

RoboChart Reference Manual

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

June 15, 2018

Abstract

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

Contents

1	Introduction	10
2	Syntax	12
2.1	RoboChart metamodel	12
3	Well-formedness Conditions	22
3.1	Core Language	22
3.1.1	Robotic Platforms	22
3.1.2	Interfaces	23
3.1.3	Modules	23
3.1.4	Connection	23
3.1.5	Controllers	23
3.1.6	State Machines	24
3.1.7	States	24
3.1.8	Initial junctions	25
3.1.9	Junction	25
3.1.10	Final states	25
3.1.11	Transitions	25
3.1.12	Operations	25
3.2	Timed Language	25
3.2.1	Timed Expressions	25
3.2.2	Timed Statements	26
4	Semantics	27
4.1	Detailed Semantics: Core Language	28

4.1.1	Modules	28
4.1.2	Controllers	33
4.1.3	State machines	36
4.1.4	Statements	45
4.1.5	Expressions	50
4.2	Detailed Semantics: Timed Language	58
4.2.1	State machines	58
	Clocks	59
	Waiting Conditions	64
	Trigger Deadlines	77
	Memory	78
4.2.2	States	80
4.2.3	Timed statements	84
5	Conclusions	86
A	Complete Metamodel: Core Language	87
A.1	Robotic Platforms	87
A.2	Interfaces	88
A.3	Variables	88
A.4	Constants	89
A.5	Events	89
A.6	Required, Provided and Defined Interfaces	89
A.7	State-Machines	90
A.8	States	91
A.9	Final states	91
A.10	Junction node	92
A.11	Initial nodes	92
A.12	Transitions	92
A.13	Controllers	93
A.14	Connection	94
A.15	Modules	95
A.16	Statements	95
A.17	Expressions	96
A.18	Type Declaration	101
A.19	Primitive Types	101
A.20	Datatypes	101

A.21 Enumeration	102
A.22 Type Constructors	102
B Complete Metamodel: Timed Language	104
B.1 Clock	104
B.2 Timed Statements	104
B.3 Timed Expressions	105
B.4 Timed Triggers	105
C Credits	106
Index of Semantic Rules	110
Index of Calls to Semantic Rules	112

List of Figures

2.1 Metamodel of RoboChart packages	13
2.2 Metamodel of RoboChart modules	14
2.3 Metamodel of RoboChart controllers	14
2.4 Metamodel of RoboChart context for elements and operations	15
2.5 Metamodel of RoboChart state machines	16
2.6 Concrete syntax of transitions	16
2.7 Metamodel of RoboChart triggers	17
2.8 Concrete syntax of triggers	17
2.9 Metamodel of RoboChart types	18
2.10 Metamodel of RoboChart time primitives	21

List of rules

Untimed semantics	28
1 Semantics of modules	28
2 Memory channels	28
3 Function allVariables	29
4 Function requiredVariables	29
5 Function allLocalVariables	29
6 Module Memory	30
7 Composition of controllers	31
8 Renaming controller	31
9 Renaming controller events	32
10 Buffer	32
11 Semantics of controllers	33
12 Controller Memory	34
13 Composition of machines	35
14 Renaming state machine	35
15 Renaming machine events	36
16 Semantics of state machine	36
17 Function substates	36
18 Initialisation	37
19 Get and Set channels	37
20 Composition of states	37
21 Trigger events	38
22 Get and set local channels	38
23 Flow events	38
24 Semantics of states	38
25 Semantics of simple states	39
26 Semantics of composite states	40
27 Semantics of final states	40
28 Synchronisation events between parent state and substates	41
29 Triggers of substates	41
30 Restricted semantics of states	41
31 Semantics of transitions	42

32	Compile target	42
33	Exit substates	43
34	State-machine Memory	43
35	Semantics of triggers	44
36	Event for transition trigger	44
37	Memory transitions	44
38	Function transitionsFrom	44
39	Function allTransitions	45
40	Function allSTMTransitions	45
41	Semantics of statements	45
42	Read state of an expression	46
43	Semantics of statements in context	46
44	Function usedVariables	47
45	Semantics of assignment	47
46	Semantics of call statement	48
47	Semantics of if statements	48
48	Semantics of send event statements	49
49	Semantics of sequential composition	49
50	Semantics of skip	49
51	Semantics of actions	50
52	Semantics of expressions	50
53	Semantics of and expression	50
54	Semantics of array expression	51
55	Semantics of boolean expression	51
56	Semantics of call expression	52
57	Semantics of concatenation expression	52
58	Semantics of not equal expression	52
59	Semantics of division	53
60	Semantics of equality	53
61	Semantics of greater or equal expression	53
62	Semantics of greater than	53
63	Semantics of if and only if expression	54
64	Semantics of implication	54
65	Semantics of integer expression	54
66	Semantics of less or equal expression	54
67	Semantics of less than	55

68	Semantics of minus	55
69	Semantics of modulus	55
70	Semantics of multiplication	55
71	Semantics of arithmetic negation	56
72	Semantics of logical negation	56
73	Semantics of or expression	56
74	Semantics of parenthesised expression	56
75	Semantics of plus	57
76	Semantics of range expression	57
77	Semantics of sequence expression	57
78	Semantics of set expression	57
79	Semantics of tuple expression	58
	Timed Semantics	58
80	Semantics of state machine	59
81	allClockVariables function	59
82	clockResets function	60
83	stmClocks function	60
84	alphaClockReset function	60
85	alphaClockReset function	60
86	alphaClockReset function	61
87	alphaClockReset function	61
88	alphaClockResetCallArgs function	61
89	alphaClockReset function	61
90	alphaClockReset function	62
91	alphaClockReset function	62
92	alphaClockReset function	62
93	alphaClockReset function	62
94	alphaClockReset function	63
95	alphaClockReset function	63
96	alphaClockReset function	63
97	alphaClockReset function	63
98	alphaClockReset function	64
99	alphaClockReset function	64
100	wc function	64
101	wc function (ParExp)	65

102	wc function (Not)	65
103	wc function (CallExp)	65
104	wcArgSeq function	66
105	wc function (And)	66
106	wc function (Or)	67
107	wc function (Implies)	67
108	wc function (Iff)	67
109	wc function (GreaterThan)	68
110	wc function (GreaterOrEqual)	68
111	wc function (LessThan)	69
112	wc function (LessOrEqual)	69
113	wc function (Equals)	70
114	compileWC function	70
115	getClockReset function	70
116	getClockReset function	71
117	getClockReset function	71
118	compileWC function	72
119	compileWC function	73
120	compileWC function	74
121	compileWC function	75
122	compileWC function	76
123	deadlineEvents function	77
124	triggerEvent function	77
125	State-machine Memory	78
126	allTriggerDeadlineTransitions function	79
127	memoryTransition function	79
128	Memory deadline	79
129	Timed semantics of states	80
130	Timed semantics of simple states	81
131	Timed semantics of composite states	82
132	Semantics of trigger deadlines	83
133	Timed composition of states	83
134	Semantics of timed statements	84
135	Function usedVariables for timed semantics	84
136	Semantics of Deadlines	84
137	Semantics of Wait	84

138 Semantics of Clock Reset 85

Introduction

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

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

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

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

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

In this report, we formalise the semantics of the core and timed constructs of RoboChart using CSP [9]. Importantly, CSP is a front end for a mathematical model that supports a number

analysis techniques such as model-checking, which provide a high degree of automation, as well as more powerful (but not automatic) verification based on interactive and theorem proving, namely, Hoare and He's Unifying Theories of Programming [7] (UTP). Use of CSP enables model checking with FDR [5]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [11] and probability [14].

Chapter 2 describes RoboChart models, and Chapter 3 defines their well-formedness conditions. Chapter 4 presents their semantics of RoboChart in CSP. Chapter 5 describes the API available for modelling robotic systems. Chapter 6 presents a number of models specified in RoboChart. Finally, Chapter 8 concludes with a summary of the results and future work.

Syntax

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

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

2.1 RoboChart metamodel

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

The metamodel is automatically generated from a syntax definition. It includes a notion of a MachineContainer, which, as the name 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

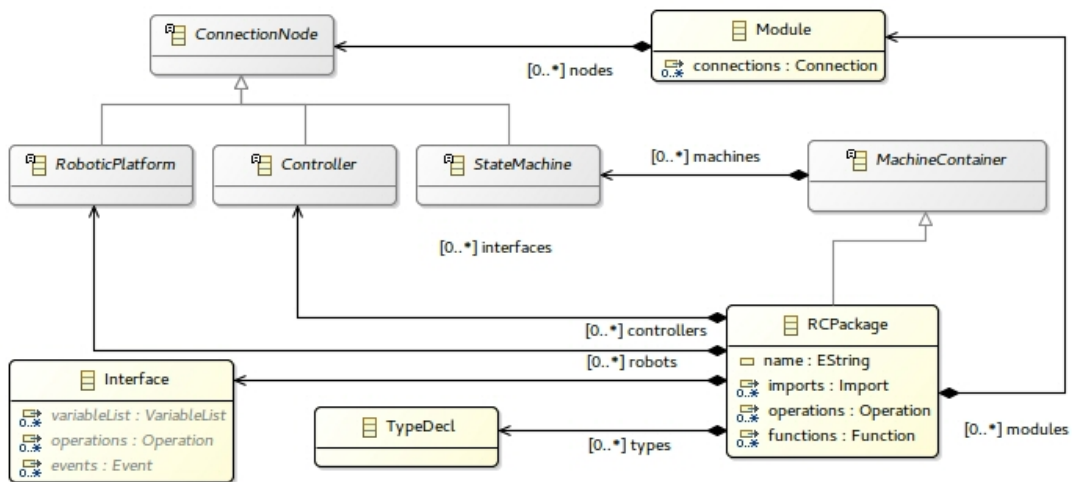


Figure 2.1: Metamodel of RoboChart packages

source (efrom) and a target (eto) events. They can be asynchronous and bidirectional, as indicated by the boolean attributes `async` and `bidirec`. An event may or not have a `Type`, which defines the values that can be communicated via the connection, if any.

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

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

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

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

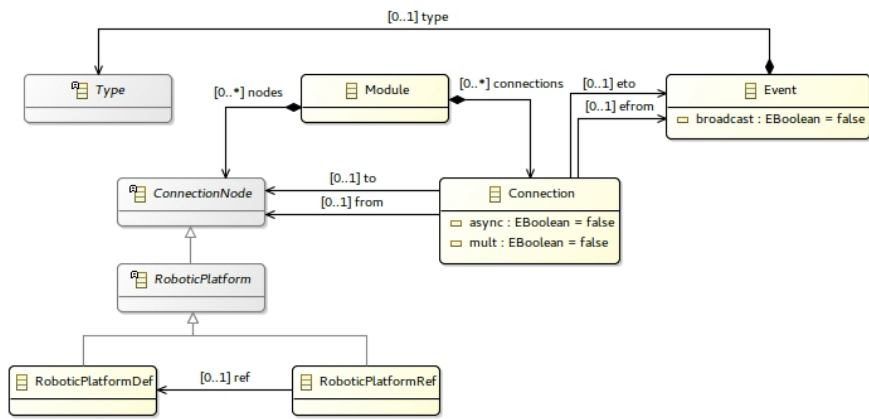


Figure 2.2: Metamodel of RoboChart modules

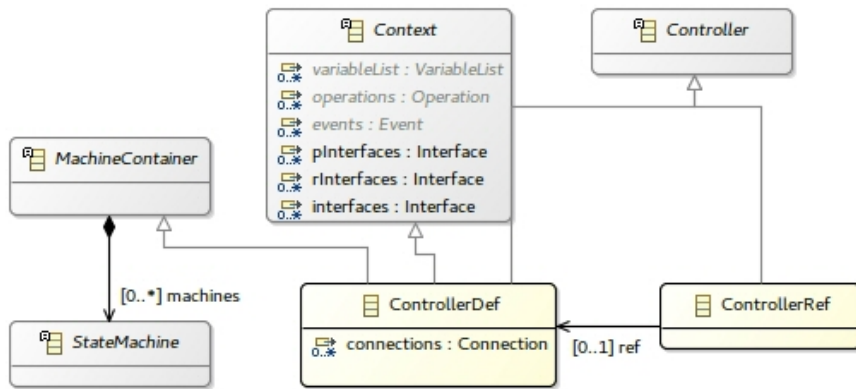


Figure 2.3: Metamodel of RoboChart controllers

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

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

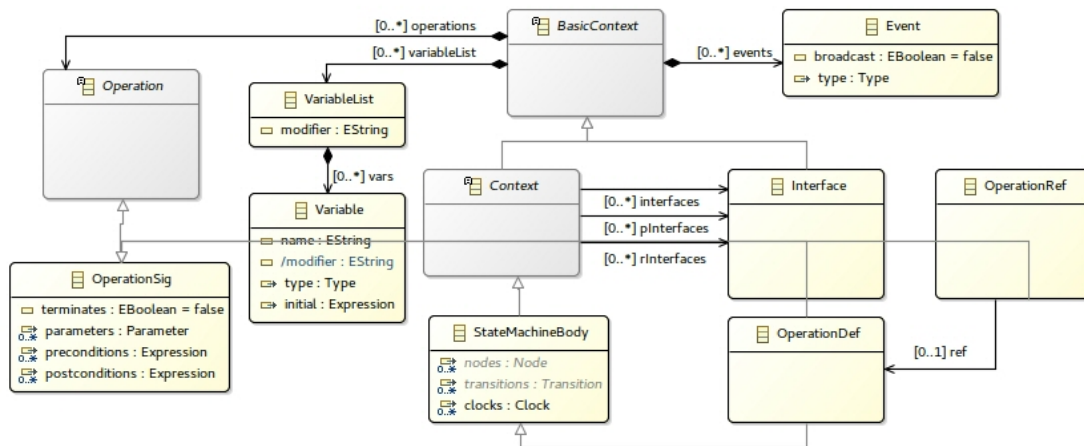


Figure 2.4: Metamodel of RoboChart context for elements and operations

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

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

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

Transitions are directed connections between two nodes: a source and a target. They may be triggered by an event, guarded by a condition, and contain an action that is executed when the transition is taken. We can also specify start and end deadlines for a transition.

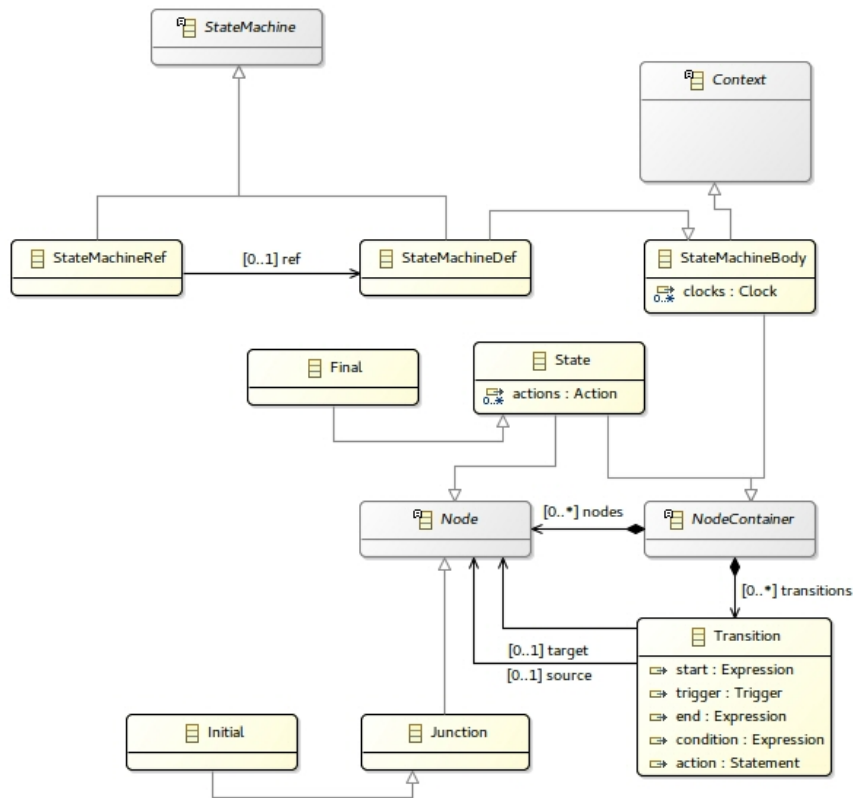


Figure 2.5: Metamodel of RoboChart state machines

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

Figure 2.6: Concrete syntax of transitions

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

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

The concrete syntax of triggers is shown in Figure 2.8. It consists of an optional input, output, sync or simple trigger, followed by an optional variable that record the time instant the trigger occurs, and a (potentiall empty) list of clock resets. The concrete syntax of the different types of triggers is shown in Table 2.1.

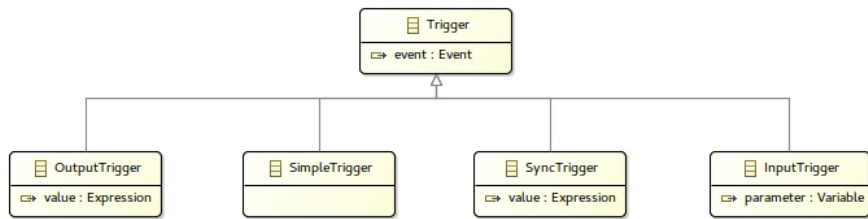


Figure 2.7: Metamodel of RoboChart triggers

`(Input|Output|Sync|Simple)? (@Variable)? 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.

Element	Concrete Syntax	Comment
Integer	$(0..9)^+$	
Float	$(0..9)^+.(0..9)^+$	
String	"..."	Quoted values
Boolean	true false	
Reference	N	N is the name of a variable or constant.
Sequence	$\langle e_1, e_2, \dots \rangle$	Sequence with values e_i .
Set	$\{e_1, e_2, \dots\}$	Set with values e_i .
Set Comprehension	$\{x:T \mid P @ e\}$	Set containing values e , calculated from elements of type T, for which the predicate P holds.
Interval	$[e_1, e_2]$ or (e_3, e_4)	Closed interval between e_1 and e_2 , and open interval between e_3 and e_4 , or a combination of both.
Tuple	(e_1, e_2, \dots)	Tuple containing elements e_i .
Array	$e[i]$	The i -th element of array e .
Selection	$e.n$	The n field of record e .
Negation	$-e$	Arithmetical negation of expression e .
Concatenation	$e_1^e_2$	Concatenate sequences e_1 and e_2 .
Modulo	$e_1 \% e_2$	Remainder of dividing e_1 by e_2 .
Division	e_1 / e_2	Division of e_1 by e_2 .
Multiplication	$e_1 * e_2$	Multiplication of e_1 by e_2 .
Sum	$e_1 + e_2$	Sum of e_1 and e_2 .
Subtraction	$e_1 - e_2$	Subtraction of e_1 and e_2 .
Conditional	if c then e else f end	If condition c is true, e_1 else e_2 .
Local definition	let n == e @ f	Define locally n and use it to calculate f .
Definite description	the x: T P @ e	The value e calculated based on the unique x for which P holds.
Lambda expression	lambda x: T P @ e	The anonymous function that takes values of type T for which P holds, to values e calculated based on x .
Equality	$e_1 == e_2$	True if both expressions are equal.
Different	$e_1 != e_2$	True if both expressions are different.
Greater than	$e_1 > e_2$	True if e_1 is greater than e_2 .
Greater than or equal to	$e_1 >= e_2$	True if e_1 is greater than or equal to e_2 .
Less than	$e_1 < e_2$	True if e_1 is less than e_2 .
Less than or equal to	$e_1 <= e_2$	True if e_1 is less than or equal to e_2 .

Table 2.3: Concrete syntax of expressions (1)

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 e 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 @ e	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 channel ev.
Synchronisation	ev	Synchronise on event ev.
Sequential composition	S1; S2	Execute S1, and then S2.

Table 2.5: Concrete syntax of statements

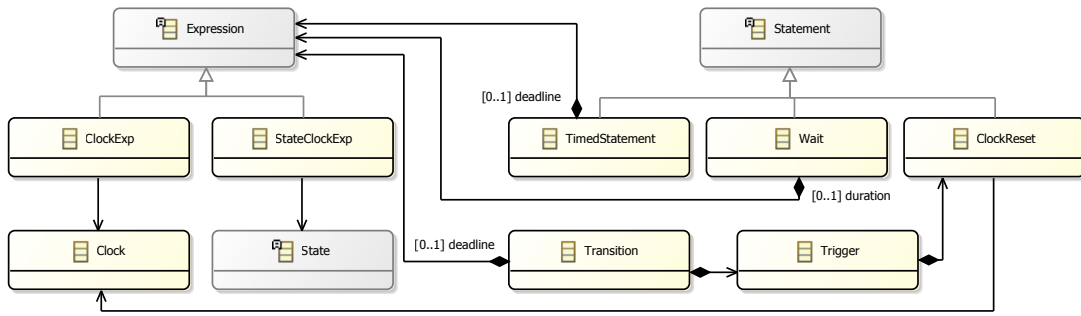


Figure 2.10: Metamodel of RoboChart time primitives

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

Table 2.6: Concrete syntax of time primitives

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

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

Well-formedness Conditions

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

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

We present the conditions related to each of the elements of the core language in Section 3.1. We also provide here justifications for the restrictions. We follow that with conditions on the timed language in Section 3.2. Finally, in Section ??, we provide a formalisation of all conditions using the Z notation of axiomatic descriptions [12].

3.1 Core Language

3.1.1 Robotic Platforms

- Robotic platforms cannot require interfaces.
- Defined interfaces can only have events
- The names of variables, operations, and events are unique to the platform.

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

3.1.2 Interfaces

- Provided and required interfaces contain only variables and operations
- Defined interfaces contain only variables and events
- Names of variables, events and operations are unique.

3.1.3 Modules

- A module must contain exactly one robotic platform and at least one controller.
- All variables and operations required by the module's controllers must be provided by the platform.

3.1.4 Connection

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

- Connections of a module must associate only events of the robotic platform and its controllers.
- Connections involving a robotic platform are always asynchronous.
- Connections of a controller must associate only its events and those of its state machines.
- Only events of the same type may be connected.

3.1.5 Controllers

- A controller must contain at least one state machine.
- Controllers cannot provide variables or operations to other controllers.
- Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.

- All variables required by the controller's state-machines must be provided or required by the controller.
- All operations required by the controller's state-machines must be required or defined by the controller.
- Operations cannot not be required by a controller, but those required by its state machines must be fully defined within the controller.
- All variables required by the controller's state-machines must be provided or required by the controller.
- All operations required by the controller's state-machines must be required or defined by the controller.
- The names of variables, operations, and events are unique to the controller.

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

3.1.6 State Machines

- State machines cannot have provided interfaces
- Operations in state machines can only be required, not defined.
- Every state machine must have exactly one initial junction.
- State machines must contain at least one state.
- The names of variables, operations, and events are unique to the machine.

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

3.1.7 States

- If a state has a non-empty set of nodes, then conditions 3 and 4 of *state machines* apply.
- A state has at most one of each type of action: entry, during, and exit,

3.1.8 Initial junctions

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

3.1.9 Junction

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

3.1.10 Final states

- Final states cannot be the source of transitions.

3.1.11 Transitions

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

3.1.12 Operations

- All state-machine conditions apply to operation definitions.

3.2 Timed Language

3.2.1 Timed Expressions

- Expressions involving `since(C)` and `sinceEntry(S)` are only permitted in transition guards.
- The clock `C` in an expression `since(C)` may only reference a clock declared within the expression's containing state-machine.

- The state S in an expression $\text{sinceEntry}(S)$ may only reference a state within the containing expression's state-machine. When the name S is ambiguous, because, for instance, there is a state and a substate with the same name in the state machine, the fully qualified name of the state S must be used.
- The expressions $\text{since}(C)$ or $\text{sinceEntry}(S)$ may only be compared with a constant expression, and only when using one of the following operators: $>$, $<$, $>=$, $<=$, $==$. A consequence of this restriction is that no expression can compare the value of two clocks as given by $\text{since}(C)$ or $\text{sinceEntry}(S)$.

3.2.2 Timed Statements

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

Semantics

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

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

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

Rule 3. Function allVariables

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

$$\begin{aligned} & \underline{\bigcup\{l : c.\text{variableList} \bullet l.\text{vars}\}} \cup \\ & \underline{\bigcup\{i : c.\text{PInterfaces} \bullet \bigcup\{l : i.\text{variableList} \bullet l.\text{vars}\}\}} \cup \\ & \underline{\bigcup\{i : c.\text{RInterfaces} \bullet \bigcup\{l : i.\text{variableList} \bullet l.\text{vars}\}\}} \end{aligned}$$

Rule 4. Function requiredVariables

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

$$\underline{\bigcup\{i : c.\text{RInterfaces} \bullet \bigcup\{l : i.\text{variableList} \bullet l.\text{vars}\}\}}$$

Rule 5. Function allLocalVariables

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

$$\underline{\bigcup\{l : c.\text{variableList} \bullet l.\text{vars}\}} \cup \underline{\bigcup\{i : c.\text{PInterfaces} \bullet \bigcup\{l : i.\text{variableList} \bullet l.\text{vars}\}\}}$$

Rule 6. Module Memory

$\text{modMemory}(c : \text{Module}) : \text{CSPPProcess} =$

let $\text{Memory}(\underline{\text{vars}}) \hat{=} \square v : \underline{\text{lvars}} \bullet \underline{\text{set_vid}}(v, \text{rp})?x \rightarrow$
 $(\S c : \underline{\text{rcontrollers}}(v) \bullet \underline{\text{set_Ext_vid}}(v, c)!x \rightarrow \text{Skip}); \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x])$
within $\text{Memory}(\underline{\text{varvalues}})$
where
 $\text{rp} = \underline{\text{roboticPlatformDefinition}}(m)$
 $\text{ctrls} = \underline{\text{m.controllers}}$
 $\underline{\text{lvars}} = \underline{\text{allLocalVariables}^2}(c)$
 $\underline{\text{vars}} = \langle v : \underline{\text{lvars}} \bullet \underline{\text{name}}(v) \rangle$
 $\underline{\text{varvalues}} = \langle v : \underline{\text{lvars}} \bullet \underline{\text{initial}}(v) \rangle$
 $\underline{\text{rcontrollers}} = \lambda v \bullet \{c : \underline{\text{ctrls}} \mid v \in \underline{\text{requiredVariables}^2}(c)\}$

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 7. Composition of controllers

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

$\text{if } \#ctrls = 1$
 then
 $\quad \text{renamingController}^1(\text{head } ctrl\text{s}, \text{cons})$
 else
 $\quad \text{renamingController}^2(\text{head } ctrl\text{s}, \text{cons})$
 $\quad \quad \llbracket \text{connevt\text{s}} \rrbracket$
 $\quad \text{composeControllers}^2(m, rp, \text{tail } ctrl\text{s}, \text{cons})$

where

$\text{connevt\text{s}} = \text{renCtrlEvt\text{s}}^1(\text{head } ctrl\text{s}, \text{cons}) \cap$
 $\quad \cup \{c : \text{tail } ctrl\text{s} \bullet \text{renCtrlEvt\text{s}}^2(c, \text{cons})\}$

Rule 8. Renaming controller

$\text{renamingController}(c : \text{Controller}, \text{cons} : \text{Set}(\text{Connection})) : \text{CSPPProcess} =$

$\llbracket c \rrbracket \llbracket c' \rrbracket \left[\left[\begin{array}{l} \{x : \text{internalConns} \cup \text{fromPlatform} \bullet \text{eventId}(e.\text{eto}) \leftarrow \text{eventId}(e.\text{efrom})\} \\ \cup \{x : \text{toPlatform} \bullet \text{eventId}(e.\text{efrom}) \leftarrow \text{eventId}(e.\text{eto})\} \end{array} \right] \right]$

where

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

Rule 9. Renaming controller events

$\text{renCtrlEvs}(c : \text{Controller}, \text{cons} : \text{Set}(\text{Connection})) : \text{ChannelSet} =$

$\{\underline{x : \text{internalConns} \bullet \text{eventId}(e.\text{efrom})}\} \cup \{\text{end_}\}$

where

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

Rule 10. Buffer

$\text{buffer}(c : \text{Connection}) : \text{CSPPProcess} =$

if $c.\text{efrom.type} \neq \text{null}$

then

let $\text{Buffer}(\langle \rangle) \hat{=} \underline{\text{eventId}(c.\text{efrom})?x} \rightarrow \text{Buffer}(\langle x \rangle)$

$\text{Buffer}(\langle v \rangle) \hat{=} \underline{\text{eventId}(c.\text{efrom})?x} \rightarrow \text{Buffer}(\langle x \rangle) \sqcap \underline{\text{eventId}(c.\text{eto})!v} \rightarrow \text{Buffer}(\langle \rangle)$

within $\text{Buffer}(\langle \rangle)$

else

let $\text{Buffer}(\text{false}) \hat{=} \underline{\text{eventId}(c.\text{efrom})} \rightarrow \text{Buffer}(\text{true})$

$\text{Buffer}(\text{true}) \hat{=} \underline{\text{eventId}(c.\text{efrom})} \rightarrow \text{Buffer}(\text{true}) \sqcap \underline{\text{eventId}(c.\text{eto})} \rightarrow \text{Buffer}(\text{false})$

within $\text{Buffer}(\text{false})$

4.1.2 Controllers

Rule 11. Semantics of controllers

$\llbracket c : \text{ControllerDef} \rrbracket_{\mathcal{C}} : \text{CSPPProcess} =$

$$\left(\left(\frac{\text{composeMachines}^1(c, ms, cs)}{\llbracket \text{lvars} \cup \text{rvars} \rrbracket} \right) \setminus \left(\frac{\text{hvars}}{\cup} \right) \right) \Theta_{\{\text{end_}\}} \text{Skip}$$

where

$ms = c.machines$

$cs = c.connections$

$\text{lvars} = \{ \underline{v} : \text{allLocalVariables}^3(c) \bullet \text{set_vid}(v, c) \}$

$\text{rvars} = \{ \underline{v} : \text{requiredVariables}^3(c) \bullet \text{set_Ext_vid}(v, c) \}$

$\text{hvars} = \{ \underline{v} : \text{allLocalVariables}^4(c) \bullet \text{set_vid}(v, c) \} \cup$

$\{ \underline{v} : \text{allLocalVariables}^5(c) \bullet \text{get_vid}(v, c) \}$

Rule 12. Controller Memory

$\text{ctrlMemory}(c : \text{ControllerDef}) : \text{CSPPProcess} =$

$$\text{let } \text{Memory}(\underline{\text{vars}}) \hat{=} \left(\begin{array}{l} \square v : \underline{\text{lvars}} \bullet \text{set_vid}(v, c)?x \rightarrow \\ \quad (\S m : \underline{\text{rmachines}}(v) \bullet \text{set_Ext_vid}(v, m)!x \rightarrow \text{Skip}); \\ \quad \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x]) \\ \square \\ \square v : \underline{\text{rvars}} \bullet \text{set_Ext_vid}(v, c)?x \rightarrow \\ \quad (\S m : \underline{\text{rmachines}}(v) \bullet \text{set_Ext_vid}(v, m)!x \rightarrow \text{Skip}); \\ \quad \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x]) \end{array} \right)$$

within

$\text{Memory}(\underline{\text{varvalues}})$

where

$\underline{\text{ms}} = c.\text{machines}$

$\underline{\text{lvars}} = \text{allLocalVariables}^6(c)$

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

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

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

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

Rule 13. Composition of machines

$\text{composeMachines}(c : \text{Controller}, ms : \text{Seq}(\text{StateMachine}), cons : \text{Set}(\text{Connection})) :$

$\text{CSPPProcess} =$

if #ms = 1

then

renamingMachine¹(head ms, cons)

else

renamingMachine²(head ms, cons)

[[connevts]]

composeMachines²(c, tail ms, cons)

where

connevts = renamingStmEvs¹(head ms, cons, c) \cap $\bigcup\{m : \text{tail ms} \bullet \text{renamingStmEvs}^2(m, cons, c)\}$

Rule 14. Renaming state machine

$\text{renamingMachine}(m : \text{StateMachine}, cons : \text{Set}(\text{Connection})) : \text{CSPPProcess} =$

[[m]]
STM' $\left[\left[\begin{array}{l} \{x : \text{internalConns} \cup \text{fromController} \bullet \text{eventId}(e.eto) \leftarrow \text{eventId}(e.efrom)\} \\ \cup \{x : \text{toController} \bullet \text{eventId}(e.efrom) \leftarrow \text{eventId}(e.eto)\} \end{array} \right] \right]$

where

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

toController = $\{x : cons \bullet x.from = m \wedge x.to \in \text{Controller}\}$

fromController = $\{x : cons \bullet x.to = m \wedge x.from \in \text{Controller}\}$

Rule 15. Renaming machine events
$$\text{renStmEvts}(m : \text{StateMachine}, \text{cons} : \text{Set}(\text{Connection})) : \text{ChannelSet} =$$

$$\{x : \text{internalConns} \bullet \text{eventId}(e.\text{efrom})\} \cup \{\text{end}_ \}$$

where

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

4.1.3 State machines

Rule 16. Semantics of state machine
$$\llbracket \text{stm} : \text{StateMachineDef} \rrbracket_{STM} : \text{CSPPProcess} =$$

$$\left(\begin{array}{l} \left(\begin{array}{l} \text{initialisation}^1(\text{stm}) \\ \llbracket \text{flowevts} \rrbracket \\ \text{composeStates}^1(\langle x : \text{stm.nodes} \mid x \in \text{State} \rangle, \text{stm}) \end{array} \right) \\ \setminus \{ \text{enter}, \text{entered}, \text{exit}, \text{exited} \} \\ \llbracket \text{getsetChannels}^1(\text{stm}) \cup \text{trigEvents}^1(\text{stm}) \rrbracket \\ \text{stmMemory}^1(\text{stm}) \\ \setminus \text{getsetLocalChannels}^1(\text{stm}) \cup \{ \text{internal}_ \} \\ \ominus_{\{ \text{end}_ \}} \text{Skip} \end{array} \right)$$

where

$$\text{flowevts} =$$

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

Rule 17. Function substates
$$\text{substates}(n : \text{NodeContainer}) : \text{Set}(\text{State}) =$$

$$\text{x.nodes} \cap (\text{State} \cup \text{Final})$$

Rule 18. Initialisation

initialisation(n : NodeContainer) : CSPPProcess =

if (#n.nodes > 0) then
 $\llbracket \iota t : \text{Transition} \mid t.\text{source} = (\iota x : \text{n.nodes} \mid x \in \text{Initial}), n, \text{true} \rrbracket^{\text{Skip, Skip}}$

 \mathcal{T}'
else *Skip*

Rule 19. Get and Set channels

getsetChannels(s : StateMachineDef) : ChannelSet =

$\{ \underline{v} : \text{allVariables}^1(s) \bullet \text{get_vid}(v) \} \cup \{ \underline{v} : \text{allVariables}^2(s) \bullet \text{set_vid}(v) \}$

Rule 20. Composition of states

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

if #ss = 1
then
 restrictedState¹(p, head ss)
else
 $\left(\begin{array}{c} \text{restrictedState}^2(p, \text{head ss}) \\ \llbracket \text{shflowevts} \rrbracket \\ \text{composeStates}^2(\text{tail ss}, p) \end{array} \right) \setminus \text{shflowevts}$

where

shflowevts = flowEvents¹(head ss, p) \cap $\bigcup \{ x : \text{tail ss} \bullet \text{flowEvents}^2(x, p) \}$

Rule 21. Trigger events

trigEvents(s : StateMachineDef) : ChannelSet =

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

Rule 22. Get and set local channels

getsetLocalChannels(s : StateMachineDef) : ChannelSet =

$$\{v : \text{allLocalVariables}^7(s) \bullet \text{get_vid}(v)\} \cup \{v : \text{allLocalVariables}^8(s) \bullet \text{set_vid}(v)\} \\ \cup \{v : \text{requiredVariables}^6(s) \bullet \text{get_vid}(v)\}$$

Rule 23. Flow events

flowEvents(s : State, p : NodeContainer) : ChannelSet =

$$\cup \{x : \text{substates}^3(p); y : \{\text{id}(s)\} \bullet \{ \\ \text{enter.y.x}, \text{entered.y.x}, \text{exit.y.x}, \text{exited.y.x}, \\ \text{enter.x.y}, \text{entered.x.y}, \text{exit.x.y}, \text{exited.x.y}, \\ \}\}$$

Rule 24. Semantics of states

[[s : State]]_S : CSPPProcess =

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

Rule 28. Synchronisation events between parent state and substates

flowTriggerEvents(s : State) : ChannelSet =

$$\begin{aligned} & (\{e : Event; t : TIDS \bullet e.t\} \setminus \underline{\text{substatesTriggers}^1(s)}) \cup \\ & \underline{\cup\{x : SIDS \setminus \text{substates}^7(s); y \in \text{states}(s) \bullet \{enter.x.y, entered.x.y, exit.x.y, exited.x.y\}\}} \end{aligned}$$

Rule 29. Triggers of substates

substatesTriggers(s : State) : ChannelSet =

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

Rule 30. Restricted semantics of states

restrictedState(p : NodeContainer, s : State) : CSPProcess =

$$\underline{\llbracket s \rrbracket}_{s'} \llbracket \underline{\text{all_other_transitions_S} \setminus \text{all_transitions_PS}} \rrbracket \textit{Skip}$$

where

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

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

$$\underline{\text{all_transitions_PS}} = \{e : Event; tid : TIDS \bullet \text{eventId}(e).tid\}$$

$$\underline{\setminus\{t : \text{allTransitions}^4(p) \bullet \text{eventId}(t.trigger.event).id(t)\}}$$

Rule 31. Semantics of transitions
$$\llbracket t : \text{Transition}, \text{origin} : \text{NodeContainer}, \text{initial} : \text{boolean} \rrbracket_{\mathcal{T}}^{P, Q} : \text{CSPPProcess} =$$

if $\text{src} \in \text{State}$

$$\frac{\llbracket t.\text{trigger} \rrbracket^{\text{id}(t)} ; \text{exit}!\underline{\text{id}(\text{src})}!\underline{\text{id}(\text{src})} \rightarrow \text{exitSubstates}^3(\text{src}); \llbracket \text{src.exit} \rrbracket}{\text{Trigger}^1} ; \frac{\text{exited}!\underline{\text{id}(\text{src})}!\underline{\text{id}(\text{src})} \rightarrow \llbracket t.\text{action} \rrbracket}{\text{Action}^8} ; \text{compileTarget}^1(\text{tgt}, \text{src}, \text{false}, P, Q)$$

else if $\text{src} \in \text{Initial}$

$$\frac{\llbracket t.\text{trigger} \rrbracket^{\text{id}(t)} ; \llbracket t.\text{action} \rrbracket}{\text{Trigger}^2} ; \text{compileTarget}^2(\text{tgt}, \text{parent}(\text{src}), \text{true}, P, Q)$$

else if $\text{src} \in \text{Junction}$

$$\frac{\llbracket t.\text{trigger} \rrbracket^{\text{id}(t)} ; \llbracket t.\text{action} \rrbracket}{\text{Trigger}^3} ; \text{compileTarget}^3(\text{tgt}, \text{origin}, \text{initial}, P, Q)$$

where

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

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

Rule 32. Compile target
$$\text{compileTarget}(\text{tgt} : \text{Node}, o : \text{NodeContainer}, i : \text{boolean}, P : \text{CSPPProcess}, Q : \text{CSPPProcess}) : \text{CSPPProcess} =$$

if $(\text{tgt} \in \text{State})$ then

$$\frac{\text{if } (\text{tgt} = o) \text{ then } \text{enter}!\underline{\text{id}(o)}!\underline{\text{id}(\text{tgt})} \rightarrow Q}{\text{else } \text{enter}!\underline{\text{id}(o)}!\underline{\text{id}(\text{tgt})} \rightarrow \text{entered}!\underline{\text{id}(o)}!\underline{\text{id}(\text{tgt})} \rightarrow \left(\frac{\text{if } (i) \text{ then } \text{Skip}}{\text{else } P} \right)}$$

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

$$\frac{\square \{ t : \text{transitionsFrom}^4(\text{tgt}) \bullet \llbracket t, o, i \rrbracket^{P, Q} \}}{\mathcal{T}^4}$$

Rule 33. Exit substates

$\text{exitSubstates}(s : \text{NodeContainer}) : \text{CSPPProcess} =$

$$\text{exit!}\underline{\text{id}(s)}?z : \{x : \text{substates}^8(s) \bullet \text{id}(x)\} \rightarrow \text{exited!}\underline{\text{id}(s)}!z \rightarrow \text{Skip}$$

Rule 34. State-machine Memory

$\text{stmMemory}(s : \text{StateMachine}) : \text{CSPPProcess} =$

let $\text{Memory}(\underline{\text{vars}}) \hat{=}$

$$\left(\begin{array}{l} \square v : \underline{\text{lvars}} \bullet \left(\begin{array}{l} \underline{\text{get_vid}(v, s)!name(v)} \rightarrow \text{Memory}(\underline{\text{vars}}) \\ \square \\ \underline{\text{set_vid}(v, s)?x} \rightarrow \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x]) \end{array} \right) \\ \square \\ \square v : \underline{\text{rvars}} \bullet \left(\begin{array}{l} \underline{\text{get_vid}(v, s)!name(v)} \rightarrow \text{Memory}(\underline{\text{vars}}) \\ \square \\ \underline{\text{set_vid}(v, s)?x} \rightarrow \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x]) \\ \square \\ \underline{\text{set_Ext_vid}(v, s)?x} \rightarrow \text{Memory}(\underline{\text{vars}}[\text{name}(v) := x]) \end{array} \right) \\ \square \\ \square t : \underline{\text{allTransitions}}^5(s) \bullet \underline{\text{memoryTransition}}^1(t, \text{Memory}(\underline{\text{vars}})) \end{array} \right)$$

within

$\text{Memory}(\underline{\text{varvalues}})$

where

$\underline{\text{rvars}} = \underline{\text{requiredVariables}}^7(s)$

$\underline{\text{lvars}} = \underline{\text{allLocalVariables}}^9(s)$

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

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

Rule 35. Semantics of triggers
$$\llbracket t : \text{Trigger} \rrbracket_{\text{Trigger}}^{\text{tid}} : \text{CSPPProcess} =$$

$$\begin{array}{l} \text{if } t.\text{event.type} \neq \text{null} \\ \quad \underline{\text{id}(t.\text{event}).\text{tid}?x} \rightarrow \text{set_vid}(t.\text{parameter})!x \rightarrow \text{Skip} \\ \text{else} \\ \quad \underline{\text{id}(t.\text{event}).\text{tid}} \rightarrow \text{Skip} \end{array}$$

Rule 36. Event for transition trigger
$$\text{triggerEvent}(t : \text{Trigger}, \text{tid} : \text{TIDS}) : \text{CSPEvent} =$$

$$\underline{\text{id}(t.\text{event}).\text{tid}}$$

Rule 37. Memory transitions
$$\text{memoryTransition}(t : \text{Transition}, P : \text{CSPPProcess}) : \text{CSPPProcess} =$$

$$\begin{array}{l} \text{if } (t.\text{condition} \neq \text{null}) \text{ then} \\ \quad \underline{\llbracket t.\text{condition} \rrbracket_{\text{Expr}^4}} \ \& \ \underline{\llbracket t.\text{trigger} \rrbracket_{\text{Trigger}^4}^{\text{id}(t)}} ; P \\ \text{else} \\ \quad \underline{\llbracket t.\text{trigger} \rrbracket_{\text{Trigger}^5}^{\text{id}(t)}} ; P \end{array}$$

Rule 38. Function transitionsFrom
$$\text{transitionsFrom}(s : \text{State}) : \text{Set}(\text{Transition}) =$$

$$\underline{\{t : \text{parent}(s).\text{transitions} \mid t.\text{source} = s \bullet t\}}$$

Rule 39. Function allTransitions

$\text{allTransitions}(s : \text{State}) : \text{Set}(\text{Transition}) =$

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

Rule 40. Function allSTMTransitions

$\text{allSTMTransitions}(s : \text{StateMachineDef}) : \text{Set}(\text{Transition}) =$

$$\underline{s.\text{transitions} \cup \bigcup \{x : s.\text{nodes} \mid s \in \text{State} \bullet \text{allTransitions}^7(x)\}}$$

4.1.4 Statements

Rule 41. Semantics of statements

$\llbracket s : \text{Statement} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$

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

The semantics of statements, in general, has the format

$$\text{get_}x_1?x_1 \rightarrow \dots \rightarrow \text{get_}x_n?x_n \rightarrow P$$

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

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

Rule 42. Read state of an expression

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

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

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

Rule 43. Semantics of statements in context

$\llbracket s : \text{Statement} \rrbracket_{\text{StatementInContext}} : \text{CSPPProcess} =$

$\text{readState}^2(\text{usedVariables}^1(s), \llbracket s \rrbracket_{\text{Statement}^1})$

Rule 44. Function usedVariables

$\text{usedVariables}(s : \text{Statement}) : \text{CSPPProcess} =$

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

Rule 45. Semantics of assignment

$\llbracket s : \text{Assignment} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$

$\text{set_vid}(s.\text{left})! \llbracket s.\text{right} \rrbracket_{\text{Expr}^2} \rightarrow \text{Skip}$

Rule 46. Semantics of call statement
$$\llbracket s : \text{Call} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$$

$$\text{op.name} \text{Call} \rightarrow \text{body}; \text{op.name} \text{Ret} \rightarrow \text{Skip}$$
where
$$\text{op} = \text{s.operation}$$
$$\text{opdef} = \text{findOperationDefinition}(\text{op})$$
$$\text{body} = \left(\begin{array}{l} \text{if } (\text{opdef} = \text{null}) \text{ then} \\ \quad \text{Skip} \sqcap \text{Stop} \\ \text{else} \\ \quad \llbracket \text{opdef} \rrbracket_{\text{STM}^2} (\{x : \text{s.args} \bullet \llbracket x \rrbracket_{\text{Expr}^3}\}) \end{array} \right);$$

Rule 47. Semantics of if statements
$$\llbracket s : \text{IfStatement} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$$

$$\left(\begin{array}{l} \text{if } \llbracket \text{s.expression} \rrbracket_{\text{Expr}^4} \\ \quad \text{then } \llbracket \text{s.then} \rrbracket_{\text{StatementInContext}^1} \\ \quad \text{else if } (\text{s.else} \neq \text{null}) \text{ then } \llbracket \text{s.else} \rrbracket_{\text{StatementInContext}^2} \text{ else } \text{Skip} \end{array} \right)$$

Rule 48. Semantics of send event statements
$$\llbracket s : \text{SendEvent} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$$

$$\begin{aligned} & \text{if } (\text{type} = \text{INPUT}) \text{ then} \\ & \quad \frac{\text{eventId}(\text{event})? \text{par.name} \rightarrow \text{set_vid}(\text{par})! \text{par.name} \rightarrow \text{Skip}}{\text{eventId}(\text{event})? \text{par.name} \rightarrow \text{set_vid}(\text{par})! \text{par.name} \rightarrow \text{Skip}} \\ & \text{if } (\text{type} = \text{OUTPUT}) \text{ then} \\ & \quad \frac{\text{eventId}(\text{event})! \llbracket \text{value} \rrbracket_{\text{Expr}^5} \rightarrow \text{Skip}}{\text{eventId}(\text{event})! \llbracket \text{value} \rrbracket_{\text{Expr}^5} \rightarrow \text{Skip}} \\ & \text{if } (\text{type} = \text{SIMPLE}) \text{ then} \\ & \quad \text{eventId}(\text{event}) \rightarrow \text{Skip} \\ & \text{else} \\ & \quad \frac{\text{eventId}(\text{event}). \llbracket \text{value} \rrbracket_{\text{Expr}^6} \rightarrow \text{Skip}}{\text{eventId}(\text{event}). \llbracket \text{value} \rrbracket_{\text{Expr}^6} \rightarrow \text{Skip}} \end{aligned}$$
where
$$\begin{aligned} \text{type} &= \text{s.trigger.type} \\ \text{event} &= \text{s.trigger.event} \\ \text{value} &= \text{s.trigger.value} \\ \text{par} &= \text{s.trigger.parameter} \end{aligned}$$

Rule 49. Semantics of sequential composition
$$\llbracket s : \text{SeqStatement} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$$

$$\frac{\{ x : \text{s.statements} \bullet \llbracket x \rrbracket_{\text{StatementLnContext}^3} \}}{\{ x : \text{s.statements} \bullet \llbracket x \rrbracket_{\text{StatementLnContext}^3} \}}$$

Rule 50. Semantics of skip
$$\llbracket s : \text{Skip} \rrbracket_{\text{Statement}} : \text{CSPPProcess} =$$

$$\text{Skip}$$

Rule 51. Semantics of actions

$\llbracket a : \text{Action} \rrbracket_{\text{Action}} : \text{CSPProcess} =$

$\frac{\llbracket a.\text{action} \rrbracket}{\text{StatementInContext}^{\neq}}$

4.1.5 Expressions

Rule 52. Semantics of expressions

$\llbracket s : \text{Expression} \rrbracket_{\text{Expr}} : \text{CSPExpression} =$

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

Rule 53. Semantics of and expression

$\llbracket s : \text{And} \rrbracket_{\text{Expr}} : \text{CSPExpression} =$

$\frac{\llbracket s.\text{left} \rrbracket_{\text{Expr}^7} \wedge \llbracket s.\text{right} \rrbracket_{\text{Expr}^8}}$

Rule 54. Semantics of array expression

$\llbracket s : \text{ArrayExp} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPExpression} =$

$$\frac{\llbracket s.\text{value} \rrbracket_{\mathcal{E}_{\text{Expr}^0}} \left(\{ p : s.\text{parameters} \bullet \llbracket p \rrbracket_{\mathcal{E}_{\text{Expr}^0}} \} \right)}{\quad}$$

Rule 55. Semantics of boolean expression

$\llbracket s : \text{BooleanExp} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPExpression} =$

$\text{if } (s.\text{value} = \text{TRUE}) \text{ then } \underline{\text{true}} \text{ else } \underline{\text{false}}$

Rule 56. Semantics of call expression
$$\llbracket s : \text{CallExp} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$$

if (name = 'size' \wedge (head s.args **has type** SetType)) then
$$\text{card}(\llbracket \text{head s.args} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f1}})$$

else if (name = 'size' \wedge (head s.args **has type** SeqType)) then
$$\text{length}(\llbracket \text{head s.args} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f2}})$$

else
$$\text{name}(\{a : \text{s.args} \bullet \llbracket a \rrbracket_{\mathcal{E}_{\text{Expr}}^{f3}}\})$$

where

$$\text{name} = \text{s.function.name}$$

Rule 57. Semantics of concatenation expression
$$\llbracket s : \text{Cat} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$$

$$\llbracket \text{s.left} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f4}} \hat{\ } \llbracket \text{s.right} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f5}}$$

Rule 58. Semantics of not equal expression
$$\llbracket s : \text{Different} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$$

$$\llbracket \text{s.left} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f6}} \neq \llbracket \text{s.right} \rrbracket_{\mathcal{E}_{\text{Expr}}^{f7}}$$

Rule 59. Semantics of division

$\llbracket s : \text{Div} \rrbracket_{\mathcal{E}Expr} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{18}}}{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{19}}}$$

Rule 60. Semantics of equality

$\llbracket s : \text{Equals} \rrbracket_{\mathcal{E}Expr} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{20}}}{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{21}}}$$

Rule 61. Semantics of greater or equal expression

$\llbracket s : \text{GreaterOrEqual} \rrbracket_{\mathcal{E}Expr} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{22}}}{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{23}}}$$

Rule 62. Semantics of greater than

$\llbracket s : \text{GreaterThan} \rrbracket_{\mathcal{E}Expr} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{24}}}{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{25}}}$$

Rule 63. Semantics of if and only if expression

$\llbracket s : \text{Iff} \rrbracket_{\mathcal{E}xpr} : \text{CSPExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}xpr^{26}}}{\text{---}} \Leftrightarrow \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}xpr^{27}}}{\text{---}}$$

Rule 64. Semantics of implication

$\llbracket s : \text{Implies} \rrbracket_{\mathcal{E}xpr} : \text{CSPExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}xpr^{28}}}{\text{---}} \Rightarrow \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}xpr^{29}}}{\text{---}}$$

Rule 65. Semantics of integer expression

$\llbracket s : \text{IntegerExp} \rrbracket_{\mathcal{E}xpr} : \text{CSPExpression} =$

s.value

Rule 66. Semantics of less or equal expression

$\llbracket s : \text{LessOrEqual} \rrbracket_{\mathcal{E}xpr} : \text{CSPExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}xpr^{30}}}{\text{---}} \leq \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}xpr^{31}}}{\text{---}}$$

Rule 67. Semantics of less than

$\llbracket s : \text{LessThan} \rrbracket_{\mathcal{E}Expr} : \text{CSPEExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{32}}}{\text{Expr}^{32}} < \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{33}}}{\text{Expr}^{33}}$$

Rule 68. Semantics of minus

$\llbracket s : \text{Minus} \rrbracket_{\mathcal{E}Expr} : \text{CSPEExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{34}}}{\text{Expr}^{34}} - \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{35}}}{\text{Expr}^{35}}$$

Rule 69. Semantics of modulus

$\llbracket s : \text{Modulus} \rrbracket_{\mathcal{E}Expr} : \text{CSPEExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{36}}}{\text{Expr}^{36}} \text{ mod } \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{37}}}{\text{Expr}^{37}}$$

Rule 70. Semantics of multiplication

$\llbracket s : \text{Mult} \rrbracket_{\mathcal{E}Expr} : \text{CSPEExpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\mathcal{E}Expr^{38}}}{\text{Expr}^{38}} \times \frac{\llbracket s.\text{right} \rrbracket_{\mathcal{E}Expr^{39}}}{\text{Expr}^{39}}$$

Rule 71. Semantics of arithmetic negation

$\llbracket s : \text{Neg} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$

$$\frac{-\llbracket s.\text{exp} \rrbracket}{\mathcal{E}_{\text{Expr}}^{40}}$$

Rule 72. Semantics of logical negation

$\llbracket s : \text{Not} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$

$$\frac{\neg \llbracket s.\text{exp} \rrbracket}{\mathcal{E}_{\text{Expr}}^{41}}$$

Rule 73. Semantics of or expression

$\llbracket s : \text{Or} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket}{\mathcal{E}_{\text{Expr}}^{42}} \vee \frac{\llbracket s.\text{right} \rrbracket}{\mathcal{E}_{\text{Expr}}^{43}}$$

Rule 74. Semantics of parenthesised expression

$\llbracket s : \text{ParExp} \rrbracket_{\mathcal{E}_{\text{Expr}}} : \text{CSPEXpression} =$

$$\frac{(\llbracket s.\text{exp} \rrbracket)}{\mathcal{E}_{\text{Expr}}^{44}}$$

where

Rule 75. Semantics of plus

$\llbracket s : \text{Plus} \rrbracket_{\text{Expr}} : \text{CSPEXpression} =$

$$\frac{\llbracket s.\text{left} \rrbracket_{\text{Expr}^{45}}}{\text{Expr}^{45}} + \frac{\llbracket s.\text{right} \rrbracket_{\text{Expr}^{46}}}{\text{Expr}^{46}}$$

Rule 76. Semantics of range expression

$\llbracket s : \text{RangeExp} \rrbracket_{\text{Expr}} : \text{CSPEXpression} =$

$$\{x : \mathbb{N} \mid \frac{\llbracket s.\text{lrange} \rrbracket_{\text{Expr}^{47}}}{\text{Expr}^{47}} \text{ rel1 } x \wedge x \text{ rel2 } \frac{\llbracket s.\text{rrange} \rrbracket_{\text{Expr}^{48}}}{\text{Expr}^{48}}\}$$

where

$\text{rel1} = \text{if } (e.\text{linterval} = \text{I}) \text{ then } \leq \text{ else } <$

$\text{rel2} = \text{if } (e.\text{rinterval} = \text{J}) \text{ then } \geq \text{ else } >$

Rule 77. Semantics of sequence expression

$\llbracket s : \text{SeqExp} \rrbracket_{\text{Expr}} : \text{CSPEXpression} =$

$$\langle \{x : s.\text{values} \bullet \frac{\llbracket x \rrbracket_{\text{Expr}^{49}}}{\text{Expr}^{49}}\} \rangle$$

Rule 78. Semantics of set expression

$\llbracket s : \text{SetExp} \rrbracket_{\text{Expr}} : \text{CSPEXpression} =$

$$\{ \{x : s.\text{values} \bullet \frac{\llbracket x \rrbracket_{\text{Expr}^{50}}}{\text{Expr}^{50}}\} \}$$

Rule 79. Semantics of tuple expression
$$\llbracket s : \text{TupleExp} \rrbracket_{\mathcal{E}_{\text{CSP}}} : \text{CSPExpression} =$$

$$\left(\{x : s.\text{values} \bullet \llbracket x \rrbracket_{\mathcal{E}_{\text{CSP}}}\} \right)$$

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 $\text{stmClocks}(\text{stm}, \text{wcs})$, which relies on the calculation of wcs , a partial function from transitions to pairs, where the first component is the guard with occurrences of `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 $\text{stmMemory}(\text{stm}, \text{wcs})$ 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 80. Semantics of state machine
$$\llbracket \text{stm} : \text{StateMachineDef} \rrbracket_{STM} : \text{TimedCSPProcess} =$$

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

where

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

$$\text{clockMemSync} = \left(\begin{array}{l} \{ t : \text{Transition} \mid t \in \text{dom wcs} \bullet \text{triggerEvent}^1(t) \} \\ \cup \\ \{ v : \text{allClockVariables}^1(\text{wcs}) \bullet \text{setWC_vid}(v) \} \end{array} \right)$$

$$\text{flowevts} =$$

$$\cup \{ x : \text{SIDS} \setminus \text{substates}^9(\text{stm}); y : \text{substates}^{10}(\text{stm}) \bullet \{ \text{enter}._x._y, \text{entered}._x._y, \text{exit}._x._y, \text{exited}._x._y \} \}$$

Clocks

Functions related to clocks are formalised in this section.

Rule 81. allClockVariables function
$$\text{allClockVariables}(\text{wcs} : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \mathbb{P} \text{Variable}$$

$$\text{allClockVariables}(\text{wcs}) =$$

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

Rule 82. clockResets function

clockResets(wcs : Transition → (Expression, WC)) : ChannelSet

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

Rule 83. stmClocks function

stmClocks(wcs : Transition → (Expression, WC)) : TimedCSPPProcess =

$$\| (t, e, v) : \{t : \text{Transition}, e : \text{Expression}, v : \text{Variable} \mid t \in \text{dom wcs} \wedge (e \mapsto v) \in \pi_2(\text{wcs}(t))\} \\ \bullet \llbracket \alpha\text{WC}(t, e, v) \rrbracket \text{compileWC}(t, e, v)$$

where

$$\underline{\alpha\text{WC}(t, e, v)} = \{\text{triggerEvent}^2(t), \text{setWC_vid}(v)\} \cup \text{alphaClockReset}^2(e)$$

Rule 84. alphaClockReset function

alphaClockReset(e : Expression) : ChannelSet =

This rule is defined by multiple rules according to the subtype of the expression:
(85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97).

Rule 85. alphaClockReset function

alphaClockReset(e : ParExp) : ChannelSet =

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

Rule 86. alphaClockReset function

alphaClockReset(e : Not) : ChannelSet =

alphaClockReset(e) = alphaClockReset⁴(e.exp)

Rule 87. alphaClockReset function

alphaClockReset(e : CallExp) : ChannelSet =

alphaClockReset(e) = alphaClockResetCallArgs¹(e.args)

Rule 88. alphaClockResetCallArgs function

alphaClockResetCallArgs(s : seq Expression) : ChannelSet =

if #(s) > 0 then

alphaClockReset⁵(head(s)) ∪ alphaClockResetCallArgs²(tail(s))

else

 ∅

endif

Rule 89. alphaClockReset function

alphaClockReset(e : And) : ChannelSet =

alphaClockReset(e) = alphaClockReset⁶(e.left) ∪ alphaClockReset⁷(e.right)

Rule 90. alphaClockReset function

alphaClockReset(e : Or) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^8(e.\text{left}) \cup \alpha\text{ClockReset}^9(e.\text{right})$$

Rule 91. alphaClockReset function

alphaClockReset(e : Implies) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{10}(e.\text{left}) \cup \alpha\text{ClockReset}^{11}(e.\text{right})$$

Rule 92. alphaClockReset function

alphaClockReset(e : Iff) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{12}(e.\text{left}) \cup \alpha\text{ClockReset}^{13}(e.\text{right})$$

Rule 93. alphaClockReset function

alphaClockReset(e : GreaterThan) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{14}(e.\text{left}) \cup \alpha\text{ClockReset}^{15}(e.\text{right})$$

Rule 94. alphaClockReset function

alphaClockReset(e : GreaterOrEqual) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{16}(e.\text{left}) \cup \alpha\text{ClockReset}^{17}(e.\text{right})$$

Rule 95. alphaClockReset function

alphaClockReset(e : LessThan) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{18}(e.\text{left}) \cup \alpha\text{ClockReset}^{19}(e.\text{right})$$

Rule 96. alphaClockReset function

alphaClockReset(e : LessOrEqual) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{20}(e.\text{left}) \cup \alpha\text{ClockReset}^{21}(e.\text{right})$$

Rule 97. alphaClockReset function

alphaClockReset(e : Equals) : ChannelSet =

$$\alpha\text{ClockReset}(e) = \alpha\text{ClockReset}^{22}(e.\text{left}) \cup \alpha\text{ClockReset}^{23}(e.\text{right})$$

Rule 98. alphaClockReset function

$\alpha\text{ClockReset}(e : \text{ClockExp}) : \text{ChannelSet} =$

$\alpha\text{ClockReset}(e) = \{\{\text{clockReset.id}(e.\text{clock})\}\}$

Rule 99. alphaClockReset function

$\alpha\text{ClockReset}(e : \text{StateClockExp}) : \text{ChannelSet} =$

$\alpha\text{ClockReset}(e) = \{\{x : \text{SIDS} \mid \text{entered.id}(x).\text{id}(e.\text{state})\}\}$

Waiting Conditions

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

Rule 100. wc function

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

This rule is split into multiple rules according to the expression subtype:

(101, 102, 103, 105, 106, 107, 108, 109, 110, 111, 112, 113).

In every other case this function is defined by $\text{wc}(e) = (e, \emptyset)$

Rule 101. wc function (ParExp)

$wc(e : \text{ParExp}) : (\text{Expression}, \text{Expression} \rightarrow \text{Variable}) =$

$$wc(e) = (e_{\text{new}}, \pi_2(wc_{\text{result}}))$$

where

$$wc_{\text{result}} = wc^1(e.\text{exp})$$

$$e_{\text{new}} = e \oplus (\text{exp} \mapsto \pi_1(wc_{\text{result}}))$$

Rule 102. wc function (Not)

$wc(e : \text{Not}) : (\text{Expression}, \text{Expression} \rightarrow \text{Variable}) =$

$$wc(e) = (e_{\text{new}}, \pi_2(wc_{\text{result}}))$$

where

$$wc_{\text{result}} = wc^2(e.\text{exp})$$

$$e_{\text{new}} = e \oplus (\text{exp} \mapsto \pi_1(wc_{\text{result}}))$$

Rule 103. wc function (CallExp)

$wc(e : \text{CallExp}) : (\text{Expression}, \text{Expression} \rightarrow \text{Variable}) =$

$$wc(e) = (e_{\text{new}}, \pi_2(wc_{\text{result}}))$$

where

$$wc_{\text{result}} = wcArgSeq^1(e.\text{args})$$

$$e_{\text{new}} = e \oplus (\text{args} \mapsto \pi_1(wc_{\text{result}}))$$

Rule 104. wcArgSeq function

$\text{wcArgSeq}(s : \text{seq}(\text{Expression})) : (\text{seq Expression}, \text{Expression} \mapsto \text{Variable}) =$

if $\#(s) > 0$ then

$$\langle \langle \pi_1(\text{wc}_{\text{head}}) \rangle \hat{\cap} \pi_1(\text{wc}_{\text{tail}}), \pi_2(\text{wc}_{\text{head}}) \cup \pi_2(\text{wc}_{\text{tail}}) \rangle$$

where

$$\text{wc}_{\text{head}} = \text{wc}^3(\text{head}(s))$$

$$\text{wc}_{\text{tail}} = \text{wcArgSeq}^2(\text{tail}(s))$$

else

$$\langle \langle \rangle, \emptyset \rangle$$

endif

Rule 105. wc function (And)

$\text{wc}(e : \text{And}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{wc}(e) = \langle e_{\text{new}}, \pi_2(\text{wc}_{\text{left}}) \cup \pi_2(\text{wc}_{\text{right}}) \rangle$$

where

$$\text{wc}_{\text{left}} = \text{wc}^4(e.\text{left})$$

$$\text{wc}_{\text{right}} = \text{wc}^5(e.\text{right})$$

$$e_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(\text{wc}_{\text{left}}), \text{right} \mapsto \pi_1(\text{wc}_{\text{right}}))$$

Rule 106. wc function (Or)

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

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

where

$$\text{wc}_{\text{left}} = \text{wc}^6(\text{e.left})$$

$$\text{wc}_{\text{right}} = \text{wc}^7(\text{e.right})$$

$$\text{e}_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(\text{wc}_{\text{left}}), \text{right} \mapsto \pi_1(\text{wc}_{\text{right}}))$$

Rule 107. wc function (Implies)

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

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

where

$$\text{wc}_{\text{left}} = \text{wc}^8(\text{e.left})$$

$$\text{wc}_{\text{right}} = \text{wc}^9(\text{e.right})$$

$$\text{e}_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(\text{wc}_{\text{left}}), \text{right} \mapsto \pi_1(\text{wc}_{\text{right}}))$$

Rule 108. wc function (Iff)

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

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

where

$$\text{wc}_{\text{left}} = \text{wc}^{10}(\text{e.left})$$

$$\text{wc}_{\text{right}} = \text{wc}^{11}(\text{e.right})$$

$$\text{e}_{\text{new}} = e \oplus (\text{left} \mapsto \pi_1(\text{wc}_{\text{left}}), \text{right} \mapsto \pi_1(\text{wc}_{\text{right}}))$$

Rule 109. wc function (GreaterThan)

$wc(e : \text{GreaterThan}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{if } \left(\begin{array}{l} e.\text{left} \in \text{ClockExp} \vee e.\text{right} \in \text{ClockExp} \\ \vee \\ e.\text{left} \in \text{StateClockExp} \vee e.\text{right} \in \text{StateClockExp} \end{array} \right) \text{ then}$$

$$\underline{(b, \{e \mapsto b\})}$$

else

$$\underline{(e, \emptyset)}$$

endif

where

\underline{b} is a fresh identifier

Rule 110. wc function (GreaterOrEqual)

$wc(e : \text{GreaterOrEqual}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{if } \left(\begin{array}{l} e.\text{left} \in \text{ClockExp} \vee e.\text{right} \in \text{ClockExp} \\ \vee \\ e.\text{left} \in \text{StateClockExp} \vee e.\text{right} \in \text{StateClockExp} \end{array} \right) \text{ then}$$

$$\underline{(b, \{e \mapsto b\})}$$

else

$$\underline{(e, \emptyset)}$$

endif

where

\underline{b} is a fresh identifier

Rule 111. wc function (LessThan)

$wc(e : \text{LessThan}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{if } \left(\begin{array}{l} e.\text{left} \in \text{ClockExp} \vee e.\text{right} \in \text{ClockExp} \\ \vee \\ e.\text{left} \in \text{StateClockExp} \vee e.\text{right} \in \text{StateClockExp} \end{array} \right) \text{ then}$$

$$\underline{(b, \{e \mapsto b\})}$$

else

$$\underline{(e, \emptyset)}$$

endif

where

\underline{b} is a fresh identifier

Rule 112. wc function (LessOrEqual)

$wc(e : \text{LessOrEqual}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{if } \left(\begin{array}{l} e.\text{left} \in \text{ClockExp} \vee e.\text{right} \in \text{ClockExp} \\ \vee \\ e.\text{left} \in \text{StateClockExp} \vee e.\text{right} \in \text{StateClockExp} \end{array} \right) \text{ then}$$

$$\underline{(b, \{e \mapsto b\})}$$

else

$$\underline{(e, \emptyset)}$$

endif

where

\underline{b} is a fresh identifier

Rule 113. wc function (Equals)

$wc(e : \text{Equals}) : (\text{Expression}, \text{Expression} \mapsto \text{Variable}) =$

$$\text{if } \left(\begin{array}{l} e.\text{left} \in \text{ClockExp} \vee e.\text{right} \in \text{ClockExp} \\ \vee \\ e.\text{left} \in \text{StateClockExp} \vee e.\text{right} \in \text{StateClockExp} \end{array} \right) \text{ then}$$

$$\quad \underline{(b, \{e \mapsto b\})}$$

else

$$\quad \underline{(e, \emptyset)}$$

endif

where

b is a fresh identifier

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

Rule 114. compileWC function

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

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

Rule 115. getClockReset function

$\text{getClockReset}(e : \text{Expression}) : \text{TimedCSPPProcess} =$

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

Rule 116. getClockReset function

getClockReset(e : StateClockExp) : TimedCSPPProcess =

entered?x?.id(e.state) → Skip

Rule 117. getClockReset function

getClockReset(e : ClockExp) : TimedCSPPProcess =

clockReset.id(e.clock) → Skip

Rule 118. compileWC function

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

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

let

$\text{Reset} = \text{getClockReset}^1(e.\text{left}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^3(t)\}) \\ \Delta_{[e.\text{right}]} \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^4(t)\}) \\ \text{Expr}^{52} \end{array} \right) \Delta \text{Reset}$

within

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

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

let

$\text{Reset} = \text{getClockReset}^2(e.\text{right}); \text{setWC_vid}(v)!true \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^5(t)\}) \\ \Delta_{[e.\text{left}]} \text{setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^6(t)\}) \\ \text{Expr}^{53} \end{array} \right) \Delta \text{Reset}$

within

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

endif

Rule 119. compileWC function

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

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

let

$\text{Reset} = \text{getClockReset}^3(e.\text{left}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^7(t)\}) \\ \Delta(\llbracket e.\text{right} \rrbracket_{\text{Expr}^{54}} + 1) \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^8(t)\}) \end{array} \right) \Delta \text{Reset}$

within

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

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

let

$\text{Reset} = \text{getClockReset}^4(e.\text{right}); \text{setWC_vid}(v)!true \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^9(t)\}) \\ \Delta(\llbracket e.\text{left} \rrbracket_{\text{Expr}^{55}} + 1) \text{setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^{10}(t)\}) \end{array} \right) \Delta \text{Reset}$

within

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

endif

Rule 120. compileWC function

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

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

let

$\text{Reset} = \text{getClockReset}^5(e.\text{left}); \text{setWC_vid}(v)!true \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{11}(t)\}) \\ \Delta_{[e.\text{right}]} \text{setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^{12}(t)\}) \\ \text{Expr}^{56} \end{array} \right) \Delta \text{Reset}$

within

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

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

let

$\text{Reset} = \text{getClockReset}^6(e.\text{right}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{13}(t)\}) \\ \Delta_{[e.\text{left}]} \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^{14}(t)\}) \\ \text{Expr}^{57} \end{array} \right) \Delta \text{Reset}$

within

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

endif

Rule 121. compileWC function

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

if (e.left \in ClockExp \vee e.left \in StateClockExp) then

let

$\text{Reset} = \text{getClockReset}^7(e.\text{left}); \text{setWC_vid}(v)!true \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{15}(t)\}) \\ \Delta(\llbracket e.\text{right} \rrbracket_{\text{Expr}^{58}} + 1) \text{setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^{16}(t)\}) \end{array} \right) \Delta \text{Reset}$

within

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

else if (e.right \in ClockExp \vee e.right \in StateClockExp) then

let

$\text{Reset} = \text{getClockReset}^8(e.\text{right}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{17}(t)\}) \\ \Delta(\llbracket e.\text{left} \rrbracket_{\text{Expr}^{59}} + 1) \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^{18}(t)\}) \end{array} \right) \Delta \text{Reset}$

within

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

endif

Rule 122. compileWC function

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

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

let

$\text{Reset} = \text{getClockReset}^9(e.\text{left}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{19}(t)\}) \\ \frac{\Delta_{[[e.\text{right}]]} \quad \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^{20}(t)\})}{\text{Expr}^{60}} \\ \frac{\Delta_{[[e.\text{right}]]} \quad +1 \text{ setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^{21}(t)\})}{\text{Expr}^{61}} \end{array} \right) \Delta \text{Reset}$$

within

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

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

let

$\text{Reset} = \text{getClockReset}^{10}(e.\text{right}); \text{setWC_vid}(v)!false \rightarrow \text{Monitor}$

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\text{triggerEvent}^{22}(t)\}) \\ \frac{\Delta_{[[e.\text{left}]]} \quad \text{setWC_vid}(v)!true \rightarrow \text{RUN}(\{\text{triggerEvent}^{23}(t)\})}{\text{Expr}^{62}} \\ \frac{\Delta_{[[e]]} \quad +1 \text{ setWC_vid}(v)!false \rightarrow \text{RUN}(\{\text{triggerEvent}^{24}(t)\})}{\text{Expr}^{63}} \end{array} \right) \Delta \text{Reset}$$

within

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

endif

Trigger Deadlines

Rule 123. **deadlineEvents** function

deadlineEvents(s : StateMachineDef) : ChannelSet =

deadlineEvents(stm) = {t : allSTMTransitions²(s) | t.end ≠ null • *deadline*.id(t)}

Rule 124. **triggerEvent** function

triggerEvent(t : Transition) : CSPEvent =

if t.trigger ≠ null

 triggerEvent³(t.trigger, id(t))

else

 *internal*_.id(t)

Rule 125. State-machine Memory

$$\text{stmMemory}(s : \text{StateMachine}, \text{wcs} : \text{Transition} \rightarrow (\text{Expression}, \text{WC})) : \text{TimedCSPProcess} =$$

let $\text{Memory}(\underline{\text{vars}}) \hat{=}$

$$\left(\begin{array}{l} \square v : \underline{\text{lvars}} \bullet \left(\begin{array}{l} \underline{\text{get_vid}}(v, s) ! \underline{\text{name}}(v) \rightarrow \text{Memory}(\underline{\text{vars}}) \\ \square \\ \underline{\text{set_vid}}(v, s) ? x \rightarrow \text{Memory}(\underline{\text{vars}}[\underline{\text{name}}(v) := x]) \end{array} \right) \\ \square \\ \square v : \underline{\text{rvars}} \bullet \left(\begin{array}{l} \underline{\text{get_vid}}(v, s) ! \underline{\text{name}}(v) \rightarrow \text{Memory}(\underline{\text{vars}}) \\ \square \\ \underline{\text{set_vid}}(v, s) ? x \rightarrow \text{Memory}(\underline{\text{vars}}[\underline{\text{name}}(v) := x]) \\ \square \\ \underline{\text{set_Ext_vid}}(v, s) ? x \rightarrow \text{Memory}(\underline{\text{vars}}[\underline{\text{name}}(v) := x]) \end{array} \right) \\ \square \\ \square v : \underline{\text{cvars}} \bullet \underline{\text{setWC_vid}}(v) ? x \rightarrow \text{Memory}(\underline{\text{vars}}[\underline{\text{name}}(v) := x]) \\ \square \\ \square t : \underline{\text{allSTMTransitions}}^3(s) \bullet \underline{\text{memoryTransition}}^1(t, \text{wcs}); \text{Memory}(\underline{\text{vars}}) \\ \square \\ \square t : \underline{\text{allTriggerDeadlineTransitions}}^1(\text{stm}) \bullet \underline{\text{memoryDeadline}}^1(t, \text{wcs}); \text{Memory}(\underline{\text{vars}}) \end{array} \right)$$

within

$\text{Memory}(\underline{\text{varvalues}})$

where

$\underline{\text{rvars}} = \underline{\text{requiredVariables}}^8(s)$

$\underline{\text{lvars}} = \underline{\text{allLocalVariables}}^{10}(s)$

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

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

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

Rule 126. allTriggerDeadlineTransitions function

$\text{allTriggerDeadlineTransitions}(s : \text{StateMachineDef}) : \mathbb{P} \text{Transition} =$

$\text{allTriggerDeadlineTransitions}(s) = \{t : \text{allSTMTransitions}^4(s) \mid t.\text{end} \neq \text{null}\}$

Rule 127. memoryTransition function

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

if (t.condition \neq null) then
$$\frac{(\llbracket \pi_1(\text{wcs}(t)) \rrbracket) \& \llbracket t.\text{trigger} \rrbracket^{\text{id}(t)}}{\text{Expr}^{64} \quad \text{Trigger}^6}$$
else
$$\frac{\llbracket t.\text{trigger} \rrbracket^{\text{id}(t)}}{\text{Trigger}^7}$$

Rule 128. Memory deadline

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

if (t.condition \neq null) then
$$\frac{(\llbracket \pi_1(\text{wcs}(t)) \rrbracket) \& \text{deadline!id}(t)\text{!on} \rightarrow \text{Skip}}{\text{Expr}^{65}}$$
$$\square$$
$$\frac{(\neg \llbracket \pi_1(\text{wcs}(t)) \rrbracket) \& \text{deadline!id}(t)\text{!off} \rightarrow \text{Skip}}{\text{Expr}^{66}}$$
else
$$\text{deadline!id}(t)\text{!on} \rightarrow \text{Skip}$$

4.2.2 States

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

Rule 129. Timed semantics of states

$\llbracket s : \text{State} \rrbracket_S : \text{TimedCSPPProcess} =$

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

Rule 130. Timed semantics of simple states
$$\llbracket s : \text{State} \rrbracket_S : \text{TimedCSPPProcess} =$$

let
$$\text{Inactive} \hat{=} \text{enter}?x : \text{sids!id}(s) \rightarrow \text{Activating}(x)$$
$$\text{Activating}(o) \hat{=} \llbracket s.\text{entry} \rrbracket \xrightarrow{\text{Action}^{f1}} ; \text{initialisation}^5(s); \text{entered!o!id}(s) \rightarrow$$
$$\xrightarrow{\text{Action}^{f2}} (\text{triggerDeadlines}^1(s) \parallel \llbracket s.\text{during} \rrbracket ; \text{Stop}) \Delta$$
$$\left(\begin{array}{l} \square t : \text{transitionsFrom}^5(s) \bullet \llbracket t, s, \text{false} \rrbracket^{\text{Inactive, Activating}} \xrightarrow{\mathcal{T}^5} \\ \square \\ \square e : \text{Event} \bullet \text{if}(e.\text{type} == \text{null}) \\ \quad \text{then event!d}(e)?x : \text{tids} \rightarrow \text{exit}; \text{Inactive} \\ \quad \text{else event!d}(e)?x : \text{tids}?y \rightarrow \text{exit}; \text{Inactive} \end{array} \right)$$
within
$$\text{Inactive}$$
where
$$\#\text{substates}^{11}(s) = 0$$
$$\text{sids} = \text{SIDS} \setminus \{\text{id}(s)\}$$
$$\text{exit} = \text{exit}?x : \text{sids!id}(s) \rightarrow \text{exitSubstates}^4(s); \llbracket s.\text{exit} \rrbracket \xrightarrow{\text{Action}^{f3}} ; \text{exited!x!id}(s) \rightarrow \text{Skip}$$
$$\text{tids} = \text{TIDS} \setminus \text{tIDS}(s)$$

Rule 131. Timed semantics of composite states
$$\llbracket s : \text{State} \rrbracket_S : \text{TimedCSPPProcess} =$$

let
$$\text{Inactive} \hat{=} \text{enter}?x : \text{sids!id}(s) \rightarrow \text{Activating}(x)$$
$$\text{Activating}(o) \hat{=} \llbracket \text{s.entry} \rrbracket \text{Action}^{f4} ; \text{initialisation}^6(s); \text{entered!o!id}(s) \rightarrow$$
$$\text{(triggerDeadlines}^2(s) \parallel \llbracket \text{s.during} \rrbracket \text{Action}^{f5} ; \text{Stop}) \Delta$$
$$\left(\begin{array}{l} \square t : \text{transitionsFrom}^6(s) \bullet \llbracket t, s, \text{false} \rrbracket^{\text{Inactive, Activating}} \mathcal{T}^6 \\ \square \\ \square e : \text{Event} \bullet \text{if}(e.\text{type} == \text{null}) \\ \quad \text{then event!d}(e)?x : \text{tids} \rightarrow \text{exit}; \text{Inactive} \\ \quad \text{else event!d}(e)?x : \text{tids}?y \rightarrow \text{exit}; \text{Inactive} \end{array} \right)$$
within
$$(\text{Inactive} \llbracket \text{flowtrigevts} \rrbracket \text{composeStates}^1(\text{substates}^{12}(s), s))$$
where
$$\# \text{substates}^{13}(s) > 0$$
$$\text{flowtrigevts} = \text{flowTriggerEvents}^3(s)$$
$$\text{sids} = \text{SIDS} \setminus \{\text{id}(s)\}$$
$$\text{exit} = \text{exit}?x : \text{sids!id}(s) \rightarrow \text{exitSubstates}^5(s); \llbracket \text{s.exit} \rrbracket \text{Action}^{f6} ; \text{exited!x!id}(s) \rightarrow \text{Skip}$$
$$\text{tids} = \text{TIDS} \setminus \text{tIDS}(s)$$

Rule 132. Semantics of trigger deadlines

$\text{triggerDeadlines}(s : \text{State}) : \text{TimedCSPPProcess} =$

$$\lll t : \underline{\text{tDS}} \bullet \left(\mu X \bullet \left(\begin{array}{l} \underline{\text{deadline!id}(t)!on} \rightarrow \\ \underline{\text{readState}}^3 \left(\begin{array}{l} \underline{\text{usedV}(t.\text{end})}, \\ \underline{\text{deadline!id}(t)!off} \rightarrow \text{Skip} \blacktriangleright \lll \underline{\text{t.end}} \lll \end{array} \right) \end{array} \right) ; X \right)$$

where

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

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

Rule 133. Timed composition of states

$\text{composeStates}(ss : \text{seq State}, p : \text{NodeContainer}) : \text{TimedCSPPProcess} =$

if $\#ss = 1$

then

$\underline{\text{restrictedState}}^3(p, \text{head } ss)$

else

$$\left(\begin{array}{l} \underline{\text{restrictedState}}^4(p, \text{head } ss) \\ \lll \underline{\text{shflowevts}} \lll \\ \underline{\text{composeStates}}^2(\text{tail } ss, p) \end{array} \right)$$

where

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

4.2.3 Timed statements

Rule 134. Semantics of timed statements

$\llbracket s : \text{Statement} \rrbracket_{\text{Statement}} : \text{TimedCSPPProcess} =$

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

Rule 135. Function usedVariables for timed semantics

$\text{usedVariables}(s : \text{Statement}) : \text{CSPPProcess} =$

if $s \in \text{TimedStatement}$ then
 $\text{usedV}(s.\text{end}) \cup \text{usedVariables}(s.\text{stmt})$
else if $s \in \text{Wait}$ then
 $\text{usedV}(s.\text{duration})$
else
 $\text{usedVariables}^2(s)$

Rule 136. Semantics of Deadlines

$\llbracket s : \text{TimedStatement} \rrbracket_{\text{Statement}} : \text{TimedCSPPProcess} =$

$\llbracket s.\text{stmt} \rrbracket_{\text{Statement}^2} \blacktriangleright \llbracket s.\text{end} \rrbracket_{\text{Expr}^{68}}$

Rule 137. Semantics of Wait

$\llbracket s : \text{Wait} \rrbracket_{\text{Statement}} : \text{TimedCSPPProcess} =$

$\text{Wait}(\llbracket s.\text{duration} \rrbracket_{\text{Expr}^{69}})$

Rule 138. Semantics of Clock Reset

$\llbracket s : \text{ClockReset} \rrbracket \text{Statement} : \text{TimedCSPProcess} =$

$\text{clockReset!id}(s.\text{clock}) \rightarrow \text{Skip}$

Conclusions

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

We have described the semantics for the core constructs of RoboChart. It uses CSP, but we envisage its extension to use *Circus* [2], a process algebra that combines Z [12] and CSP, and includes time constructs [11]. Use of *Circus* and its UTP foundation will enable use of theorem proving as well as model checking.

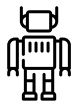
An approach for writing object-oriented simulations of RoboChart diagrams has also been defined. Automatic generation of simulations is possible and part of our future work. Verification of correctness of simulations will use the object-oriented version of *Circus* [3], with a semantics given by the UTP theory in [13].

RoboChart itself misses support for modelling the environment and the robotic platforms in model detail. It is also in our plans to take inspiration from hybrid automata [6] to extend the notation, and from the UTP model of continuous variables [4] to define the semantics.

Complete Metamodel: Core Language

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

A.1 Robotic Platforms



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


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

```

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

```

A.2 Interfaces


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

```

class Interface extends NamedElement, BasicContext;

```

A.3 Variables

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

```

class VariableList {
    attribute modifier : String[?];
    property vars : Variable[*|1] { ordered composes };
}
class Variable extends NamedExpression {
    attribute name : String[?];
    property type : Type[?] { composes };
    property initial : Expression[?] { composes };
    attribute modifier : String[?] { derived transient volatile };
}

```


A.4 Constants

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

```
Variable with modifier == 'con'
```

A.5 Events



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

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

A.6 Required, Provided and Defined Interfaces

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

based solely on the assumption that a potential target platform supports changing linear and angular speed by setting specific variables.

rInterfaces of Context

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

pInterfaces of Context

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

interfaces of Context

A.7 State-Machines



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


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

```

}
class Node extends NamedElement;
abstract class NodeContainer {
    property nodes : Node[*|1] { ordered composes };
    property transitions : Transition[*|1] { ordered composes };
}

```

A.8 States

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


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

```

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

```

A.9 Final states

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

```

class Final extends State;

```

A.10 Junction node

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

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

```
class Junction extends Node;
```

A.11 Initial nodes

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

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

```
class Initial extends Junction;
```

A.12 Transitions

→ A transition defines one possible path between two nodes in a state-machine or composite state. It contains source and target nodes as well as, optionally, a trigger in the form of an event, a boolean condition, and an action. The transition can only be executed if the event in the trigger is available and the condition is true.

The transition action is executed after the source state (if it exists) is exited, but before the target state (if it exists) is entered. Notice that source and target states are not necessarily available. For example transitions between junction nodes have neither.

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

A.13 Controllers



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

directly with robotic platform.

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

A.14 Connection

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

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

A.15 Modules

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

A.16 Statements

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

```

class ParStmt extends Statement {
  property stmt : Statement[?] { composes };
}
class Call extends Statement {
  property operation : Operation[?];
  property args : Expression[*|1] { ordered composes };
}

```

A.17 Expressions

```

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

```



```

    property variables : Variable[*|1] { ordered composes };
    property suchthat : Expression[?] { composes };
    property predicate : Expression[?] { composes };
}
class Exists extends Expression {
    property variables : Variable[*|1] { ordered composes };
    property suchthat : Expression[?] { composes };
    property predicate : Expression[?] { composes };
    attribute unique : Boolean[?];
}
class LambdaExp extends Expression {
    property variables : Variable[*|1] { ordered composes };
    property suchthat : Expression[?] { composes };
    property expression : Expression[?] { composes };
}
class DefiniteDescription extends Expression {
    property variables : Variable[*|1] { ordered composes };
    property suchthat : Expression[?] { composes };
    property expression : Expression[?] { composes };
}
class IfExpression extends Expression {
    property condition : Expression[?] { composes };
    property ifexp : Expression[?] { composes };
    property elseexp : Expression[?] { composes };
}
class Declaration {
    attribute name : String[?];
    property value : Expression[?] { composes };
}
class LetExpression extends Expression {
    property declarations : Declaration[*|1] { ordered composes };
    property expression : Expression[?] { composes };
}
class And extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}

```

```

}
class Not extends Expression {
  property exp : Expression[?] { composes };
}
class InExp extends Expression {
  property member : Expression[?] { composes };
  property set : Expression[?] { composes };
}
class TypeExp extends Expression {
  property type : Type[?] { composes };
}
class Equals extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Different extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class GreaterThan extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class GreaterOrEqual extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class LessThan extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class LessOrEqual extends Expression {
  property left : Expression[?] { composes };
  property right : Expression[?] { composes };
}
class Plus extends Expression {

```

```

    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Minus extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Modulus extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Mult extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Div extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Cat extends Expression {
    property left : Expression[?] { composes };
    property right : Expression[?] { composes };
}
class Neg extends Expression {
    property exp : Expression[?] { composes };
}
class Selection extends Expression {
    property receiver : Expression[?] { composes };
    property member : Member[?];
}
class IntegerExp extends Expression {
    attribute value : ecore::EInt[?];
}
class FloatExp extends Expression {
    attribute value : ecore::EFloat[?];
}

```

```

class StringExp extends Expression {
    attribute value : String[?];
}
class BooleanExp extends Expression {
    attribute value : String[?];
}
class VarExp extends Expression {
    property value : Variable[?];
}
class RefExp extends Expression {
    property ref : NamedExpression[?];
}
class EnumExp extends Expression {
    property type : Enumeration[?];
    property constant : Constant[?];
}
class ParExp extends Expression {
    property exp : Expression[?] { composes };
}
class SeqExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}
class SetExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}
class SetComp extends Expression {
    property variables : Variable[*|1] { ordered composes };
    property predicate : Expression[?] { composes };
    property expression : Expression[?] { composes };
}
class SetRange extends Expression {
    property start : Expression[?] { composes };
    property end : Expression[?] { composes };
}
class TupleExp extends Expression {
    property values : Expression[*|1] { ordered composes };
}

```

```

}
class RangeExp extends Expression {
  attribute linterval : String[?];
  property lrange : Expression[?] { composes };
  property rrange : Expression[?] { composes };
  attribute rinterval : String[?];
}
class CallExp extends Expression {
  property function : Function[?];
  property args : Expression[*|1] { ordered composes };
}
class ElseExp extends Expression;

```

A.18 Type Declaration

```
class TypeDecl extends NamedElement;
```

A.19 Primitive Types

T A primitive type

```
class PrimitiveType extends TypeDecl;
```

A.20 Datatypes



A datatype

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

```

F A field

```
class Field extends Member,NamedExpression;
```

A.21 Enumeration

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

A.22 Type Constructors

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

```
    property domain : Type[?] { composes };
}
class SeqType extends Type {
    property domain : Type[?] { composes };
}
class TypeRef extends Type {
    property ref : TypeDecll[?];
}
```

Complete Metamodel: Timed Language

B.1 Clock



A clock

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

B.2 Timed Statements

```
class Statement {
  property start : Expression[?] { composes };
  property stmt : Statement[?] { composes };
  property end : Expression[?] { composes };
}
class Wait extends Statement {
  property duration : Expression[?] { composes };
}
```


B.3 Timed Expressions

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

B.4 Timed Triggers


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

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


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](#)

 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] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [8] Object Management Group. *OMG Unified Modeling Language*, March 2015.
- [9] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
- [10] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.
- [11] 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.
- [12] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

- [13] 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.
- [14] 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, 50
- allClockVariables (**timed**), 59
- allLocalVariables, 29
- allSTMTransitions, 45
- allTransitions, 45
- allTriggerDeadlineTransitions (**timed**), 79
- allVariables, 29
- alphaClockReset, 60
 - And (**timed**), 61
 - CallExp (**timed**), 61
 - ClockExp (**timed**), 64
 - Equals (**timed**), 63
 - GreaterOrEqual (**timed**), 63
 - GreaterThan (**timed**), 62
 - Iff (**timed**), 62
 - Implies (**timed**), 62
 - LessOrEqual (**timed**), 63
 - LessThan (**timed**), 63
 - Not (**timed**), 61
 - Or (**timed**), 62
 - ParExp (**timed**), 60
 - StateClockExp (**timed**), 64
- alphaClockResetCallArgs (**timed**), 61
- buffer, 32
- C (Controller), 33
- clockResets (**timed**), 60
- compileTarget, 42
- composeWC (**timed**), 70, 72, 73, 74, 75, 76
- composeControllers, 31
- composeMachines, 35
- composeStates, 37, 83
- ctrlMemory, 34
- deadlineEvents (**timed**), 77
- exitSubstates, 43
- Expr (Expression), 50
 - And Expression, 50
 - Array Expression, 51
 - Boolean Expression, 51
 - Call Expression, 52
 - Concatenation Expression, 52
 - Division Expression, 53
 - Equals Expression, 53
 - Greater or Equal Expression, 53
 - Greater Than Expression, 53
 - If and Only If Expression, 54
 - Implies Expression, 54
 - Integer Expression, 54
 - Less or Equal Expression, 54
 - Less Than Expression, 55
 - Minus Expression, 55

- Modulus Expression, 55
- Multiplication Expression, 55
- Negation Expression, 56
- Not Equal Expression, 52
- Not Expression, 56
- Or Expression, 56
- Parenthesised Expression, 56
- Plus Expression, 57
- Range Expression, 57
- Sequence Expression, 57
- Set Expression, 57
- Tuple Expression, 58

- flowEvents, 38
- flowTriggerEvents, 41

- getClockReset (**timed**), 70, 71, 71
- getsetChannels, 37
- getsetLocalChannels, 38

- initialisation, 37

- M (Module), 28
- memoryChannels, 28
- memoryDeadline (**timed**), 79
- memoryTransition, 44
 - timed**, 79
- modMemory, 30

- renamingController, 31
- renamingMachine, 35
- renCtrlEvts, 32
- renStmEvts, 36
- requiredVariables, 29
- restrictedState, 41

- S (State), 38

- timed**, 80
- Composite State, 40
 - timed**, 82
- Final State, 40
- Simple State, 39
 - timed**, 81
- Statement, 45, 46, 47
 - timed**, 84, 84
 - ClockReset, 85
 - Wait, 84
- Call Statement, 48
- If Statement, 48
- Send Event, 49
- Sequential Composition, 49
- Skip, 49
- StatementInContext, 46
- STM (State Machine), 36
 - timed**, 59
- stmClocks (**timed**), 60
- stmMemory, 43
 - timed**, 78
- substates, 36
- substatesTriggers, 41

- T (Transition), 42
- transitionsFrom, 44
- trigEvents, 38
- Trigger, 44
- triggerDeadlines (**timed**), 83
- triggerEvent, 44, 77

- usedVariables, 47, 84

- wc (**timed**), 64, 65, 65, 65, 66, 67, 67, 67, 68, 68, 69, 69, 70
- wcArgSeq (**timed**), 66

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 , 39¹, 39², 39³, 40⁴, 40⁵, 40⁶,
42¹⁰, 42⁷, 42⁸, 42⁹, 81¹¹, 81¹²,
81¹³, 82¹⁴, 82¹⁵, 82¹⁶
- allClockVariables (timed) , 59¹, 78²
- allLocalVariables , 28¹, 30², 33³, 33⁴,
33⁵, 34⁶, 38⁷, 38⁸, 43⁹, 78¹⁰
- allSTMTransitions , 59¹, 77², 78³, 79⁴
- allTransitions , 38¹, 41², 41³, 41⁴, 43⁵,
45⁶, 45⁷
- allTriggerDeadlineTransitions (timed) ,
78¹
- allVariables , 37¹, 37²
- alphaClockReset (timed) , 60¹, 60², 60³,
61⁴, 61⁵, 61⁶, 61⁷, 62¹⁰, 62¹¹,
62¹², 62¹³, 62¹⁴, 62¹⁵, 62⁸, 62⁹,
63¹⁶, 63¹⁷, 63¹⁸, 63¹⁹, 63²⁰,
63²¹, 63²², 63²³
- alphaClockResetCallArgs (timed) , 61¹,
61²
- buffer , 28¹
- C (Controller) , 31¹
- clockResets (timed) , 59¹, 59²
- compileTarget , 42¹, 42², 42³
- composeControllers , 28¹, 31²
- composeMachines , 33¹, 35²
- composeStates , 36¹, 37², 40³, 59⁴
- composeStates (timed) , 82¹, 83²
- ctrlMemory , 33¹
- deadlineEvents (timed) , 59¹, 59²
- exitSubstates , 39¹, 40², 42³, 81⁴, 82⁵
- Expr (Expression) , 44¹, 47², 48³, 48⁴,
49⁵, 49⁶, 50⁷, 50⁸, 51¹⁰, 51⁹,
52¹¹, 52¹², 52¹³, 52¹⁴, 52¹⁵,
52¹⁶, 52¹⁷, 53¹⁸, 53¹⁹, 53²⁰,
53²¹, 53²², 53²³, 53²⁴, 53²⁵,
54²⁶, 54²⁷, 54²⁸, 54²⁹, 54³⁰,
54³¹, 55³², 55³³, 55³⁴, 55³⁵,
55³⁶, 55³⁷, 55³⁸, 55³⁹, 56⁴⁰,
56⁴¹, 56⁴², 56⁴³, 56⁴⁴, 57⁴⁵,
57⁴⁶, 57⁴⁷, 57⁴⁸, 57⁴⁹, 57⁵⁰,
58⁵¹, 72⁵², 72⁵³, 73⁵⁴, 73⁵⁵,
74⁵⁶, 74⁵⁷, 75⁵⁸, 75⁵⁹, 76⁶⁰,
76⁶¹, 76⁶², 76⁶³, 79⁶⁴, 79⁶⁵,
79⁶⁶, 83⁶⁷, 84⁶⁸, 84⁶⁹
- flowEvents , 37¹, 37², 83³, 83⁴
- flowTriggerEvents , 39¹, 40², 82³
- getClockReset (timed) , 72¹, 72², 73³,
73⁴, 74⁵, 74⁶, 75⁷, 75⁸, 76¹⁰, 76⁹
- getsetChannels , 36¹, 59²
- getsetLocalChannels , 36¹, 59²
- initialisation , 36¹, 39², 40³, 59⁴, 81⁵, 82⁶
- memoryChannels , 28¹, 28²

memoryDeadline (timed) , 78¹
 memoryTransition , 43¹
 memoryTransition (timed) , 78¹
 modMemory , 28¹

readState , 46¹, 46², 83³
 renamingController , 31¹, 31²
 renamingMachine , 35¹, 35²
 renCtrlEvs , 31¹, 31²
 renStmEvs , 35¹, 35²
 requiredVariables , 28¹, 30², 33³, 34⁴,
 34⁵, 38⁶, 43⁷, 78⁸
 restrictedState , 37¹, 37², 83³, 83⁴

S (State) , 41¹
 Statement , 46¹, 84²
 StatementInContext , 48¹, 48², 49³, 50⁴
 STM (State machine) , 35¹, 48²
 stmClocks (timed) , 59¹
 stmMemory , 36¹
 stmMemory (timed) , 59¹

substates , 36¹, 36², 38³, 39⁴, 40⁵, 40⁶,
 41⁷, 43⁸, 59¹⁰, 59⁹, 81¹¹, 82¹²,
 82¹³
 substatesTriggers , 41¹

T (Transition) , 37¹, 39², 40³, 42⁴, 81⁵,
 82⁶
 transitionsFrom , 39¹, 40², 41³, 42⁴, 81⁵,
 82⁶, 83⁷
 trigEvents , 36¹, 59², 59³
 Trigger , 42¹, 42², 42³, 44⁴, 44⁵, 79⁶, 79⁷
 triggerDeadlines (timed) , 81¹, 82²
 triggerEvent , 38¹, 41², 77³
 triggerEvent (timed) , 59¹, 60², 72³, 72⁴,
 72⁵, 72⁶, 73¹⁰, 73⁷, 73⁸, 73⁹,
 74¹¹, 74¹², 74¹³, 74¹⁴, 75¹⁵,
 75¹⁶, 75¹⁷, 75¹⁸, 76¹⁹, 76²⁰,
 76²¹, 76²², 76²³, 76²⁴

usedVariables , 46¹, 84²

wc (timed) , 65¹, 65², 66³, 66⁴, 66⁵, 67¹⁰,
 67¹¹, 67⁶, 67⁷, 67⁸, 67⁹
 wcArgSeq (timed) , 65¹, 66²

Index

Connection

- async, 13
- bidirec, 13
- Description, 12
- efrom, 13
- eto, 13
- from, 12
- to, 12

ConnectionNode

- Description, 12

Event

- Type, 13

MachineContainer

- Description, 12

Module

- Description, 13

RCPackage

- Description, 12

Robotic Platform

- Description, 12
- Metamodel, 87
- RoboticPlatformDef, 12
- RoboticPlatformRef, 12
- Well Formedness, 22