} == \text{`}\mathbf{var}\text{`} \bullet l.\mathsf{vars}\} \cup$$
$$\bigcup\{i : c.\mathsf{PInterfaces} \bullet \bigcup\{l : i.\mathsf{variableList} \mid l.\mathsf{modifier} == \text{`}\mathbf{var}\text{`} \bullet l.\mathsf{vars}\}\} \cup$$
$$\bigcup\{i : c.\mathsf{RInterfaces} \bullet \bigcup\{l : i.\mathsf{variableList} \mid l.\mathsf{modifier} == \text{`}\mathbf{var}\text{`} \bullet l.\mathsf{vars}\}\}$$

Rule 6. Function requiredVariables

$\mathsf{requiredVariables}(c : \mathsf{Context}) : \mathsf{Set}(\mathsf{Variable}) =$

$$\bigcup\{i : c.\mathsf{RInterfaces} \bullet \bigcup\{l : i.\mathsf{variableList} \mid l.\mathsf{modifier} == \text{`}\mathbf{var}\text{`} \bullet l.\mathsf{vars}\}\}$$

Rule 7. Function allLocalVariables

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

$$\bigcup\{l : c.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{var} \bullet l.\text{vars}\} \cup$$
$$\bigcup\{i : c.\text{PInterfaces} \bullet \bigcup\{l : i.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{var}\text{‘} \bullet l.\text{vars}\}\}$$

Rule 8. Function allConstants

$\text{allConstants}(c : \text{Context}) : \text{Set}(\text{Variable}) =$

$$\bigcup\{l : c.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const}\text{‘} \bullet l.\text{vars}\} \cup$$
$$\bigcup\{i : c.\text{PInterfaces} \bullet \bigcup\{l : i.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const}\text{‘} \bullet l.\text{vars}\}\} \cup$$
$$\bigcup\{i : c.\text{RInterfaces} \bullet \bigcup\{l : i.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const}\text{‘} \bullet l.\text{vars}\}\}$$

Rule 9. Function requiredConstants

$\text{requiredConstants}(c : \text{Context}) : \text{Set}(\text{Variable}) =$

$$\bigcup\{i : c.\text{RInterfaces} \bullet \bigcup\{l : i.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const}\text{‘} \bullet l.\text{vars}\}\}$$

Rule 10. Function allLocalConstants

$\text{allLocalConstants}(c : \text{Context}) : \text{Set}(\text{Variable}) =$

$$\bigcup\{l : c.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const} \bullet l.\text{vars}\} \cup$$
$$\bigcup\{i : c.\text{PInterfaces} \bullet \bigcup\{l : i.\text{variableList} \mid l.\text{modifier} == \text{‘}\mathbf{const}\text{‘} \bullet l.\text{vars}\}\}$$

Rule 11. Module Memory

$\mathsf{modMemory}(\mathsf{m} : \mathsf{Module}) : \mathsf{CSPProcess} =$

$$\begin{array}{l}
\textbf{let } \mathit{Memory}(\mathsf{vars}) \widehat{=} \Box\, v : \mathsf{lvars} \bullet \mathit{set_}\mathsf{vid}(\mathsf{v})?x \rightarrow \\
\quad (\text{\textcircled{;}}\, c : \mathsf{rcontrollers}(\mathsf{v}) \bullet \mathit{set_Ext_}\mathsf{vid}(\mathsf{v}, \mathsf{c}).x \rightarrow \mathit{Skip});\ \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\
\textbf{within} \\
\quad \mathsf{constInit}^{1}(\mathsf{rp});\ \mathit{Memory}(\mathsf{varvalues})
\end{array}$$

where

$\mathsf{rp} = \mathsf{roboticPlatformDefinition}(\mathsf{m})$

$\mathsf{ctrls} = \mathsf{m.controllers}$

$\mathsf{lvars} = \mathsf{allLocalVariables}^{2}(\mathsf{rp})$

$\mathsf{vars} = \langle \mathsf{v} : \mathsf{lvars} \bullet \mathsf{name}(\mathsf{v}) \rangle$

$\mathsf{varvalues} = \langle \mathsf{v} : \mathsf{lvars} \bullet \mathsf{initial}(\mathsf{v}) \rangle$

$\mathsf{rcontrollers} = \lambda\, \mathsf{v} \bullet \{\mathsf{c} : \mathsf{ctrls} \mid \mathsf{v} \in \mathsf{requiredVariables}^{2}(\mathsf{rp})\}$

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

Rule 12. Constants Initialisation for Controllers and Modules

$\mathsf{constInit}(\mathsf{node} : \mathsf{ConnectionNode}) : \mathsf{CSPProcess} =$

$$|||\, c : \mathsf{consts} \bullet \left(\begin{array}{l}
\text{if } \mathsf{c.initial} \neq \mathsf{NULL} \text{ then} \\
\quad \mathit{set_}\mathsf{vid}(\mathsf{c})!\llbracket \mathsf{c.initial} \rrbracket_{\mathcal{E}xpr^{f}} \rightarrow \mathit{Skip} \\
\text{else} \\
\quad \mathit{set_}\mathsf{vid}(\mathsf{c})?\mathsf{name}(\mathsf{c}) \rightarrow \mathit{Skip}
\end{array} \right)$$

where

$\mathsf{consts} = \mathsf{allConstants}^{1}(\mathsf{node})$

The function *initial* picks an initial value of the appropriate type for a variable. If the variable defines an initial value, this value is used.

Rule 13. Composition of controllers

$\mathsf{composeControllers}(m : \mathsf{Module}, ctrls : \mathsf{Seq}(\mathsf{Controller}), cons : \mathsf{Set}(\mathsf{Connection})) : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\text{if}\ \#ctrls = 1 \\
\text{then} \\
\quad \mathsf{renamingController}^1(m, \text{head}\ ctrls, cons) \\
\text{else} \\
\quad \mathsf{renamingController}^2(m, \text{head}\ ctrls, cons) \\
\qquad [|connevts|] \\
\quad \mathsf{composeControllers}^2(m, \text{tail}\ ctrls, cons)
\end{array}
$$

where

$$
\begin{array}{ll}
connevts = & \mathsf{renCtrlEvts}^1(m, \text{head}\ ctrls, cons) \cap \\
& \bigcup\{c : \text{tail}\ ctrls \bullet \mathsf{renCtrlEvts}^2(m, c, cons)\}
\end{array}
$$

Rule 14. Renaming controller

$\mathsf{renamingController}(m : \mathsf{Module}, c : \mathsf{Controller}, cons : \mathsf{Set}(\mathsf{Connection})) : \mathsf{CSPProcess} =$

$$
[\![c]\!]_{\mathcal{C}'}\left[\!\!\left[
\begin{array}{l}
\{e : internalConns \bullet eventId(e.eto).\mathit{in} \leftarrow eventId(e.efrom).\mathit{out}\} \\
\cup \{e : internalConns \bullet eventId(e.eto).\mathit{out} \leftarrow eventId(e.efrom).\mathit{in}\} \\
\cup \{e : fromPlatform \bullet eventId(e.eto) \leftarrow eventId(e.efrom)\} \\
\cup \{e : toPlatform \bullet eventId(e.efrom) \leftarrow eventId(e.eto)\} \\
\cup \{v : \mathsf{requiredConstants}^1(c) \bullet \mathit{set}_vid(v) \leftarrow \mathit{set}_vid(v, m)\}
\end{array}
\right]\!\!\right]
$$

where

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

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

Rule 15. Renaming controller events

$\mathsf{renCtrlEvts}(\mathsf{m} : \mathsf{Module}, \mathsf{c} : \mathsf{Controller}, \mathsf{cons} : \mathsf{Set}(\mathsf{Connection})) : \mathsf{ChannelSet} =$

$\{\!|\, \mathsf{x} : \mathsf{internalConns} \bullet \mathsf{eventId}(\mathsf{e.efrom}) \,|\!\} \cup \{\mathit{end}\} \cup$

$\{\!|\, \mathsf{x} : \mathsf{requiredConstants}^2(\mathsf{c}) \bullet \mathit{set_}\mathsf{vid}(\mathsf{x}, \mathsf{m}) \,|\!\}$

where

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

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

Rule 16. Buffer

$\mathsf{buffer}(\mathsf{c} : \mathsf{Connection}) : \mathsf{CSPProcess} =$

---

if c.mult then

$\mathsf{singleBuffer}(\mathsf{c.efrom}, \mathsf{c.eto}) \,|||\, \mathsf{singleBuffer}(\mathsf{c.eto}, \mathsf{c.efrom})$

else

$\mathsf{singleBuffer}(\mathsf{c.efrom}, \mathsf{c.eto})$

Rule 17. Single buffer

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

$$
\begin{array}{l}
\text{if efrom.type} \neq \text{null} \\
\text{then} \\
\quad \textbf{let} \quad \mathit{Buffer}(\langle\rangle) \mathrel{\widehat=} \text{eventId}(\text{efrom}).\mathit{out}?x \rightarrow \mathit{Buffer}(\langle x \rangle) \\
\quad \qquad \mathit{Buffer}(\langle v \rangle) \mathrel{\widehat=} \text{eventId}(\text{efrom}).\mathit{out}?x \rightarrow \mathit{Buffer}(\langle x \rangle) \;\Box\; \text{eventId}(\text{eto}).\mathit{in}!v \rightarrow \mathit{Buffer}(\langle\rangle) \\
\quad \textbf{within } \mathit{Buffer}(\langle\rangle) \\
\text{else} \\
\quad \textbf{let} \quad \mathit{Buffer}(\mathit{false}) \mathrel{\widehat=} \text{eventId}(\text{efrom}).\mathit{out} \rightarrow \mathit{Buffer}(\mathit{true}) \\
\quad \qquad \mathit{Buffer}(\mathit{true}) \mathrel{\widehat=} \text{eventId}(\text{efrom}).\mathit{out} \rightarrow \mathit{Buffer}(\mathit{true}) \;\Box\; \text{eventId}(\text{eto}).\mathit{in} \rightarrow \mathit{Buffer}(\mathit{false}) \\
\quad \textbf{within } \mathit{Buffer}(\mathit{false})
\end{array}
$$

4.1.2 Controllers

Rule 18. Semantics of controllers

$[\![\text{c : ControllerDef}]\!]_{\mathcal{C}}$: CSPProcess =

$$
\left(\left(\begin{array}{l}
\text{composeMachines}^{1}(\text{c, ms, cs}) \\
\quad [\![\text{lvars} \cup \text{rvars} \cup \text{lconsts} \cup \text{rconsts}]\!] \\
\text{ctrlMemory}^{1}(\text{c})
\end{array}\right) \setminus \left(\begin{array}{l}
\text{lvars} \cup \text{rvars} \cup \\
\text{lconsts}
\end{array}\right)\right) \Theta_{\{end\}} \mathit{Skip}
$$

where

$$
\begin{array}{l}
\text{ms} = \langle \text{x : c.machines} \rangle \\
\text{cs} = \text{c.connections} \\
\text{lvars} = \{\!|\, \text{v : allLocalVariables}^{3}(\text{c}) \bullet \mathit{set}_\text{vid}(\text{v}) \,|\!\} \\
\text{rvars} = \{\!|\, \text{v : requiredVariables}^{3}(\text{c}) \bullet \mathit{set}_\mathit{Ext}_\text{vid}(\text{v}) \,|\!\} \\
\text{lconsts} = \{\!|\, \text{v : allLocalConstants}^{2}(\text{c}) \bullet \mathit{set}_\text{vid}(\text{v}) \,|\!\} \\
\text{rconsts} = \{\!|\, \text{v : requiredConstants}^{3}(\text{c}) \bullet \mathit{set}_\text{vid}(\text{v}) \,|\!\}
\end{array}
$$

The state machine synchronise with the memory controller on the *set* events of all constants of the controller (required and local), but only the *set* events of the local constants are hidden.

The *set* events of the required variables are later renamed and synchronised with the memory of the module.

Rule 19. Controller Memory

ctrlMemory(c : ControllerDef) : CSPProcess =

$$
\textbf{let } \mathit{Memory}(\mathsf{vars}) \mathrel{\widehat=} \left(\begin{array}{l}
\Box\, v : \mathsf{lvars} \bullet \mathit{set_}\mathsf{vid}(\mathsf{v})?x \to \\
\quad (\mathbin{;}\, m : \mathsf{rmachines}(\mathsf{v}) \bullet \mathit{set_Ext_}\mathsf{vid}(\mathsf{v},\mathsf{m})!x \to \mathit{Skip}); \\
\quad \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\
\Box \\
\Box\, v : \mathsf{rvars} \bullet \mathit{set_Ext_}\mathsf{vid}(\mathsf{v})?x \to \\
\quad (\mathbin{;}\, m : \mathsf{rmachines}(\mathsf{v}) \bullet \mathit{set_Ext_}\mathsf{vid}(\mathsf{v},\mathsf{m})!x \to \mathit{Skip}); \\
\quad \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}])
\end{array}\right)
$$

within

$\mathsf{constInit}^2(\mathsf{c});\ \mathit{Memory}(\mathsf{varvalues}))$

where

$\mathsf{ms} = \mathsf{c.machines}$

$\mathsf{lvars} = \mathsf{allLocalVariables}^4(\mathsf{c})$

$\mathsf{rvars} = \mathsf{requiredVariables}^4(\mathsf{c})$

$\mathsf{vars} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \bullet \mathsf{name}(\mathsf{v}) \rangle$

$\mathsf{varvalues} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \bullet \mathsf{initial}(\mathsf{v}) \rangle$

$\mathsf{rmachines} = \lambda\, \mathsf{v} \bullet \{\mathsf{m} : \mathsf{ms} \mid \mathsf{v} \in \mathsf{requiredVariables}^5(\mathsf{m})\}$

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

Rule 20. Composition of machines

$\mathsf{composeMachines}(c : \mathsf{Controller}, ms : \mathsf{Seq}(\mathsf{StateMachineDef}), cons : \mathsf{Set}(\mathsf{Connection})) : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\mathsf{if}\ \#ms = 1 \\
\mathsf{then} \\
\quad \mathsf{renamingMachine}^1(c, \mathsf{head}\ ms, cons) \\
\mathsf{else} \\
\quad \mathsf{renamingMachine}^2(c, \mathsf{head}\ ms, cons) \\
\qquad [| connevts |] \\
\quad \mathsf{composeMachines}^2(c, \mathsf{tail}\ ms, cons)
\end{array}
$$

where

$$connevts = \mathsf{renStmEvts}^1(c, \mathsf{head}\ ms, cons) \cap \bigcup\{m : \mathsf{tail}\ ms \bullet \mathsf{renStmEvts}^2(c, m, cons)\}$$

Rule 21. Renaming state machine

$\mathsf{renamingMachine}(c : \mathsf{Controller}, m : \mathsf{StateMachineDef}, cons : \mathsf{Set}(\mathsf{Connection})) : \mathsf{CSPProcess} =$

$$
[\![m]\!]_{\mathcal{STM}'}
\left[\!\!\left[
\begin{array}{l}
\{e : internalConns \bullet \mathsf{eventId}(e.eto).\mathit{in} \leftarrow \mathsf{eventId}(e.efrom).\mathit{out}\} \\
\cup \{e : internalConns \bullet \mathsf{eventId}(e.eto).\mathit{out} \leftarrow \mathsf{eventId}(e.efrom).\mathit{in}\} \\
\cup \{e : fromController \bullet \mathsf{eventId}(e.eto) \leftarrow \mathsf{eventId}(e.efrom)\} \\
\cup \{e : toController \bullet \mathsf{eventId}(e.efrom) \leftarrow \mathsf{eventId}(e.eto)\} \\
\cup \{v : \mathsf{requiredConstants}^4(m) \bullet \mathit{set_}\mathsf{vid}(v) \leftarrow \mathit{set_}\mathsf{vid}(v, c)\}
\end{array}
\right]\!\!\right]
$$

where

$$
\begin{array}{l}
internalConns = \{x : cons \bullet \{x.from, x.to\} \subseteq \mathsf{StateMachine} \wedge m \in \{x.from, x.to\}\} \\
toController = \{x : cons \bullet x.from = m \wedge x.to \in \mathsf{Controller}\} \\
fromController = \{x : cons \bullet x.to = m \wedge x.from \in \mathsf{Controller}\}
\end{array}
$$

Rule 22. Renaming machine events

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

$$\{\mathsf{x} : \mathsf{internalConns} \bullet \mathsf{eventId}(\mathsf{e.efrom})\} \cup \{\mathit{end}\} \cup$$
$$\{\mathsf{x} : \mathsf{requiredConstants}^5(\mathsf{m}) \bullet \mathit{set_}\mathsf{vid}(\mathsf{x}, \mathsf{c})\}$$

where

$$\mathsf{internalConns} = \{\mathsf{x} : \mathsf{cons} \bullet \{\mathsf{x.from}, \mathsf{x.to}\} \subseteq \mathsf{StateMachine} \wedge \mathsf{m} \in \{\mathsf{x.from}, \mathsf{x.to}\}\}$$

4.1.3 State machines

Rule 23. Semantics of state machine

$[\![\mathsf{stm} : \mathsf{StateMachineDef}]\!]_{\mathcal{STM}} : \mathsf{CSPProcess} =$

$$\left(\begin{array}{l}\left(\begin{array}{l}\mathsf{initialisation}^1(\mathsf{stm}) \\ \quad [\![\mathsf{flowevts}]\!]| \\ \mathsf{composeStates}^1(\langle \mathsf{x} : \mathsf{stm.nodes} \mid \mathsf{x} \in \mathsf{State}\rangle, \mathsf{stm})\end{array}\right) \\ \setminus \{|\mathit{enter}, \mathit{entered}, \mathit{exit}, \mathit{exited}|\} \\ |[\mathsf{getsetChannels}^1(\mathsf{stm}) \cup \mathsf{trigEvents}^1(\mathsf{stm})]| \\ \mathsf{stmMemory}^1(\mathsf{stm})\end{array}\right)$$
$$[\![\ \mathsf{renameTriggerEvents}^1(\mathsf{stm})\ ]\!]$$
$$\setminus\ \mathsf{getsetLocalChannels}^1(\mathsf{stm}) \cup \{|\mathit{internal}|\}$$
$$\Theta_{\{end\}} \mathit{Skip}$$

where

$\mathsf{flowevts} =$

$$\bigcup\{\mathsf{x} : \mathit{SIDS} \setminus \mathsf{states}^1(\mathsf{stm});\ \mathsf{y} : \mathsf{states}^2(\mathsf{stm}) \bullet \{|\mathit{enter}.\underline{\mathsf{x}}.\underline{\mathsf{y}}, \mathit{entered}.\underline{\mathsf{x}}.\underline{\mathsf{y}}, \mathit{exit}.\underline{\mathsf{x}}.\underline{\mathsf{y}}, \mathit{exited}.\underline{\mathsf{x}}.\underline{\mathsf{y}}|\}\}$$

Rule 24. Rename Transition Trigger Events

$\mathsf{renameTriggerEvents(stm : StateMachineDef) : RenamingSet} =$

$$\{e : \mathsf{allEvents}^1(m);\ tid : TIDS \bullet eventId(e).tid \leftarrow eventId(e)\}$$

Rule 25. Function states

$\mathsf{states(n : NodeContainer) : Set(State)} =$

$$x.nodes \cap (State \cup Final)$$

Rule 26. Initialisation

$\mathsf{initialisation(n : NodeContainer) : CSPProcess} =$

$$\begin{array}{l} \text{if } (\#n.nodes > 0) \text{ then} \\ \quad [\![\iota t : Transition \mid t.source = (\iota x : n.nodes \mid x \in Initial), n, true]\!]^{Skip,Skip}_{\mathcal{T}'} \\ \text{else } \mathit{Skip} \end{array}$$

Rule 27. Get and Set channels

$\mathsf{getsetChannels(s : StateMachineDef) : ChannelSet} =$

$$\{\!|v : \mathsf{allVariables}^1(s) \bullet \mathit{get}_vid(v)|\!\} \cup \{\!|v : \mathsf{allVariables}^2(s) \bullet \mathit{set}_vid(v)|\!\} \cup$$
$$\{\!|v : \mathsf{allConstants}^2(s) \bullet \mathit{get}_vid(v)|\!\} \cup \{\!|v : \mathsf{allConstants}^3(s) \bullet \mathit{set}_vid(v)|\!\}$$

Rule 28. Composition of states

$composeStates(ss : \mathbf{seq}\ State, p : NodeContainer) : CSPProcess =$

$$
\begin{array}{l}
\text{if } \#ss = 1 \\
\text{then} \\
\quad restrictedState^1(p, head\ ss) \\
\text{else} \\
\quad \left(\begin{array}{l} restrictedState^2(p, head\ ss) \\ \quad [|cflowevts|] \\ composeStates^2(tail\ ss, p) \end{array} \right) \setminus cflowevts
\end{array}
$$

where

$$cflowevts = flowEvents^1(head\ ss, p) \cap \bigcup\{x : tail\ ss \bullet flowEvents^2(x, p)\}$$

Rule 29. Trigger events

$trigEvents(s : StateMachineDef) : ChannelSet =$

$$\{\!| t : allTransitions^1(s) \bullet triggerEvent^1(t.trigger, id(t)) |\!\}$$

Rule 30. Get and set local channels

$getsetLocalChannels(s : StateMachineDef) : ChannelSet =$

$$
\begin{array}{l}
\{\!| v : allVariables^3(s) \bullet get_vid(v) |\!\} \cup \{\!| v : allLocalVariables^5(s) \bullet set_vid(v) |\!\} \cup \\
\{\!| v : allConstants^4(s) \bullet get_vid(v) |\!\} \cup \{\!| v : allLocalConstants^3(s) \bullet set_vid(v) |\!\}
\end{array}
$$

The state machine hides all *get* events for both constants and variables, but only hides the *set* events for local constants and variables.

Rule 31. Flow events

$\mathsf{flowEvents}(\mathsf{s} : \mathsf{State}, \mathsf{p} : \mathsf{NodeContainer}) : \mathsf{ChannelSet} =$

$$\bigcup \{x : \mathsf{states}^3(p);\ y : \{id(s)\} \bullet \{\!|$$
$$enter.y.x, entered.y.x, exit.y.x, exited.y.x,$$
$$enter.x.y, entered.x.y, exit.x.y, exited.x.y,$$
$$|\!\}\}$$

Rule 32. Semantics of states

$[\![s : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{CSPProcess} =$

This function is split in multiple rules according to the type of states.

Rule 33. Semantics of simple states

$[\![\mathsf{s} : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{CSPProcess} =$

let

$Inactive \mathrel{\widehat=} enter?o : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow Activating(o)$

$Activating(o) \mathrel{\widehat=} \underbrace{[\![\mathsf{s.entry}]\!]}_{\mathcal{A}ction^{1}}; \mathsf{initialisation}^{2}(\mathsf{s}); entered.o.\mathsf{id}(\mathsf{s}) \rightarrow$

$(\underbrace{[\![\mathsf{s.during}]\!]}_{\mathcal{A}ction^{2}}; Stop) \triangle$

$$\left(\begin{array}{l} \Box\, t : \mathsf{transitionsFrom}^{1}(\mathsf{s}) \bullet \underbrace{[\![t, \mathsf{s}, \mathsf{false}]\!]}_{\mathcal{T}^{2}}^{Inactive, Activating} \\ \Box \\ \Box\, e : \mathsf{Event} \bullet \begin{array}{l} \mathsf{if}(\mathsf{e.type} == \mathsf{null}) \\ \mathsf{then}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids}?d \rightarrow \mathsf{exit};\ Inactive \\ \mathsf{else}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids}?d?y \rightarrow \mathsf{exit};\ Inactive \end{array} \end{array}\right)$$

within

$Inactive$

where

$\#\mathsf{states}^{4}(\mathsf{s}) = 0$

$\mathsf{sids} = SIDS \setminus \{\mathsf{id}(\mathsf{s})\}$

$\mathsf{exit} = exit?as : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow \underbrace{[\![\mathsf{s.exit}]\!]}_{\mathcal{A}ction^{3}}; exited.as.\mathsf{id}(\mathsf{s}) \rightarrow Skip$

$\mathsf{tids} = TIDS \setminus \mathsf{tIDS}(\mathsf{s})$

Rule 34. Semantics of composite states

$[\![\mathsf{s} : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{CSPProcess} =$

let

$$Inactive \mathrel{\widehat=} enter?o : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow Activating(o)$$

$$Activating(o) \mathrel{\widehat=} \underbrace{[\![\mathsf{s.entry}]\!]}_{\mathcal{A}ction^4} ;\ \mathsf{initialisation}^3(\mathsf{s});\ entered.o.\mathsf{id}(\mathsf{s}) \rightarrow$$

$$(\underbrace{[\![\mathsf{s.during}]\!]}_{\mathcal{A}ction^5} ;\ Stop) \bigtriangleup$$

$$\left(\begin{array}{l} \Box\, t : \mathsf{transitionsFrom}^2(\mathsf{s}) \bullet [\![t, \mathsf{s}, \mathsf{false}]\!]_{\mathcal{T}^3}^{Inactive, Activating} \\ \Box \\ \Box\, e : \mathsf{Event} \bullet \quad \mathsf{if}(\mathsf{e.type} == \mathsf{null}) \\ \qquad\qquad\qquad \mathsf{then\ eventId}(\mathsf{e})?x : \mathsf{tids}?d \rightarrow \mathsf{exit};\ Inactive \\ \qquad\qquad\qquad \mathsf{else\ eventId}(\mathsf{e})?x : \mathsf{tids}?d?y \rightarrow \mathsf{exit};\ Inactive \end{array} \right)$$

within

$$(Inactive\ [\![\ \mathsf{flowtrigevts}\ ]\!]\ \mathsf{composeStates}^3(\langle \mathsf{x} : \mathsf{states}^5(\mathsf{s}) \rangle, \mathsf{s})) \setminus \mathsf{flowevts}$$

where

$$\#\mathsf{states}^6(\mathsf{s}) > 0$$

$$\mathsf{flowevts} = \bigcup \{ \mathsf{x} : SIDS \setminus \mathsf{states}^7(\mathsf{s});\ \mathsf{y} : \mathsf{states}^8(\mathsf{s}) \bullet \{\!| enter.\mathsf{x}.\mathsf{y}, entered.\mathsf{x}.\mathsf{y}, exit.\mathsf{x}.\mathsf{y}, exited.\mathsf{x}.\mathsf{y} |\!\} \}$$

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

$$\mathsf{sids} = SIDS \setminus \{ \mathsf{id}(\mathsf{s}) \}$$

$$\mathsf{exit} = exit?as : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow \mathsf{exitSubstates}^1(\mathsf{s});\ \underbrace{[\![\mathsf{s.exit}]\!]}_{\mathcal{A}ction^6} ;\ exited.as.\mathsf{id}(\mathsf{s}) \rightarrow Skip$$

$$\mathsf{tids} = TIDS \setminus \mathsf{tIDS}(\mathsf{s})$$

Rule 35. Semantics of final states

$[\![\mathsf{s} : \mathsf{Final}]\!]_{\mathcal{S}} : \mathsf{CSPProcess} =$

$$enter?x : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow entered.x.\mathsf{id}(\mathsf{s}) \rightarrow \left(\begin{array}{l} \mathsf{if}\ (\mathsf{parent}(\mathsf{s}) \in \mathsf{StateMachine}) \\ \mathsf{then}\ end \rightarrow Skip \\ \mathsf{else}\ Stop \end{array} \right)$$

Rule 36. Synchronisation events between parent state and substates

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

$$(\{\!|e : Event;\ t : TIDS \bullet e.t|\!\} \setminus \mathsf{substatesTriggers}^{1}(\mathsf{s})) \cup \bigcup\{\mathsf{x} : SIDS \setminus \mathsf{states}^{9}(\mathsf{s});\ \mathsf{y} : \mathsf{states}^{10}(\mathsf{s}) \bullet \{\!|enter.\underline{\mathsf{x}}.\underline{\mathsf{y}}, entered.\underline{\mathsf{x}}.\underline{\mathsf{y}}, exit.\underline{\mathsf{x}}.\underline{\mathsf{y}}, exited.\underline{\mathsf{x}}.\underline{\mathsf{y}}|\!\}\}$$

Rule 37. Triggers of substates

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

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

Rule 38. Restricted semantics of states

$\mathsf{restrictedState(p : NodeContainer, s : State) : CSPProcess} =$

$$[\![\mathsf{s}]\!]_{\mathcal{S}'} \ [\![\ \mathsf{all_other_transitions_S} \setminus \mathsf{all_transitions_PS}\ ]\!]\ Skip$$

where

$$\mathsf{tidsfromwithin} = \{\mathsf{t} : \mathsf{transitionsFrom}^{3}(\mathsf{s}) \cup \mathsf{allTransitions}^{3}(\mathsf{s}) \bullet \mathsf{id(t)}\}$$

$$\mathsf{all_other_transitions_S} = \{\mathsf{e : Event;\ tid : TIDS} \setminus \mathsf{tidsfromwithin} \bullet \mathsf{eventId(e).tid}\}$$

$$\mathsf{all_transitions_PS} = \{\mathsf{e : Event;\ tid : TIDS} \bullet \mathsf{eventId(e).tid}\} \setminus \{\mathsf{t} : \mathsf{allTransitions}^{4}(\mathsf{p}) \bullet \mathsf{eventId(t.trigger.event).id(t)}\}$$

Rule 39. Semantics of transitions

$[\![\mathsf{t : Transition, origin : NodeContainer, initial : boolean}]\!]_{\mathcal{T}}^{P,Q} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\mathsf{if\ src} \in \mathsf{State} \\
\quad [\![\mathsf{t.trigger}]\!]_{\mathit{Trigger}^{1}}^{\mathsf{id(t)}} \,;\, \mathit{exit}.\mathsf{id(src).id(src)} \rightarrow \mathsf{exitSubstates}^{2}(\mathsf{src});\ [\![\mathsf{src.exit}]\!]_{\mathit{Action}^{7}} \,; \\
\quad \mathit{exited}.\mathsf{id(src).id(src)} \rightarrow [\![\mathsf{t.action}]\!]_{\mathit{Action}^{8}} \,;\ \mathsf{compileTarget}^{1}(\mathsf{tgt, src, false})^{P,Q} \\
\mathsf{else\ if\ src} \in \mathsf{Initial} \\
\quad \mathit{internal}.\mathsf{id(t)} \rightarrow [\![\mathsf{t.action}]\!]_{\mathit{Action}^{9}} \,;\ \mathsf{compileTarget}^{2}(\mathsf{tgt, parent(src), true})^{P,Q} \\
\mathsf{else\ if\ src} \in \mathsf{Junction} \\
\quad \mathit{internal}.\mathsf{id(t)} \rightarrow [\![\mathsf{t.action}]\!]_{\mathit{Action}^{10}} \,;\ \mathsf{compileTarget}^{3}(\mathsf{tgt, origin, initial})^{P,Q}
\end{array}
$$

where

$\mathsf{src} = \mathsf{t.source}$

$\mathsf{tgt} = \mathsf{t.target}$

Rule 40. Compile target

$\mathsf{compileTarget(tgt : Node, o : NodeContainer, i : boolean)}^{P,Q} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\mathsf{if\ (tgt} \in \mathsf{State)\ then} \\
\quad \mathsf{if\ (tgt = o)\ then}\ \mathit{enter}.\mathsf{id(o).id(tgt)} \rightarrow Q \\
\quad \mathsf{else}\ \mathit{enter}.\mathsf{id(o).id(tgt)} \rightarrow \mathit{entered}.\mathsf{id(o).id(tgt)} \rightarrow \left(\begin{array}{l} \mathsf{if\ (i)\ then}\ \mathit{Skip} \\ \mathsf{else}\ P \end{array} \right) \\
\mathsf{else\ if\ (tgt} \in \mathsf{Junction)\ then} \\
\quad \Box \{\mathsf{t : transitionsFrom}^{4}(\mathsf{tgt}) \bullet [\![\mathsf{t, o, i}]\!]_{\mathcal{T}^{4}}^{P,Q}\}
\end{array}
$$

Rule 41. Exit substates

exitSubstates(s : NodeContainer) : CSPProcess =

---

if #states(s) > 0 then

$exit.\mathrm{id}(s)?z : \{x : \mathrm{states}^{11}(s) \bullet \mathrm{id}(x)\} \rightarrow exited.\mathrm{id}(s).z \rightarrow Skip$

else

$Skip$

Rule 42. State Machine Memory

$\mathsf{stmMemory}(\mathsf{stm} : \mathsf{StateMachineDef}) : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\textbf{let } Memory(\mathsf{vars}) \mathrel{\widehat=} \\
\left(\begin{array}{l}
\Box\, v : \mathsf{lvars} \bullet \left(\begin{array}{l}
get_\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{vars}) \\
\Box \\
set_\mathsf{vid}(\mathsf{v})?x \to Memory(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}])
\end{array}\right) \\
\Box \\
\Box\, v : \mathsf{rvars} \bullet \\
\quad \left(\begin{array}{l}
get_\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{vars}) \\
\Box \\
set_\mathsf{vid}(\mathsf{v})?x \to Memory(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\
\Box \\
set_Ext_\mathsf{vid}(\mathsf{v})?x \to Memory(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}])
\end{array}\right) \\
\Box \\
\Box\, v : \mathsf{allConstants}^{5}(\mathsf{stm}) \bullet get_\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{vars}) \\
\Box \\
\Box\, t : \mathsf{allTransitions}^{5}(\mathsf{stm}) \bullet \mathsf{memoryTransition}^{1}(\mathsf{t});\ Memory(\mathsf{vars})
\end{array}\right) \\
\textbf{within} \\
\quad \mathsf{constInitSTM}^{1}(\mathsf{consts}, \mathsf{stm}, Memory(\mathsf{varvalues}))
\end{array}
$$

where

$\mathsf{rvars} = \mathsf{requiredVariables}^{6}(\mathsf{stm})$

$\mathsf{lvars} = \mathsf{allLocalVariables}^{6}(\mathsf{stm})$

$\mathsf{consts} = \langle \mathsf{v} : \mathsf{allConstants}^{6}(\mathsf{stm}) \bullet \mathsf{v} \rangle$

$\mathsf{vars} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \bullet \mathsf{name}(\mathsf{v}) \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name}(\mathsf{v}) \rangle$

$\mathsf{varvalues} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \bullet \mathsf{initial}(\mathsf{v}) \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name}(\mathsf{v}) \rangle$

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

Rule 43. Constants Initialisation for State Machines

$$\mathsf{constInitSTM}(cs : \mathsf{Seq}_1(\mathsf{Variable}), stm : \mathsf{StateMachineDef}, P : \mathsf{CSPProcess}) : \mathsf{CSPProcess} =$$

$$\mathsf{buildScope}^1(undefc, stm, \left(\begin{array}{l}\textbf{let}\ c : defs \bullet name(c) = [\![c.initial]\!]_{\mathsf{Expr}^2} \\ \textbf{within}\ P\end{array}\right))$$

where

$$defc = \langle x : consts \mid x.initial \neq null \rangle$$

$$undefc = \langle x \bullet consts \mid x.initial = null \rangle$$

Rule 44. Build Scope

$$\mathsf{buildScope}(cs : \mathsf{Seq}_1(\mathsf{Variable}), ctx : \mathsf{ConnectionNode}, P : \mathsf{CSPProcess}) : \mathsf{CSPProcess} =$$

$$\begin{array}{l} \text{if } (\text{head } cs).initial == NULL \text{ then} \\ \quad set_vid(\text{head } cs, ctx)?name(\text{head } cs) \rightarrow \left(\begin{array}{l}\text{if } \#cs == 1 \text{ then } P \\ \text{else } \mathsf{buildScope}^2(\text{tail } cs, ctx, P)\end{array}\right) \\ \text{else} \\ \quad \left(\begin{array}{l}\text{if } \#cs == 1 \text{ then } P \\ \text{else } \mathsf{buildScope}^3(\text{tail } cs, ctx, P)\end{array}\right)\end{array}$$

Rule 45. Semantics of triggers

$$[\![t : \mathsf{Trigger}]\!]^{tid}_{\mathit{Trigger}} : \mathsf{CSPProcess} =$$

$$\begin{array}{l} \text{if } t.event.type \neq null \\ \quad eventId(t.event).tid.\mathit{in}?x__ \rightarrow set_vid(t.parameter)!x__ \rightarrow \mathit{Skip} \\ \text{else} \\ \quad eventId(t.event).tid.\mathit{in} \rightarrow \mathit{Skip}\end{array}$$

Rule 46. Semantics of triggers for memory

triggerForMemory(t : Trigger, tid : TIDS) : CSPProcess =

if t.event.type ≠ null

  eventId(t.event).tid.$in?x__ \rightarrow Skip$

else

  eventId(t.event).tid.$in \rightarrow Skip$

Rule 47. Event for transition trigger

triggerEvent(t : Trigger, tid : TIDS) : CSPEvent =

eventId(t.event).tid

Rule 48. Memory transitions

memoryTransition(t : Transition) : CSPProcess =

if (t.condition ≠ null) then

  ($[\![$t.condition$]\!]_{\mathcal{E}xpr^3}$) & triggerForMemory[1](t.trigger, id(t))

else

  triggerForMemory[2](t.trigger, id(t))

Rule 49. Function transitionsFrom

transitionsFrom(s : Node) : Set(Transition) =

{t : parent(s).transitions | t.source = s • t}

Rule 50. Function allTransitions

$\mathsf{allTransitions}(s : \mathsf{NodeContainer}) : \mathsf{Set}(\mathsf{Transition}) =$

$s.\mathsf{transitions} \cup \bigcup\{x : s.\mathsf{nodes} \mid s \in \mathsf{State} \bullet \mathsf{allTransitions}^6(x)\}$

4.1.4 Statements

Rule 51. Semantics of statements

$[\![s : \mathsf{Statement}]\!]_{Statement} : \mathsf{CSPProcess} =$

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

The semantics of statements, in general, has the format

$$get_x_1?x_1 \rightarrow \ldots \rightarrow get_x_n?x_n \rightarrow P$$

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 52. Read state of an expression

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

$\mathsf{if}\ (\#vs = 0)\ \mathsf{then}$
$\quad P$
else
$\quad get_\mathsf{vid}(\mathsf{head}\ vs)?(\mathsf{head}\ vs).\mathsf{name} \rightarrow \mathsf{readState}^1(\mathsf{tail}\ vs, P)$

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

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

Rule 53. Semantics of statements in context

$[\![s : \mathsf{Statement}]\!]_{StatementInContext} : \mathsf{CSPProcess} =$

$\mathsf{readState}^2(\mathsf{usedVariables}^1(s), [\![s]\!]_{\mathsf{Statement}^1})$

Rule 54. Function usedVariables

$\mathsf{usedVariables}(s : \mathsf{Statement}) : \mathsf{Set}(\mathsf{Variable}) =$

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

Rule 55. Function usedVariables

$\mathsf{usedVariables}(\mathsf{e} : \mathsf{Expression}) : \mathsf{Set}(\mathsf{Variable}) =$

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

Rule 56. Semantics of assignment

$[\![\mathsf{s} : \mathsf{Assignment}]\!]_{Statement} : \mathsf{CSPProcess} =$

$$set_\mathsf{vid}(\mathsf{s.left})![\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{4}} \rightarrow Skip$$

Rule 57. Semantics of call statement

$[\![\mathsf{s} : \mathsf{Call}]\!]_{Statement} : \mathsf{CSPProcess} =$

$$\mathsf{op.name}Call \rightarrow \mathsf{body};\ \mathsf{op.name}Ret \rightarrow Skip$$

where

$$\mathsf{op} = \mathsf{s.operation}$$
$$\mathsf{opdef} = \mathsf{findOperationDefinition}(\mathsf{op})$$
$$\mathsf{body} = \begin{pmatrix} \mathsf{if}\ (\mathsf{opdef} = \mathsf{null})\ \mathsf{then} \\ \quad (Skip \sqcap Stop) \\ \mathsf{else} \\ \quad [\![\mathsf{opdef}]\!]_{\mathcal{STM}^{2}} (\{\mathsf{x} : \mathsf{s.args} \bullet [\![\mathsf{x}]\!]_{\mathcal{E}xpr^{5}}\}) \end{pmatrix};$$

Rule 58. Semantics of if statements

$[\![\text{s : IfStatement}]\!]_{Statement} : \text{CSPProcess} =$

$$\left(\begin{array}{l} \textbf{if } [\![\text{s.expression}]\!]_{Expr^{6}} \\ \textbf{then } [\![\text{s.then}]\!]_{StatementInContext^{1}} \\ \textbf{else } \text{if } (\text{s.else} \neq \text{null}) \text{ then } [\![\text{s.else}]\!]_{StatementInContext^{2}} \text{ else } Skip \end{array}\right)$$

Rule 59. Semantics of send event statements

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

$$\begin{array}{l} \text{if (type} = \text{INPUT) then} \\ \quad \text{eventId(event)}.in?\text{par.name} \rightarrow set_\text{vid(par)}!\text{par.name} \rightarrow Skip \\ \text{if (type} = \text{OUTPUT) then} \\ \quad \text{eventId(event)}.out![\![\text{value}]\!]_{Expr^{7}} \rightarrow Skip \\ \text{if (type} = \text{SIMPLE) then} \\ \quad \text{eventId(event)}.out \rightarrow Skip \\ \text{else} \\ \quad \text{eventId(event)}.out.[\![\text{value}]\!]_{Expr^{8}} \rightarrow Skip \end{array}$$

where

$$\begin{array}{l} \text{type} = \text{s.trigger.type} \\ \text{event} = \text{s.trigger.event} \\ \text{value} = \text{s.trigger.value} \\ \text{par} = \text{s.trigger.parameter} \end{array}$$

Rule 60. Semantics of sequential composition

$[\![s : \mathsf{SeqStatement}]\!]_{\mathit{Statement}} : \mathsf{CSPProcess} =$

$⨟ \{x : \mathsf{s.statements} \bullet [\![x]\!]_{\mathit{StatementInContext}^{3}}\}$

Rule 61. Semantics of skip

$[\![s : \mathsf{Skip}]\!]_{\mathit{Statement}} : \mathsf{CSPProcess} =$

Skip

Rule 62. Semantics of actions

$[\![a : \mathsf{Action}]\!]_{\mathit{Action}} : \mathsf{CSPProcess} =$

$[\![\mathsf{a.action}]\!]_{\mathit{StatementInContext}^{4}}$

4.1.5 Expressions

Rule 63. Semantics of expressions

$[\![s : \mathsf{Expression}]\!]_{\mathit{Expr}} : \mathsf{CSPExpression} =$

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

Rule 64. Semantics of and expression

$[\![\mathsf{s : And}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{9}} \wedge [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{10}}$

Rule 65. Semantics of array expression

$[\![\mathsf{s : ArrayExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.value}]\!]_{\mathcal{E}xpr^{11}} (\{ \mathsf{p : s.parameters} \bullet [\![\mathsf{p}]\!]_{\mathcal{E}xpr^{12}} \})$

Rule 66. Semantics of boolean expression

$[\![\mathsf{s : BooleanExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

if (s.value = TRUE) then *true* else *false*

Rule 67. Semantics of call expression

$[\![\mathsf{s : CallExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

if (name = '*size*' $\wedge$ (head s.args **has type** SetType)) then

$\quad$ *card*($[\![\text{head s.args}]\!]_{\mathcal{E}xpr^{13}}$)

else if (name = '*size*' $\wedge$ (head s.args **has type** SeqType)) then

$\quad$ *length*($[\![\text{head s.args}]\!]_{\mathcal{E}xpr^{14}}$)

else

$\quad$ name($\{ a : s.args \bullet [\![a]\!]_{\mathcal{E}xpr^{15}} \}$)

where

name = s.function.name

Rule 68. Semantics of concatenation expression

$[\![\mathsf{s : Cat}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![s.left]\!]_{\mathcal{E}xpr^{16}} \frown [\![s.right]\!]_{\mathcal{E}xpr^{17}}$$

Rule 69. Semantics of not equal expression

$[\![\mathsf{s : Different}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![s.left]\!]_{\mathcal{E}xpr^{18}} \neq [\![s.right]\!]_{\mathcal{E}xpr^{19}}$$

Rule 70. Semantics of division

$[\![\mathsf{s} : \mathsf{Div}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{20}} / [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{21}}$

Rule 71. Semantics of equality

$[\![\mathsf{s} : \mathsf{Equals}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{22}} = [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{23}}$

Rule 72. Semantics of greater or equal expression

$[\![\mathsf{s} : \mathsf{GreaterOrEqual}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{24}} \geq [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{25}}$

Rule 73. Semantics of greater than

$[\![\mathsf{s} : \mathsf{GreaterThan}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{26}} > [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{27}}$

Rule 74. Semantics of if and only if expression

$[\![\text{s : Iff}]\!]_{\mathcal{E}xpr} : \text{CSPExpression} =$

$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{28}} \Leftrightarrow [\![\text{s.right}]\!]_{\mathcal{E}xpr^{29}}$

Rule 75. Semantics of implication

$[\![\text{s : Implies}]\!]_{\mathcal{E}xpr} : \text{CSPExpression} =$

$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{30}} \Rightarrow [\![\text{s.right}]\!]_{\mathcal{E}xpr^{31}}$

Rule 76. Semantics of integer expression

$[\![\text{s : IntegerExp}]\!]_{\mathcal{E}xpr} : \text{CSPExpression} =$

s.value

Rule 77. Semantics of less or equal expression

$[\![\text{s : LessOrEqual}]\!]_{\mathcal{E}xpr} : \text{CSPExpression} =$

$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{32}} \leq [\![\text{s.right}]\!]_{\mathcal{E}xpr^{33}}$

Rule 78. Semantics of less than

$[\![\mathsf{s} : \mathsf{LessThan}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{34}} < [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{35}}$$

Rule 79. Semantics of minus

$[\![\mathsf{s} : \mathsf{Minus}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{36}} - [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{37}}$$

Rule 80. Semantics of modulus

$[\![\mathsf{s} : \mathsf{Modulus}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{38}} \ \mathsf{mod} \ [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{39}}$$

Rule 81. Semantics of multiplication

$[\![\mathsf{s} : \mathsf{Mult}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{40}} \times [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{41}}$$

Rule 82. Semantics of arithmetic negation

$[\![\mathsf{s : Neg}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$- [\![\mathsf{s.exp}]\!]_{\mathcal{E}xpr^{42}}$

Rule 83. Semantics of logical negation

$[\![\mathsf{s : Not}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$\neg [\![\mathsf{s.exp}]\!]_{\mathcal{E}xpr^{43}}$

Rule 84. Semantics of or expression

$[\![\mathsf{s : Or}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{44}} \vee [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{45}}$

Rule 85. Semantics of parenthesised expression

$[\![\mathsf{s : ParExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$([\![\mathsf{s.exp}]\!]_{\mathcal{E}xpr^{46}})$

where

Rule 86. Semantics of plus

$[\![\mathsf{s : Plus}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^{47}} + [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{48}}$$

Rule 87. Semantics of range expression

$[\![\mathsf{s : RangeExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$\{x : \mathbb{N} \mid [\![\mathsf{s.lrange}]\!]_{\mathcal{E}xpr^{49}} \ \mathsf{rel1}\ x \wedge x\ \mathsf{rel2}\ [\![\mathsf{s.rrange}]\!]_{\mathcal{E}xpr^{50}} \}$$

where

$\mathsf{rel1} = \mathsf{if}\ (\mathsf{e.linterval} = [\,)\ \mathsf{then}\ \leq\ \mathsf{else}\ <$

$\mathsf{rel2} = \mathsf{if}\ (\mathsf{e.linterval} =]\,)\ \mathsf{then}\ \geq\ \mathsf{else}\ >$

Rule 88. Semantics of sequence expression

$[\![\mathsf{s : SeqExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$\langle \{ \mathsf{x : s.values} \bullet [\![\mathsf{x}]\!]_{\mathcal{E}xpr^{51}} \} \rangle$$

Rule 89. Semantics of set expression

$[\![\mathsf{s : SetExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$\{ \{ \mathsf{x : s.values} \bullet [\![\mathsf{x}]\!]_{\mathcal{E}xpr^{52}} \} \}$$

Rule 90. Semantics of tuple expression
$[\![s : \mathsf{TupleExp}]\!]_{\mathcal{E}xpr} : \mathsf{CSPExpression} =$

$$(\{x : s.values \bullet [\![x]\!]_{\mathcal{E}xpr^{53}}\})$$

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

Rule 91. Semantics of state machine

$[\![\mathsf{stm} : \mathsf{StateMachineDef}]\!]_{\mathcal{STM}} : \mathsf{TimedCSPProcess} =$

$$
\left(
\begin{array}{l}
\left(
\begin{array}{l}
\mathsf{initialisation}^{4}(\mathsf{stm}) \\
\quad [\![\mathsf{flowevts}]\!] \\
\mathsf{composeStates}^{4}(\langle \mathsf{x} : \mathsf{stm.nodes} \mid \mathsf{x} \in \mathsf{State}\rangle, \mathsf{stm})
\end{array}
\right) \\
\setminus \{\!| enter, exit, exited |\!\} \\
[\![\mathsf{getsetChannels}^{2}(\mathsf{stm}) \cup \mathsf{trigEvents}^{2}(\mathsf{stm}) \cup \mathsf{clockResets}^{1}(\mathsf{wcs}) \cup \mathsf{deadlineEvents}^{1}(\mathsf{stm})]\!] \\
\mathsf{constInitSTM}^{1}(\mathsf{consts}, \mathsf{stm}, \left(\mathsf{stmMemory}^{1}(\mathsf{stm}, \mathsf{wcs}) \; [\![\mathsf{clockMemSync}]\!] \; \mathsf{stmClocks}^{1}(\mathsf{stm}) \right)) \\
\setminus (\mathsf{clockMemSync} \setminus \mathsf{trigEvents}^{3}(\mathsf{stm}))
\end{array}
\right)
$$

$$[\![\mathsf{renameTriggerEvents}^{2}(\mathsf{stm})]\!]$$

$$\setminus \mathsf{getsetLocalChannels}^{2}(\mathsf{stm}) \cup \mathsf{clockResets}^{2}(\mathsf{wcs}) \cup \mathsf{deadlineEvents}^{2}(\mathsf{stm}) \cup \{\!| internal, entered |\!\}$$

$$\Theta_{\{end\}} Skip$$

where

$$\mathsf{wcs} = \{\mathsf{t} : \mathsf{allTransitions}^{7}(\mathsf{stm}) \mid \mathsf{t.condition} \neq \mathsf{null} \bullet \mathsf{t} \mapsto \mathsf{wc}(\mathsf{t.condition})\}$$

$$\mathsf{clockMemSync} = \left(
\begin{array}{l}
\{\!| \mathsf{t} : \mathsf{Transition} \mid \mathsf{t} \in \mathrm{dom}\, \mathsf{wcs} \bullet \mathsf{triggerEvent}^{1}(\mathsf{t}) |\!\} \\
\cup \\
\{\!| \mathsf{v} : \mathsf{allClockVariables}^{1}(\mathsf{wcs}) \bullet set WC_\mathsf{vid}(\mathsf{v}) |\!\}
\end{array}
\right)$$

$$\mathsf{flowevts} =$$

$$\bigcup \{\mathsf{x} : \mathsf{SIDS} \setminus \mathsf{states}^{12}(\mathsf{stm});\ \mathsf{y} : \mathsf{states}^{13}(\mathsf{stm}) \bullet \{\!| enter.\mathsf{x}.\mathsf{y}, entered.\mathsf{x}.\mathsf{y}, exit.\mathsf{x}.\mathsf{y}, exited.\mathsf{x}.\mathsf{y} |\!\}\}$$

$$\mathsf{consts} = \langle \mathsf{v} : \mathsf{allConstants}^{7}(\mathsf{stm}) \bullet \mathsf{v} \rangle$$

Rule 92. Constants Initialisation for State Machines

constInitSTM(cs : Seq_1(Variable), stm : StateMachineDef, P : CSPProcess) : CSPProcess =

$$\text{buildScope}^1(\text{undefc}, \text{stm}, \left(\begin{array}{l}\textbf{let}\ c : \text{defs} \bullet \text{name}(c) = [\![c.\text{initial}]\!]_{\text{Expr}^{54}} \\ \textbf{within}\ P\end{array}\right))$$

where

defc = ⟨x : consts | x.initial ≠ null⟩

undefc = ⟨x • consts | x.initial = null⟩

Rule 93. Build Scope

buildScope(cs : Seq_1(Variable), ctx : ConnectionNode, P : CSPProcess) : CSPProcess =

if (head cs).initial == NULL then

$$0 \blacktriangleleft \textit{set_}\text{vid}(\text{head cs}, \text{ctx})?\text{name}(\text{head cs}) \rightarrow \left(\begin{array}{l}\text{if } \#\text{cs} == 1 \text{ then } P \\ \text{else constInitSTM}^2(\text{tail cs}, \text{ctx}, P)\end{array}\right)$$

else

$$\left(\begin{array}{l}\text{if } \#\text{cs} == 1 \text{ then } P \\ \text{else constInitSTM}^3(\text{tail cs}, \text{ctx}, P)\end{array}\right)$$

Clocks

Functions related to clocks are formalised in this section.

Rule 94. allClockVariables function

$\mathsf{allClockVariables}(\mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathbb{P}\,\mathsf{Variable}$

$$\mathsf{allClockVariables}(\mathsf{wcs}) = \{\mathsf{t} : \mathsf{Transition}, \mathsf{e} : \mathsf{Expression}, \mathsf{v} : \mathsf{Variable} \mid \mathsf{t} \in \operatorname{dom} \mathsf{wcs} \wedge (\mathsf{e} \mapsto \mathsf{v}) \in \pi_2(\mathsf{wcs}(\mathsf{t})) \bullet \mathsf{v}\}$$

Rule 95. clockResets function

$\mathsf{clockResets}(\mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathsf{ChannelSet}$

$$\mathsf{clockResets}(\mathsf{stm}) = \bigcup \left\{ \begin{array}{l} \mathsf{t} : \mathsf{Transition}, \mathsf{e} : \mathsf{Expression}, \mathsf{v} : \mathsf{Variable} \mid \\ \mathsf{t} \in \operatorname{dom} \mathsf{wcs} \wedge (\mathsf{e} \mapsto \mathsf{v}) \in \pi_2(\mathsf{wcs}(\mathsf{t})) \\ \bullet\ \mathsf{alphaClockReset}^1(\mathsf{e}) \end{array} \right\}$$

Rule 96. stmClocks function

$\mathsf{stmClocks}(\mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathsf{TimedCSPProcess} =$

$$\begin{array}{l} \|\ (t, e, v) : \{\mathsf{t} : \mathsf{Transition}, \mathsf{e} : \mathsf{Expression}, \mathsf{v} : \mathsf{Variable} \mid \mathsf{t} \in \operatorname{dom} \mathsf{wcs} \wedge (\mathsf{e} \mapsto \mathsf{v}) \in \pi_2(\mathsf{wcs}(\mathsf{t}))\} \\ \quad \bullet\ [\![\alpha\mathsf{WC}(\mathsf{t}, \mathsf{e}, \mathsf{v})]\!]\ \mathsf{compileWC}^1(\mathsf{t}, \mathsf{e}, \mathsf{v}) \end{array}$$

where

$$\alpha\mathsf{WC}(\mathsf{t}, \mathsf{e}, \mathsf{v}) = \{\!|\mathsf{triggerEvent}^2(\mathsf{t}), \mathit{setWC_}\mathsf{vid}(\mathsf{v})|\!\} \cup \mathsf{alphaClockReset}^2(\mathsf{e})$$

Rule 97. alphaClockReset function

$\mathsf{alphaClockReset}(\mathsf{e} : \mathsf{Expression}) : \mathsf{ChannelSet} =$

This rule is defined by multiple rules according to the subtype of the expression: (98, 99, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110).

Rule 98. alphaClockReset function

$\mathsf{alphaClockReset}(e : \mathsf{ParExp}) : \mathsf{ChannelSet} =$

$\mathsf{alphaClockReset}(e) = \mathsf{alphaClockReset}^{3}(e.exp)$

Rule 99. alphaClockReset function

$\mathsf{alphaClockReset}(e : \mathsf{Not}) : \mathsf{ChannelSet} =$

$\mathsf{alphaClockReset}(e) = \mathsf{alphaClockReset}^{4}(e.exp)$

Rule 100. alphaClockReset function

$\mathsf{alphaClockReset}(e : \mathsf{CallExp}) : \mathsf{ChannelSet} =$

$\mathsf{alphaClockReset}(e) = \mathsf{alphaClockResetCallArgs}^{1}(e.args)$

Rule 101. alphaClockResetCallArgs function

$\mathsf{alphaClockResetCallArgs}(s : \mathrm{seq}\ \mathsf{Expression}) : \mathsf{ChannelSet} =$

if $\#(s) > 0$ then

$\quad \mathsf{alphaClockReset}^{5}(head(s)) \cup \mathsf{alphaClockResetCallArgs}^{2}(tail(s))$

else

$\quad \emptyset$

endif

Rule 102. alphaClockReset function

alphaClockReset(e : And) : ChannelSet =

alphaClockReset(e) = alphaClockReset[6](e.left) ∪ alphaClockReset[7](e.right))

Rule 103. alphaClockReset function

alphaClockReset(e : Or) : ChannelSet =

alphaClockReset(e) = alphaClockReset[8](e.left) ∪ alphaClockReset[9](e.right))

Rule 104. alphaClockReset function

alphaClockReset(e : Implies) : ChannelSet =

alphaClockReset(e) = alphaClockReset[10](e.left) ∪ alphaClockReset[11](e.right))

Rule 105. alphaClockReset function

alphaClockReset(e : Iff) : ChannelSet =

alphaClockReset(e) = alphaClockReset[12](e.left) ∪ alphaClockReset[13](e.right))

Rule 106. alphaClockReset function

alphaClockReset(e : GreaterThan) : ChannelSet =

alphaClockReset(e) = alphaClockReset[14](e.left) ∪ alphaClockReset[15](e.right))

Rule 107. alphaClockReset function

alphaClockReset(e : GreaterOrEqual) : ChannelSet =

alphaClockReset(e) = alphaClockReset[16](e.left) ∪ alphaClockReset[17](e.right))

Rule 108. alphaClockReset function

alphaClockReset(e : LessThan) : ChannelSet =

alphaClockReset(e) = alphaClockReset[18](e.left) ∪ alphaClockReset[19](e.right))

Rule 109. alphaClockReset function

alphaClockReset(e : LessOrEqual) : ChannelSet =

alphaClockReset(e) = alphaClockReset[20](e.left) ∪ alphaClockReset[21](e.right))

Rule 110. alphaClockReset function
$\mathsf{alphaClockReset}(\mathsf{e} : \mathsf{Equals}) : \mathsf{ChannelSet} =$

$$\mathsf{alphaClockReset}(\mathsf{e}) = \mathsf{alphaClockReset}^{22}(\mathsf{e.left}) \cup \mathsf{alphaClockReset}^{23}(\mathsf{e.right}))$$

Rule 111. alphaClockReset function
$\mathsf{alphaClockReset}(\mathsf{e} : \mathsf{ClockExp}) : \mathsf{ChannelSet} =$

$$\mathsf{alphaClockReset}(\mathsf{e}) = \{\!| \mathit{clockReset}.\mathsf{id}(\mathsf{e.clock}) |\!\}$$

Rule 112. alphaClockReset function
$\mathsf{alphaClockReset}(\mathsf{e} : \mathsf{StateClockExp}) : \mathsf{ChannelSet} =$

$$\mathsf{alphaClockReset}(\mathsf{e}) = \{\!| \mathsf{x} : \mathsf{SIDS} \mid \mathit{entered}.\mathsf{id}(\mathsf{x}).\mathsf{id}(\mathsf{e.state}) |\!\}$$

Rule 113. Timed semantics of triggers
$[\![\mathsf{t} : \mathsf{Trigger}]\!]^{\mathsf{tid}}_{\mathit{Trigger}} : \mathsf{CSPProcess} =$

$$[\![\mathsf{t}]\!]^{\mathsf{tid}}_{\mathit{Trigger}^2} \; ; \; \left(|\!|\!| \, c : \mathsf{t.reset} \bullet \mathit{ClockReset}.\mathsf{id}(\mathsf{c.clock}) \rightarrow \mathit{Skip} \right)$$

Waiting Conditions

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

Rule 114. wc function

$\mathsf{wc}(e : \mathsf{Expression}) : (\mathsf{Expression}, \mathsf{Expression} \nrightarrow \mathsf{Variable}) =$

$\mathsf{wc}(\,(e)\,) = (\,(\pi_1(\mathsf{wc}(e)))\,,\,\pi_2(\mathsf{wc}(e)))$

$\mathsf{wc}(\,\neg\, e\,) = (\,\neg\,\pi_1(\mathsf{wc}(e))\,,\,\pi_2(\mathsf{wc}(e)))$

$\mathsf{wc}(\,f(args)\,) = (\,f(\pi_1(\mathsf{wcArgSeq}^1(args)))\,,\pi_2(\mathsf{wcArgSeq}^2(args)))$

$\mathsf{wc}(\,e_1 \wedge e_2\,) = (\,\pi_1(\mathsf{wc}(e_1)) \wedge \pi_1(\mathsf{wc}(e_1))\,,\,\pi_2(\mathsf{wc}(e_1)) \cup \pi_2(\mathsf{wc}(e_2))$

$\mathsf{wc}(\,e_1 \vee e_2\,) = (\,\pi_1(\mathsf{wc}(e_1)) \vee \pi_1(\mathsf{wc}(e_1))\,,\,\pi_2(\mathsf{wc}(e_1)) \cup \pi_2(\mathsf{wc}(e_2))$

$\mathsf{wc}(\,e_1 => e_2\,) = (\,\pi_1(\mathsf{wc}(e_1)) => \pi_1(\mathsf{wc}(e_1))\,,\,\pi_2(\mathsf{wc}(e_1)) \cup \pi_2(\mathsf{wc}(e_2))$

$\mathsf{wc}(\,e_1 \text{ iff } e_2\,) = (\,\pi_1(\mathsf{wc}(e_1)) \text{ iff } \pi_1(\mathsf{wc}(e_1))\,,\,\pi_2(\mathsf{wc}(e_1)) \cup \pi_2(\mathsf{wc}(e_2))$

$\mathsf{wc}(\,\mathsf{since}(C) > e\,) = (\,b\,,\,\{\,\mathsf{since}(C) > e\,\} \mapsto b)$

$\mathsf{wc}(\,\mathsf{sinceEntry}(S) > e\,) = (\,b\,,\,\{\,\mathsf{sinceEntry}(S) > e\,\} \mapsto b)$

$\mathsf{wc}(\,e > \mathsf{since}(C)\,) = (\,b\,,\,\{\,e > \mathsf{since}(C)\,\} \mapsto b)$

$\mathsf{wc}(\,e > \mathsf{sinceEntry}(S)\,) = (\,b\,,\,\{\,e > \mathsf{sinceEntry}(S)\,\} \mapsto b)$

$\mathsf{wc}(\,e_1 > e_2\,) = (\,e_1 > e_2\,,\emptyset)$

$\mathsf{wc}(\,\mathsf{since}(C) >= e\,) = (\,b\,,\,\{\,\mathsf{since}(C) >= e\,\} \mapsto b)$

$\mathsf{wc}(\,\mathsf{sinceEntry}(S) >= e\,) = (\,b\,,\,\{\,\mathsf{sinceEntry}(S) >= e\,\} \mapsto b)$

$\mathsf{wc}(\,e >= \mathsf{since}(C)\,) = (\,b\,,\,\{\,e >= \mathsf{since}(C)\,\} \mapsto b)$

$\mathsf{wc}(\,e >= \mathsf{sinceEntry}(S)\,) = (\,b\,,\,\{\,e >= \mathsf{sinceEntry}(S)\,\} \mapsto b)$

$\mathsf{wc}(\,e_1 >= e_2\,) = (\,e_1 >= e_2\,,\emptyset)$

$\mathsf{wc}(\,\mathsf{since}(C) < e\,) = (\,b\,,\,\{\,\mathsf{since}(C) < e\,\} \mapsto b)$

$\mathsf{wc}(\,\mathsf{sinceEntry}(S) < e\,) = (\,b\,,\,\{\,\mathsf{sinceEntry}(S) < e\,\} \mapsto b)$

$\mathsf{wc}(\,e < \mathsf{since}(C)\,) = (\,b\,,\,\{\,e < \mathsf{since}(C)\,)\} \mapsto b)$

$\mathsf{wc}(\,e < \mathsf{sinceEntry}(S)\,) = (\,b\,,\,\{\,e < \mathsf{sinceEntry}(S)\,\} \mapsto b)$

$\mathsf{wc}(\,e_1 < e_2\,) = (\,e_1 < e_2\,,\emptyset)$

$\mathsf{wc}(\,\mathsf{since}(C) <= e\,) = (\,b\,,\,\{\,\mathsf{since}(C) <= e\,\} \mapsto b)$

$\mathsf{wc}(\,\mathsf{sinceEntry}(S) <= e\,) = (\,b\,,\,\{\,\mathsf{sinceEntry}(S) <= e\,\} \mapsto b)$

$\mathsf{wc}(\,e <= \mathsf{since}(C)\,) = (\,b\,,\,\{\,e <= \mathsf{since}(C)\,\} \mapsto b)$

$\mathsf{wc}(\,e <= \mathsf{sinceEntry}(S)\,) = (\,b\,,\,\{\,e <= \mathsf{sinceEntry}(S)\,\} \mapsto b)$

$\mathsf{wc}(\,e_1 <= e_2\,) = (\,e_1 <= e_2\,,\emptyset)$

$\mathsf{wc}(\,\mathsf{since}(C) == e\,) = (\,b\,,\,\{\,\mathsf{since}(C) == e\,\} \mapsto b)$

$\mathsf{wc}(\,\mathsf{sinceEntry}(S) == e\,) = (\,b\,,\,\{\,\mathsf{sinceEntry}(S) == e\,\} \mapsto b)$

$\mathsf{wc}(\,e == \mathsf{since}(C)\,) = (\,b\,,\,\{\,e == \mathsf{since}(C)\,\} \mapsto b)$

$\mathsf{wc}(\,e == \mathsf{sinceEntry}(S)\,) = (\,b\,,\,\{\,e == \mathsf{sinceEntry}(S)\,\} \mapsto b)$

$\mathsf{wc}(\,e_1 == e_2\,) = (\,e_1 == e_2\,,\emptyset)$

where

b is a fresh identifier.

Rule 115. wcArgSeq function

$\mathsf{wcArgSeq}(s : \mathrm{seq}(\mathsf{Expression})) : (\mathrm{seq}\,\mathsf{Expression}, \mathsf{Expression} \nrightarrow \mathsf{Variable}) =$

$\mathsf{if}\ \#(s) > 0\ \mathsf{then}$

$\quad (\langle \pi_1(wc_{head}) \rangle \frown \pi_1(wc_{tail}), \pi_2(wc_{head}) \cup \pi_2(wc_{tail}))$

where

$\quad wc_{head} = \mathsf{wc}^1(\mathsf{head}(s))$

$\quad wc_{tail} = \mathsf{wcArgSeq}^3(\mathsf{tail}(s))$

else

$\quad (\langle\rangle, \emptyset)$

endif

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

Rule 116. compileWC function

$\mathsf{compileWC}(\mathsf{t} : \mathsf{Transition}, \mathsf{e} : \mathsf{Expression}, \mathsf{v} : \mathsf{Variable}) : \mathsf{TimedCSPProcess} =$

$\mathsf{compileWC}(\mathsf{t}, \mathsf{since}(\mathsf{C}) >= \mathsf{e}, \mathsf{v}) =$

let

$$Reset = clockReset.\mathsf{id}(\mathsf{C}) \rightarrow setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{3}(\mathsf{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{55}}{[\![\mathsf{e}]\!]}} \quad setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{4}(\mathsf{t})|\}) \end{array}\right) \triangle\, Reset$$

within

$$setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow Monitor$$

$\mathsf{compileWC}(\mathsf{t}, \mathsf{e} >= \mathsf{since}(\mathsf{C}), \mathsf{v}) =$

let

$$Reset = clockReset.\mathsf{id}(\mathsf{C}) \rightarrow setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{5}(\mathsf{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{56}}{[\![\mathsf{e}]\!]}} \quad setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{6}(\mathsf{t})|\}) \end{array}\right) \triangle\, Reset$$

within

$$setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow Monitor$$

$\mathsf{compileWC}(\mathsf{t}, \mathsf{sinceEntry}(\mathsf{S}) >= \mathsf{e}, \mathsf{v}) =$

let

$$Reset = entered?x.\mathsf{id}(\mathsf{S}) \rightarrow setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{7}(\mathsf{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{57}}{[\![\mathsf{e}]\!]}} \quad setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{8}(\mathsf{t})|\}) \end{array}\right) \triangle\, Reset$$

within

$$setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow Monitor$$

$\mathsf{compileWC}(\mathsf{t}, \mathsf{e} >= \mathsf{sinceEntry}(\mathsf{S}), \mathsf{v}) =$

let

$$Reset = entered?x.\mathsf{id}(\mathsf{S}) \rightarrow setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{9}(\mathsf{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{58}}{[\![\mathsf{e}]\!]}} \quad setWC_\mathsf{vid}(\mathsf{v})!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{10}(\mathsf{t})|\}) \end{array}\right) \triangle\, Reset$$

within

$$setWC_\mathsf{vid}(\mathsf{v})!true \rightarrow Monitor$$

Rule 116. compileWC function

compileWC(t : Transition, e : Expression, v : Variable) : TimedCSPProcess =

compileWC(t, since(C) >= e , v) =

let

$Reset = clockReset.\mathsf{id}(C) \rightarrow setWC_\mathsf{vid}(v)!false \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{11}(t)|\}) \\ \quad \triangle_{\underbrace{[\![e]\!]}_{\mathcal{E}xpr^{59}}+1} setWC_\mathsf{vid}(v)!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{12}(t)|\}) \end{array} \right) \triangle Reset$$

within

$setWC_\mathsf{vid}(v)!false \rightarrow Monitor$

compileWC(t, e > since(C) , v) =

let

$Reset = clockReset.\mathsf{id}(C) \rightarrow setWC_\mathsf{vid}(v)!true \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{13}(t)|\}) \\ \quad \triangle_{\underbrace{[\![e]\!]}_{\mathcal{E}xpr^{60}}+1} setWC_\mathsf{vid}(v)!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{14}(t)|\}) \end{array} \right) \triangle Reset$$

within

$setWC_\mathsf{vid}(v)!true \rightarrow Monitor$

compileWC(t, sinceEntry(S) > e , v) =

let

$Reset = entered?x.\mathsf{id}(S) \rightarrow setWC_\mathsf{vid}(v)!false \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{15}(t)|\}) \\ \quad \triangle_{\underbrace{[\![e]\!]}_{\mathcal{E}xpr^{61}}+1} setWC_\mathsf{vid}(v)!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{16}(t)|\}) \end{array} \right) \triangle Reset$$

within

$setWC_\mathsf{vid}(v)!false \rightarrow Monitor$

compileWC(t, e > sinceEntry(S) , v) =

let

$Reset = entered?x.\mathsf{id}(S) \rightarrow setWC_\mathsf{vid}(v)!true \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{17}(t)|\}) \\ \quad \triangle_{\underbrace{[\![e]\!]}_{\mathcal{E}xpr^{62}}+1} setWC_\mathsf{vid}(v)!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{18}(t)|\}) \end{array} \right) \triangle Reset$$

within

$setWC_\mathsf{vid}(v)!true \rightarrow Monitor$

Rule 116. compileWC function

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

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

let

$$Reset = clockReset.\text{id}(\text{C}) \rightarrow setWC_\text{vid}(\text{v})!true \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\text{triggerEvent}^{19}(\text{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{63}}{[\![\text{e}]\!]}} \quad setWC_\text{vid}(\text{v})!false \rightarrow RUN(\{|\text{triggerEvent}^{20}(\text{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$$setWC_\text{vid}(\text{v})!true \rightarrow Monitor$$

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

let

$$Reset = clockReset.\text{id}(\text{C}) \rightarrow setWC_\text{vid}(\text{v})!false \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\text{triggerEvent}^{21}(\text{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{64}}{[\![\text{e}]\!]}} \quad setWC_\text{vid}(\text{v})!true \rightarrow RUN(\{|\text{triggerEvent}^{22}(\text{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$$setWC_\text{vid}(\text{v})!false \rightarrow Monitor$$

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

let

$$Reset = entered?x.\text{id}(\text{S}) \rightarrow setWC_\text{vid}(\text{v})!true \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\text{triggerEvent}^{23}(\text{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{65}}{[\![\text{e}]\!]}} \quad setWC_\text{vid}(\text{v})!false \rightarrow RUN(\{|\text{triggerEvent}^{24}(\text{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$$setWC_\text{vid}(\text{v})!true \rightarrow Monitor$$

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

let

$$Reset = entered?x.\text{id}(\text{S}) \rightarrow setWC_\text{vid}(\text{v})!false \rightarrow Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{|\text{triggerEvent}^{25}(\text{t})|\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{66}}{[\![\text{e}]\!]}} \quad setWC_\text{vid}(\text{v})!true \rightarrow RUN(\{|\text{triggerEvent}^{26}(\text{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$$setWC_\text{vid}(\text{v})!false \rightarrow Monitor$$

Rule 116. compileWC function

$\mathsf{compileWC(t : Transition, e : Expression, v : Variable) : TimedCSPProcess} =$

$\mathsf{compileWC(t,\ since(C) < e\ , v)} =$

let

$Reset = clockReset.\mathsf{id(C)} \rightarrow setWC_\mathsf{vid(v)}!true \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{27}(\mathsf{t})|\}) \\ \quad \triangle_{(\underbrace{[\![\mathsf{e}]\!]}_{\mathcal{E}xpr^{67}}+1)} setWC_\mathsf{vid(v)}!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{28}(\mathsf{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!true \rightarrow Monitor$

$\mathsf{compileWC(t,\ e < since(C)\ , v)} =$

let

$Reset = clockReset.\mathsf{id(C)} \rightarrow setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{29}(\mathsf{t})|\}) \\ \quad \triangle_{(\underbrace{[\![\mathsf{e}]\!]}_{\mathcal{E}xpr^{68}}+1)} setWC_\mathsf{vid(v)}!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{30}(\mathsf{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$\mathsf{compileWC(t,\ sinceEntry(S) < e\ , v)} =$

let

$Reset = entered?x.\mathsf{id(S)} \rightarrow setWC_\mathsf{vid(v)}!true \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{31}(\mathsf{t})|\}) \\ \quad \triangle_{(\underbrace{[\![\mathsf{e}]\!]}_{\mathcal{E}xpr^{69}}+1)} setWC_\mathsf{vid(v)}!false \rightarrow RUN(\{|\mathsf{triggerEvent}^{32}(\mathsf{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!true \rightarrow Monitor$

$\mathsf{compileWC(t,\ e < sinceEntry(S)\ , v)} =$

let

$Reset = entered?x.\mathsf{id(S)} \rightarrow setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$$Monitor = \left(\begin{array}{l} RUN(\{|\mathsf{triggerEvent}^{33}(\mathsf{t})|\}) \\ \quad \triangle_{(\underbrace{[\![\mathsf{e}]\!]}_{\mathcal{E}xpr^{70}}+1)} setWC_\mathsf{vid(v)}!true \rightarrow RUN(\{|\mathsf{triggerEvent}^{34}(\mathsf{t})|\}) \end{array} \right) \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

Rule 116. compileWC function

$\mathsf{compileWC(t : Transition, e : Expression, v : Variable) : TimedCSPProcess} =$

$\mathsf{compileWC(t,\ since(C) == e\ , v)} =$

let

$Reset = clockReset.\mathsf{id(C)} \rightarrow setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$$Monitor = \begin{pmatrix} RUN(\{\mid \mathsf{triggerEvent}^{35}\mathsf{(t)} \mid\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{71}}{[\![e]\!]}} \quad setWC_\mathsf{vid(v)}!true \rightarrow RUN(\{\mid \mathsf{triggerEvent}^{36}\mathsf{(t)} \mid\}) \\ \qquad \triangle_{(\underset{\mathcal{E}xpr^{72}}{[\![e]\!]} +1)} setWC_\mathsf{vid(v)}!false \rightarrow RUN(\{\mid \mathsf{triggerEvent}^{37}\mathsf{(t)} \mid\}) \end{pmatrix} \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$\mathsf{compileWC(t,\ e == since(C)\ , v)} =$

let

$Reset = clockReset.\mathsf{id(C)} \rightarrow setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

$$Monitor = \begin{pmatrix} RUN(\{\mid \mathsf{triggerEvent}^{38}\mathsf{(t)} \mid\}) \\ \quad \triangle_{\underset{\mathcal{E}xpr^{73}}{[\![e]\!]}} \quad setWC_\mathsf{vid(v)}!true \rightarrow RUN(\{\mid \mathsf{triggerEvent}^{39}\mathsf{(t)} \mid\}) \\ \qquad \triangle_{(\underset{\mathcal{E}xpr^{74}}{[\![e]\!]} +1)} setWC_\mathsf{vid(v)}!false \rightarrow RUN(\{\mid \mathsf{triggerEvent}^{40}\mathsf{(t)} \mid\}) \end{pmatrix} \triangle\ Reset$$

within

$setWC_\mathsf{vid(v)}!false \rightarrow Monitor$

Rule 116. compileWC function

compileWC(t : Transition, e : Expression, v : Variable) : TimedCSPProcess =

compileWC(t, sinceEntry(C) == e , v) =

let

$$Reset = entered?x.\mathsf{id}(S) \to setWC_\mathsf{vid}(v)!false \to Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{\!|\mathsf{triggerEvent}^{41}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{\mathcal{E}xpr^{75}}} \; setWC_\mathsf{vid}(v)!true \to RUN(\{\!|\mathsf{triggerEvent}^{42}(t)|\!\}) \\ \quad\quad \triangle_{([\![e]\!]_{\mathcal{E}xpr^{76}}+1)} \; setWC_\mathsf{vid}(v)!false \to RUN(\{\!|\mathsf{triggerEvent}^{43}(t)|\!\}) \end{array} \right) \triangle \, Reset$$

within

$$setWC_\mathsf{vid}(v)!false \to Monitor$$

compileWC(t, e == sinceEntry(S) , v) =

let

$$Reset = entered?x.\mathsf{id}(S) \to setWC_\mathsf{vid}(v)!false \to Monitor$$

$$Monitor = \left(\begin{array}{l} RUN(\{\!|\mathsf{triggerEvent}^{44}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{\mathcal{E}xpr^{77}}} \; setWC_\mathsf{vid}(v)!true \to RUN(\{\!|\mathsf{triggerEvent}^{45}(t)|\!\}) \\ \quad\quad \triangle_{([\![e]\!]_{\mathcal{E}xpr^{78}}+1)} \; setWC_\mathsf{vid}(v)!false \to RUN(\{\!|\mathsf{triggerEvent}^{46}(t)|\!\}) \end{array} \right) \triangle \, Reset$$

within

$$setWC_\mathsf{vid}(v)!false \to Monitor$$

Rule 117. triggerEvent function

triggerEvent(t : Transition) : CSPEvent =

if t.trigger $\neq$ null

  triggerEvent[3](t.trigger, id(t))

else

  internal.id(t)

Trigger deadline events

Rule 118. deadlineEvents function

$\mathsf{deadlineEvents}(s : \mathsf{StateMachineDef}) : \mathsf{ChannelSet} =$

$\mathsf{deadlineEvents}(stm) = \{\!| t : \mathsf{allTransitions}^{8}(s) \mid t.end \neq null \bullet \textit{deadline}.\mathrm{id}(t) |\!\}$

Rule 119. State-machine Memory

$\mathsf{stmMemory}(\mathsf{stm} : \mathsf{StateMachineDef}, \mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathsf{TimedCSPProcess} =$

$$
\textbf{let } \mathit{Memory}(\mathsf{vars}) \mathrel{\widehat=}
\left(
\begin{array}{l}
\Box\, v : \mathsf{lvars} \bullet \left(\begin{array}{l} \mathit{get_}\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to \mathit{Memory}(\mathsf{vars}) \\ \Box \\ \mathit{set_}\mathsf{vid}(\mathsf{v})?x \to \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \end{array}\right) \\
\Box \\
\Box\, v : \mathsf{rvars} \bullet \\
\quad \left(\begin{array}{l} \mathit{get_}\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to \mathit{Memory}(\mathsf{vars}) \\ \Box \\ \mathit{set_}\mathsf{vid}(\mathsf{v})?x \to \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\ \Box \\ \mathit{set_Ext_}\mathsf{vid}(\mathsf{v})?x \to \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \end{array}\right) \\
\Box \\
\Box\, v : \mathsf{allConstants}^{8}(\mathsf{stm}) \bullet \mathit{get_}\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to \mathit{Memory}(\mathsf{vars}) \\
\Box \\
\Box\, t : \mathsf{allTransitions}^{9}(\mathsf{stm}) \bullet \mathsf{memoryTransition}^{1}(\mathsf{t}, \mathsf{wcs}) \,;\, \mathit{Memory}(\mathsf{vars})) \\
\Box \\
\Box\, v : \mathsf{cvars} \bullet \mathit{setWC_}\mathsf{vid}(\mathsf{v})?x \to \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\
\Box \\
\Box\, t : \mathsf{allDeadlineTransitions}^{1}(\mathsf{stm}) \bullet \mathsf{memoryDeadline}^{1}(\mathsf{t}, \mathsf{wcs});\, \mathit{Memory}(\mathsf{vars})
\end{array}
\right)
$$

within

$\quad \mathit{Memory}(\mathsf{varvalues})$

where

$\mathsf{rvars} = \mathsf{requiredVariables}^{7}(\mathsf{stm})$

$\mathsf{lvars} = \mathsf{allLocalVariables}^{7}(\mathsf{stm})$

$\mathsf{consts} = \langle \mathsf{v} : \mathsf{allConstants}^{9}(\mathsf{stm}) \bullet \mathsf{v} \rangle$

$\mathsf{cvars} = \mathsf{allClockVariables}^{2}(\mathsf{wcs})$

$\mathsf{vars} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{name}(\mathsf{v}) \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name}(\mathsf{v}) \rangle$

$\mathsf{varvalues} = \langle \mathsf{v} : \mathsf{rvars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{initial}(\mathsf{v}) \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name}(\mathsf{v}) \rangle$

Rule 120. allDeadlineTransitions function

$\mathsf{allDeadlineTransitions}(s : \mathsf{StateMachineDef}) : \mathbb{P}\,\mathsf{Transition} =$

$\mathsf{allDeadlineTransitions}(s) = \{t : \mathsf{allTransitions}^{10}(s) \mid t.\mathsf{end} \neq \mathsf{null}\}$

Rule 121. memoryTransition function

$\mathsf{memoryTransition}(t : \mathsf{Transition}, \mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathsf{TimedCSPProcess}$

$$\begin{array}{l} \mathsf{if}\ (t.\mathsf{condition} \neq \mathsf{null})\ \mathsf{then} \\ \quad ([\![\pi_1(\mathsf{wcs}(t))]\!]_{\mathcal{E}xpr^{79}})\ \&\ \mathsf{triggerForMemory}^{3}(t.\mathsf{trigger}, \mathsf{id}(t)) \\ \mathsf{else} \\ \quad \mathsf{triggerForMemory}^{4}(t.\mathsf{trigger}, \mathsf{id}(t)) \end{array}$$

Rule 122. Memory deadline

$\mathsf{memoryDeadline}(t : \mathsf{Transition}, \mathsf{wcs} : \mathsf{Transition} \nrightarrow (\mathsf{Expression}, \mathsf{WC})) : \mathsf{TimedCSPProcess}$

$$\begin{array}{l} \mathsf{if}\ (t.\mathsf{condition} \neq \mathsf{null})\ \mathsf{then} \\ \quad ([\![\pi_1(\mathsf{wcs}(t))]\!]_{\mathcal{E}xpr^{80}})\ \&\ \mathit{deadline}.\mathsf{id}(t).\mathit{on} \rightarrow \mathit{Skip} \\ \quad \Box \\ \quad (\neg\ [\![\pi_1(\mathsf{wcs}(t))]\!]_{\mathcal{E}xpr^{81}})\ \&\ \mathit{deadline}.\mathsf{id}(t).\mathit{off} \rightarrow \mathit{Skip} \\ \mathsf{else} \\ \quad \mathit{deadline}.\mathsf{id}(t).\mathit{on} \rightarrow \mathit{Skip} \end{array}$$

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 123. Semantics of states

$[\![\mathsf{s} : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{TimedCSPProcess} =$

This function is split in multiple rules according to the type of states.

Rule 124. Semantics of simple states

$[\![\mathsf{s} : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{TimedCSPProcess} =$

let

$$Inactive \mathrel{\widehat=} enter?x : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow Activating(x)$$

$$Activating(o) \mathrel{\widehat=} [\![\mathsf{s.entry}]\!]_{\mathcal{A}ction^{11}} ; \mathsf{initialisation}^{5}(\mathsf{s}); entered.o.\mathsf{id}(\mathsf{s}) \rightarrow$$

$$([\![\mathsf{s.during}]\!]_{\mathcal{A}ction^{12}} ; Stop \mid\mid\mid triggerDeadlines^{1}(s)) \triangle$$

$$\left(\begin{array}{ll} \Box\, t : \mathsf{transitionsFrom}^{5}(\mathsf{s}) \bullet [\![\mathsf{t}, \mathsf{s}, \mathsf{false}]\!]^{\mathsf{Inactive},\mathsf{Activating}}_{\mathcal{T}^{5}} & \\ \Box & \\ \Box\, e : \mathsf{Event} \bullet & \mathsf{if}(\mathsf{e.type} == \mathsf{null}) \\ & \mathsf{then}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids} \rightarrow \mathsf{exit};\ Inactive \\ & \mathsf{else}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids}?y \rightarrow \mathsf{exit};\ Inactive \end{array} \right)$$

within

$$Inactive$$

where

$$\#\mathsf{states}^{14}(\mathsf{s}) = 0$$

$$\mathsf{sids} = SIDS \setminus \{\mathsf{id}(\mathsf{s})\}$$

$$\mathsf{exit} = exit?x : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow \mathsf{exitSubstates}^{3}(\mathsf{s}); [\![\mathsf{s.exit}]\!]_{\mathcal{A}ction^{13}} ; exited.x.\mathsf{id}(\mathsf{s}) \rightarrow Skip$$

$$\mathsf{tids} = TIDS \setminus \mathsf{tIDS}(\mathsf{s})$$

Rule 125. Semantics of composite states

$[\![\mathsf{s} : \mathsf{State}]\!]_{\mathcal{S}} : \mathsf{TimedCSPProcess} =$

let

$Inactive \mathrel{\widehat=} enter?x : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow Activating(x)$

$Activating(o) \mathrel{\widehat=} \underbrace{[\![\mathsf{s.entry}]\!]}_{\mathcal{A}ction^{14}} ; \ \mathsf{initialisation}^{6}(\mathsf{s}); \ entered.o.\mathsf{id}(\mathsf{s}) \rightarrow$

$([\![s.during]\!]_{\mathcal{A}ction^{15}} ; \ Stop \ ||| \ \mathsf{triggerDeadlines}^{2}(\mathsf{s})) \ \triangle$

$$\left(\begin{array}{ll} \Box\, t : \mathsf{transitionsFrom}^{6}(\mathsf{s}) \bullet & \underbrace{[\![\mathsf{t}, \mathsf{s}, \mathsf{false}]\!]^{\mathsf{Inactive},\mathsf{Activating}}}_{\mathcal{T}^{6}} \\ \Box & \\ \Box\, e : \mathsf{Event} \bullet & \mathsf{if}(\mathsf{e.type} == \mathsf{null}) \\ & \mathsf{then}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids} \rightarrow \mathsf{exit}; \ Inactive \\ & \mathsf{else}\ \mathsf{eventId}(\mathsf{e})?x : \mathsf{tids}?y \rightarrow \mathsf{exit}; \ Inactive \end{array} \right)$$

within

$(Inactive \ [\![\ \mathsf{flowtrigevts} \ ]\!] \ \mathsf{composeStates}^{1}(\langle x : \mathsf{states}^{15}(\mathsf{s}) \rangle, \mathsf{s}))$

where

$\#\mathsf{states}^{16}(\mathsf{s}) > 0$

$\mathsf{flowtrigevts} = \mathsf{flowTriggerEvents}^{2}(\mathsf{s})$

$\mathsf{sids} = SIDS \setminus \{\mathsf{id}(\mathsf{s})\}$

$\mathsf{exit} = exit?x : \mathsf{sids}.\mathsf{id}(\mathsf{s}) \rightarrow \mathsf{exitSubstates}^{4}(\mathsf{s}); \ \underbrace{[\![\mathsf{s.exit}]\!]}_{\mathcal{A}ction^{16}} ; \ exited.x.\mathsf{id}(\mathsf{s}) \rightarrow Skip$

$\mathsf{tids} = TIDS \setminus \mathsf{tIDS}(\mathsf{s})$

Rule 126. Semantics of trigger deadlines

triggerDeadlines(s : State) : TimedCSPProcess =

$$\begin{array}{l}
\textbf{let} \\
\quad Deadline(t) \mathrel{\widehat=} \left(\begin{array}{l} deadline.\mathsf{id}(\mathsf{t}).on \rightarrow \\ \quad \mathsf{readState}^{3} \left(\begin{array}{l} \mathsf{usedVariables}^{9}(\mathsf{t.end}), \\ (deadline.\mathsf{id}(\mathsf{t}).off \rightarrow Skip) \blacktriangleright [\![\mathsf{t.end}]\!]_{\mathcal{E}xpr^{82}} \end{array} \right); \\ \quad\quad Deadline(t) \end{array} \right) \\
\textbf{within} \\
\quad ||| \, t : \mathsf{tDS} \bullet Deadline(t)
\end{array}$$

where

$\mathsf{tDS} = \{\mathsf{t} : \mathsf{transitionsFrom}^{7}(\mathsf{s}) \mid \mathsf{t.end} \neq \mathsf{null}\}$

The composition of states is also largely unchanged when compared to the untimed Rule 28 except that the set shflowevts is not hidden, so as to allow a parent to observe all of its children's flow events, and the state-machine to observe *entered* events required to reset an implicit clock in the case of sinceEntry(s).

Rule 127. Composition of states

$\mathsf{composeStates}(ss : \mathbf{seq}\ \mathsf{State}, p : \mathsf{NodeContainer}) : \mathsf{TimedCSPProcess} =$

$\mathsf{if}\ \#ss = 1$

then

$\quad \mathsf{restrictedState}^{3}(p, \mathsf{head}\ ss)$

else

$$\left(\begin{array}{l} \mathsf{restrictedState}^{4}(p, \mathsf{head}\ ss) \\ \quad [|\mathsf{shflowevts}|] \\ \mathsf{composeStates}^{2}(\mathsf{tail}\ ss, p) \end{array} \right)$$

where

$\mathsf{shflowevts} = \mathsf{flowEvents}^{3}(\mathsf{head}\ ss, p) \cap \bigcup \{x : \mathsf{tail}\ ss \bullet \mathsf{flowEvents}^{4}(x, p)\}$

4.2.3 Timed statements

Rule 128. Semantics of statements

$[\![s : \mathsf{Statement}]\!]_{\mathit{Statement}} : \mathsf{TimedCSPProcess} =$

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

Rule 129. Semantics of statement deadlines

$[\![s : \mathsf{TimedStatement}]\!]_{\mathit{Statement}} : \mathsf{TimedCSPProcess} =$

$[\![s.\mathsf{stmt}]\!]_{\mathit{Statement}^{2}} \blacktriangleright [\![s.\mathsf{end}]\!]_{\mathit{Expr}^{83}}$

Rule 130. Semantics of wait

$[\![\mathsf{s} : \mathsf{Wait}]\!]_{\mathcal{S}tatement} : \mathsf{TimedCSPProcess} =$

$[\![\mathsf{s.duration}]\!]_{\mathcal{W}ait}$

where

$[\![\mathsf{e} : \mathsf{RangeExp}]\!]_{\mathcal{W}ait} = \sqcap n : [\![\mathsf{e}]\!]_{\mathcal{E}xpr} \bullet \mathit{Wait}(n)$

$[\![\mathsf{e} : \mathsf{Expression}]\!]_{\mathcal{W}ait} = \mathit{Wait}([\![\mathsf{e}]\!]_{\mathcal{E}xpr})$

Rule 131. Semantics of clock reset

$[\![\mathsf{s} : \mathsf{ClockReset}]\!]_{\mathcal{S}tatement} : \mathsf{TimedCSPProcess} =$

$\mathit{clockReset}.\mathsf{id(s.clock)} \rightarrow \mathit{Skip}$

Rule 132. Semantics of assignment

$[\![\mathsf{s} : \mathsf{Assignment}]\!]_{\mathcal{S}tatement} : \mathsf{CSPProcess} =$

$(\mathit{set_}\mathsf{vid(s.left)}![\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^{84}} \rightarrow \mathit{Skip}) \blacktriangleright 0$

Rule 133. Semantics of call statement

$[\![\mathsf{s : Call}]\!]_{\mathcal{Statement}} : \mathsf{CSPProcess} =$

$$(\mathsf{op.name}Call \rightarrow Skip \blacktriangleright 0);\ \mathsf{body};\ (\mathsf{op.name}Ret \rightarrow Skip) \blacktriangleright 0$$

where

$\mathsf{op} = \mathsf{s.operation}$

$\mathsf{opdef} = \mathsf{findOperationDefinition(op)}$

$\mathsf{SkipAnytime} = (Wait(1);\ \mathsf{SkipAnytime}) \sqcap Skip$

$$\mathsf{body} = \left(\begin{array}{l} \mathsf{if\ (opdef = null)\ then} \\ \quad (\mathsf{SkipAnytime} \sqcap Stop) \\ \mathsf{else} \\ \quad [\![\mathsf{opdef}]\!]_{\mathcal{STM}^f} (\{\mathsf{x : s.args} \bullet [\![\mathsf{x}]\!]_{\mathcal{E}xpr^{85}}\}) \end{array}\right);$$

Chapter 5

Collections

This chapter describe the RoboChart extensions designed to support modelling, analysis and simulation of collections of robots. Section 5.1 describes the extensions of the metamodel of RoboChart, Section 5.2 describes the conditions that characterise well-formed RoboChart collections, and Section 5.3 specifies the semantics of collections based on the untimed and timed semantics in Section 4.

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

5.1 Metamodel

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

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

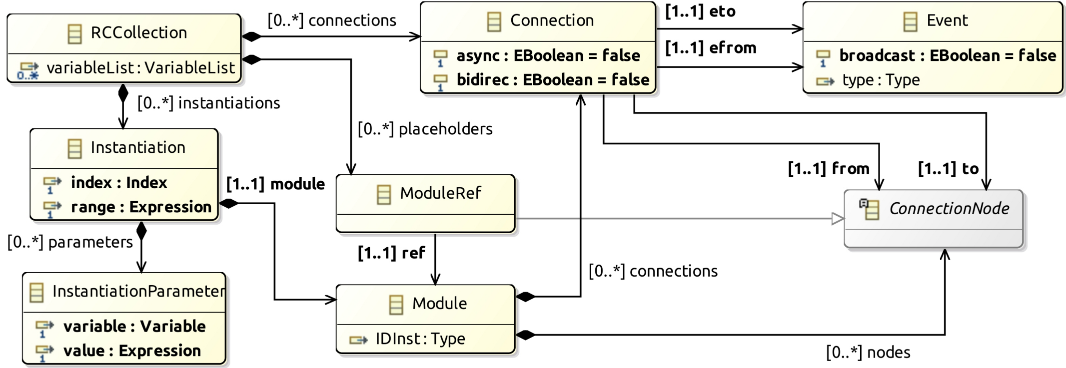


Figure 5.1: Metamodel for collections: RCCollection and Event

Figure 5.1 shows the part of the extensions of the metamodel. Collections are specified by RCCollectionss, which include:

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

An Instantiation can include InstantiationParameters that are used to initialise constants of a module. Events are extended with a boolean attribute that is used to determine whether or not it a broadcast event.

Finally, Triggers (shown in Figure 5.2) are extended with two new attributes:

- _from is used to identify a variable in which to record the identifier of the source of the communication; and
- _predicate is used to restrict the potential targets of communication. For example, an empty predicate is equivalent to *true*, and results on the message being sent to all possible

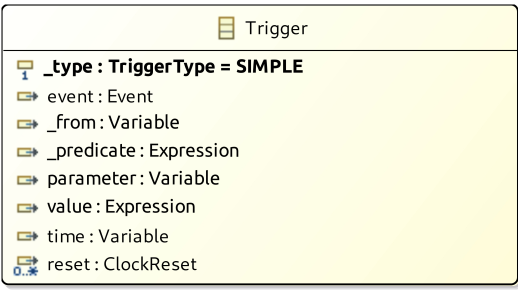


Figure 5.2: Metamodel for collections: Trigger

targets (determined by the set of identifiers that characterise the target event), while a predicate such as from = v sends the message only to the target whose identifier is recorded in the variable v.

5.2 Well-formedness Conditions

5.2.1 RCCollection

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

5.2.2 Instantiation

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

5.2.3 Event

- *Events connected in a collection must be broadcast events.* Events connected in a collection model some form of communication, which in its most general form is a broadcast. Further restriction over the patterns of communication can be modelled internally using the _predicate attribute of triggers.

- *Connections (at any level) involving a broadcast event in one end must link to another broadcast event.* The broadcast nature of the event is part of its type and affects the semantics of triggers, therefore the ends of the connection must be compatible.

5.2.4 Trigger

- *A transition trigger must not record a value for* _predicate. In the context of broadcast communications, transition triggers are interpreted as input communications, in which restriction of the targets is meaningless.

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

Rule 1. Semantics of Collections

$[\![\mathsf{c} : \mathsf{RCCollection}]\!]_{\mathcal{C}ol} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\interleave\ \mathsf{inst} : \mathsf{c.instantiations} \bullet \interleave\ i : \mathsf{inst.range} \bullet [\![\mathsf{inst.module}]\!]_{\mathcal{M}'}(i) \\
\quad [|\{|e_1, e_2 \mid (e_1, e_2) \leftarrow \mathsf{connectedEvents(c)}|\}|] \\
\left(
\begin{array}{l}
\left(
\begin{array}{l}
\interleave\ \mathsf{conn} : \mathsf{c.connections} \bullet \interleave\ (i,j) : \mathsf{inds(conn, c)} \bullet \\
\quad \mathsf{BBuffer}^1(\mathsf{eventId(conn.efrom), i, eventId(conn.eto), j})
\end{array}
\right) \\
\interleave \\
\left(
\begin{array}{l}
\interleave\ \mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \land \mathsf{heterogeneous} \bullet \interleave\ (i,j) : \mathsf{inds(conn, c)} \bullet \\
\quad \mathsf{BBuffer}^2(\mathsf{eventId(conn.eto), j, eventId(conn.efrom), i})
\end{array}
\right)
\end{array}
\right)
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{connectedEvents(c : Collection)} : \mathbb{P}(\mathsf{Event} \times \mathsf{Event}) = \\
\quad \{\mathsf{conn} : \mathsf{c.connections} \bullet (\mathsf{eventId(conn.efrom), eventId(conn.eto)})\} \\
\quad \cup \\
\quad \{\mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \bullet (\mathsf{eventId(conn.eto), eventId(conn.efrom)})\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{inds(conn : Connection, c : Collection)} : \mathbb{P}(\mathsf{ID} \times \mathsf{ID}) = \\
\quad \mathsf{if\ conn.from.ref = conn.to.ref\ then} \\
\quad\quad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)} \setminus \{i : \mathsf{range(conn.to, c)} \bullet (i, i)\} \\
\quad \mathsf{else} \\
\quad\quad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)}
\end{array}
$$

$$\mathsf{range(m : Module, c : Collection)} : \mathbb{P}\,\mathsf{ID} = (\iota\, \mathsf{i} : \mathsf{c.instantiations} \mid \mathsf{i.module} = \mathsf{m}).\mathsf{range}$$

$$\mathsf{heterogeneous} = (\mathsf{c.from.ref} \neq \mathsf{c.to.ref})$$

Rule 2. Broadcast Buffer

$\mathsf{BBuffer}(\mathsf{e_{in}} : \mathsf{Event}, \mathsf{i} : \mathsf{ID}, \mathsf{e_{out}} : \mathsf{Event}, \mathsf{j} : \mathsf{ID}) : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\textit{let} \\
\quad \textit{BufferEmpty} = \mathsf{prefixIn} \rightarrow \textit{BufferFull}(x) \\
\quad \textit{BufferFull}(v) = \mathsf{prefixIn} \rightarrow \textit{BufferFull}(x) \;\square\; \mathsf{prefixOut} \rightarrow \textit{BufferEmpty} \\
\textit{within} \\
\quad \textit{BufferEmpty}
\end{array}
$$

where

$\mathsf{e_{in}.broadcast}$

$\mathsf{prefixIn} = \mathsf{if}\ \mathsf{e_{in}.type} \neq \mathsf{null}\ \mathsf{then}\ \mathsf{eventId}(\mathsf{e_{in}})?x\ \mathsf{else}\ \mathsf{eventId}(\mathsf{e_{in}})$

$\mathsf{prefixOut} = \mathsf{if}\ \mathsf{e_{out}.type} \neq \mathsf{null}\ \mathsf{then}\ \mathsf{eventId}(\mathsf{e_{out}})!v\ \mathsf{else}\ \mathsf{eventId}(\mathsf{e_{out}})$

Rule 3. Semantics of triggers

$[\![\mathsf{t} : \mathsf{Trigger}]\!]^{\mathsf{tid}}_{\mathit{Trigger}} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
\mathsf{if}\ \mathsf{t.type} = \mathsf{INPUT}\ \mathsf{then} \\
\quad \mathsf{eventId}(\mathsf{t.event}).\textit{tid}?f__!\textit{id}_?x__ \rightarrow \textit{set}_\mathsf{vid}(\mathsf{t._from})!f__ \rightarrow \textit{set}_\mathsf{vid}(\mathsf{t.parameter})!x__ \rightarrow \textit{Skip} \\
\mathsf{else\ if}\ \mathsf{t.type} = \mathsf{SIMPLE}\ \mathsf{then} \\
\quad \mathsf{eventId}(\mathsf{t.event}).\textit{tid}?f__!\textit{id}_ \rightarrow \textit{set}_\mathsf{vid}(\mathsf{t._from})!f__ \rightarrow \textit{Skip} \\
\mathsf{else}\ \textit{These cases do not occur when the event is broadcast}
\end{array}
$$

where

$\mathsf{t.event.broadcast}$

Rule 4. Semantics of send event statements

$[\![\mathsf{s} : \mathsf{SendEvent}]\!]_{\mathcal{S}tatement} : \mathsf{CSPProcess} =$

if t.type = OUTPUT $\vee$ t.type = SYNC then

$\quad ||| \; i : \{x : ID \mid [\![\mathsf{t._predicate}]\!]_{\mathsf{Expr}^{86}}\} \bullet \mathsf{eventId(t.event)}!id_!i![\![\mathsf{t.value}]\!]_{\mathsf{Expr}^{87}} \rightarrow Skip$

else if t.type = SIMPLE then

$\quad ||| \; i : \{x : ID \mid [\![\mathsf{t._predicate}]\!]_{\mathsf{Expr}^{88}}\} \bullet \mathsf{eventId(t.event)}!id_!i \rightarrow Skip$

else *These cases do not occur when the event is broadcast*

where

s.trigger.broadcast

Chapter 6

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 [15] and CSP, and includes time constructs [14]. 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 [16].

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.

Appendix A

RoboChart diagrams - an informal overview

To illustrate the concepts, we present the model of a robot for chemical detection based on that in [7] [1]. In our example, the robot employs a random walk and, upon detection of a chemical source, it turns on a light and drops a flag.

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

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

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

The operations move(lv,a) and shortRandomWalk() are defined separately, just to indicate that they terminate; randomWalk() is left undefined. Further elaboration of the model may include a definition for these operations via state machines, or via pre and postconditions. Such def-

[1] http://tinyurl.com/hdaws7o

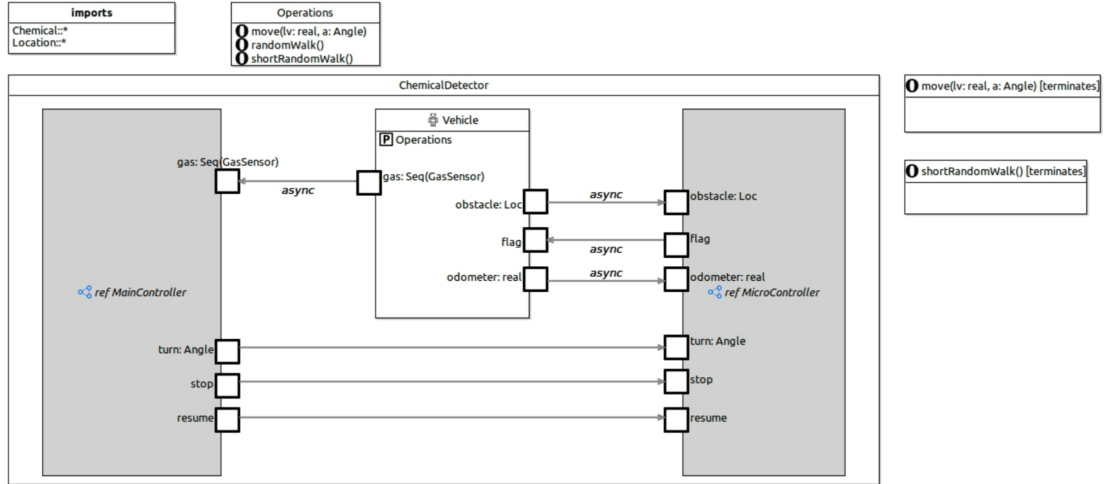


Figure A.1: Chemical Detector

initions would have the purpose to support reasoning, but since these are operations that we declare to be provided by the platform, they do not need to be implemented. The fact that Vehicle declares Operations as a provided (P) interface makes this clear.

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

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

Communication with a robotic platform is always asynchronous, but communication between controllers can be synchronous or asynchronous. In our example, the communication between the controllers and the platform via the gas, obstacle, flag, and odometers events is asynchronous, as indicated by the label async on the arrows that represent the connections. That label is used on all connections with a robotic platform.

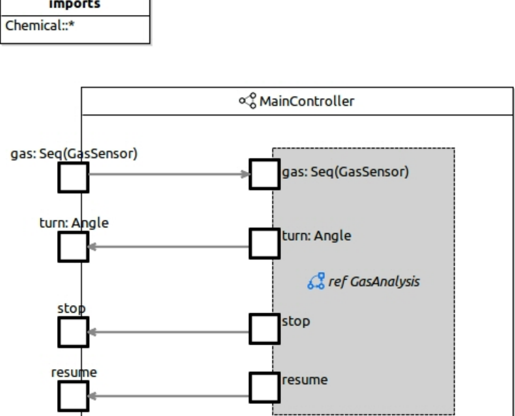


Figure A.2: MainController

The connections between the controllers are all synchronous in this example. This is an abstraction, since typically controllers of a robot communicate asynchronously.

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

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

The events of a controller can be connected to those of the state machines that defines it. Communication between states machines is always synchronous, since parallelism at this level is used for convenience of modelling, rather than to indicate concrete designs.

MainController is defined by a single state machine GasAnalysis. It is referenced in the definition of MainController in Figure A.2, and defined in Figure A.4. The controller in this case just relays its events to and from the state machine.

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

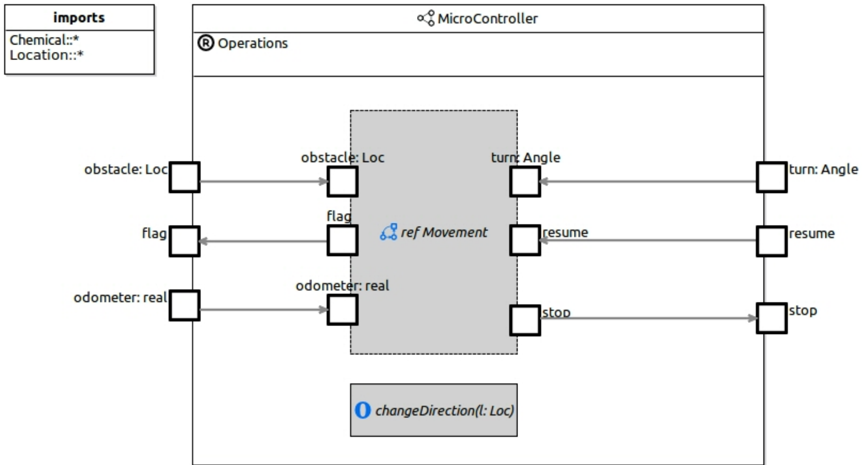


Figure A.3: MicroController

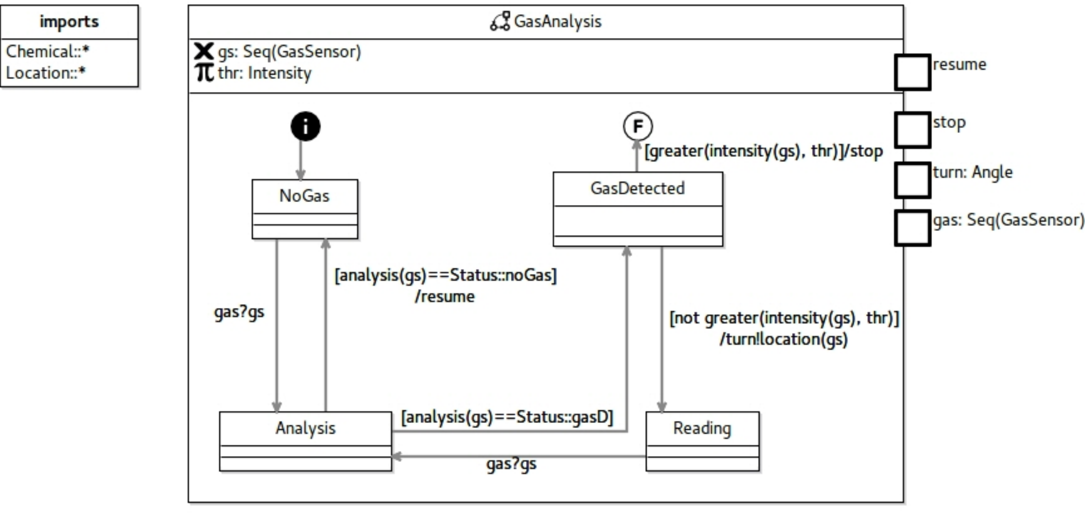


Figure A.4: GasAnalysis state machine

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

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

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

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

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

RoboChart state machines are standard, but restricted and with a well defined semantics. They can have composed states, junctions, and entry, during, exit, and transition actions defined using a well defined action language. Features of UML state machines [10] deemed not essential for robotics are not included, resulting in a streamlined semantics.

Definitions of a model can be organised in packages. Like in UML, they are just containers. They do not correspond to a concept or abstraction, and so do not have an interface. An imports mechanism controls scope of the definitions. All packages that do not have a package name conceptually compose the same package. Elements defined in the unnamed package are available in all other packages. Elements defined in a package with a name can only be used if

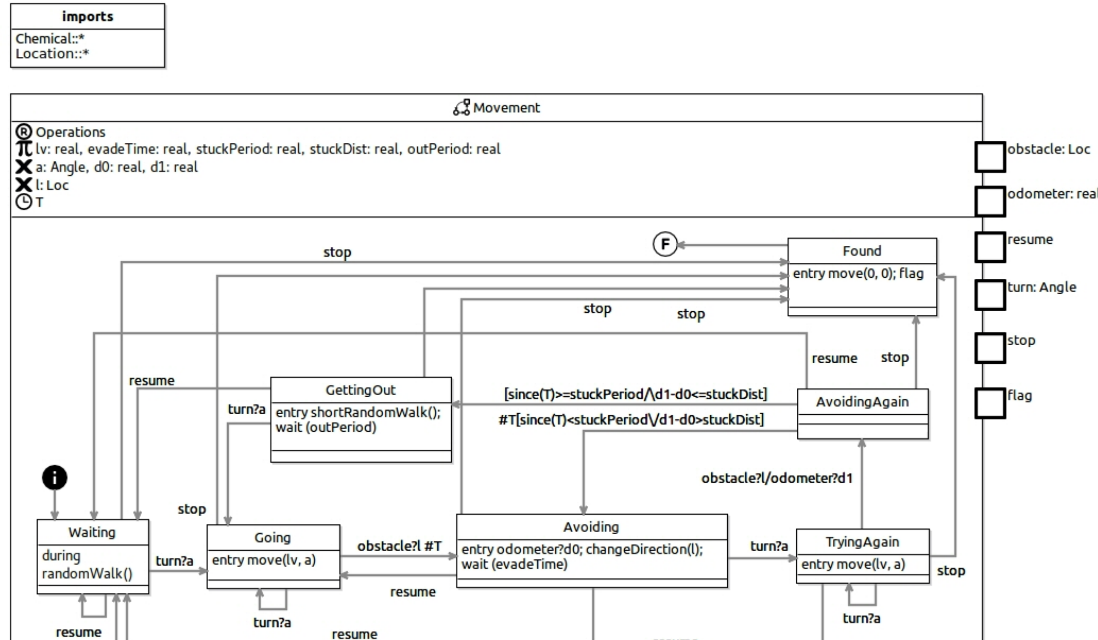


Figure A.5: Movement state machine

they are explicitly imported. A model is identified by one module and all the other elements there. Figures A.6 and A.7 define two packages used in our example.

In Chemical, we specify a data model for handling the gas sensors. This involves a number of types, some of which are just named, like Chem, and a number of functions acting on those types. Functions are either left undefined, like greater, or defined by pre and postconditions, like intensity. To define these conditions, RoboChart provides a simple predicate language.

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

A.1 Time primitives

RoboChart operations take zero time, and enabled transitions take place as soon as they can be triggered. Time constraints need to be explicitly defined. In Table 2.6 we summarize the syntax of all timed constructs that can be used in the definition of state machines.

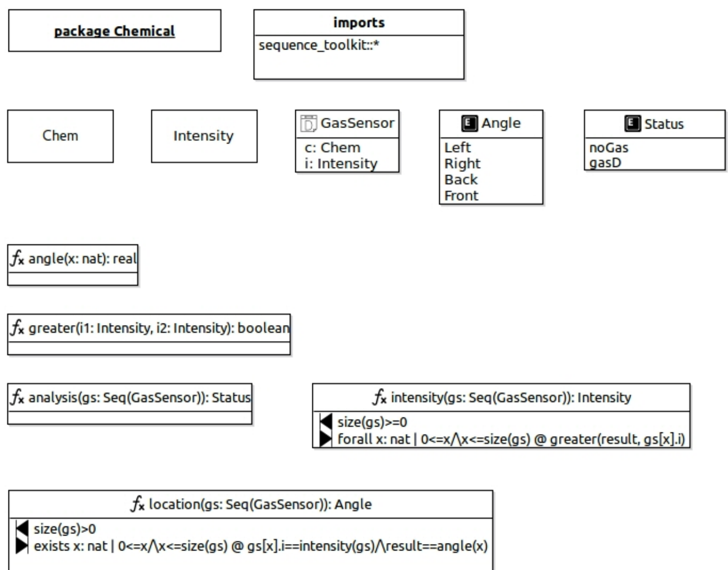


Figure A.6: Chemical package

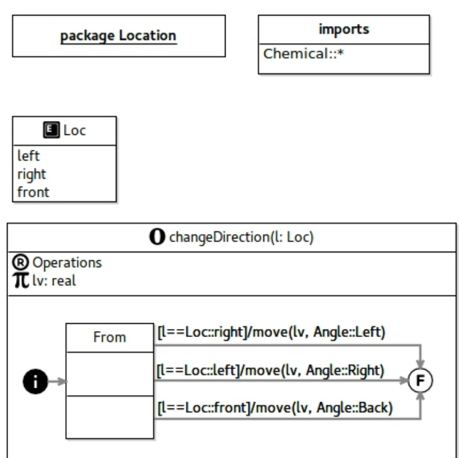


Figure A.7: Location package

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

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

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

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

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

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

When the robot is started, it transitions from the initial state, denoted by a black circle, to the state MovingForward, while resetting (#C) a clock C and assigning 1 to a local variable segment. A RoboChart state machine is self-contained, in that it declares all the variables, events, and operations that it uses. In Figure A.8 two constants linear and angular are defined, to represent the linear and angular speed, respectively. The local variable segment records how many sides of the square have been covered so far; the robot stops when it completes

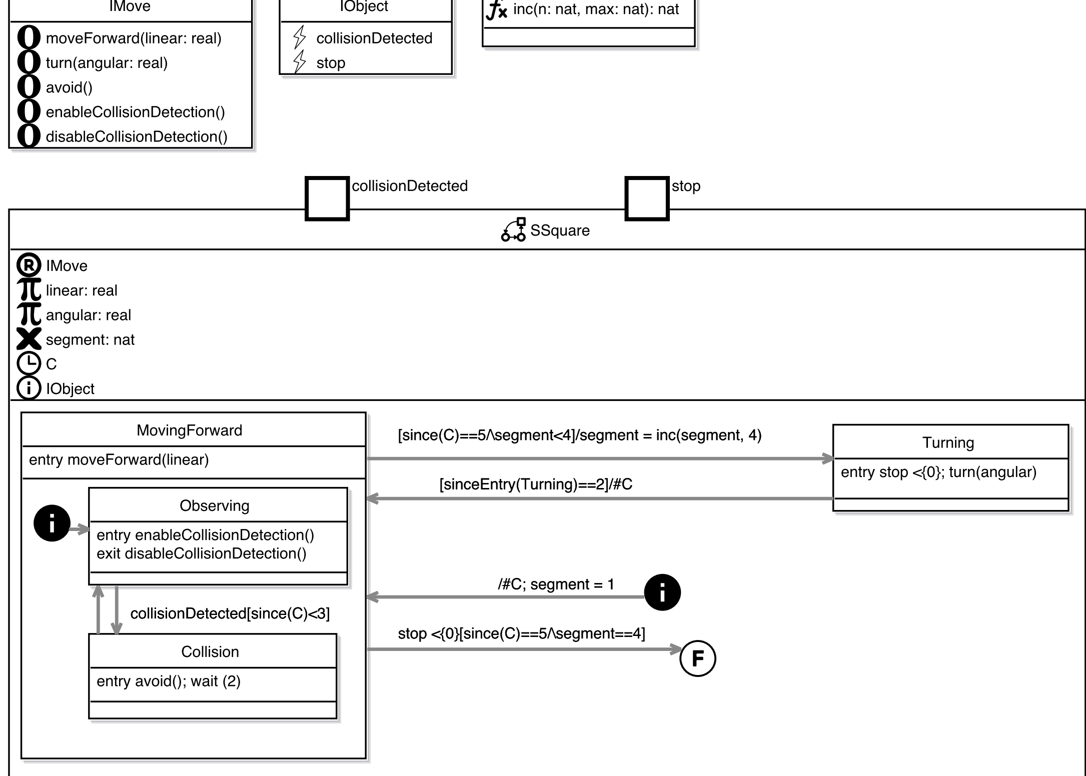


Figure A.8: Square-trajectory robot

the square (segment == 4). This is achieved by sending an event stop to the platform and transitioning to the final state: a white circle. The event stop is given a deadline 0, indicating that it is expected that the robotic platform is always ready to accept this event immediately.

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

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

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

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

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

Appendix B

Complete Metamodel

This appendix contains the complete metamodel specified in Ecore and formatted by the tool `OCLinEcore`. The syntax of the representation used in this appendix is available here.

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

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

package robochart:robochart = 'http://www.robocalc.circus/text/RoboChart'
    {
class RCPackage extends MachineContainer {
  attribute name : String[?];
  property imports : Import[*|1] { composes };
  property interfaces : Interface[*|1] { ordered composes };
  property robots : RoboticPlatform[*|1] { ordered composes };
  property types : TypeDecl[*|1] { ordered composes };
  property controllers : Controller[*|1] { ordered composes };
  property modules : Module[*|1] { ordered composes };
  property operations : Operation[*|1] { ordered composes };
  property functions : Function[*|1] { ordered composes };
  property collections : RCCollection[*|1] { ordered composes };
}
abstract class MachineContainer
{
  property machines : StateMachine[*|1] { ordered composes };
}
abstract class OperationContainer;
class Import
{
  attribute importedNamespace : String[1];
}
abstract class NamedElement
{
  attribute name : String[1];
```

```
}
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 };
}
class VariableList
{
  attribute modifier : String[1];
  property vars : Variable[*|1] { ordered composes };
}
// The attributed modifier of a variable is derived from the modifier of
// a variable list.
class Variable extends NamedExpression
{
  attribute name : String[1];
  property type : Type[1] { composes };
  property initial : Expression[?] { composes };
  attribute modifier : String[?] { derived transient volatile };
}
abstract class Operation extends NamedElement;
class OperationSig extends Operation
{
  property parameters : Parameter[*|1] { ordered composes };
  attribute terminates : Boolean[1];
  property preconditions : Expression[*|1] { ordered composes };
  property postconditions : Expression[*|1] { ordered composes };
}
class Parameter extends Variable;
class OperationRef extends OperationSig,ConnectionNode
{
  property ref : OperationDef[1];
}
class OperationDef extends OperationSig,StateMachineBody,ConnectionNode;
class Function extends NamedElement,NamedExpression
{
  property parameters : Parameter[*|1] { ordered composes };
```

```
  property type : Type[1] { composes };
  property preconditions : Expression[*|1] { ordered composes };
  property postconditions : Expression[*|1] { ordered composes };
}
class Event extends NamedElement
{
  property type : Type[?] { composes };
  attribute broadcast : Boolean[1];
}
class Member extends NamedElement
{
  property type : Type[1] { composes };
}
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[1] { composes };
  property range : Type[1] { composes };
}
class RelationType extends Type
{
  property domain : Type[1] { composes };
  property range : Type[1] { composes };
}
class SetType extends Type
{
  property domain : Type[1] { composes };
}
class SeqType extends Type
{
  property domain : Type[1] { composes };
}
class TypeRef extends Type
{
  property ref : TypeDecl[1];
}
```

```
class TypeDecl extends NamedElement;
class PrimitiveType extends TypeDecl;
class DataType extends TypeDecl
{
  property fields : Field[*|1] { ordered composes };
}
class Field extends Member,NamedExpression;
class Enumeration extends TypeDecl
{
  property constants : Constant[*|1] { ordered composes };
}
abstract class NamedExpression;
class Constant extends TypeDecl,NamedExpression
{
  property types : Type[*] { ordered composes };
}
class Interface extends NamedElement,BasicContext;
abstract class RoboticPlatform extends ConnectionNode;
class RoboticPlatformDef extends NamedElement,Context,RoboticPlatform;
class RoboticPlatformRef extends NamedElement,RoboticPlatform
{
  property ref : RoboticPlatformDef[1];
}
abstract class ConnectionNode;
abstract class Controller extends ConnectionNode;
class ControllerDef extends NamedElement,Context,Controller,
    MachineContainer
{
  property connections : Connection[*|1] { ordered composes };
}
class ControllerRef extends NamedElement,Controller
{
  property ref : ControllerDef[1];
}
abstract class NodeContainer
{
  property nodes : Node[*|1] { ordered composes };
  property transitions : Transition[*|1] { ordered composes };
}
class StateMachineBody extends Context,NodeContainer
{
  property clocks : Clock[*|1] { ordered composes };
}
abstract class StateMachine extends ConnectionNode;
```

```
class StateMachineDef extends NamedElement,StateMachineBody,StateMachine;
class StateMachineRef extends Node,StateMachine
{
  property ref : StateMachineDef[1];
}
abstract class Node extends NamedElement;
class State extends Node,NodeContainer
{
  property actions : Action[*|1] { ordered composes };
}
class Junction extends Node;
class Initial extends Junction;
class Final extends State;
class ProbabilisticJunction extends Node;
class Transition extends NamedElement
{
  property source : Node[1];
  property target : Node[1];
  property start : Expression[?] { composes };
  property trigger : Trigger[?] { composes };
  property end : Expression[?] { composes };
  property condition : Expression[?] { composes };
  property action : Statement[?] { composes };
  property probability : Expression[?] { composes };
}
class ClockReset extends Statement
{
  property clock : Clock[1];
}
class Clock extends NamedElement;
abstract class Action
{
  property action : Statement[1] { composes };
}
class EntryAction extends Action;
class DuringAction extends Action;
class ExitAction extends Action;
abstract class Statement;
class TimedStatement extends Statement
{
  property start : Expression[?] { composes };
  property stmt : Statement[1] { composes };
  property end : Expression[?] { composes };
}
```

```
class Wait extends Statement
{
  property duration : Expression[1] { composes };
}
class Skip extends Statement;
class IfStmt extends Statement
{
  property expression : Expression[1] { composes };
  property _'then' : Statement[1] { composes };
  property _'else' : Statement[1] { composes };
}
class Assignment extends Statement
{
  property left : Assignable[1] { composes };
  property right : Expression[1] { composes };
}
class SendEvent extends Statement
{
  property trigger : Trigger[1] { composes };
}
class SeqStatement extends Statement
{
  property statements : Statement[*|1] { ordered composes };
}
class Call extends Statement
{
  property operation : Operation[1];
  property args : Expression[*|1] { ordered composes };
}
abstract class Expression;
class ResultExp extends Expression;
class ArrayExp extends Expression
{
  property value : Expression[1] { composes };
  property parameters : Expression[*|1] { composes };
}
class ClockExp extends Expression
{
  property clock : Clock[1];
}
class StateClockExp extends Expression
{
  property state : State[1];
}
```

```
class Iff extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Implies extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Or extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[?] { composes };
}
class Forall extends Expression
{
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[?] { composes };
  property predicate : Expression[1] { composes };
}
class Exists extends Expression
{
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[?] { composes };
  property predicate : Expression[1] { composes };
  attribute unique : Boolean[1];
}
class LambdaExp extends Expression
{
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[?] { composes };
  property expression : Expression[1] { composes };
}
class DefiniteDescription extends Expression
{
  property variables : Variable[*|1] { ordered composes };
  property suchthat : Expression[?] { composes };
  property expression : Expression[1] { composes };
}
class IfExpression extends Expression
{
  property condition : Expression[1] { composes };
  property ifexp : Expression[1] { composes };
```

```
  property elseexp : Expression[1] { composes };
}
class Declaration extends NamedExpression
{
  attribute name : String[1];
  property value : Expression[1] { composes };
}
class LetExpression extends Expression
{
  property declarations : Declaration[*|1] { ordered composes };
  property expression : Expression[1] { composes };
}
class And extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Not extends Expression
{
  property exp : Expression[1] { composes };
}
class InExp extends Expression
{
  property member : Expression[1] { composes };
  property set : Expression[1] { composes };
}
class TypeExp extends Expression
{
  property type : Type[1] { composes };
}
class Equals extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Different extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class GreaterThan extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
```

```
}
class GreaterOrEqual extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class LessThan extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class LessOrEqual extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Plus extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Minus extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Modulus extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Mult extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Div extends Expression
{
  property left : Expression[1] { composes };
  property right : Expression[1] { composes };
}
class Cat extends Expression
{
  property left : Expression[1] { composes };
```

```
  property right : Expression[1] { composes };
}
class Neg extends Expression
{
  property exp : Expression[1] { composes };
}
class Selection extends Expression
{
  property receiver : Expression[1] { composes };
  property member : Member[1];
}
class IntegerExp extends Expression
{
  attribute value : ecore::EInt[1];
}
class FloatExp extends Expression
{
  attribute value : ecore::EFloat[1];
}
class StringExp extends Expression
{
  attribute value : String[1];
}
class BooleanExp extends Expression
{
  attribute value : String[1];
}
class VarExp extends Expression
{
  property value : Variable[1];
}
class RefExp extends Expression
{
  property ref : NamedExpression[1];
}
class ToExp extends Expression;
class FromExp extends Expression;
class IdExp extends Expression;
class AsExp extends Expression
{
  property exp : Expression[1] { composes };
  property type : Type[1] { composes };
}
class IsExp extends Expression
```

```
{
  property exp : Expression[1] { composes };
  property type : Type[1] { composes };
}
class EnumExp extends Expression
{
  property type : Enumeration[1];
  property constant : Constant[1];
}
class ParExp extends Expression
{
  property exp : Expression[1] { 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 : Expression[1] { composes };
  property args : Expression[*|1] { ordered composes };
}
class ElseExp extends Expression;
abstract class Assignable;
class VarSelection extends Assignable
{
  property receiver : Assignable[1] { composes };
  property member : Member[1];
}
class ArrayAssignable extends Assignable
{
  property value : Assignable[1] { composes };
  property parameters : Expression[*|1] { composes };
}
class VarRef extends Assignable
{
  property name : Variable[1];
}
class Connection
{
  property from : ConnectionNode[1];
  property efrom : Event[1];
  property to : ConnectionNode[1];
  property eto : Event[1];
  attribute async : Boolean[1];
  attribute bidirec : Boolean[1];
}
class Module extends NamedElement
{
  property connections : Connection[*|1] { ordered composes };
  property nodes : ConnectionNode[*|1] { ordered composes };
  property IDInst : Type[?] { composes };
}
class ModuleRef extends NamedElement,ConnectionNode
{
  property ref : Module[1];
}
class InstantiationParameter
{
  property variable : Variable[1];
  property value : Expression[1] { composes };
}
```

```
class Index extends NamedExpression
{
  attribute name : String[1];
}
class Instantiation
{
  property index : Index[1] { composes };
  property range : Expression[1] { composes };
  property module : Module[1];
  property parameters : InstantiationParameter[*|1] { ordered composes };
}
enum Nature { serializable }
{
  literal BROADCAST;
  literal P2P;
}
class RCCollection extends NamedElement
{
  property instantiations : Instantiation[*|1] { ordered composes };
  property placeholders : ModuleRef[*|1] { ordered composes };
  property connections : Connection[*|1] { ordered composes };
  property variableList : VariableList[*|1] { ordered composes };
}
```

Appendix C

Mathematical Toolkit

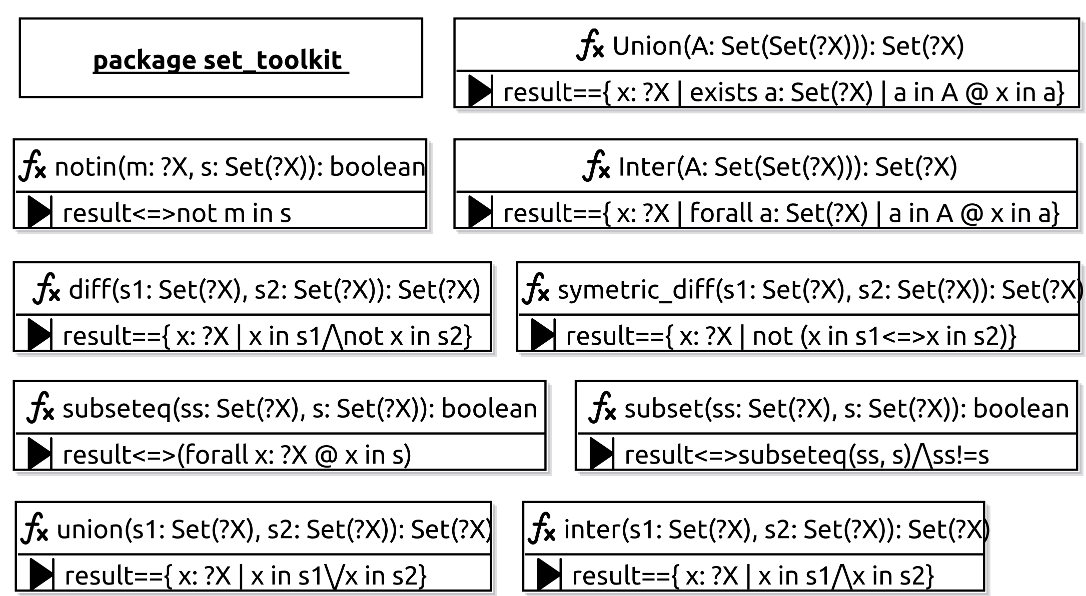
This section presents the functions of the Z mathematical toolkit as modelled in RoboChart.

A package named core contains a number of primitive types and is imported by default in every RoboChart model. They are shown in Table C.1.

Type	Syntax
Natural numbers	nat
Integers	int
Strings	string
Booleans	boolean
Real numbers	real

Table C.1: Primitive types in core package.

The functions are grouped in the set, relation, function, and sequence toolkits.



package relation_toolkit

imports

```
set_toolkit::*
```

```
fx relcomp(r: ?X<->?Y, s: ?Y<->?Z): ?X<->?Z
result=={ p: ?X*?Y, q: ?Y*?Z | p in r/\q in s/\p[2]==q[1] @ maplet(p[1], q[2])}
```

```
fx tr_closure(r: ?X<->?X): ?X<->?X
result==Inter({ s: ?X<->?X | subseteq(r, s)/\subseteq(relcomp(r, s), s)})
```

```
fx dom(r: ?X<->?Y): Set(?X)
result=={ p: ?X*?Y | p in r @ p[1]}
```

```
fx dres(a: Set(?X), r: ?X<->?Y): ?X<->?Y
result=={ p: ?X*?Y | p in r/\p[1] in a}
```

```
fx rimage(r: ?X<->?Y, a: Set(?X)): Set(?Y)
result=={ p: ?X*?Y | p in r/\p[1] in a @ p[2]}
```

```
fx override(r: ?X<->?Y, s: ?X<->?Y): ?X<->?Y
result==union(dsub(dom(s), r), s)
```

```
fx first(p: ?X*?Y): ?X
result==p[1]
```

```
fx rinv(r: ?X<->?Y): ?Y<->?X
result=={ p: ?X*?Y | p in r @ maplet(p[2], p[1])}
```

```
fx refl_tr_closure(r: ?X<->?X): ?X<->?X
result==union(tr_closure(r), id())
```

```
fx second(p: ?X*?Y): ?Y
result==p[2]
```

```
fx rres(r: ?X<->?Y, b: Set(?Y)): ?X<->?Y
result=={ p: ?X*?Y | p in r/\p[2] in b}
```

```
fx dsub(a: Set(?X), r: ?X<->?Y): ?X<->?Y
result=={ p: ?X*?Y | p in r/\not p[1] in a}
```

```
fx maplet(x: ?X, y: ?Y): ?X*?Y
result==(|x, y|)
```

```
fx funcomp(s: ?Y->?Z, r: ?X->?Y): ?X<->?Z
result==relcomp(r, s)
```

```
fx ran(r: ?X<->?Y): Set(?Y)
result=={ p: ?X*?Y | p in r @ p[2]}
```

```
fx id(): ?X<->?X
result=={ x: ?X | maplet(x, x)}
```

```
fx rsub(r: ?X<->?Y, b: Set(?Y)): ?X<->?Y
result=={ p: ?X*?Y | p in r/\not p[2] in b}
```

```
package function_toolkit

imports
relation_toolkit::*
set_toolkit::*

fx isFinite(s: Set(?X)): boolean
result<=>(exists n: nat, g: nat->?X @ isBijection(g)/\dom(g)=={ x: nat | x>=1/\x<=n}/\ran(g)==s)

fx isInjection(f: ?X->?Y): boolean
result<=>(forall p: ?X*?Y, q: ?X*?Y | p in f/\q in f @ p[1]==q[1]<=>p[2]==q[2])

fx disjoint(f: ?L<->Set(?X)): boolean
result<=>(forall p: ?L*Set(?X), q: ?L*Set(?X) | p in f/\q in f/\p!=q @ inter(p[2], q[2])=={})

fx isTotal(f: ?X->?Y): boolean
result<=>(forall x: ?X @ exists y: ?Y @ (|x, y|) in f)

fx isFiniteFunction(f: ?X->?Y): boolean
result<=>isFinite(f)

fx isFiniteInjection(f: ?X->?Y): boolean
result<=>isFinite(f)/\isInjection(f)

fx isBijection(f: ?X->?Y): boolean
result<=>isTotalInjection(f)/\isTotalSurjection(f)

fx isTotalSurjection(f: ?X->?Y): boolean
result<=>isTotal(f)/\isSurjection(f)

fx isTotalInjection(f: ?X->?Y): boolean
result<=>isTotal(f)/\isInjection(f)

fx partitions(f: ?L<->Set(?X), a: Set(?X)): boolean
result<=>(disjoint(f)/\Union(ran(f))==a)

fx isSurjection(f: ?X->?Y): boolean
result<=>(ran(f)=={ y: ?Y})
```

```
package sequence_toolkit

imports
function_toolkit::*
relation_toolkit::*
set_toolkit::*

fx dconcat(s: Seq(Seq(?X))): Seq(?X)
result==if size(s)==0 then <> else the t: Seq(?X) | concat(head(s), dconcat(tail(s)))==t @ t end

fx iter(n: int, r: ?X<->?X): ?X<->?X
result==if n==0 then id() else if n>0 then relcomp(r, iter(n-1, r)) else iter(-n, rinv(r)) end end

fx size(a: Set(?X)): nat
isFinite(a)
result==(the n: nat | (exists f: nat->?X | isBijection(f) @ dom(f)==range(1, n)/\ran(f)==a) @ n)

fx squash(f: int->?X): Seq(?X)
isFiniteFunction(f)
result=={ p: int*?X | p in f @ maplet(size({ i: int | i in dom(f) @ i<=p[1]}), p[2])}

fx max(a: Set(int)): int
hasMax(a)
result in a/\(forall n: int | n in a @ result>=n)

fx concat(s: Seq(?X), t: Seq(?X)): Seq(?X)
result==union(s, { n: int | n in dom(t) @ maplet(n+size(s), t[n])})

fx reverse(s: Seq(?X)): Seq(?X)
result==(lambda n: int | n in dom(s) @ s[size(s)-n+1])

fx infix(s: Seq(?X), t: Seq(?X)): boolean
result<=>(exists u: Seq(?X), v: Seq(?X) @ concat(u, concat(s, v))==t)

fx suffix(s: Seq(?X), t: Seq(?X)): boolean
result<=>(exists u: Seq(?X) @ concat(u, s)==t)

fx hasMax(a: Set(int)): boolean
result<=>(exists m: int | m in a @ (forall n: int | n in a @ m>=n))

fx front(s: Seq(?X)): Seq(?X)
result==dsub({size(s)}, s)

fx range(x: int, y: int): Seq(int)
result=={ i: int | x<=i/\i<=y}

fx hasMin(a: Set(int)): boolean
result<=>(exists m: int | m in a @ (forall n: int | n in a @ m<=n))

fx filter(s: Seq(?X), a: Set(?X)): Seq(?X)
result==squash(rres(s, a))

fx isNonEmpty(s: Seq(?X)): boolean
result<=>size(s)>0

fx tail(s: Seq(?X)): Seq(?X)
result==(lambda n: nat | n in range(1, size(s)-1) @ s[n+1])

fx extract(a: Set(int), s: Seq(?X)): Seq(?X)
result==squash(dres(a, s))

fx prefix(s: Seq(?X), t: Seq(?X)): boolean
result<=>subseteq(s, t)

fx min(a: Set(int)): int
hasMin(a)
result in a/\(forall n: int | n in a @ result<=n)

fx head(s: Seq(?X)): ?X
isNonEmpty(s)
s[1]

fx last(s: Seq(?X)): ?X
isNonEmpty(s)
s[size(s)]

fx isInjectiveSequence(s: Seq(?X)): boolean
result<=>isInjection(s)
```

Appendix D

Credits

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

Icon made by Iconnice from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Sarfraz Shoukat from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Dario Ferrando from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Lyolya from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Google from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

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

Icon made by Icomoon 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

π 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 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 Popcic from www.flaticon.com is licensed by CC 3.0 BY

Bibliography

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.

[4] S. Foster, B. Thiele, A. L. C. Cavalcanti, and J. C. P. Woodcock. Towards a UTP semantics for Modelica. In *Unifying Theories of Programming*, Lecture Notes in Computer Science. Springer, 2016.

[5] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 : A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

[6] T. A. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[7] J. A. Hilder, N. D. L. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. A. Kilgour, J. Timmis, and A. M. Tyrrell. Chemical detection using the receptor density algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1730–1741, 2012.

[8] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[9] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. Robochart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, pages 1–53, 2019.

[10] Object Management Group. OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Technical report, OMG, 2011.

[11] Object Management Group. *OMG Unified Modeling Language*, March 2015.

[12] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[13] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.

[14] A. Sherif, A. L. C. Cavalcanti, J. He, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.

[15] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[16] F. Zeyda, T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. A modular theory of object orientation in higher-order UTP. In *Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 627–642. Springer, 2014.

[17] H. Zhu, J. W. Sanders, He Jifeng, and S. Qin. Denotational Semantics for a Probabilistic Timed Shared-Variable Language. In B. Wolff, M.-C. Gaudel, and A. Feliachi, editors, *Unifying Theories of Programming*, volume 7681 of *Lecture Notes in Computer Science*, pages 224–247. Springer, 2013.

Index of Semantic Rules

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

Action, 54
allClockVariables (**timed**), 65
allConstants, 31
allDeadlineTransitions (**timed**), 80
allEvents, 30
allLocalConstants, 31
allLocalVariables, 31
allTransitions, 50
allVariables, 30
alphaClockReset, 65
  And (**timed**), 67
  CallExp (**timed**), 66
  ClockExp (**timed**), 69
  Equals (**timed**), 69
  GreaterOrEqual (**timed**), 68
  GreaterThan (**timed**), 68
  Iff (**timed**), 67
  Implies (**timed**), 67
  LessOrEqual (**timed**), 68
  LessThan (**timed**), 68
  Not (**timed**), 66
  Or (**timed**), 67
  ParExp (**timed**), 66
  StateClockExp (**timed**), 69
alphaClockResetCallArgs (**timed**), 66

BBuffer (**collection**), 92
buffer, 34
buildScope, 48
  timed, 64

C (Controller), 35
Cln (Collection,**collection**), 91
clockResets (**timed**), 65
compileTarget, 45
compileWC (**timed**), 72–77
composeControllers, 33
composeMachines, 37
composeStates, 40, 84
constInit, 32
constInitSTM, 48
  timed, 64
ctrlMemory, 36

deadlineEvents (**timed**), 78

exitSubstates, 46
Expr (Expression), 54
  And Expression, 55
  Array Expression, 55
  Boolean Expression, 55
  Call Expression, 56
  Concatenation Expression, 56
  Division Expression, 57

Equals Expression, 57
Greater or Equal Expression, 57
Greater Than Expression, 57
If and Only If Expression, 58
Implies Expression, 58
Integer Expression, 58
Less or Equal Expression, 58
Less Than Expression, 59
Minus Expression, 59
Modulus Expression, 59
Multiplication Expression, 59
Negation Expression, 60
Not Equal Expression, 56
Not Expression, 60
Or Expression, 60
Parenthesised Expression, 60
Plus Expression, 61
Range Expression, 61
Sequence Expression, 61
Set Expression, 61
Tuple Expression, 62

flowEvents, 41
flowTriggerEvents, 44

getsetChannels, 39
getsetLocalChannels, 40

hiddenModuleChannels, 29

initialisation, 39

M (Module), 29
memoryChannels, 30
memoryDeadline (**timed**), 80
memoryTransition, 49
timed, 80
modMemory, 32

renameTriggerEvents, 39
renamingController, 33
renamingMachine, 37
renCtrlEvts, 34
renStmEvts, 38
requiredConstants, 31
requiredVariables, 30
restrictedState, 44

S (State), 41
timed, 81
Composite State, 43
timed, 82
Final State, 43
Simple State, 42
timed, 81
singleBuffer, 35
Statement, 50, 50, 52
timed, 84, 84, 85
Call Statement, 86
ClockReset, 85
Wait, 85
Call Statement, 52
If Statement, 53
Send Event, 53
collection, 93
Sequential Composition, 54
Skip, 54
StatementInContext, 51
states, 39
STM (State Machine), 38
timed, 63
stmClocks (**timed**), 65
stmMemory, 47
timed, 79
substatesTriggers, 44

T (Transition), 45
transitionsFrom, 49
trigEvents, 40
Trigger, 48, 69
    collection, 92
triggerDeadlines (**timed**), 83
triggerEvent, 49, 77
triggerForMemory, 49

usedVariables, 51, 52

wc (**timed**), 70
wcArgSeq (**timed**), 71

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 , 42[1], 42[2], 42[3], 43[4], 43[5], 43[6], 45[10], 45[7], 45[8], 45[9], 81[11], 81[12], 81[13], 82[14], 82[15], 82[16]

allClockVariables (timed) , 63[1], 79[2]

allConstants , 32[1], 39[2], 39[3], 40[4], 47[5], 47[6], 63[7], 79[8], 79[9]

allDeadlineTransitions (timed) , 79[1]

allEvents , 39[1]

allLocalConstants , 30[1], 35[2], 40[3]

allLocalVariables , 30[1], 32[2], 35[3], 36[4], 40[5], 47[6], 79[7]

allTransitions , 40[1], 44[2], 44[3], 44[4], 47[5], 50[6], 63[7], 78[8], 79[9], 80[10]

allVariables , 39[1], 39[2], 40[3]

alphaClockReset (timed) , 65[1], 65[2], 66[3], 66[4], 66[5], 67[10], 67[11], 67[12], 67[13], 67[6], 67[7], 67[8], 67[9], 68[14], 68[15], 68[16], 68[17], 68[18], 68[19], 68[20], 68[21], 69[22], 69[23]

alphaClockResetCallArgs (timed) , 66[1], 66[2]

BBuffer , 91[1], 91[2]

buffer , 29[1]

buildScope , 48[1], 48[2], 48[3]

buildScope (timed) , 64[1]

C (Controller) , 33[1]

clockResets (timed) , 63[1], 63[2]

compileTarget , 45[1], 45[2], 45[3]

compileWC (timed) , 65[1]

composeControllers , 29[1], 33[2]

composeMachines , 35[1], 37[2]

composeStates , 38[1], 40[2], 43[3], 63[4]

composeStates (timed) , 82[1], 84[2]

constInit , 32[1], 36[2]

constInitSTM , 47[1]

constInitSTM (timed) , 63[1], 64[2], 64[3]

ctrlMemory , 35[1]

deadlineEvents (timed) , 63[1], 63[2]

exitSubstates , 43[1], 45[2], 81[3], 82[4]

Expr (Expression) , 32[1], 48[2], 49[3], 52[4], 52[5], 53[6], 53[7], 53[8], 55[10], 55[11], 55[12], 55[9], 56[13], 56[14], 56[15], 56[16], 56[17], 56[18], 56[19], 57[20], 57[21], 57[22], 57[23], 57[24], 57[25], 57[26], 57[27], 58[28], 58[29], 58[30], 58[31], 58[32], 58[33], 59[34], 59[35], 59[36], 59[37], 59[38], 59[39], 59[40], 59[41], 60[42], 60[43], 60[44], 60[45], 60[46], 61[47], 61[48], 61[49], 61[50], 61[51], 61[52], 62[53], 64[54], 72[55], 72[56], 72[57], 72[58], 73[59], 73[60], 73[61], 73[62], 74[63], 74[64], 74[65], 74[66], 75[67], 75[68], 75[69], 75[70], 76[71], 76[72], 76[73], 76[74], 77[75], 77[76],

77[77], 77[78], 80[79], 80[80], 80[81], 83[82], 84[83], 85[84], 86[85], 93[86], 93[87], 93[88]

flowEvents , 40[1], 40[2], 84[3], 84[4]
flowTriggerEvents , 43[1], 82[2]

getsetChannels , 38[1], 63[2]
getsetLocalChannels , 38[1], 63[2]

hiddenModuleChannels , 29[1]

initialisation , 38[1], 42[2], 43[3], 63[4], 81[5], 82[6]

M (Module) , 91[1]
memoryChannels , 29[1], 29[2]
memoryDeadline (timed) , 79[1]
memoryTransition , 47[1]
memoryTransition (timed) , 79[1]
modMemory , 29[1]

readState , 50[1], 51[2], 83[3]
renameTriggerEvents , 38[1], 63[2]
renamingController , 33[1], 33[2]
renamingMachine , 37[1], 37[2]
renCtrlEvts , 33[1], 33[2]
renStmEvts , 37[1], 37[2]
requiredConstants , 33[1], 34[2], 35[3], 37[4], 38[5]
requiredVariables , 30[1], 32[2], 35[3], 36[4], 36[5], 47[6], 79[7]
restrictedState , 40[1], 40[2], 84[3], 84[4]

S (State) , 44[1]
Statement , 51[1], 84[2]
StatementInContext , 53[1], 53[2], 54[3], 54[4]
states , 38[1], 38[2], 41[3], 42[4], 43[5], 43[6], 43[7], 43[8], 44[10], 44[9], 46[11], 63[12], 63[13], 81[14], 82[15], 82[16]
STM (State machine) , 37[1], 52[2]
STM (State machine,timed) , 86[1]
stmClocks (timed) , 63[1]
stmMemory , 38[1]
stmMemory (timed) , 63[1]
substatesTriggers , 44[1]

T (Transition) , 39[1], 42[2], 43[3], 45[4], 81[5], 82[6]
transitionsFrom , 42[1], 43[2], 44[3], 45[4], 81[5], 82[6], 83[7]
trigEvents , 38[1], 63[2], 63[3]
Trigger , 45[1], 69[2]
triggerDeadlines (timed) , 81[1], 82[2]
triggerEvent , 40[1], 44[2], 77[3]
triggerEvent (timed) , 63[1], 65[2], 72[10], 72[3], 72[4], 72[5], 72[6], 72[7], 72[8], 72[9], 73[11], 73[12], 73[13], 73[14], 73[15], 73[16], 73[17], 73[18], 74[19], 74[20], 74[21], 74[22], 74[23], 74[24], 74[25], 74[26], 75[27], 75[28], 75[29], 75[30], 75[31], 75[32], 75[33], 75[34], 76[35], 76[36], 76[37], 76[38], 76[39], 76[40], 77[41], 77[42], 77[43], 77[44], 77[45], 77[46]
triggerForMemory , 49[1], 49[2], 80[3], 80[4]

usedVariables , 51[1], 51[2], 51[3], 51[4], 51[5], 51[6], 51[7], 51[8], 83[9]

wc (timed) , 71[1]
wcArgSeq (timed) , 70[1], 70[2], 71[3]

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
- RoboticPlatformDef, 12
- RoboticPlatformRef, 12
- Well Formedness, 22