RoboChart Reference Manual

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

Contents

II Assertions

III Collections

IV Appendices

List of rules

1. Introduction

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

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

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

The time primitives of RoboChart allow time budgets and deadlines to be specified for operations and events directly as part of a state machine. Constraints can be specified in association with the relative-time elapsed since the occurrence of events or the entering of states. Our time primitives are inspired by constructs of timed automata [26] and Timed CSP [29].

UML [25] state machines are popular. RoboChart, however, is customised for robotic applications,

via the extra notions of robotic platform, controller, and a specialised API. Moreover, RoboChart provides support for time and probabilistic specifications that to make it suitable for verification and automatic generation of simulations.

In this report, we formalise the semantics of the core and timed constructs of RoboChart using CSP [7]. Importantly, CSP is a front end for a mathematical model that supports a number analysis techniques such as model-checking, which provide a high degree of automation, as well as more powerful (but not automatic) verification based on interactive and theorem proving, namely, Hoare and He's Unifying Theories of Programming [10] (UTP). Use of CSP enables model checking with FDR [31]. On the other hand, the underlying UTP model makes our core semantics adequate for extension to deal with time [6] and probability [13].

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

RoboChart

2. Language Structure

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

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

2.1 RoboChart metamodel

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

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

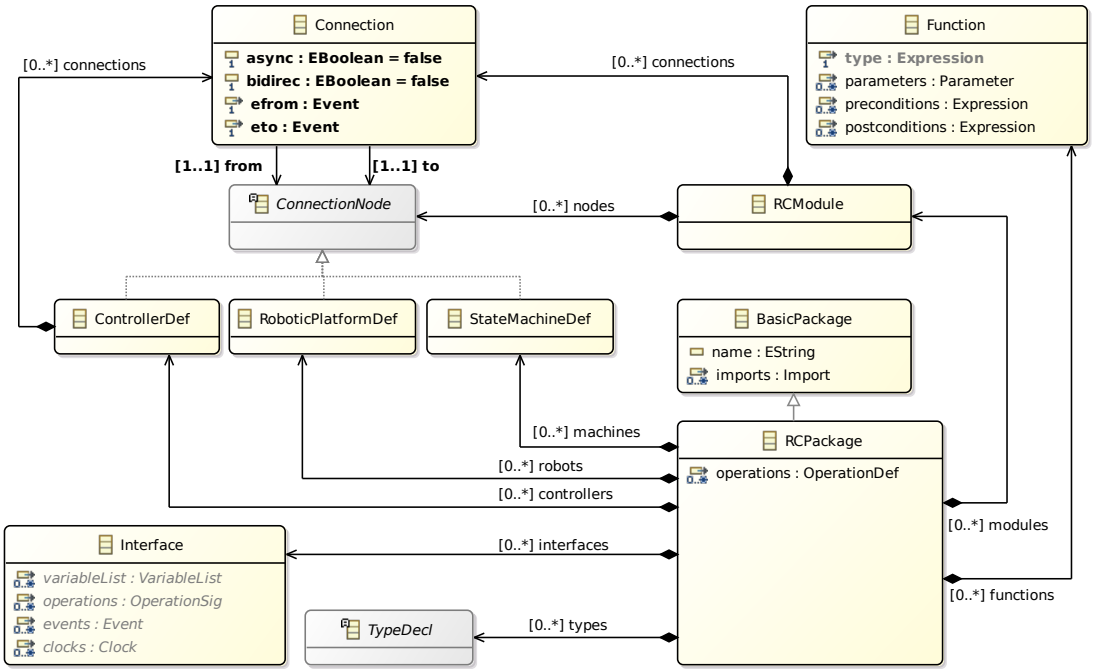

Figure 2.1: Metamodel of RoboChart packages

2.1.1 Module

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

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

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

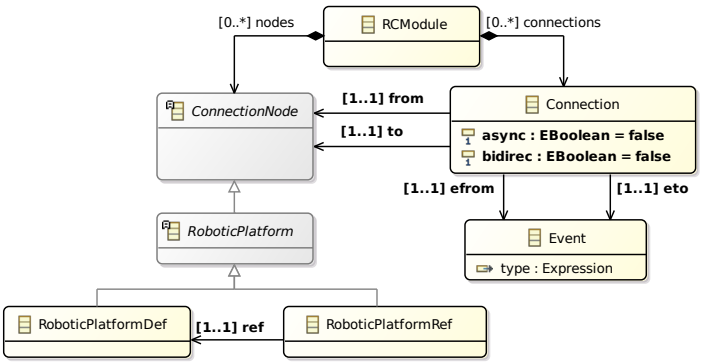

Figure 2.2: Metamodel of RoboChart modules

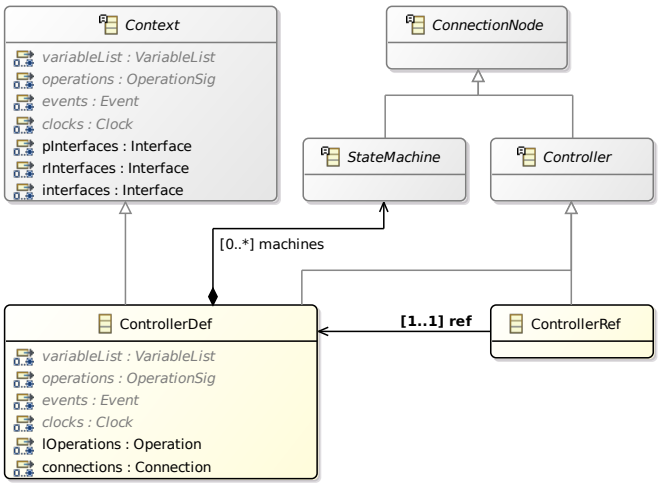

Figure 2.3: Metamodel of RoboChart controllers

2.1.2 Controller

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

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

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

Figure 2.4 also gives the metamodel for an Operation. It has an OperationSignature, which

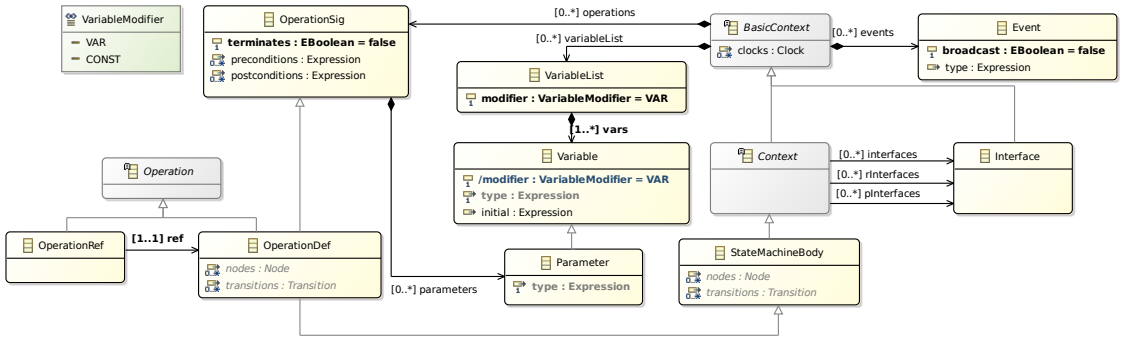

Figure 2.4: Metamodel of RoboChart context for elements and operations

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 conjunction is the actual postcondition. An Operation can also be defined by a reference or by a StateMachineBody.

2.1.3 State Machine

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

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

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

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

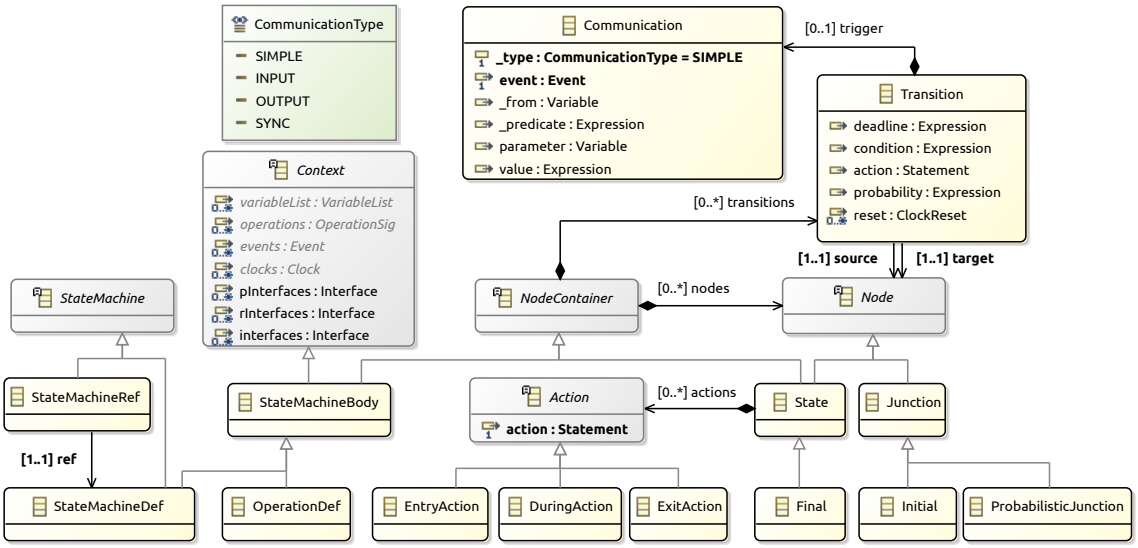

Figure 2.5: Metamodel of RoboChart state machines

composed states have a machine to define behaviour while in that state.

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

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

Syntax 2.1.1 — Transition Label.

```
Label ::= TriggerLabel TimedLabel ('['Expression']')? ('/'Statement)?
TriggerLabel ::= (Communication|'p{'Expression'}')?
TimedLabel ::= ClockReset* ('<{'Expression'}')?
ClockReset ::= '#'Clock
```

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

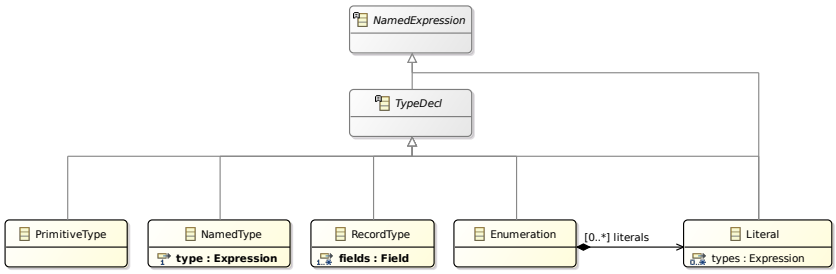

Figure 2.6: Metamodel of RoboChart type declarations

Syntax 2.1.2 — Communication.

```
Communication ::= (Input|Output|Sync|Simple)
Input ::= Event '?' Variable
Output ::= Event '!' Expression
Sync ::= Event '.' Expression
Simple ::= Event
```

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

Communication Type	Meaning
Input Communication (I)	Receives any value from the event and stores it on the variable.
Output Communication (O)	Sends the value of the expression through the event.
Sync Communication (Sync)	Synchronises on the event with the value of the expression.
Simple Communication (S)	Synchronises on the event.

Table 2.1 : **Types of Communications.**

2.1.4 Type Declaration

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

2.1.5 Expression

For simplicity, RoboChart adopts the type system of the Z Standard [1, 16]. This provides a simple and solid basis for implementation of all tools, including model checkers and theorem provers. It

also means that we do not need to specify the type system for RoboChart separately. Accordingly, the Expression language is also based on that of Z.

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

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

2.1.6 Action and Statement

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

Syntax 2.1.3 — Statements.

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

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

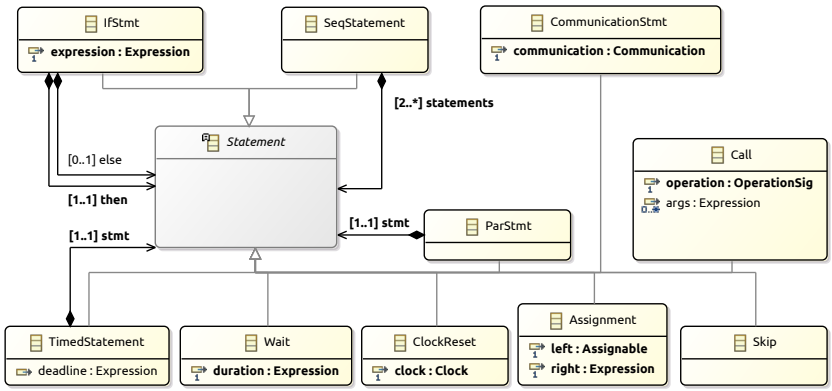

Figure 2.7: Metamodel of RoboChart Statements

Syntax 2.1.4 — Actions.

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

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

Table 2.2 : **Statements.**

2.2 Timed Primitives

The time primitives are described separately in Figure 2.8. They are shown in the previous section for completeness, and summarised and explained here. The timed primitives appear in the syntax of expressions, statements, and transitions. A ClockExpression since is a condition involving a clock. A StateClockExpression is a sinceEntry expression. A TimedStatement defines a deadline to terminate as an Expression. A Wait and ClockReset are also statements. The delay defined by Wait is that defined by the attribute duration as an Expression. Finally, a Transition may define a deadline as an Expression, and may have zero or more ClockResets. Table 2.3 gives the concrete syntax.

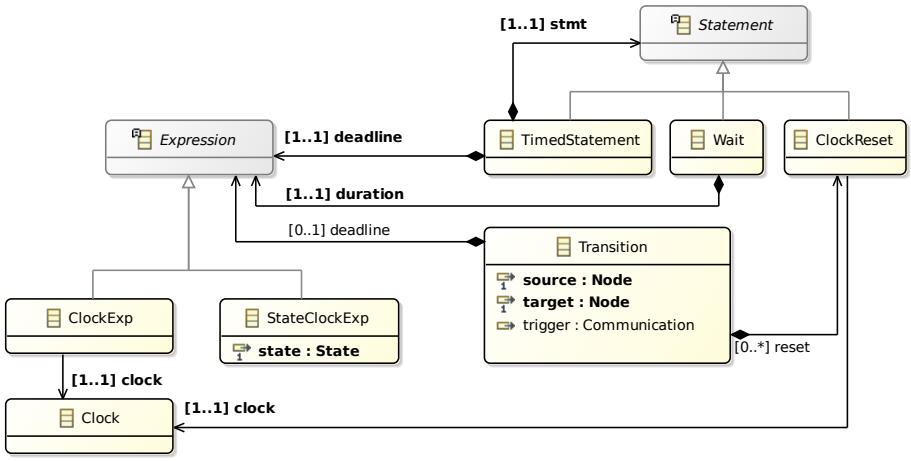

Figure 2.8: Metamodel of RoboChart time primitives

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

Table 2.3 : **Timed Primitives.**

We observe that a TimedStatement defines a deadline to terminate, but not a deadline to start. The possibility to specify a deadline to start was considered, however, because statements like assignment and operation calls are immediate, only an event synchronisation or a Wait statement could introduce a delayed start. For example, consider the case where we have an assignment of expression e to variable x, sequentially composed with a call to operation op as x:=e ; op(), then a deadline to start would be imposed on the assignment, which is immediate, and thus would be redundant. Another scenario arises if, instead, we consider the example Wait(d) ; op(). A starting deadline could constrain Wait(d), however, if we were to specify this statement as an operation waitOp(), then the starting deadline on waitOp() would be satisfied immediately, whereas this would not be the case for Wait(d) ; op().

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

3. Well-formedness Conditions

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

Well formedness requires well typedness. Here, however, we do not focus on this aspect, except where this is not standard for an expression or statement. The type system of RoboChart is the type system of Z[16].

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

3.1 Core Language

3.1.1 Robotic Platforms

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

We note that variables and operations declared directly in the platform, outside an interface, are

considered as if declared in a provided interface, for the reasons already explained above. Events declared directly in the platform, on the other hand, are defined.

3.1.2 Interfaces

I1 Provided interfaces contain only variables and operations, and required interfaces contain only variables, operation and clocks.

I2 Defined interfaces contain only variables, events and clocks.

I3 Names of variables, events, clocks and operations are unique.

3.1.3 Modules

M1 A module must contain exactly one robotic platform, at least one controller, and no state machines or operations.

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

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

3.1.4 → Connection

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

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

Cn2 Connections involving a robotic platform are always asynchronous.

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

Cn4 Only events of the same type may be connected.

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

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

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

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

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

Cn10 The to-event of a connection must be an event of its to-context and the from-event of a connection must be an event of its from-context.

Cn11 Connections between state machines are always synchronous.

3.1.5 Controllers

C1 A controller must contain at least one state machine.

C2 Controllers cannot provide variables or operations to other controllers.

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

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

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

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

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

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

C9 Controllers cannot have clocks.

C10 Controllers cannot define interfaces that contain clocks.

C11 Controllers cannot require interfaces that contain clocks.

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

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

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

3.1.6 State Machines

STM1 State machines cannot have provided interfaces

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

STM3 Every state machine must have exactly one initial junction.

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

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

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

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

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

STM9 A state machine that requires an operation, including a machine that defines another operation, must define all events defined in that operation.

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

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

3.1.7 States

S1 If a state has a non-empty set of nodes, then conditions 3 and 4 of *state machines* apply.
S2 A state has at most one of each type of action: entry, during, and exit,

3.1.8 Initial junctions

IJ1 An initial junction does not have incoming transitions.
IJ2 An initial junction must have exactly one outgoing transition.
IJ3 All junction conditions apply.

3.1.9 Junction

J1 A junction must contain at least one outgoing transition.
J2 The guards of the transitions out of a junction must form a cover.
J3 Transitions starting in junctions cannot have triggers.

3.1.10 Final states

FS1 Final states cannot be the source of transitions.
FS2 Final states cannot have actions.
FS3 Final states cannot have states or transitions.

3.1.11 Communications

Tg1 *A communication of type* SIMPLE *has neither the* parameter *attribute not the* value *attribute set.* This is a pure synchronisations and does not involve exchange of values.
Tg2 *A communication of type* SIMPLE *must use a typeless event.* This is a pure synchronisations

and does not involve exchange of values.

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

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

3.1.12 → Transitions

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

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

3.1.13 O Operations

O1 All state-machine conditions apply to operation definitions.

O2 Recursive and mutually recursive operations are not allowed.

3.1.14 X Variables

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

3.1.15 Expressions

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

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

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

3.2 Timed Language

3.2.1 State Machines

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

3.2.2 → Transitions

TT1 A transition with a deadline must have a trigger.

3.2.3 Timed Expressions

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

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

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

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

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

Name Disambiguation in State Clock Expressions

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

3.2.4 Timed Statements

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

TS2 In a non-deterministic wait((m,n)) the condition m < n must hold.

3.3 Probabilistic Language

3.3.1 → Transitions

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

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

3.3.2 Ⓟ Probabilistic Junction

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

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

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

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

4. Semantics

For the purpose of this semantics, the functions *vid*, *eventId*, and *id* calculate unique identifiers for their parameters, which are, respectively, variables, events 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 represented in the trigger by a null value, and the semantic rules have been adapted to handle it appropriately.

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

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

4.1 Detailed Semantics: Core Language

4.1.1 Modules

Rule 1. Semantics of modules

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

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

where

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

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

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

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

Rule 2. Hidden Module channels

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

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

where

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

Rule 3. Memory channels

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

$\{\mathsf{v} : \mathsf{allLocalVariables}^1(\mathsf{rp}) \bullet set\_\mathsf{vid}(\mathsf{v})\} \cup$

$\{\mathsf{v} : \mathsf{allLocalConstants}^1(\mathsf{rp}) \bullet set\_\mathsf{vid}(\mathsf{v})\} \cup$

$\bigcup\{\mathsf{c} : \mathsf{m.controllers} \bullet \{\mathsf{v} : \mathsf{requiredVariables}^1(\mathsf{c}) \bullet set\_EXT\_\mathsf{vid}(\mathsf{v}, \mathsf{c})\}\}$

where

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

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

Rule 4. Function allEvents

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

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

Rule 5. Function allVariables

allVariables(c : Context) : Set(Variable) =

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

Rule 6. Function requiredVariables

requiredVariables(c : Context) : Set(Variable) =

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

Rule 7. Function allLocalVariables

allLocalVariables(c : Context) : Set(Variable) =

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

Rule 8. Function allConstants

allConstants(c : Context) : Set(Variable) =

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

Rule 9. Function requiredConstants

requiredConstants(c : Context) : Set(Variable) =

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

Rule 10. Function allLocalConstants

allLocalConstants(c : Context) : Set(Variable) =

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

Rule 11. Function requiredOperations

requiredOperations(c : Context) : Set(Operation) =

$\bigcup\{i : c.RInterfaces \bullet i.operations\}$

Rule 12. Module Memory

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

$$
\begin{array}{l}
\textbf{let}\ \mathit{Memory}(\mathsf{vars}) \mathrel{\widehat{=}} \Box\, v : \mathsf{lvars} \bullet \mathit{set\_}\mathsf{vid}(\mathsf{v})?x \rightarrow \\
\quad (\mathop{\S}_{\S}\, c : \mathsf{rcontrollers}(\mathsf{v}) \bullet \mathit{set\_Ext\_}\mathsf{vid}(\mathsf{v},\mathsf{c}).x \rightarrow \mathit{Skip});\ \mathit{Memory}(\mathsf{vars}[\mathsf{name}(\mathsf{v}) := \mathsf{x}]) \\
\textbf{within} \\
\quad \mathsf{constInit}^1(\mathsf{rp});\ \mathit{Memory}(\mathsf{varvalues}) \\
\textbf{where} \\
\quad \mathsf{rp} = \mathsf{roboticPlatformDefinition}(\mathsf{m}) \\
\quad \mathsf{ctrls} = \mathsf{m.controllers} \\
\quad \mathsf{lvars} = \mathsf{allLocalVariables}^2(\mathsf{rp}) \\
\quad \mathsf{vars} = \langle \mathsf{v} : \mathsf{lvars} \bullet \mathsf{name}(\mathsf{v}) \rangle \\
\quad \mathsf{varvalues} = \langle \mathsf{v} : \mathsf{lvars} \bullet \mathsf{initial}(\mathsf{v}) \rangle \\
\quad \mathsf{rcontrollers} = \lambda\, \mathsf{v} \bullet \{\mathsf{c} : \mathsf{ctrls} \mid \mathsf{v} \in \mathsf{requiredVariables}^2(\mathsf{c})\}
\end{array}
$$

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

Rule 13. Constants Initialisation for Controllers and Modules

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

$$
\begin{array}{l}
\Big|\!\Big|\!\Big|\, c : \mathsf{consts} \bullet \left(\begin{array}{l}
\mathsf{if}\ \mathsf{c.initial} \neq \mathsf{NULL}\ \mathsf{then} \\
\quad \mathit{set\_}\mathsf{vid}(\mathsf{c})!\llbracket \mathsf{c.initial} \rrbracket_{\mathcal{E}_{expr}} \rightarrow \mathit{Skip} \\
\mathsf{else} \\
\quad \mathit{set\_}\mathsf{vid}(\mathsf{c})?\mathsf{name}(\mathsf{c}) \rightarrow \mathit{Skip}
\end{array} \right) \\
\textbf{where} \\
\quad \mathsf{consts} = \mathsf{allConstants}^1(\mathsf{node})
\end{array}
$$

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 14. Composition of controllers

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

$$
\begin{array}{l}
\textbf{if}\ \#\mathsf{ctrls} = 1 \\
\textbf{then} \\
\quad \mathsf{renamingController}^1(\mathsf{m}, \mathsf{head}\,\mathsf{ctrls}, \mathsf{cons}) \\
\textbf{else} \\
\quad \mathsf{renamingController}^2(\mathsf{m}, \mathsf{head}\,\mathsf{ctrls}, \mathsf{cons}) \\
\qquad \llbracket \mathsf{connevts} \rrbracket \\
\quad \mathsf{composeControllers}^2(\mathsf{m}, \mathsf{tail}\,\mathsf{ctrls}, \mathsf{cons}) \\
\textbf{where} \\
\quad \mathsf{connevts} = \mathsf{renCtrlEvts}^1(\mathsf{m}, \mathsf{head}\,\mathsf{ctrls}, \mathsf{cons}) \cap \\
\qquad\qquad \bigcup\{\mathsf{c} : \mathsf{tail}\,\mathsf{ctrls} \bullet \mathsf{renCtrlEvts}^2(\mathsf{m}, \mathsf{c}, \mathsf{cons})\}
\end{array}
$$

Rule 15. Renaming controller

renamingController(m : Module, c : Controller, cons : Set(Connection)) : CSPProcess =

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

where

internalConns = {x : cons • {x.from, x.to} ⊆ Controller ∧ ¬ x.async ∧ c ∈ {x.from, x.to}}

toPlatform = {x : cons • x.from = c ∧ x.to ∈ RoboticPlatform}

fromPlatform = {x : cons • x.to = c ∧ x.from ∈ RoboticPlatform}

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

Rule 16. Renaming controller events

renCtrlEvts(m : Module, c : Controller, cons : Set(Connection)) : ChannelSet =

$\{\!|$x : internalConns • eventId(x.efrom)$|\!\}$ ∪ $\{end\}$ ∪

$\{\!|$x : requiredConstants^2(c) • $\mathit{set}_$vid(x, m)$|\!\}$

where

internalConns = {x : cons • {x.from, x.to} ⊆ Controller ∧ ¬ x.async ∧ c ∈ {x.from, x.to}}

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

Rule 17. Buffer

buffer(c : Connection) : CSPProcess =

if c.mult **then**

singleBuffer(c.efrom, c.eto) ||| singleBuffer(c.eto, c.efrom)

else

singleBuffer(c.efrom, c.eto)

Rule 18. Single buffer

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

if efrom.type $\neq$ null
then
  let $Buffer(\langle\rangle) \widehat{=}$ eventId(efrom).$out?x \rightarrow Buffer(\langle x\rangle)$
    $Buffer(\langle v\rangle) \widehat{=}$ eventId(efrom).$out?x \rightarrow Buffer(\langle x\rangle) \Box$ eventId(eto).$in!v \rightarrow Buffer(\langle\rangle)$
  within $Buffer(\langle\rangle)$
else
  let $Buffer(false) \widehat{=}$ eventId(efrom).$out \rightarrow Buffer(true)$
    $Buffer(true) \widehat{=}$ eventId(efrom).$out \rightarrow Buffer(true) \Box$ eventId(eto).$in \rightarrow Buffer(false)$
  within $Buffer(false)$

4.1.2 Controllers

Rule 19. Semantics of controllers

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

let
  for each op : c.lOperations •
    nproc(op)({x : op.parameters • x.name}) = $[\![$opdef$]\!]^{nops}_{\mathcal{STM}'}$

  for each op : requiredOperations^1(c) •
    nproc(op)({x : op.parameters • x.name}) =
      o.name$Call$.{x : op.parameters • x.name} $\rightarrow SKIP$
within

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

where
  ms = $\langle$x : c.machines$\rangle$
  cs = c.connections
  opdef = findOperationDefinition(op, c.lOperations)
  nops = {op : c.lOperations • id(op) $\mapsto$ nproc(op)} $\cup$
    {op : requiredOperations^2(c) • id(op) $\mapsto$ nproc(op)}
  lvars = $\{\!|$v : allLocalVariables^3(c) • $set_$vid(v)$|\!\}$
  rvars = $\{\!|$v : requiredVariables^3(c) • $set_Ext_$vid(v)$|\!\}$
  lconsts = $\{\!|$v : allLocalConstants^2(c) • $set_$vid(v)$|\!\}$
  rconsts = $\{\!|$v : requiredConstants^3(c) • $set_$vid(v)$|\!\}$

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

Rule 20. Controller Memory

$\mathsf{ctrlMemory}(\mathsf{c} : \mathsf{ControllerDef}) : \mathsf{CSPProcess} =$

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

within

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

where

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

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

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

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

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

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

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

Rule 21. Composition of machines

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

$\quad \mathsf{if} \#\mathsf{ms} = 1$

$\quad \mathsf{then}$

$\qquad \mathsf{renamingMachine}^1(\mathsf{c}, \mathsf{head\,ms}, \mathsf{cons})^{\mathsf{nops}}$

$\quad \mathsf{else}$

$\qquad \mathsf{renamingMachine}^2(\mathsf{c}, \mathsf{head\,ms}, \mathsf{cons})^{\mathsf{nops}}$

$\qquad\quad [\![\mathsf{connevts}]\!]$

$\qquad \mathsf{composeMachines}^2(\mathsf{c}, \mathsf{tail\,ms}, \mathsf{cons})^{\mathsf{nops}}$

where

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

Rule 22. Renaming state machine

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

$$[\![\mathsf{m}]\!]^{\mathsf{nops}}_{\mathcal{STM}^2} \left[\!\!\left[\begin{array}{l} \{\mathsf{e} : \mathsf{internalConns} \bullet \mathsf{eventId}(\mathsf{e.eto}).\mathit{in} \leftarrow \mathsf{eventId}(\mathsf{e.efrom}).\mathit{out}\} \\ \cup \{\mathsf{e} : \mathsf{internalConns} \bullet \mathsf{eventId}(\mathsf{e.eto}).\mathit{out} \leftarrow \mathsf{eventId}(\mathsf{e.efrom}).\mathit{in}\} \\ \cup \{\mathsf{e} : \mathsf{fromController} \bullet \mathsf{eventId}(\mathsf{e.eto}) \leftarrow \mathsf{eventId}(\mathsf{e.efrom})\} \\ \cup \{\mathsf{e} : \mathsf{toController} \bullet \mathsf{eventId}(\mathsf{e.efrom}) \leftarrow \mathsf{eventId}(\mathsf{e.eto})\} \\ \cup \{\mathsf{v} : \mathsf{requiredConstants}^4(\mathsf{m}) \bullet \mathit{set\_}\mathsf{vid}(\mathsf{v}) \leftarrow \mathit{set\_}\mathsf{vid}(\mathsf{v},\mathsf{c})\} \end{array} \right]\!\!\right]$$

where

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

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

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

Rule 23. Renaming machine events

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

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

$\{\mathsf{x} : \mathsf{requiredConstants}^5(\mathsf{m}) \bullet \mathit{set\_}\mathsf{vid}(\mathsf{x},\mathsf{c})\}$

where

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

4.1.3 State machines

Rule 24. Semantics of state machine

$[\![\mathsf{stm} : \mathsf{StateMachineBody}]\!]^{\mathsf{nops}}_{\mathcal{STM}} : \mathsf{CSPProcess} =$

The rule is split according to the type of stm.

Rule 25. Semantics of state machine

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

$$\left(\left(\begin{array}{l} \left(\begin{array}{l} \left(\mathsf{stateful}^1(\mathsf{stm}) \setminus \{\mathit{end}\} ; \mathit{SStop} \triangle \mathit{end} \rightarrow \mathit{Skip} \right) \\ [\![\mathsf{v} : \mathsf{requiredVariables}^6(\mathsf{stm}) \mid \mathit{share} \leftarrow \mathit{set\_EXT\_}\mathsf{vid}(\mathsf{v})]\!] \\ |[\{\mathit{shared}\}]| \\ \mathit{Skip} \end{array} \right) \\ |[\mathsf{sharedVarSync} \cup \mathsf{sharedConstSync} \cup \{\mathit{end}\}]| \\ \left(\mathsf{sharedVarMemory}^1(\mathsf{stm}) \; |[\{\mathit{end}\}]| \; \mathsf{sharedConstMemory}^1(\mathsf{stm}) \right) \end{array} \right) \right)$$

$\setminus (\mathsf{sharedVarHide} \cup \mathsf{sharedConstSync})$

where

$\mathsf{sharedVarSync} = \{\!| \mathsf{v} : \mathsf{requiredVariables}^7(\mathsf{stm}) \bullet \mathit{get\_}\mathsf{vid}(\mathsf{v}), \mathit{set\_}\mathsf{vid}(\mathsf{v}), \mathit{set\_EXT\_}\mathsf{vid}(\mathsf{v}) |\!\}$

$\mathsf{sharedVarHide} = \{\!| \mathsf{v} : \mathsf{requiredVariables}^8(\mathsf{stm}) \bullet \mathit{get\_}\mathsf{vid}(\mathsf{v}) |\!\}$

$\mathsf{sharedConstSync} = \{\!| \mathsf{v} : \mathsf{requiredConstants}^6(\mathsf{stm}) \bullet \mathit{get\_}\mathsf{vid}(\mathsf{v}) |\!\}$

The semantic rule for operations is simpler, in that it does not have a model of the shared variables.

This is because the memory for state machines needs to conform to the semantics of Controllers, whereas operations do not engage in *set_EXT_* events directly. Instead, they synchronise on *shared* events, as well as on *set* and *get* events for all required variables.

Rule 26. Get and set channels

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

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

Rule 27. Semantics of defined operations

$[\![\mathsf{stm} : \mathsf{OperationDef}]\!]^{\mathsf{nops}}_{\mathcal{STM}} : \mathsf{CSPProcess} =$

$$\mathsf{stateful}^2(\mathsf{stm}) \setminus \{end\}$$

where

$\mathsf{stm.nodes} > 0$

Rule 28. Semantics of undefined operations

$[\![\mathsf{stm} : \mathsf{OperationDef}]\!]^{\mathsf{nops}}_{\mathcal{STM}} : \mathsf{CSPProcess} =$

let

$$Aux = SStop \triangle (Skip \sqcap Stop \sqcap ((Events \sqcap Variables); Aux))$$
$$Variables = \left(\begin{array}{l} \textbf{if } \#\mathsf{vars} > 0 \textbf{ then} \quad \bigsqcap \mathsf{e} : \mathsf{vars} \bullet set\_\mathsf{vid}(\mathsf{v}) \rightarrow Skip \sqcap get\_\mathsf{vid}(\mathsf{v}) \rightarrow Skip \\ \textbf{else} \quad Stop \end{array} \right)$$
$$Events = \left(\begin{array}{l} \textbf{if } \#\mathsf{events} > 0 \textbf{ then} \quad \bigsqcap \mathsf{e} : \mathsf{events} \bullet e \rightarrow Skip \\ \textbf{else} \quad Stop \end{array} \right)$$

within

Aux

where

$\mathsf{stm.nodes} = 0$

$\mathsf{stm.terminates} = \mathsf{false}$

$\mathsf{vars} = \mathsf{allVariables}^3(\mathsf{stm})$

$\mathsf{events} = \mathsf{allEvents}^1(\mathsf{stm})$

Rule 29. Stateful component of state machine body

stateful(stm : StateMachineBody)^{nops} : CSPProcess =

let

$$Stateful = \left(\left(\begin{array}{l} MachineBody \\ [\![\text{getsetLocalChannels}^1(\text{stm}) \cup \{end\}]\!] \\ \left(\text{varMemory}^1(\text{stm}) \; [\![\{end\}]\!] \; \text{constMemory}^1(\text{stm}) \right) \end{array} \right) \right) \\ \setminus \text{getsetLocalChannels}^2(\text{stm})$$

$$MachineBody = \left(\underbrace{[\![\text{stm}]\!]^{\text{nops}}}_{\mathcal{NC}'} \; [\![\{\!| interrupt |\!\}]\!] \; Skip \right) \setminus \text{enteredSS}$$

within

$Stateful$

where

enteredSS = $\{\!|$ x : stm.nodes $\bullet entered.\text{id}(x) |\!\}$

Rule 30. Get and set local channels

getsetLocalChannels(s : StateMachineBody) : ChannelSet =

$\{\!|$ v : allLocalVariables^5(s) $\bullet get\_\text{vid}(v), set\_\text{vid}(v) |\!\} \cup$
v : allLocalConstants^3(s) $\bullet get\_\text{vid}(v), set\_\text{vid}(v) |\!\}$

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

Memory

Rule 31. Variable memory

varMemory(stm : Context) : CSPProcess =

if #allLocalVariables^6(stm) > 0
$[\![\{end\}]\!]$ v : allLocalVariables^7(stm) $\bullet$ Memory^1(v)
else
$end \rightarrow Skip$

Rule 32. Constant memory

constMemory(stm : Context) : CSPProcess =

if #allLocalConstants^4(stm) > 0
$[\![\{end\}]\!]$ v : allLocalConstants^5(stm) $\bullet$ Constant^1(v)
else
$end \rightarrow Skip$

Rule 33. Shared variables memory

sharedVarMemory(stm : Context) : CSPProcess =

if #requiredVariables^{9}(stm) > 0

$|[\{end\}]|$ v : requiredVariables^{10}(stm) • sharedMemory^{1}(v)

else

$end \rightarrow Skip$

Rule 34. Shared constants memory

sharedConstMemory(stm : Context) : CSPProcess =

if #requiredConstants^{7}(stm) > 0

$|[\{end\}]|$ v : requiredConstants^{8}(stm) • sharedConstant^{1}(v)

else

$end \rightarrow Skip$

Rule 35. Individual variable memory

Memory(v : Variable) : CSPProcess =

let

$Memory(x) = get\_\text{vid}(\text{v})!x \rightarrow Memory(x) \Box set\_\text{vid}(\text{v})?x \rightarrow Memory(x) \Box end \rightarrow Skip$

within

if v.initial $\neq$ null $\quad Memory\left([\![\text{x.initial}]\!]_{\mathcal{E}xpr}\right)$

else $\quad \sqcap i : [\![\text{v.type}]\!]_{\mathcal{E}xpr} \mathrel{:\!\!\circ} \mathbb{P}\,\tau \bullet Memory(i)$

Rule 36. Individual constant

Constant(v : Variable) : CSPProcess =

let

$Constant(x) = get\_\text{vid}(\text{v})!x \rightarrow Constant(x) \Box end \rightarrow Skip$

within

if v.initial $\neq$ null $\quad Constant\left([\![\text{v.initial}]\!]_{\mathcal{E}xpr}\right)$

else $\quad \sqcap i : [\![\text{v.type}]\!]_{\mathcal{E}xpr} \mathrel{:\!\!\circ} \mathbb{P}\,\tau \bullet Constant(i)$

Rule 37. Individual shared variable memory

sharedMemory(v : Variable) : CSPProcess =

let

$$Memory(x) = \left(\begin{array}{l} get\_\mathsf{vid(v)}!x \rightarrow Memory(x) \\ \Box\ set\_\mathsf{vid(v)}?x \rightarrow Memory(x) \\ \Box\ set\_EXT\_\mathsf{vid(v)}?x \rightarrow Memory(x) \\ \Box\ end \rightarrow Skip \end{array} \right)$$

within

$$\begin{array}{l} \underline{\mathbf{if}}\ \mathsf{v.initial} \neq \mathsf{null} \quad Memory\left(\llbracket \mathsf{v.initial} \rrbracket_{\mathcal{E}xpr}\right) \\ \underline{\mathbf{else}} \quad \sqcap i : \llbracket \mathsf{v.type} \rrbracket_{\mathcal{E}xpr} \mathrel{\mathrel{:}\!\!\mid} \mathbb{P}\,\tau \bullet Memory(i) \end{array}$$

Rule 38. Individual shared constant memory

sharedConstant(v : Variable) : CSPProcess =

let

$$Memory(x) = \left(\begin{array}{l} get\_\mathsf{vid(v)}!x \rightarrow Memory(x) \\ \Box\ end \rightarrow Skip \end{array} \right)$$

within

$$set\_\mathsf{vid(v)}?x \rightarrow Memory(x)$$

4.1.4 NodeContainers

The semantics of a node container is split into three rules for convenience.

Rule 39. Node container semantics

$\llbracket \mathsf{nc} : \mathsf{NodeContainer} \rrbracket^{\mathsf{nops}}_{\mathcal{NC}}$: CSPProcess =

$\mathsf{ncBehaviour}^{1}(\mathsf{nc})^{\mathsf{nops}}$

Rule 40. Node container behaviour

ncBehaviour(nc : NodeContainer)^{nops} : CSPProcess =

$(\mathsf{ncCoreBehaviour}^{1}(\mathsf{nc})^{\mathsf{nops}} \setminus \{\!|enter, exit, exited, internal|\!\}) \llbracket \mathsf{renameTriggerEvents}^{1}(\mathsf{nc}) \rrbracket$

Rule 41. Node container core behaviour

$$\mathsf{ncCoreBehaviour}(\mathsf{nc} : \mathsf{NodeContainer})^{\mathsf{nops}} : \mathsf{CSPProcess} =$$

$$\left(\begin{array}{l} \left(\mathsf{composeNodes}^1(\mathsf{nc.nodes})^{\mathsf{nops}} [\![\mathsf{tidsR}(\mathsf{nc})]\!]\right) \\ [\![\mathsf{vars}^1(\mathsf{nc.nodes}) \mid \mathsf{flowevts} \cup \mathsf{transSync}(\mathsf{nc}) \cup \{end\} \mid \mathsf{vtguards}(\mathsf{nc}) : \emptyset]\!] \\ \left(enter.\mathsf{initial}(\mathsf{nc.nodes}) \rightarrow \mathsf{transitions}^1(\mathsf{nc})^{\mathsf{nops}}\right) \end{array}\right)$$

where

$$\mathsf{tidsR}(\mathsf{nc}) = \left(\begin{array}{l} \{\mathsf{t} : \mathsf{nc.transitions} \bullet interrupt.\mathsf{id}(\mathsf{t.source}) \leftarrow \mathsf{tevent}^1(\mathsf{t})\} \\ \cup \{\mathsf{n} : \mathsf{nc.nodes} \mid \mathsf{n} \in \mathsf{State} \bullet interrupt.\mathsf{id}(\mathsf{n}) \leftarrow interrupt.\mathsf{id}(\mathsf{nc})\} \end{array}\right)$$

$$\mathsf{flowevts} = \{\!| enter, exit, exited, interrupt |\!\}$$

$$\mathsf{transSync}(\mathsf{nc}) = \{\!| \mathsf{t} : \mathsf{nc.transitions} \bullet \mathsf{tevent}^2(\mathsf{t}) |\!\}$$

$$\mathsf{vtguards}(\mathsf{nc}) = \{\mathsf{t} : \mathsf{nc.transitions} \mid \exists \mathsf{v} \bullet \mathsf{v} \in \mathsf{usedVariables}^1(\mathsf{t.condition}) \bullet \mathsf{v}\}$$

We observe that in the meta parallel composition, the variables that are considered to be written in parallel by the semantics of the transitions is the empty set. This assumption is possible given that there is sequentiality between the moment a transition is taken (and some node is interrupted) and the possibility to change a variable in a transition, either via an InputCommunication trigger, or as a Transition's action.

Rule 42. Rename transition trigger events

$$\mathsf{renameTriggerEvents}(\mathsf{nc} : \mathsf{NodeContainer}) : \mathsf{RenamingSet} =$$

$$\{\mathsf{t} : \mathsf{ts} \bullet \mathsf{eventId}(\mathsf{t.trigger.event})\_\_.\mathsf{id}(\mathsf{t.source}) \leftarrow \mathsf{eventId}(\mathsf{t.trigger.event})\}$$

where

$$\mathsf{ts} = \{\mathsf{t} : \mathsf{nc.transitions} \mid \mathsf{t.trigger} \neq \mathsf{null}\}$$

Rule 43. Transition events

$$\mathsf{tevent}(\mathsf{t} : \mathsf{Transition}) : \mathsf{CSPEvent} =$$

if $\mathsf{t.trigger} = \mathsf{null}$

$\quad internal.\mathsf{id}(\mathsf{t.source})$

else

$\quad \mathsf{eventId}(\mathsf{t.trigger.event})\_\_.\mathsf{id}(\mathsf{t.source}).in$

Rule 44. Function usedVariables

usedVariables(s : Statement) : Set(Variable) =

if s ∈ Assignment then
  usedVariables^2(s.right)
else if s ∈ Call then
  ⋃{x : s.args • usedVariables^3(x)}
else if s ∈ IfStatement then
  usedVariables^4(s.expression)
else if s ∈ CommunicationStmt ∧ s.communication.type ∈ {SYNC, OUTPUT} then
  usedVariables^5(s.communication.value)
else if s ∈ ParStmt
  usedVariables^6(s.stmt)
else if s ∈ SeqStatement
  usedVariables^7(s.statements)
else if s ∈ TimedStatement
  usedVariables^8(s.stmt) ∪ usedVariables^9(s.deadline)
else
  {}

Rule 45. Function usedVariables

usedVariables(e : Expression) : Set(Variable) =

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

Rule 46. Function vars

vars(ns : ℙNode) : Set(NamedElement) =

⋃{n : ns • vars^2(n)}

Rule 47. Function vars

vars(ts : ℙTransition) : Set(NamedElement) =

⋃{t : ts • vars^3(t)}

Rule 48. Function vars

vars(s : Node) : Set(NamedElement) =

if s ∈ State
  ⋃{a : s.actions • vars^4(a.action)} ∪ vars^5(s.nodes) ∪ vars^6(s.transitions)
else
  ∅

Rule 49. Function vars

vars(t : Transition) : Set(NamedElement) =

vars^{7}(t.condition) ∪ vars^{8}(t.action) ∪ vars^{9}(t.deadline)

The following definition is similar to usedVariables^{10}, but takes into account the variables required by operations called in statements. This is required in order to ensure that operations can synchronise on state updates via *share*.

Rule 50. Function vars

vars(s : Statement) : Set(NamedElement) =

if s ∈ Assignment
  usedVariables^{11}(s.left) ∪ usedVariables^{12}(s.right)
else if s ∈ Call
  ⋃{x : s.args • usedVariables^{13}(x)} ∪ requiredVariables^{11}(s.operation)
else if s ∈ IfStatement
  usedVariables^{14}(s.expression) ∪ vars^{10}(s.then) ∪ vars^{11}(s.else)
else if s ∈ CommunicationStmt ∧ s.communication.type ∈ {SYNC, OUTPUT}
  usedVariables^{15}(s.communication.value)
else if s ∈ ParStmt
  vars^{12}(s.stmt)
else if s ∈ SeqStatement
  vars^{13}(s.statements)
else if s ∈ TimedStatement then
  vars^{14}(s.stmt) ∪ vars^{15}(s.deadline)
else
  {}

Rule 51. Function vars

vars(s : Seq(Statement)) : Set(NamedElement) =

if s ≠ ⟨⟩
  vars^{16}(head(s)) ∪ vars^{17}(tail(s))
else
  ∅

Composition

The composition of nodes in a NodeContainer is the replicated generalised parallel composition of the semantics of nodes n in nc.nodes, synchronising on events *share* and *end*.

Rule 52. Composition of nodes

$\mathsf{composeNodes}(\mathsf{nodes} : \mathsf{Set}(\mathsf{Node}))^{\mathsf{nops}} : \mathsf{CSPProcess} =$

$\textbf{if}\ \mathsf{nodes} = \emptyset$

$\quad SStop \triangle end \rightarrow Skip$

$\textbf{else}$

$\quad |[\{share, end\}]|\ \mathsf{n} : \mathsf{nodes} \bullet [\![\mathsf{n}]\!]^{\mathsf{nops}}_{\mathcal{N}'}$

4.1.5 Transitions

Rule 53. Semantics of Transitions

$\mathsf{transitions}(\mathsf{nc} : \mathsf{NodeContainer})^{\mathsf{nops}} : \mathsf{CSPProcess} =$

$\textbf{let}$

$$Trans \mathrel{\widehat{=}} \mathsf{readStateA}^{1}\left(\begin{array}{l} \mathsf{vguards}, \\ \left(\begin{array}{l} \left(\left(\left(\Box\, t : \mathsf{nc.transitions} \bullet [\![\mathsf{t}]\!]_{\mathcal{T}'}\right) \Box\, share \rightarrow Skip\right) ; Trans\right) \\ \Box \left(\begin{array}{l} interrupt.\mathsf{id}(\mathsf{nc}) \rightarrow (SStop \triangle exit \rightarrow Skip); \\ \quad (SStop \triangle exited \rightarrow end \rightarrow Skip) \end{array}\right) \\ \Box\, end \rightarrow Skip \end{array}\right) \end{array}\right)$$

$\textbf{within}$

$\quad Trans$

$\textbf{where}$

$\quad \mathsf{vguards} = \{\mathsf{t} : \mathsf{nc.transitions}, \mathsf{v} : \mathsf{NamedElement} \mid \mathsf{v} \in \mathsf{usedVariables}^{16}(\mathsf{t.condition}) \bullet \mathsf{v}\}$

Observe that in *Trans*, the process also offers the possibility to engage in the event $interrupt.\mathsf{id}(\mathsf{nc})$, so that a parent NodeContainer of *nc* may also interrupt the current state, and *end*, so that a final state may offer to terminate.

Rule 54. Semantics of a Transition

$[\![\mathsf{t} : \mathsf{Transition}]\!]^{\mathsf{nops}}_{\mathcal{T}} : \mathsf{CSPProcess} =$

$\textbf{if}\ \mathsf{src} \in \mathsf{State} \wedge \mathsf{t.condition} \neq \mathsf{null}$

$$\left(\begin{array}{l} \left([\![\mathsf{t.condition}]\!]_{\mathcal{E}xpr}\right) \,\&\, \mathsf{trigger}^{1}(\mathsf{t});\ (SStop \triangle exit \rightarrow Skip); \\ (SStop \triangle (exited \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatementInContext'}\ ;\ enter.\mathsf{id}(\mathsf{t.target}) \rightarrow Skip)) \end{array}\right)$$

$\textbf{else if}\ \mathsf{src} \in \mathsf{State} \wedge \mathsf{t.condition} = \mathsf{null}$

$$\left(\begin{array}{l} \mathsf{trigger}^{2}(\mathsf{t});\ (SStop \triangle exit \rightarrow Skip); \\ (SStop \triangle (exited \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatementInContext^{2}}\ ;\ enter.\mathsf{id}(\mathsf{t.target}) \rightarrow Skip)) \end{array}\right)$$

$\textbf{else if}\ \mathsf{src} \notin \mathsf{State} \wedge \mathsf{t.condition} \neq \mathsf{null}$

$$\left([\![\mathsf{t.condition}]\!]_{\mathcal{E}xpr}\right) \,\&\, \mathsf{tevent}^{3}(\mathsf{t}) \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatementInContext^{3}}\ ;\ enter.\mathsf{id}(\mathsf{t.target}) \rightarrow Skip$$

$\textbf{else if}\ \mathsf{src} \notin \mathsf{State} \wedge \mathsf{t.condition} = \mathsf{null}$

$$\mathsf{tevent}^{4}(\mathsf{t}) \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatementInContext^{4}}\ ;\ enter.\mathsf{id}(\mathsf{t.target}) \rightarrow Skip$$

Rule 55. Semantics of a transition's trigger

$\mathsf{trigger}(\mathsf{t} : \mathsf{Transition}) : \mathsf{CSPProcess} =$

$\mathbf{if}\ \mathsf{t.trigger} = \mathsf{null}$
$\quad \mathit{internal}.\mathrm{id}(\mathsf{t.source}) \rightarrow \mathit{Skip}$
$\mathbf{else\ if}\ \mathsf{t.trigger.type} = \mathsf{CommunicationType.INPUT}$
$\quad \mathsf{eventId}(\mathsf{t.event})\_\_.\mathrm{id}(\mathsf{t.source}).\mathit{in}?x \rightarrow (\mathit{SStop} \triangle \mathit{set\_}\mathsf{vid}(\mathsf{t.trigger.parameter})!x \rightarrow \mathit{Skip})$
$\mathbf{else\ if}\ \mathsf{t.trigger.type} = \mathsf{CommunicationType.SIMPLE}$
$\quad \mathsf{eventId}(\mathsf{t.event})\_\_.\mathrm{id}(\mathsf{t.source}).\mathit{in} \rightarrow \mathit{Skip}$

4.1.6 Nodes

The semantics of nodes is split according to the type of node. The CSP events used to control the execution flow of nodes are:

- *enter*.id(n), to request the activation of a node n;
- *interrupt*.id(n), to interrupt the execution of a node n;
- *end*, to terminate.

Rule 56. Semantics of nodes

$[\![\mathsf{n} : \mathsf{Node}]\!]^{\mathsf{nops}}_{\mathcal{N}} : \mathsf{CSPProcess} =$

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

Junctions (including Initial)

The semantics of an Initial node is exactly that of a Junction.

Rule 57. Semantics of junctions

$[\![\mathsf{n} : \mathsf{Junction}]\!]^{\mathsf{nops}}_{\mathcal{N}} : \mathsf{CSPProcess} =$

$\mathbf{let}$
$\quad \mathit{Inactive} \mathrel{\widehat{=}} \mathit{SStop} \triangle (\mathit{Activation} \Box \mathit{Termination})$
$\quad \mathit{Activation} \mathrel{\widehat{=}} \mathit{enter}.\mathrm{id}(\mathsf{n}) \rightarrow \mathit{Active}$
$\quad \mathit{Termination} \mathrel{\widehat{=}} \mathit{end} \rightarrow \mathit{Skip}$
$\quad \mathit{Active} \mathrel{\widehat{=}} (\mathit{SStop} \triangle \mathit{interrupt}.\mathrm{id}(\mathsf{n}) \rightarrow \mathit{Skip});\ \mathit{Inactive}$
$\mathbf{within}$
$\quad \mathit{Inactive}$

Final States

In addition to the CSP events *enter*, *interrupt* and *end*, States also engage in the following events: *entered*.id(n), used to indicate that a state n has entered; *exit*, used to synchronise the execution of a state's exit action (if any); and *exited* to indicate that a state has finished exiting.

Rule 58. Semantics of final states

$[\![\mathsf{n} : \mathsf{Final}]\!]^{\mathsf{nops}}_{\mathcal{N}} : \mathsf{CSPProcess} =$

let

$Inactive \mathrel{\widehat{=}} SStop \triangle (Activation \Box Termination)$

$Activation \mathrel{\widehat{=}} enter.\underline{\mathsf{id}(\mathsf{n})} \rightarrow Entering$

$Termination \mathrel{\widehat{=}} end \rightarrow Skip$

$Entering \mathrel{\widehat{=}} entered.\underline{\mathsf{id}(\mathsf{n})} \rightarrow Active$

$Active \mathrel{\widehat{=}} (SStop \triangle (Termination \Box interrupt.\underline{\mathsf{id}(\mathsf{n})} \rightarrow Interrupted))$

$Interrupted \mathrel{\widehat{=}} (SStop \triangle exit \rightarrow exited \rightarrow Inactive)$

within

$Inactive$

The possibility to terminate a state machine is enforced at the level of the semantics of a state machine, thus making the semantics of a final state independent from its context.

Simple States

We observe that *exited* and *exit* are parameterless, as the execution flow of states (other than via the use of during actions) is sequential. Only one state may be *interrup*ted at each level of the hierarchy, and so after an interruption, only that state is prepared to synchronise on the events *exit* and *exited*.

Rule 59. Semantics of simple states

$[\![\mathsf{s} : \mathsf{State}]\!]^{\mathsf{nops}}_{\mathcal{N}} : \mathsf{CSPProcess} =$

let

$Inactive \mathrel{\widehat{=}} SStop \triangle (Activation \Box Termination)$

$Activation \mathrel{\widehat{=}} enter.\underline{\mathsf{id}(\mathsf{s})} \rightarrow Active$

$Termination \mathrel{\widehat{=}} end \rightarrow Skip$

$Active \mathrel{\widehat{=}} \left(\underbrace{[\![\mathsf{s.entry}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^1} ; Behaviour; Exiting \right)$

$Behaviour \mathrel{\widehat{=}} entered.\underline{\mathsf{id}(\mathsf{s})} \rightarrow During$

$During \mathrel{\widehat{=}} \left(\left(\underbrace{[\![\mathsf{s.during}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^2} ; SStop \right) \triangle interrupt.\underline{\mathsf{id}(\mathsf{s})} \rightarrow Skip \right)$

$Exiting \mathrel{\widehat{=}} (SStop \triangle exit \rightarrow Skip); \underbrace{[\![\mathsf{s.exit}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^3} ; exited \rightarrow Inactive$

within

$Inactive$

where

$\#\mathsf{s.nodes} = 0$

Composite states

The process *Behaviour* accounts for the behaviour within the state, which is a composition of $\underbrace{[\![\mathsf{s}]\!]^{\mathsf{nops}}}_{\mathcal{NC}^2}$, that defines the behaviour of $\underline{\mathsf{s}}$ as a node container, including that of its states and transitions, and *During* that defines the semantics of the during action.

Rule 60. Semantics of composite states

$[\![\mathsf{s} : \mathsf{State}]\!]^{\mathsf{nops}}_{\mathcal{N}} : \mathsf{CSPProcess} =$

let

$Inactive \mathrel{\widehat{=}} SStop \triangle (Activation \Box Termination)$

$Activation \mathrel{\widehat{=}} enter.\mathsf{id}(\mathsf{s}) \rightarrow Active$

$Termination \mathrel{\widehat{=}} end \rightarrow Skip$

$$Active \mathrel{\widehat{=}} \left(\underbrace{[\![\mathsf{s.entry}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^4} ; Behaviour ; Exiting \right)$$

$$Behaviour \mathrel{\widehat{=}} \left(\left(\begin{array}{l} \underbrace{[\![\mathsf{s}]\!]^{\mathsf{nops}}}_{\mathcal{NC}^3} \\ |[\mathsf{vars}^{18}(\mathsf{s.nodes}) \mid \mathsf{enteredSS}(\mathsf{s}) \cup \{\!|end, interrupt|\!\} \mid \mathsf{vars}^{19}(\mathsf{s.during})]| \\ During \end{array} \right) \right) \setminus \mathsf{enteredSS}(\mathsf{s}) \cup \{end\}$$

$$During \mathrel{\widehat{=}} \left(\begin{array}{l} (SStop \triangle (\Box\, e : \mathsf{enteredSS}(\mathsf{s}) \bullet e \rightarrow Skip)); \\ \quad entered.\mathsf{id}(\mathsf{s}) \rightarrow \left(\left(\begin{array}{l} RUN(\mathsf{enteredSS}(\mathsf{s})) \mid\mid\mid \left(\underbrace{[\![\mathsf{s.during}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^5} ; SStop \right) \\ \triangle\, interrupt.\mathsf{id}(\mathsf{s}) \rightarrow (SStop \triangle end \rightarrow Skip) \end{array} \right) \right) \end{array} \right)$$

$$Exiting = (SStop \triangle exit \rightarrow Skip) ; \underbrace{[\![\mathsf{s.exit}]\!]^{\mathsf{nops}}}_{\mathcal{A}ction^6} ; exited \rightarrow Inactive$$

within

$Inactive$

where

$\#\mathsf{s.nodes} > 0$

$\mathsf{enteredSS}(\mathsf{s}) = \{\!|\mathsf{x} : \mathsf{s.nodes} \bullet entered.\mathsf{id}(\mathsf{x})|\!\}$

4.1.7 Statements

Rule 61. Semantics of statements

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

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

The semantics of statements, in general, has the format

$$SStop \triangle get\_x_1?x_1 \rightarrow \ldots \rightarrow SStop \triangle get\_x_n?x_n \rightarrow P$$

where the channels get_x_i read values from the memory and the process P models the actual statement. The input events $get\_x_i?x_i$ build a context where all the state components used in the expressions of the statement are declared, while $SStop$ captures the possibility for state to be changed. 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 62. Read state of an expression

readState(vs : seq(Variable), P : CSPProcess) : CSPProcess =

if (#vs = 0) then
  P
else
  $SStop \triangle get_$vid(head vs)?(head vs).name → readState^1(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.

The function readState^2 allows state updates, handled via synchronisation on *share*, to take place while individual variables are read. In cases where this lack of atomicity is undesirable, instead the following function readStateAtomic^1 is used, which refuses to engage in *share* initially if vs is non-empty.

Rule 63. Atomic read state of an expression

readStateAtomic(vs : seq(Variable), P : CSPProcess) : CSPProcess =

if (#vs = 0) then
  P
else
  $get_$vid(head vs)?(head vs).name → readStateAtomic^2(tail vs, P)

The following is a version of readStateAtomic^3 that allows the state to be changed, via *share*, before any synchronisation on $get\_$ happens.

Rule 64. Atomic read state of an expression with initial share

readStateA(vs : seq(Variable), P : CSPProcess) : CSPProcess =

if (#vs = 0) then
  P
else
  $SStop \triangle get_$vid(head vs)?(head vs).name → readStateAtomic^4(tail vs, P)

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

Rule 65. Semantics of statements in context

$[\![\text{s} : \text{Statement}]\!]^{\text{nops}}_{\mathcal{S}tatementInContext} : \text{CSPProcess} =$

$$\text{readState}^{3}\left(\text{usedVariables}^{17}(\text{s}), [\![\text{s}]\!]^{\text{nops}}_{\mathcal{S}tatement^{1}}\right)$$

Rule 66. Semantics of assignment

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

$$SStop \triangle set\_\text{vid}(\text{s.left})!\,[\![\text{s.right}]\!]_{\mathcal{E}xpr} \rightarrow Skip$$

Rule 67. Semantics of call statement

$[\![\text{s} : \text{Call}]\!]^{\text{nops}}_{\mathcal{S}tatement} : \text{CSPProcess} =$

$$\text{op.name}\,\text{nops}(\text{op})\left(\{\text{x} : \text{s.args} \bullet [\![\text{x}]\!]_{\mathcal{E}xpr}\}\right)$$

where

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

Rule 68. Semantics of if statements

$[\![\text{s} : \text{IfStatement}]\!]^{\text{nops}}_{\mathcal{S}tatement} : \text{CSPProcess} =$

$$\left(\begin{array}{l} \textbf{if}\ [\![\text{s.expression}]\!]_{\mathcal{E}xpr} \\ \textbf{then}\ \underbrace{[\![\text{s.then}]\!]^{\text{nops}}}_{\mathcal{S}tatementInContext^{5}} \\ \textbf{else}\ \underbrace{\text{if}\ (\text{s.else} \neq \text{null})\ \text{then}\ [\![\text{s.else}]\!]^{\text{nops}}}_{\mathcal{S}tatementInContext^{6}}\ \text{else}\ Skip \end{array}\right)$$

Rule 69. Semantics of communication statements

$[\![\text{s} : \text{CommunicationStmt}]\!]^{\text{nops}}_{\mathcal{S}tatement} : \text{CSPProcess} =$

$\text{if}\ (\text{type} = \text{INPUT})\ \text{then}$

$\quad SStop \triangle \text{eventId}(\text{event}).in?\text{par.name} \rightarrow (SStop \triangle set\_\text{vid}(\text{par})!\text{par.name} \rightarrow Skip)$

$\text{if}\ (\text{type} = \text{OUTPUT})\ \text{then}$

$\quad SStop \triangle \text{eventId}(\text{event}).out!\,[\![\text{value}]\!]_{\mathcal{E}xpr} \rightarrow Skip$

$\text{if}\ (\text{type} = \text{SIMPLE})\ \text{then}$

$\quad SStop \triangle \text{eventId}(\text{event}).out \rightarrow Skip$

else

$\quad SStop \triangle \text{eventId}(\text{event}).out.[\![\text{value}]\!]_{\mathcal{E}xpr} \rightarrow Skip$

where

$\text{type} = \text{s.communication.type}$

$\text{event} = \text{s.communication.event}$

$\text{value} = \text{s.communication.value}$

$\text{par} = \text{s.communication.parameter}$

Rule 70. Semantics of sequential composition

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

$\underset{\circ\!\!\!\;\circ}{\;}\{x : s.\mathsf{statements} \bullet [\![x]\!]^{\mathsf{nops}}_{\mathit{StatementInContext}^7}\}$

Rule 71. Semantics of Skip

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

$\mathit{SStop} \triangle \mathit{Skip}$

Rule 72. Semantics of ParStmt

$[\![s : \mathsf{ParStmt}]\!]^{\mathsf{nops}}_{\mathit{Statement}} : \mathsf{CSPProcess} =$

$[\![s.\mathsf{stmt}]\!]^{\mathsf{nops}}_{\mathit{StatementInContext}^8}$

Rule 73. Semantics of actions

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

$\mathbf{if}\ a \neq \mathsf{null}$

$\quad [\![a.\mathsf{action}]\!]^{\mathsf{nops}}_{\mathit{StatementInContext}^9}$

$\mathbf{else}$

$\quad \mathit{Skip}$

4.1.8 Expressions

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

4.2 Detailed Semantics: Timed Language

The semantics of modules 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. The untimed semantics of controllers, state machines, operations, and nodes is largely reused, and so we present the rules by focusing on the changes required to accommodate the timed semantics.

4.2.1 Controllers

Rule 74. Semantics of controllers

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

let

for each $op : c.\mathsf{lOperations} \bullet$

$\mathsf{nproc}(op)(\{x : op.\mathsf{parameters} \bullet x.\mathsf{name}\}) = [\![opdef]\!]^{nops}_{\mathcal{STM}^3}$

for each $op : \mathsf{requiredOperations}^3(c) \bullet$

$\mathsf{nproc}(op)(\{x : op.\mathsf{parameters} \bullet x.\mathsf{name}\}) =$

$SStop \triangle (Stop_U \Box o.\mathsf{name}Call.\{x : op.\mathsf{parameters} \bullet x.\mathsf{name}\} \rightarrow Skip)$

within

$$\left(\left(\begin{array}{c}\mathsf{composeMachines}^3(c, ms, cs)^{nops} \\ [\![\mathsf{lvars} \cup \mathsf{rvars} \cup \mathsf{lconsts} \cup \mathsf{rconsts}]\!] \\ \mathsf{ctrlMemory}^2(c)\end{array}\right) \setminus \left(\begin{array}{c}\mathsf{lvars} \cup \mathsf{rvars} \cup \\ \mathsf{lconsts}\end{array}\right)\right) \Theta_{\{end\}} Skip$$

where

$ms = \langle x : c.\mathsf{machines} \rangle$

$cs = c.\mathsf{connections}$

$opdef = \mathsf{findOperationDefinition}(op, c.\mathsf{lOperations})$

$nops = \{op : c.\mathsf{lOperations} \bullet \mathsf{id}(op) \mapsto \mathsf{nproc}(op)\} \cup$

$\{op : \mathsf{requiredOperations}^4(c) \bullet \mathsf{id}(op) \mapsto \mathsf{nproc}(op)\}$

$\mathsf{lvars} = \{\!|v : \mathsf{allLocalVariables}^8(c) \bullet set\_\mathsf{vid}(v)|\!\}$

$\mathsf{rvars} = \{\!|v : \mathsf{requiredVariables}^{12}(c) \bullet set\_Ext\_\mathsf{vid}(v)|\!\}$

$\mathsf{lconsts} = \{\!|v : \mathsf{allLocalConstants}^6(c) \bullet set\_\mathsf{vid}(v)|\!\}$

$\mathsf{rconsts} = \{\!|v : \mathsf{requiredConstants}^9(c) \bullet set\_\mathsf{vid}(v)|\!\}$

4.2.2 State machines

The main difference in the semantics of state machines, and defined operations, is that, in addition to $\mathsf{varMemory}^2$ and $\mathsf{constMemory}^2$ we have in parallel a process clocks^1. There are also changes related to the $\mathsf{transitions}^2$ rule, to account for atomic updates to clock values.

Rule 75. Timed stateful component of state machine behaviour

$\mathsf{stateful}(\mathsf{stm} : \mathsf{StateMachineBody})^{\mathsf{nops}} : \mathsf{TimedCSPProcess} =$

$$
\begin{array}{l}
\textbf{let} \\
\quad Stateful = \left(\left(\begin{array}{l} MachineBody \\ [\![\mathsf{getsetLocalChannels}^3(\mathsf{stm}) \cup \mathsf{clockSync} \cup \{end\}]\!] \\ \left(\begin{array}{l} (\mathsf{varMemory}^3(\mathsf{stm}) \; [\![\{end\}]\!] \; \mathsf{constMemory}^3(\mathsf{stm})) \\ [\![\{end\}]\!] \\ \mathsf{clocks}^2(\mathsf{stm.clocks}) \end{array} \right) \end{array} \right) \right) \\
\quad \setminus \mathsf{getsetLocalChannels}^4(\mathsf{stm}) \cup \mathsf{clockSync} \\
\quad MachineBody = \left(\begin{array}{l} [\![\mathsf{stm}]\!]^{\mathsf{nops}}_{\mathcal{NC}'} \; [\![\{\!| interrupt |\!\}]\!] \; Skip \end{array} \right) \setminus \mathsf{enteredSS} \\
\textbf{within} \\
\quad Stateful \\
\textbf{where} \\
\quad \mathsf{clockSync} = \{\!| \mathsf{c} : \mathsf{stm.clocks} \bullet get\_\mathsf{id}(\mathsf{c}), clockReset.\mathsf{id}(\mathsf{c}) |\!\} \\
\quad \mathsf{enteredSS} = \{\!| \mathsf{x} : \mathsf{stm.nodes} \bullet entered.\mathsf{id}(\mathsf{x}) |\!\}
\end{array}
$$

4.2.3 Node containers

The sematics of node container is changed to account for sinceEntry expressions by defining clocks that count the timed since entering states, with the exception of final states.

Rule 76. Semantics of node container

$[\![\mathsf{nc} : \mathsf{NodeContainer}]\!]^{\mathsf{nops}}_{\mathcal{NC}} : \mathsf{TimedCSPProcess} =$

$$
\begin{array}{l}
\left(\begin{array}{l} \mathsf{ncBehaviour}^2(\mathsf{nc})^{\mathsf{nops}} \\ [\![\mathsf{stateClockSync}(\mathsf{nc}) \cup \{end\}]\!] \\ \mathsf{clocks}^3(\mathsf{stateClocks}(\mathsf{nc})) \end{array} \right) \setminus (\mathsf{stateClockSync}(\mathsf{nc}) \setminus \{\!| entered |\!\}) \\
\textbf{where} \\
\quad \mathsf{stateClocks}(\mathsf{nc}) = \{ \mathsf{s} : \mathsf{nc.nodes} \mid \mathsf{s} \in \mathsf{State} \wedge \mathsf{s} \notin \mathsf{Final} \} \\
\quad \mathsf{stateClockSync}(\mathsf{nc}) = \{\!| \mathsf{n} : \mathsf{stateClocks}(\mathsf{nc}) \bullet entered.\mathsf{id}(\mathsf{n}), get\_\mathsf{id}(\mathsf{n}) |\!\}
\end{array}
$$

The core semantics of node containers is changed to take into account deadlines on transitions.

Rule 77. Node container core behaviour

$$\text{ncCoreBehaviour}(\text{nc} : \text{NodeContainer})^{\text{nops}} : \text{TimedCSPProcess} =$$

$$\left(\begin{array}{l} (\text{composeTimedNodes}^{1}(\text{nc})^{\text{nops}} [\![\text{tidsR}(\text{nc})]\!]) \\ |[\text{vars}^{20}(\text{nc.nodes}) \mid \text{flowevts} \cup \text{transSync}(\text{nc}) \cup \{end\} \mid \text{vtguards}(\text{nc}) : \emptyset]| \\ (enter.\text{initial}(\text{nc.nodes}) \rightarrow \text{transitions}^{3}(\text{nc})^{\text{nops}}) \end{array}\right)$$

where

$$\text{tidsR}(\text{nc}) = \left(\begin{array}{l} \{\!| t : \text{nc.transitions} \bullet interrupt.\text{id}(\text{t.src}) \leftarrow \text{tevent}^{5}(\text{t}) |\!\} \\ \cup \{\!| n : \text{nc.nodes} \mid \text{n} \in \text{State} \bullet interrupt.\text{id}(\text{n}) \leftarrow interrupt.\text{id}(\text{nc}) |\!\} \end{array}\right)$$

$$\text{flowevts} = \{\!| enter, exit, exited, interrupt |\!\}$$

$$\text{transSync}(\text{nc}) = \{\!| \text{t} : \text{nc.transitions} \bullet \text{tevent}^{6}(\text{t}) |\!\}$$

$$\text{vtguards}(\text{nc}) = \{\text{t} : \text{nc.transitions} \mid \exists \text{v} \bullet \text{v} \in \text{usedVariables}^{18}(\text{t.condition}) \bullet \text{v}\}$$

Rule 78. Composition of nodes

$$\text{composeTimedNodes}(\text{nc} : \text{NodeContainer})^{\text{nops}} : \text{TimedCSPProcess} =$$

if $\text{nc.nodes} = \emptyset$

$\quad SStop \triangle end \rightarrow Skip$

else

$\quad |[\{share, end\}]| \ \text{n} : \text{nc.nodes} \bullet \text{nodeD}^{1}(\text{n}, \{\text{t} : \text{nc.transitions} \mid \text{t.source} = \text{n}\})^{\text{nops}}$

Rule 79. Node with deadline

$$\text{nodeD}(\text{nc} : \text{NodeContainer}, \text{ts} : \text{Set}(\text{Transition}))^{\text{nops}} : \text{TimedCSPProcess} =$$

if $\text{n} \in \text{State} \wedge \#\text{tds}(\text{ts}) > 0$

$$\left(\underbrace{[\![\text{n}]\!]^{\text{nops}}}_{\mathcal{N}^{2}} |[\text{vars}(\text{n}) \mid \{\!| end, entered.\text{id}(\text{n}), interrupt.\text{id}(\text{n}) |\!\} \mid \text{vt}(\text{ts}) : \emptyset]| \ \text{stateDeadlines}^{1}(\text{n}, \text{tds}(\text{ts})) \right)$$

else

$$\left(\underbrace{[\![\text{n}]\!]^{\text{nops}}}_{\mathcal{N}^{3}} \right)$$

where

$$\text{tds}(\text{x}) = \{\text{t} : \text{x} \mid \text{t.deadline} \neq \text{null}\}$$

$$\text{vt}(\text{x}) = \left\{ \begin{array}{l} \text{t} : \text{tds}(\text{x}), \text{v} : \text{NamedElement} \mid \\ \text{v} \in \text{usedVariables}^{19}(\text{t.condition}) \cup \text{usedVariables}^{20}(\text{t.deadline}) \bullet \text{v} \end{array} \right\}$$

Rule 80. Transition deadlines of a state

stateDeadlines(n : State, ts : ℙTransitions) : TimedCSPProcess =

$$\text{if } ts = \emptyset$$
$$\left(\begin{array}{l} \textbf{let} \\ \quad P = SStop \triangle \left(end \rightarrow Skip \,\Box\, entered.\underline{id(n)} \rightarrow P \,\Box\, interrupt.\underline{id(n)} \rightarrow P \right) \\ \textbf{within} \\ \quad P \end{array}\right)$$
$$\text{else}$$
$$|[\{entered.\underline{id(n)}, end, interrupt.\underline{id(n)}, share\}]|\ t : \underline{ts} \bullet \underline{\text{transitionDeadline}^1(t)}$$

Rule 81. Transition deadline

transitionDeadline(t : Transition) : TimedCSPProcess =

$$\begin{array}{l} \textbf{let} \\ \quad Inactive = SStop \triangle (Activation \,\Box\, Termination) \\ \quad Termination = end \rightarrow Skip \\ \quad Activation = entered.\underline{id(t.source)} \rightarrow (Active \triangle interrupt.\underline{id(t.source)} \rightarrow Inactive) \\ \quad Active = \left(\begin{array}{l} UntilTrue; \\ \underline{\text{readState}^1}\left(\begin{array}{l} \underline{usedV(t.end)}, \\ \left(UntilFalse \triangle \left(Wait\left(\underline{[\![t.deadline]\!]_{\mathcal{E}xpr}}\right); UStop\right)\right) \end{array}\right); \\ Active \end{array}\right) \\ \quad UntilTrue = \underline{\text{readStateA}^1}\left(\begin{array}{l} \underline{usedV(t.condition)}, \\ \left(\begin{array}{l} \underline{[\![t.condition]\!]_{\mathcal{E}xpr}} \,\&\, Skip \\ \Box \left(not\ \underline{[\![t.condition]\!]_{\mathcal{E}xpr}}\right) \&share \rightarrow UntilTrue \end{array}\right) \triangleright_1 UntilTrue \end{array}\right) \\ \quad UntilFalse = \underline{\text{readStateA}^2}\left(\begin{array}{l} \underline{usedV(t.condition)}, \\ \left(\begin{array}{l} \left(not\ \underline{[\![t.condition]\!]_{\mathcal{E}xpr}}\right) \,\&\, Skip \\ \Box\ \underline{[\![t.condition]\!]_{\mathcal{E}xpr}} \&share \rightarrow UntilFalse \end{array}\right) \triangleright_1 UntilFalse \end{array}\right) \\ \textbf{within} \\ \quad Inactive \end{array}$$

Transitions

The rule $\underline{\text{transitions}^4}$ is changed here to accommodate the passage of time, where via the use of the timeout operator ($\triangleright_1$) the state of conditions is evaluated after the passage of every time unit. This is required to ensure that the values of time elapsed in conditions involving since(C) is updated.

Rule 82. Semantics of Transitions

$\text{transitions}(\text{nc} : \text{NodeContainer})^{\text{nops}} : \text{TimedCSPProcess} =$

let

$$Trans \mathrel{\widehat{=}} \text{readStateA}^2 \left(\begin{array}{l} \text{vtguards}(\text{nc}), \\ \left(\left(\left(\left(\left(\Box\, t : \text{nc.transitions} \bullet [\![\text{t}]\!]_{\mathcal{T}'} \right) \right) \rhd_1 Skip \right) ; Trans \right) \right. \\ \quad \Box\, share \rightarrow Skip \\ \Box \left(\begin{array}{l} interrupt.\text{id}(\text{nc}) \rightarrow \\ \quad (SStop \triangle exit \rightarrow Skip);\ (SStop \triangle exited \rightarrow end \rightarrow Skip) \end{array} \right) \\ \Box\, end \rightarrow Skip \end{array} \right)$$

within

$Trans$

where

$\text{vtguards}(\text{nc}) = \{\text{t} : \text{nc.transitions} \mid \exists \text{v} \bullet \text{v} \in \text{vars}(\text{t.condition}) \bullet \text{v}\}$

Rule 83. Semantics of a Transition

$[\![\text{t} : \text{Transition}]\!]^{\text{nops}}_{\mathcal{T}} : \text{CSPProcess} =$

if $\text{src} \in \text{State} \wedge \text{t.condition} \neq \text{null}$

$$\left(\begin{array}{l} \left([\![\text{t.condition}]\!]_{\mathcal{E}xpr} \right) \,\&\, \text{trigger}^3(\text{t});\ \text{clockResets}^1(\text{t});\ (SStop \triangle exit \rightarrow Skip); \\ (SStop \triangle (exited \rightarrow \underbrace{[\![\text{t.action}]\!]^{\text{nops}}}_{StatementInContext^{10}} ;\ enter.\text{id}(\text{tgt}) \rightarrow Skip)) \end{array} \right)$$

else if $\text{src} \in \text{State} \wedge \text{t.condition} = \text{null}$

$$\left(\begin{array}{l} \text{trigger}^4(\text{t});\ \text{clockResets}^2(\text{t});\ (SStop \triangle exit \rightarrow Skip); \\ (SStop \triangle (exited \rightarrow \underbrace{[\![\text{t.action}]\!]^{\text{nops}}}_{StatementInContext^{11}} ;\ enter.\text{id}(\text{tgt}) \rightarrow Skip)) \end{array} \right)$$

else if $\text{src} \notin \text{State} \wedge \text{t.condition} \neq \text{null}$

$$\left([\![\text{t.condition}]\!]_{\mathcal{E}xpr} \right) \,\&\, internal.\text{id}(\text{src}) \rightarrow \underbrace{[\![\text{t.action}]\!]^{\text{nops}}}_{StatementInContext^{12}} ;\ enter.\text{id}(\text{tgt}) \rightarrow Skip$$

else if $\text{src} \notin \text{State} \wedge \text{t.condition} = \text{null}$

$$internal.\text{id}(\text{src}) \rightarrow \underbrace{[\![\text{t.action}]\!]^{\text{nops}}}_{StatementInContext^{13}} ;\ enter.\text{id}(\text{tgt}) \rightarrow Skip$$

where

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

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

Rule 84. Semantics of clock resets for transitions

$\text{clockResets}(\text{t} : \text{Transition}) : \text{TimedCSPProcess} =$

if $\#\text{t.reset} > 0$

$[\![\{share\}]\!]\ \text{c} : \text{t.reset} \bullet SStop \triangle clockReset.\text{id}(\text{c.clock}) \rightarrow (SStop \triangle Skip)$

else

$SStop \triangle Skip$

4.2.4 Clocks

Rule 85. Clocks

clocks(cs : ℙ NamedElement) : TimedCSPProcess =

if #cs = 0
  $end \rightarrow Skip$
else
  $[\![\{end\}]\!]$ c : cs • clock^1(c)

Rule 86. Clock

clock(c : NamedElement) : TimedCSPProcess =

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

The function *clock* is defined over the type NamedElement, for elements of type State and Clock. Increments happen every time unit, as defined using the timeout operator $\triangleright_1$.

Rule 87. Clock

clock(c : Clock) : TimedCSPProcess =

let

$$Memory(x) = \left(\begin{array}{l} clockReset.\mathrm{id}(c) \rightarrow Memory(0) \\ \Box\ get\_\mathrm{id}(c)!x \rightarrow Memory(x) \\ \Box\ end \rightarrow Skip \end{array} \right) \triangleright_1 Memory(x+1)$$

within
  $Memory(0)$

Rule 88. Clock

clock(s : State) : TimedCSPProcess =

let

$$Memory(x) = \left(\begin{array}{l} entered.\mathrm{id}(s) \rightarrow Memory(0) \\ \Box\ get\_\mathrm{id}(s)!x \rightarrow Memory(x) \\ \Box\ end \rightarrow Skip \end{array} \right) \triangleright_1 Memory(x+1)$$

within
  $Memory(0)$

4.2.5 Statements

Rule 89. Read state of an expression

readState(vs : seq(Variable), P : CSPProcess) : TimedCSPProcess =

  if (#vs = 0) then
    P
  else
    $SStop \triangle (Stop_U \Box get_$vid(head vs)?(head vs).name $\rightarrow$ readState^2(tail vs, P))

Rule 90. Atomic read state of an expression

readStateAtomic(vs : seq(Variable), P : CSPProcess) : TimedCSPProcess =

if (#vs = 0) then
P
else
$(Stop_U \Box get\_\text{vid}(\text{head vs})?(\text{head vs}).\text{name} \rightarrow \text{readStateAtomic}^1(\text{tail vs}, \text{P}))$

Rule 91. Atomic read state of an expression with initial share

readStateA(vs : seq(Variable), P : CSPProcess) : TimedCSPProcess =

if (#vs = 0) then
P
else
$SStop \triangle (Stop_U \Box get\_\text{vid}(\text{head vs})?(\text{head vs}).\text{name} \rightarrow \text{readStateAtomic}^2(\text{tail vs}, \text{P}))$

Rule 92. Semantics of assignment

$[\![\text{s : Assignment}]\!]^{\text{nops}}_{\mathcal{S}tatement}$: TimedCSPProcess =

$SStop \triangle (Stop_U \Box set\_\text{vid}(\text{s.left})![\![\text{s.right}]\!]_{\mathcal{E}xpr} \rightarrow Skip)$

Rule 93. Semantics of statements

$[\![\text{s : Statement}]\!]^{\text{nops}}_{\mathcal{S}tatement}$: TimedCSPProcess =

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

Rule 94. Semantics of statement deadlines

$[\![\text{s : TimedStatement}]\!]^{\text{nops}}_{\mathcal{S}tatement}$: TimedCSPProcess =

$[\![\text{s.stmt}]\!]^{\text{nops}}_{\mathcal{S}tatement^2} \triangle (Wait([\![\text{s.deadline}]\!]_{Expr});\ Stop_U)$

Rule 95. Semantics of wait statement

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

$[\![[\![\text{s.duration}]\!]_{\mathcal{E}xpr}]\!]_{\mathcal{W}ait^1}$

Rule 96. Semantics of Wait

$[\![\text{e : Expression}]\!]_{\mathcal{W}ait}$: TimedCSPProcess =

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

Rule 97. Semantics of Wait

$[\![\mathsf{e} : \mathbb{P}\,\tau]\!]_{\mathcal{W}ait} : \mathsf{TimedCSPProcess} =$

$$SStop \triangle \left(\sqcap n : \underline{\mathsf{e}} \bullet Wait(n)\right)$$

Rule 98. Semantics of Wait

$[\![\mathsf{e} : \mathbb{A}]\!]_{\mathcal{W}ait} : \mathsf{TimedCSPProcess} =$

$$SStop \triangle Wait(\underline{\mathsf{e}})$$

Rule 99. Semantics of clock reset

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

$$SStop \triangle \left(clockReset.\underline{\mathsf{id}(\mathsf{s.clock})} \rightarrow Skip\right)$$

Rule 100. Semantics of deadline over statements

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

$$\underline{\mathsf{readState}^3}\left(\begin{array}{l}\underline{\mathsf{usedVariables}^{21}(\mathsf{s.deadline})},\\ \underline{[\![\mathsf{s.stmt}]\!]^{\mathsf{nops}}_{\mathcal{S}tatement^3}} \triangle \left(Wait\left(\underline{[\![\mathsf{s.deadline}]\!]_{\mathcal{E}xpr}}\right) ; UStop\right)\end{array}\right)$$

Rule 101. Semantics of clock reset

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

$$SStop \triangle clockReset.\underline{\mathsf{id}(\mathsf{s.clock})} \rightarrow Skip$$

5. Probabilistic Semantics

This chapter describes and defines the probabilistic semantics of RoboChart. We give an overview of probabilistic semantics in Section 5.1, and present its UTP semantics in Section 5.2. Then in Section 5.3, we define a translation from RoboChart to PRISM [19] (the language supported by the probabilistic model checker PRISM) in order to use PRISM to analyse RoboChart models.

5.1 Overview

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

The semantics for the probabilistic language of RoboChart is also based on UTP, and so RoboChart has a unified semantic base. Ultimately, we aim to support its verification using the theorem prover Isabelle/UTP [27], a mechanisation of UTP in the Isabelle/HOL theorem prover [32].

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

5.2 Semantic Domain

The probabilistic denotational semantics for RoboChart [33] is based on the weakest completion semantics [14] in UTP. The general idea is to use a forgetful function to embed standard designs into probabilistic designs. This embedding preserves program structure of a probabilistic programming language which includes abort, skip, assignment, conditional, nondeterministic choice, probabilistic choice, sequence, and recursion constructors. We are mechanising the UTP theory for probabilistic designs in Isabelle/UTP. The next step is to lift probabilistic designs into UTP's reactive design to develop a theory of reactive probabilistic designs.

5.3 Translation to PRISM

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

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

5.3.1 Translation requirements

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

TR-TP1 *Time primitives are not used.* In DTMCs and MDPs, transitions occur in discrete-time steps. Every transition takes one unit of time. Time primitives [23] in RoboChart, however, are capable of capturing budgets and deadlines using clocks and constructs like wait(n), which defines a waiting period of n units of time, and read?x<{2}, in which reading a value x through an event read has a deadline to take place within 2 units of time. DTMCs and MDPs do not intrinsically support clocks, time budgets and deadlines to model the corresponding RoboChart constructs.
In addition to DTMCs and MDPs, the PRISM notation also supports probabilistic timed automata (PTAs) [17, 21], which extend MDPs with the ability to model real-time behaviour through real-valued clocks [20]. The timed semantics of RoboChart, however, are based on time units [23], instead of real-valued time. The default verification method (quantitative abstraction refinement [18]) for PTAs in PRISM, therefore, cannot be used. Instead, we can use the digital clocks [22] method which uses an *integral* time model. We will extend our transformation to support PTAs in order for time primitives in RoboChart, which is part of our future work.

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

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

TR-ST1 *States cannot have* during *actions.*

To cater for these constructs, the PRISM model needs to include extra modules to deal with shared variables, buffers for asynchronous communication, operation calls, and interruptions of during actions by outgoing transitions. Dealing with these extra constructs is part of our agenda for future work.

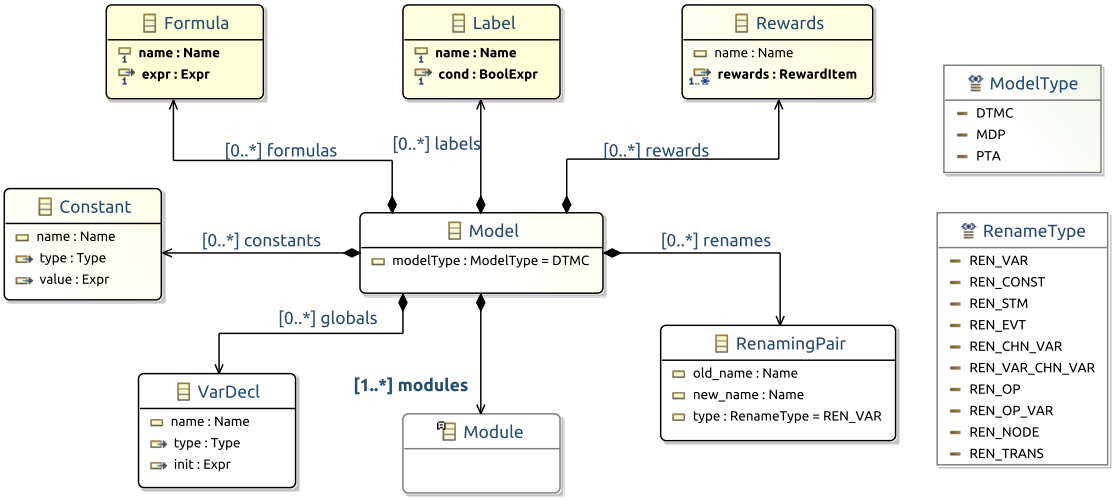

Figure 5.1: Metamodel of a PRISM model

TR-TY1 *Only primitive types and enumerations, sequences of these types, or sequences of sequences of these types are used.*

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

PRISM supports only integer, boolean, and real numbers. Refinement techniques [2, 16] are a possible solution to deal with abstract data types and constructs in RoboChart. Support of more abstract data types and expressions is part of our plans for future work.

5.3.2 Normal form

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

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

5.3.3 Notions for rules

See Table 5.1.

5.3.4 PRISM

Metamodel

See Figure 5.1 to 5.8.

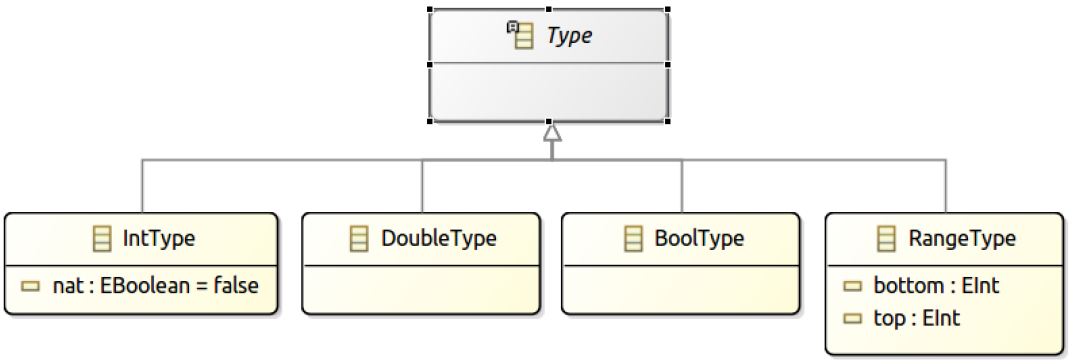

Figure 5.2: Metamodel of PRISM types

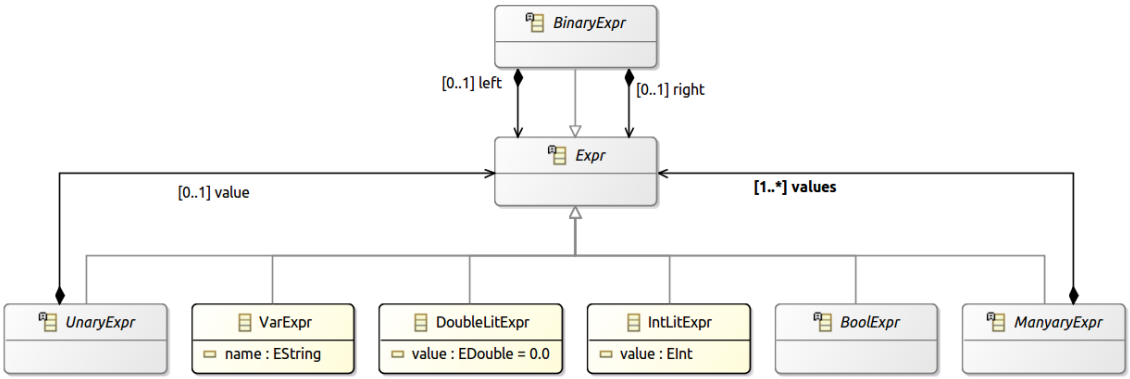

Figure 5.3: Metamodel of PRISM expressions

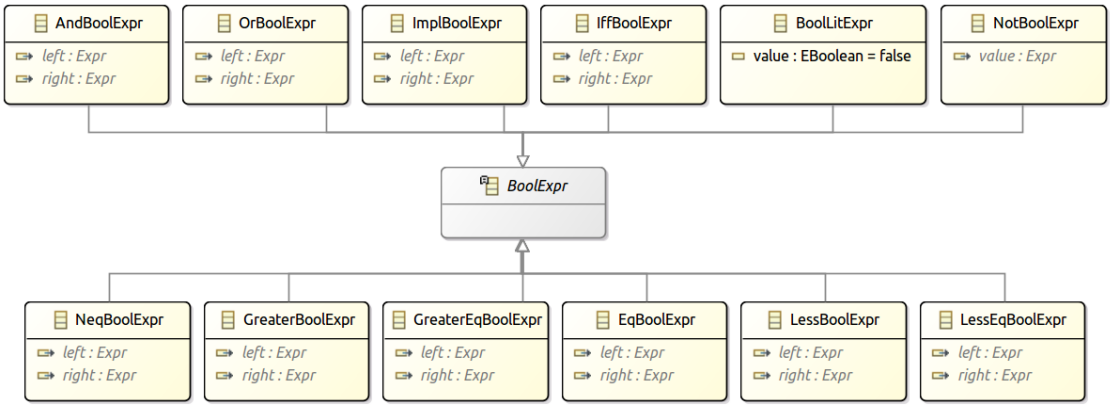

Figure 5.4: Metamodel of PRISM boolean expressions

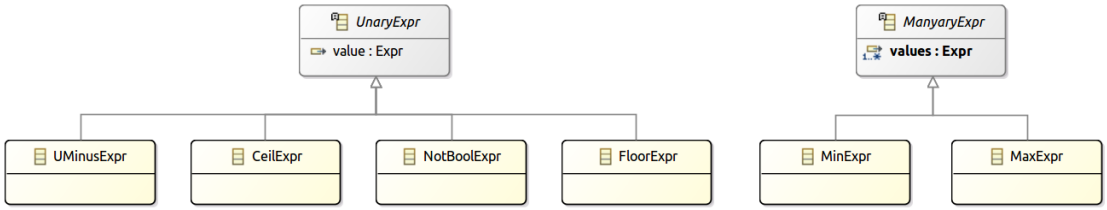

Figure 5.5: Metamodel of PRISM unary and manyary expressions

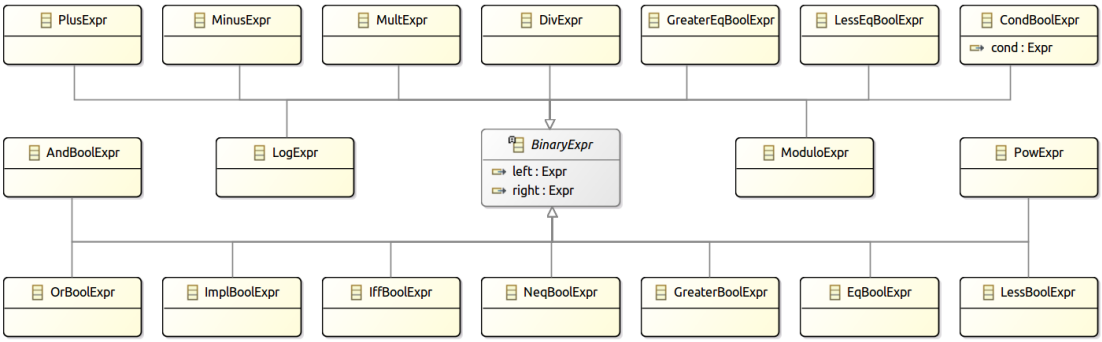

Figure 5.6: Metamodel of PRISM binary expressions

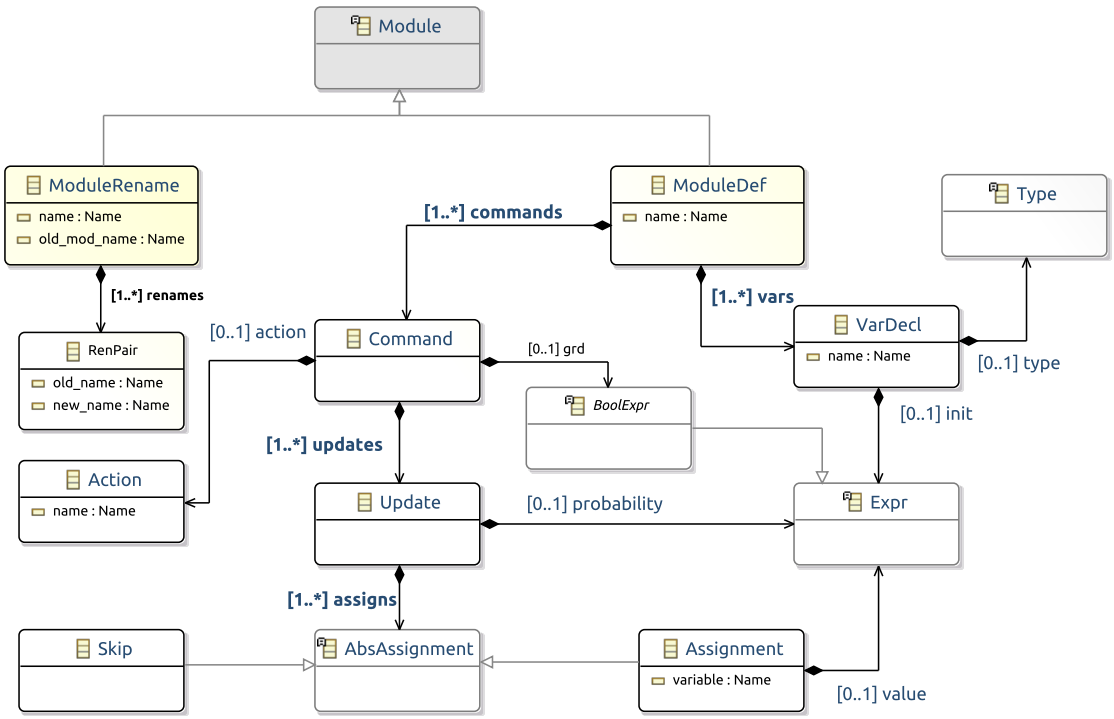

Figure 5.7: Metamodel of PRISM modules

Table 5.1: Summary of construct syntax in rules

Form	Description
Junction	The class name from the metamodel also represents a collection of objects that have the Junction type. Similarly, State, ProbabilisticJunction, Controller, and so on. Particularly, ProbJunc is an abbreviation for ProbabilisticJunction for a compact space.
$\mathsf{Class}_{\mathsf{rc}}$ $\mathsf{Class}_{\mathsf{pr}}$	Subscripts indicate classes of different metamodels, where *rc* stands for RoboChart, and *pr* for PRISM. For example, $\mathsf{Transition}_{\mathsf{rc}}$ and $\mathsf{ModuleDef}_{\mathsf{pr}}$ denote the Transition in RoboChart, and the ModuleDef in PRISM.
$\mathbb{P}X$	Power set of X.
$X \times Y$	Cartesian product.
$\mathrm{seq}\, X$	Finite sequences.
$\{x : T \mid P \bullet e(x)\}$	Set comprehension.
$\{x : T \mid P\}$	Defined as $\{x : T \mid P \bullet x\}$.
$\{x : T \bullet e\}$	Defined as $\{x : T \mid \mathit{true} \bullet e\}$.
$(\mu\, x : T \mid P)$	Definite description.
$\bigcup\{x : T \bullet e(x)\}$	Generalised union, where $e(x)$ is a set expression.
$p.n$	Selection of nth element from the tuple p.
$obj.v$	Selection of value of component v from object obj.
$\lfloor f_1 \rightsquigarrow v_1, \cdots \rfloor_C$	Record where f_i is a component name and v_i is the value associated to that component. It represents an object of the class C with its components instantiated. If a component is not specified, it is set to null or $\emptyset$ (if it is a set) by default.
$R(\!\mid S \mid\!)$	Relational image of a relation R under a set S.
$[\![i : T]\!]_{\mathcal{X}}$	Transformation of i to PRISM in the context $\mathcal{X}$. For example, $[\![\cdots]\!]_{\mathcal{M}}$ denotes the module context.
$\mathit{uname}(\mathit{par}, m)$	This function constructs a fresh unique identifier for a new element from the supplied construct *par* (of type NamedElement) and the name m (a string) of the element.
$\mathit{uname}(\mathit{par}, m, n)$	Similar to $\mathit{uname}(\mathit{par}, m)$, but this function has three parameters where both m and n are of type string.
$\mathit{id}(n)$	This function defines a unique identifier for an existing construct n (of type NamedElement). If n is null, it gives an empty name. One possible implementation is to use qualified names.
$\mathit{id}(e, \mathit{out})$	This function defines a unique identifier for an existing event e (of type Event) when considering it for input ($\mathit{out} = \mathit{false}$) or output ($\mathit{out} = \mathit{true}$). If e is null, it gives an empty name. One possible implementation is to use qualified names with a suffix "IN" or "OUT".
$\mathit{fname}(p_1 : T_1, \cdots) : p_n : T_n \rightarrow T_r$	This is a function declaration with the last parameter p_n of type T_n and the result of type T_r. It is commonly used to create a curried version of the function, an application of the function having all arguments except the last one.

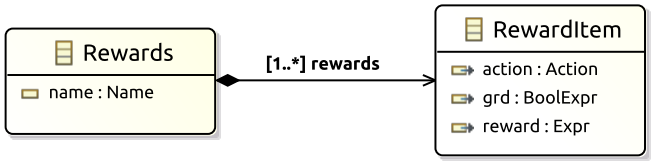

Figure 5.8: Metamodel of PRISM rewards

5.3.5 Normalisation rules

Rule 102. Normalisation of state machines

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

$$\left\lfloor \begin{array}{l} \text{name} \rightsquigarrow \text{stm.name}, \text{variableList} \rightsquigarrow \text{stm.variableList}, \\ \text{operations} \rightsquigarrow \text{stm.operations}, \text{events} \rightsquigarrow \text{stm.events}, \\ \text{pInterfaces} \rightsquigarrow \text{stm.pInterfaces}, \text{rInterfaces} \rightsquigarrow \text{stm.rInterfaces}, \\ \text{interfaces} \rightsquigarrow \text{stm.interfaces}, \text{transitions} \rightsquigarrow \text{transnodes.1}, \\ \text{nodes} \rightsquigarrow \text{transnodes.2} \end{array} \right\rfloor_{\text{StateMachineDef}}$$

where

$\text{transnodes} = [\![\text{stm}]\!]_{\mathcal{CS}'}$

Rule 103. Normalisation of states

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

$$\left\lfloor \text{name} \rightsquigarrow \text{s.name}, \text{actions} \rightsquigarrow \text{s.actions}, \text{transitions} \rightsquigarrow \text{transnodes.1}, \text{nodes} \rightsquigarrow \text{transnodes.2} \right\rfloor_{\text{State}}$$

where

$\text{transnodes} = [\![\text{s}]\!]_{\mathcal{CS}^2}$

Rule 104. Normalisation of NodeContainer

$[\![\text{s} : \text{NodeContainer}]\!]_{\mathcal{CS}} : \mathbb{P}\,\text{Transition} \times \mathbb{P}\,\text{Node} =$

$(\bigcup\{\text{tp} : \text{transpjunc} \bullet \text{tp.1}\}, (\text{loopsttr.1} \cup (\bigcup\{\text{tp} : \text{transpjunc} \bullet \text{tp.2}\}) \cup \text{compstates} \cup \text{intactnodes}))$

where

$\text{loopsttr} = \text{addLoopStateTrans}^1(\text{s})$

$$\text{junctrans} = \text{combTransJunctions}^1\left(\begin{array}{l} \text{inTransCombinableJuncs}^1(\text{s}), \\ \bigcup\left(\begin{array}{l} \text{transitionsOf}^1(\text{s.transitions}) \\ \quad (\!|\ \text{inTransCombinableJuncs}^2(\text{s})\ |\!) \end{array}\right) \end{array}\right)$$

$\text{transpjunc} = \{\text{t} : \text{s.transitions} \cup \text{junctrans.3} \cup \text{loopsttr.2} \setminus \text{junctrans.2} \bullet \text{splitTran}^1(\text{t})\}$

$\text{compstates} = \{\text{n} : \text{s.nodes} \mid \text{n} \in \text{State} \wedge \#(\text{n.nodes}) > 0 \bullet [\![\text{n}]\!]_{\mathcal{S}'}\}$

$\text{intactnodes} = \{\text{n} : \text{s.nodes} \mid \text{n} \notin \text{junctrans.1} \wedge \neg\,(\text{n} \in \text{State} \wedge \#(\text{n.nodes}) > 0)\}$

Rule 105. Normalisation of states without outgoing transitions

addLoopStateTrans (stm : NodeContainer) : $\mathbb{P}$ State $\times$ $\mathbb{P}$ Transition =

if $\exists$ n : stm.nodes $\bullet$ n $\in$ State $\wedge$ ($\forall$ t : stm.transitions $\bullet$ t.source $\neq$ n) **then**
({loopstate}, {looptran} $\cup$ newtrans)
else
($\emptyset$, $\emptyset$)
where
loopstate = newState^1 (uname(stm, "loop"))
looptran = newTransition^1 (uname(stm, "loop_self"), loopstate, loopstate)

$$\text{newtrans} = \bigcup \left\{ n : \text{stm.nodes} \;\middle|\; \begin{array}{l} n \in \text{State} \wedge (\forall t : \text{stm.transitions} \bullet t.\text{source} \neq n) \\ \bullet \left\{ \text{newTransition}^2 (\text{uname}(n, \text{"to\_loop"}), n, \text{loopstate}) \right\} \end{array} \right\}$$

Rule 106. Normal junctions whose incoming transitions are combinable

inTransCombinableJuncs (stm : NodeContainer) : $\mathbb{P}$ Junction =

$$\left\{ n : \text{stm.nodes} \;\middle|\; \left(\begin{array}{l} n \in \text{Junction} \wedge n \notin \text{Initial} \wedge n \notin \text{ProbJunc} \wedge \\ \exists t : \text{stm.transitions} \bullet t.\text{target} = n \wedge \text{isCombinableTran}^1 (t) \end{array} \right) \right\}$$

Rule 107. Combinable transition

isCombinableTran (t : Transition) : boolean =

t.action = null $\wedge$ t.source $\notin$ Initial $\wedge$ t.source $\notin$ ProbJunc

Rule 108. Incoming and outgoing transitions of a node

transitionsOf (trans : $\mathbb{P}$ Transition) : n : Node $\rightarrow$ $\mathbb{P}$ Transition =

{t : trans | t.source = n $\vee$ t.target = n}

Rule 109. Combination of transitions of junctions

combTransJunctions (juncs : $\mathbb{P}$ Junction, trans : $\mathbb{P}$ Transition) : $\mathbb{P}$ Junction $\times$ $\mathbb{P}$ Transition $\times$ $\mathbb{P}$ Transition =

if juncs = $\emptyset$ **then** ($\emptyset$, $\emptyset$, $\emptyset$) **else** (jtrans.1 $\cup$ ojstrans.1, jtrans.1 $\cup$ ojstrans.2, jtrans.3 $\cup$ ojstrans.3)
where
j $\in$ juncs

$$\text{jtrans} = \text{combTransJunction}^1 \left(j, \text{transitionsOf}^2 (\text{trans}, j) \right)$$

$$\text{ojstrans} = \text{combTransJunctions}^2 \left(\begin{array}{l} \text{juncs} \setminus \{j\}, \\ \bigcup \left(\text{transitionsOf}^3 ((\text{trans} \setminus \text{jtrans.2}) \cup \text{jtrans.3}) \llparenthesis \text{juncs} \setminus \{j\} \rrparenthesis \right) \end{array} \right)$$

Rule 110. Combination of transitions of a junction

$$\mathsf{combTransJunction}\,(j : \mathsf{Junction}, \mathsf{trans} : \mathbb{P}\,\mathsf{Transition}) : \mathbb{P}\,\mathsf{Junction} \times \mathbb{P}\,\mathsf{Transition} \times \mathbb{P}\,\mathsf{Transition} =$$

$$\textbf{if}\ \mathsf{intransother} \neq \emptyset\ \textbf{then}\ (\emptyset, \mathsf{intranscomb}, \mathsf{newtrans})\ \textbf{else}\ (\{j\}, \mathsf{trans}, \mathsf{newtrans})$$

where

$$\mathsf{intranscomb} = \{t : \mathsf{trans} \mid t.\mathsf{target} = j \wedge \mathsf{isCombinableTran}^2(t)\}$$

$$\mathsf{intransother} = \{t : \mathsf{trans} \mid t.\mathsf{target} = j\} \setminus \mathsf{intranscomb}$$

$$\mathsf{outtrans} = \{t : \mathsf{trans} \mid t.\mathsf{source} = j\}$$

$$\mathsf{newtrans} = \left\{ \begin{array}{l} ti : \mathsf{intranscomb};\ tj : \mathsf{outtrans} \bullet \\ \left\lfloor \begin{array}{l} \mathsf{name} \rightsquigarrow \mathsf{uname}(ti, tj.\mathsf{name}), \mathsf{source} \rightsquigarrow ti.\mathsf{source}, \\ \mathsf{target} \rightsquigarrow tj.\mathsf{target}, \mathsf{trigger} \rightsquigarrow ti.\mathsf{trigger}, \\ \mathsf{condition} \rightsquigarrow ti.\mathsf{condition} \wedge tj.\mathsf{condition}, \mathsf{action} \rightsquigarrow tj.\mathsf{action} \end{array} \right\rfloor_{\mathsf{Transition}} \end{array} \right\}$$

Rule 111. Transition split

$$\mathsf{splitTran}\,(t : \mathsf{Transition}) : \mathbb{P}\,\mathsf{Transition} \times \mathbb{P}\,\mathsf{ProbJunc} =$$

$$\textbf{if}\ t.\mathsf{trigger} \neq \mathsf{null} \wedge t.\mathsf{action} \neq \mathsf{null}\ \textbf{then}$$

$$\left(\left\{ \begin{array}{l} \left\lfloor \begin{array}{l} \mathsf{name} \rightsquigarrow \mathsf{uname}(t, \text{“sp\_1”}), \mathsf{source} \rightsquigarrow t.\mathsf{source}, \mathsf{target} \rightsquigarrow \mathsf{pj}, \\ \mathsf{trigger} \rightsquigarrow t.\mathsf{trigger}, \mathsf{condition} \rightsquigarrow t.\mathsf{condition} \end{array} \right\rfloor_{\mathsf{Transition}}, \\ \left\lfloor \begin{array}{l} \mathsf{name} \rightsquigarrow \mathsf{uname}(t, \text{“sp\_2”}), \mathsf{source} \rightsquigarrow \mathsf{pj}, \mathsf{target} \rightsquigarrow t.\mathsf{target}, \\ \mathsf{probability} \rightsquigarrow 1.0, \mathsf{action} = t.\mathsf{action} \end{array} \right\rfloor_{\mathsf{Transition}} \end{array} \right\}, \{\mathsf{pj}\} \right)$$

else

$$(\{t\}, \emptyset)$$

where

$$\mathsf{pj} = \lfloor \mathsf{name} \rightsquigarrow \mathsf{uname}(t, \text{“sp\_pj”}) \rfloor_{\mathsf{ProbJunc}}$$

Rule 112. New State

$$\mathsf{newState}\,(n : \mathsf{String}) : \mathsf{State} =$$

$$\lfloor \mathsf{name} \rightsquigarrow n, \mathsf{nodes} \rightsquigarrow \emptyset, \mathsf{transitions} \rightsquigarrow \emptyset, \mathsf{actions} \rightsquigarrow \emptyset \rfloor_{\mathsf{State}}$$

Rule 113. New Transition

$$\mathsf{newTransition}\,(n : \mathsf{String}, \mathsf{source} : \mathsf{Node}, \mathsf{target} : \mathsf{Node}) : \mathsf{Transition} =$$

$$\left\lfloor \begin{array}{l} \mathit{name} \rightsquigarrow n, \mathit{source} \rightsquigarrow \mathit{source}, \mathit{target} \rightsquigarrow \mathit{target}, \mathit{trigger} \rightsquigarrow \mathit{null}, \\ \mathit{condition} \rightsquigarrow \mathit{null}, \mathit{probability} \rightsquigarrow \mathit{null}, \mathit{action} \rightsquigarrow \mathit{null} \end{array} \right\rfloor_{\mathsf{Transition}}$$

5.3.6 Normal form to PRISM

Module

MODEL_TYPE is a parameter to this translation and can have two values: “DTMC” and “MDP”;

Rule 114. Module

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

$$\left[\begin{array}{l} \mathsf{modelType} \rightsquigarrow (\textbf{if } \mathsf{MODEL\_TYPE} = \text{“MDP”} \textbf{ then } \text{“MDP”} \textbf{ else } \text{“DTMC”}) \\ \mathsf{constants} \rightsquigarrow (\bigcup\{r : \mathsf{ctrlrets} \bullet r.1\}) \cup \mathsf{rprets}.1 \cup \mathsf{exitconsts}, \\ \mathsf{globals} \rightsquigarrow (\bigcup\{r : \mathsf{ctrlrets} \bullet r.2\}) \cup \mathsf{rprets}.2, \\ \mathsf{modules} \rightsquigarrow (\bigcup\{r : \mathsf{ctrlrets} \bullet r.3\}) \cup \{\mathsf{rprets}.3\} \cup \mathsf{bufmodules} \end{array}\right]_{\mathsf{Model}_{pr}}$$

where

$\mathsf{namemap} = \mathsf{renameEventsConns}^1(m, m.\mathsf{connections}, \emptyset)$

$\mathsf{bufmodules} = \mathsf{asyncConnectedEvents}^1(m, m.\mathsf{connections}, \mathsf{namemap})$

$\mathsf{ctrlrets} = [\![\_, \mathsf{namemap}]\!]_{\mathcal{C}'} (\!| \{n : m.\mathsf{nodes} \mid n \in \mathsf{Controller}\} |\!)$

$\mathsf{rpoutevents} = \{c : m.\mathsf{connections} \mid c.\mathsf{from} \in \mathsf{RoboticPlatform} \wedge c.\mathsf{efrom}.\mathsf{type} \neq \mathsf{null} \bullet c.\mathsf{efrom}\}$

$\mathsf{rprets} = [\![(\mu n : m.\mathsf{nodes} \mid n \in \mathsf{RoboticPlatform}), \mathsf{rpoutevents}, (\bigcup\{r : \mathsf{ctrlrets} \bullet r.4\}), \mathsf{namemap}]\!]_{\mathcal{R}'}$

$\mathsf{exitconsts} = \mathsf{exitSeqCtrlConsts}^1()$

Rule 115. Exit sequence control constants

$\mathsf{exitSeqCtrlConsts}() : \mathbb{P}\,\mathsf{Constant} =$

$$\{\ \mathsf{inv}, \mathsf{act\_p}, \mathsf{act\_t}, \mathsf{exited}, \mathsf{sub\_act}, \mathsf{sub\_waiting}, \mathsf{sub\_exited}\ \}$$

where

$\mathsf{inv} =$ `const int Exit_NONE = 0;`

$\mathsf{act\_p} =$ `const int Exit_ACT_Parent = 1;`

$\mathsf{act\_t} =$ `const int Exit_ACT_Trans = 2;`

$\mathsf{exited} =$ `const int Exit_EXITED = 3;`

$\mathsf{sub\_act} =$ `const int Exit_Sub_ACT = 4;`

$\mathsf{sub\_waiting} =$ `const int Exit_Sub_ACT_Waiting = 5;`

$\mathsf{sub\_exited} =$ `const int Exit_Sub_EXITED = 6;`

Robotic platforms

Rule 116. Robotic platforms

$$\left[\!\!\left[\begin{array}{l} rp : \mathsf{RoboticPlatformDef}, \mathsf{outevents} : \mathbb{P}\,\mathsf{Event}, \mathsf{opmaps} : \mathsf{OperationSig} \leftrightarrow \mathsf{Action}_{pr}, \\ \mathsf{namemap} : (\mathsf{Event}_{rc} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\!\right]_{\mathcal{R}}$$

$$: \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathsf{Module} =$$

$$(\bigcup\{r : \mathsf{constvars} \bullet r.1\}, \bigcup\{r : \mathsf{constvars} \bullet r.2\}, \mathsf{module})$$

where

$\mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}'} (\!| \mathsf{getVariableLists}^1(rp) |\!)$

$\mathsf{localvars} = \{e : \mathsf{outevents} \bullet \texttt{EVT\_}\_\mathsf{namemap}(e \mapsto \mathsf{true}) : [\![e.\mathsf{type}]\!]_t\ ;\ \}$

$$\mathsf{eventcmds} = \left\{\begin{array}{l} e : \mathsf{getEvents}^1(rp) \bullet \\ \left(\begin{array}{l} \textbf{if } e \in \mathsf{outevents} \textbf{ then} \\ \left\{\begin{array}{l} v : [\![e.\mathsf{type}]\!]_t \bullet \\ [\mathsf{namemap}(e \mapsto \mathsf{true})]\,\texttt{true} \rightarrow \texttt{EVT\_}\_\mathsf{namemap}(e \mapsto \mathsf{true})' = v\ ; \end{array}\right\} \\ \textbf{else } \{\ [\mathsf{namemap}(e \mapsto \mathsf{false})]\,\texttt{true} \rightarrow \texttt{true}\ ;\ \} \end{array}\right) \end{array}\right\}$$

$\mathsf{opcmds} = \{act : \mathrm{ran}\ \mathsf{opmaps} \bullet [act.\mathsf{name}]\,\texttt{true} \rightarrow \texttt{true}\ ;\ \}$

$\mathsf{module} =$ `module` $\mathsf{id}(rp), \mathsf{vars} \rightsquigarrow \mathsf{localvars}, \mathsf{commands} \rightsquigarrow (\bigcup \mathsf{eventcmds}) \cup \mathsf{opcmds}$ `endmodule`

Controllers

Rule 117. Controllers

$$
\begin{array}{l}
[\![\mathsf{ctrl} : \mathsf{ControllerDef}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}]\!]_{\mathcal{C}} \\
\quad : \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathbb{P}\,\mathsf{Module} \times (\mathsf{OperationSig} \leftrightarrow \mathsf{Action}) = \\
\hline
\left(\begin{array}{l} \bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.1}\} \cup (\bigcup\{\mathsf{r} : \mathsf{stmrets} \bullet \mathsf{r.1}\}), \bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.2}\}, \\ \{\mathsf{r} : \mathsf{stmrets} \bullet \mathsf{r.2}\} \cup \mathsf{bufmodules}, \mathsf{rops} \end{array} \right) \\
\hline
\underline{\textbf{where}} \\
\quad \mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}^2} \left(\!\!\left(\mathsf{getVariableLists}^2(\mathsf{ctrl}) \right)\!\!\right) \\
\quad \mathsf{newnamemap} = \mathsf{renameEventsConns}^2(\mathsf{ctrl}, \mathsf{ctrl.connections}, \mathsf{namemap}) \\
\quad \mathsf{bufmodules} = \mathsf{asyncConnectedEvents}^2(\mathsf{ctrl}, \mathsf{ctrl.connections}, \mathsf{newnamemap}) \\
\quad \mathsf{allops} = \mathsf{getRequiredOperations}^1(\mathsf{ctrl}) \cup \left\{ \mathsf{op} : \mathsf{ctrl.lOperations} \bullet \left(\begin{array}{l} \textbf{if}\ \mathsf{op} \in \mathsf{OperationDef} \\ \textbf{then}\ \mathsf{op}\ \textbf{else}\ \mathsf{op.ref} \end{array} \right) \right\} \\
\quad \mathsf{stmrets} = \left\{ \begin{array}{l} \mathsf{stm} : \mathsf{ctrl.machines} \bullet \\ \textbf{let}\ \mathsf{stmoutevents} == \\ \left\{ \mathsf{c} : \mathsf{ctrl.connections} \;\middle|\; \left(\begin{array}{l} (\mathsf{c.bidirec} \wedge \mathsf{c.to} = \mathsf{stm}) \\ \vee\ \mathsf{c.from} = \mathsf{stm} \end{array} \right) \wedge \mathsf{c.efrom.type} \neq \mathsf{null} \bullet \left(\begin{array}{l} \textbf{if}\ \mathsf{c.from} = \mathsf{stm} \\ \textbf{then}\ \mathsf{c.efrom} \\ \textbf{else}\ \mathsf{c.eto} \end{array} \right) \right\} \\ \mathsf{stminevents} == \\ \left\{ \mathsf{c} : \mathsf{ctrl.connections} \;\middle|\; \left(\begin{array}{l} (\mathsf{c.bidirec} \wedge \mathsf{c.from} = \mathsf{stm}) \\ \vee\ \mathsf{c.to} = \mathsf{stm} \end{array} \right) \wedge \mathsf{c.efrom.type} \neq \mathsf{null} \bullet \left(\begin{array}{l} \textbf{if}\ \mathsf{c.to} = \mathsf{stm} \\ \textbf{then}\ \mathsf{c.eto} \\ \textbf{else}\ \mathsf{c.efrom} \end{array} \right) \right\} \\ \bullet\ [\![\mathsf{stm}, \mathsf{stmoutevents}, \mathsf{stminevents}, \mathsf{allops}, \mathsf{newnamemap}]\!]_{\mathcal{STM}'} \end{array} \right\} \\
\hline
\quad \mathsf{rops} = \bigcup \left\{ \mathsf{stm} : \mathsf{ctrl.machines} \bullet \mathsf{opToActionMaps}^1(\mathsf{stm}) \right\}
\end{array}
$$

Rule 118. Mapping from operations required by state machines to PRISM actions

$$
\begin{array}{l}
\mathsf{opToActionMaps}(\mathsf{stm} : \mathsf{StateMachineDef}) : \mathsf{OperationSig} \leftrightarrow \mathsf{Action} = \\
\hline
\left\{ \mathsf{op} : \mathsf{getRequiredOperations}^2(\mathsf{stm}) \bullet \mathsf{op} \mapsto \lfloor \mathsf{name} \rightsquigarrow \mathsf{uname}(\mathsf{stm}, \mathsf{op.name}) \rfloor_{\mathsf{Action}_{\mathsf{pr}}} \right\}
\end{array}
$$

State machines

Rule 119. State machines

$$\left[\!\!\left[\begin{array}{l} \mathsf{stm} : \mathsf{StateMachineDef}, \mathsf{outevents} : \mathbb{P}\,\mathsf{Event}, \mathsf{inevents} : \mathbb{P}\,\mathsf{Event}, \\ \mathsf{ctrlops} : \mathbb{P}\,\mathsf{OperationSig}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array} \right]\!\!\right]_{\mathcal{STM}}$$

$$: \mathbb{P}\,\mathsf{Constant} \times \mathsf{Module} =$$

$$\bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r}.1\};$$
$$\bigcup\{\mathsf{r} : \mathsf{opdefsret} \bullet \mathsf{r}.1\};$$
$$\mathsf{stmret}.0;$$
$$\texttt{module}\ \mathsf{id}(\mathsf{stm})$$
$$\bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r}.2\};$$
$$\mathsf{outeventvars};$$
$$\mathsf{ineventvars};$$
$$\mathsf{stmret}.1;$$
$$\mathsf{ropsvars};$$
$$\bigcup\{\mathsf{r} : \mathsf{opdefsret} \bullet \mathsf{r}.2\};$$

$$\mathsf{stmret}.2;$$
$$\bigcup\{\mathsf{r} : \mathsf{opdefsret} \bullet \mathsf{r}.3\};$$
$$\texttt{endmodule}$$

where

$$\mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}^3}\ (\!|\ \mathsf{getVariableLists}^3(\mathsf{stm})\ |\!)$$

$$\mathsf{outeventvars} = \{\mathsf{e} : \mathsf{outevents} \bullet \texttt{EVT\_\_}\mathsf{namemap}(\mathsf{e} \mapsto \mathsf{true})) : [\![\mathsf{e.type}]\!]_t\ ;\ \}$$

$$\mathsf{ineventvars} = \{\mathsf{e} : \mathsf{inevents} \bullet \texttt{FIN\_\_}\mathsf{namemap}(\mathsf{e} \mapsto \mathsf{false}) : \texttt{bool init true}\ ;\ \}$$

$$\mathsf{rops} = \mathsf{getRequiredOperations}^3(\mathsf{stm})$$

$$\mathsf{calledops} = \mathsf{getAllCalledOperations}^1(\mathsf{stm}, \mathsf{ctrlops})$$

$$\mathsf{ropsvars} = \bigcup \left\{ \begin{array}{l} \mathsf{op} : \mathsf{getRequiredOperations}^4(\mathsf{stm}) \mid \mathsf{op} \in \mathsf{calledops} \bullet \\ \quad \{\mathsf{p} : \mathsf{op.parameters} \bullet \mathsf{uname}(\mathsf{stm}, \mathsf{op.name}, \mathsf{p.name}) : [\![\mathsf{p.type}]\!]_t\ ;\ \} \end{array} \right\}$$

$$\mathsf{ctrlopdefs} = \{\mathsf{op} : \mathsf{ctrlops} \mid \mathsf{op} \in \mathsf{OperationDef}\}$$

$$\mathsf{opdefsret} = \left\{ \begin{array}{l} \mathsf{op} : \mathsf{ctrlops} \mid \left(\begin{array}{l} (\exists \mathsf{rop} : \mathsf{rops} \bullet \mathsf{rop.name} = \mathsf{op.name}) \wedge \\ \mathsf{op} \in \mathsf{OperationDef} \end{array} \right) \\ \quad \bullet [\![\mathsf{op}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{OP}'} \end{array} \right\}$$

$$\mathsf{stmret} = [\![\mathsf{stm}, \mathsf{namemap}]\!]_{\mathcal{STB}'}$$

Rule 120. State machine body

$$\frac{\left[\!\left[\begin{array}{l}\mathsf{stm} : \mathsf{StateMachineBody}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef},\\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\right]_{\mathcal{STB}} : \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathbb{P}\,\mathsf{Command} = \left(\begin{array}{l}\bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r}.1\} \cup \{\mathsf{lockfree}\} \cup \mathsf{stmret}.2,\\ \bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r}.2\} \cup \{\mathsf{lockvar}\} \cup \mathsf{stmret}.3,\\ \mathsf{stmret}.4\end{array}\right)}{}$$

where

$$\mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}^{\natural}} (\!|\ \mathsf{getVariableLists}^4(\mathsf{stm})\ |\!)$$

$$\mathsf{stmret} = [\![\mathsf{stm}, \mathsf{stm}, \emptyset, 1, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{S}'}$$

$$\mathsf{lockfree} = \texttt{const int}\ \mathsf{id}(\mathsf{stm})\texttt{\_LOCK\_FREE} = 0;$$

$$\mathsf{lockvar} = \mathsf{id}(\mathsf{stm})\texttt{\_lock} : [0..(\mathsf{stmret}.1 - 1)] = \mathsf{id}(\mathsf{stm})\texttt{\_LOCK\_FREE};$$

Rule 121. Composite states

$$\frac{\left[\!\left[\begin{array}{l}\mathsf{cs} : \mathsf{NodeContainer}, \mathsf{stm} : \mathsf{StateMachineDef}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}},\\ \mathsf{trnumber} : \mathsf{int}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\right]_{\mathcal{S}} : \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathbb{P}\,\mathsf{Command} = \left(\begin{array}{l}\mathsf{subcssret}.1,\\ \left(\begin{array}{l}\{\mathsf{const0}\} \cup \mathsf{exitcsret}.2 \cup \mathsf{exitsubssret}.2 \cup \mathsf{enterstatesret}.2 \cup\\ \mathsf{constsnodesret}.2 \cup \mathsf{transret}.3 \cup \mathsf{subcssret}.2\end{array}\right),\\ \{\mathsf{scpc}, \mathsf{exit}\} \cup \mathsf{subcssret}.3,\\ \mathsf{exitcsret}.3 \cup \mathsf{exitsubssret}.3 \cup \mathsf{enterstatesret}.3 \cup \mathsf{transret}.4 \cup \mathsf{subcssret}.4\end{array}\right)}{}$$

where

$$\mathsf{const0} = \texttt{const int}\ \mathsf{id}(\mathsf{cs})\texttt{\_}\mathsf{INACTIVE} = 0;$$

$$\mathsf{i0} = \mathsf{getInitial}^1(\mathsf{cs})$$

$$\mathsf{f0} = \mathsf{getFinal}^1(\mathsf{cs})$$

$$\mathsf{scpcname} = \mathsf{id}(\mathsf{cs})\texttt{\_scpc}$$

$$\mathsf{scpc} = \mathsf{scpcname} : [0..(\mathbf{if}\ \mathsf{f0} \neq \mathsf{null}\ \mathbf{then}\ \mathsf{transret}.1\ \mathbf{else}\ (\mathsf{transret}.1 - 1))] = \mathbf{if}\ \mathsf{cs} \in \mathsf{StateMachineDef}\ \mathbf{then}\ \mathsf{id}(\mathsf{i0})\ \mathbf{else}\ \mathsf{const0.name};$$

$$\mathsf{exit} = \mathsf{id}(\mathsf{cs})\texttt{\_}\mathsf{exit} : [0..6] = \mathsf{Exit\_NONE};$$

$$\mathsf{exitcsret} = \left(\begin{array}{l}\mathbf{if}\ \mathsf{cs} \in \mathsf{StateMachineBody}\ \mathbf{then}\ (1, \emptyset, \emptyset)\\ \mathbf{else}\ \mathsf{exitCompState}^1(\mathsf{cs}, \mathsf{exit}, \mathsf{scpcname}, \mathsf{pcconstrs}, 1, \mathsf{ctrlopdefs}, \mathsf{namemap})\end{array}\right)$$

$$\mathsf{exitsubssret} = \mathsf{exitSubstates}^1\left(\begin{array}{l}\{\mathsf{s} : \mathsf{cs.nodes} \mid \mathsf{s} \in \mathsf{State}\}, \mathsf{exit}, \mathsf{scpcname},\\ \mathsf{pcconstrs}, \mathsf{exitcsret}.1, \mathsf{ctrlopdefs}, \mathsf{namemap}\end{array}\right)$$

$$\mathsf{enterstatesret} = \mathsf{enterSubstates}^1\left(\begin{array}{l}\left\{\mathsf{s} : \mathsf{cs.nodes} \,\middle|\, \mathsf{s} \in \mathsf{State} \wedge \mathsf{hasEntryAction}^1(\mathsf{s})\right\},\\ \mathsf{scpcname}, \mathsf{pcconstrs}, \mathsf{exitsubssret}.1,\\ \mathsf{ctrlopdefs}, \mathsf{namemap}\end{array}\right)$$

$$\mathsf{constsnodesret} = \mathsf{constantsOfNamedElems}^1(\mathsf{cs.nodes}, \mathsf{enterstatesret}.1)$$

$$\mathsf{transret} = \left[\!\left[\begin{array}{l}\mathsf{cs.nodes}, \mathsf{cs}, \mathsf{stm}, \mathsf{exit}, \mathsf{scpcname}, \mathsf{pcconstrs},\\ \mathsf{constsnodesret}.1, \mathsf{trnumber}, \mathsf{ctrlopdefs}, \mathsf{namemap}\end{array}\right]\!\right]_{\mathcal{TNS}'}$$

$$\mathsf{cstates} = \left\{\mathsf{s} : \mathsf{cs.nodes} \,\middle|\, \mathsf{s} \in \mathsf{State} \wedge \mathsf{isComposite}^1(\mathsf{s})\right\}$$

$$\mathsf{subcssret} = [\mathsf{cstates}, \mathsf{stm}, \mathsf{scpcname}, \mathsf{pcconstrs}, \mathsf{transret}.2, \mathsf{ctrlopdefs}, \mathsf{namemap}]_{\mathcal{SS}'}$$

The extra assignment ($\underline{\text{exit.name}}' = \texttt{Exit\_NONE}$) in $\underline{\text{cmd2}}$ in Rule 21 is to clear $\underline{\text{exit}}$ to avoid this command re-enabled.

Rule 122. Exit a composite state

$$\text{exitCompState}\left(\begin{array}{l}\text{cs} : \text{NodeContainer}, \text{exit} : \text{VarDecl}_{\text{pr}}, \text{scpcname} : \text{Name},\\ \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}}, \text{stnumber} : \text{int}, \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef},\\ \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$$(\underline{\text{exitret.1}}, \underline{\text{exitret.2}}, \{\underline{\text{cmd1}}, \underline{\text{cmd2}}\} \cup \underline{\text{exitret.4}})$$

where

$\underline{\text{exitreq}} = (\underline{\text{exit.name}} = \text{Exit\_ACT\_Parent}) \mid (\underline{\text{exit.name}} = \text{Exit\_ACT\_Trans})$

$\underline{\text{cmd1}} = []\left(\text{andExprs}^1(\underline{\text{pcconstrs}}) \,\&\, \underline{\text{exitreq}}\right) \rightarrow (1.0 : \underline{\text{exit.name}} = \text{Exit\_Sub\_ACT})\,;$

$\underline{\text{assgns}} = (\underline{\text{exit.name}}' = \text{Exit\_EXITED}) \,\&\, \left(\underline{\text{scpcname}}' = \underline{\text{id(cs)}}\text{\_INACTIVE}\right)$

$\underline{\text{exitret}} = \left[\!\left[\text{getExitAction}^1(\text{cs}), \emptyset, \text{stnumber}, \text{scpcname}, \text{null}, \text{assgns}, \text{ctrlopdefs}, \text{namemap}\right]\!\right]_{\mathcal{A}'}$

$\underline{\text{cmd2}} = []\left(\text{andExprs}^2(\underline{\text{pcconstrs}}) \,\&\, \underline{\text{exit.name}} = \text{Exit\_Sub\_EXITED}\right) \rightarrow$

$1.0 : \textbf{if}\ \underline{\text{exitret.4}} = \emptyset\ \textbf{then}\ \underline{\text{exitret.3}}\ \textbf{else}\ \underline{\text{exitret.3}} \,\&\, (\underline{\text{exit.name}}' = \text{Exit\_NONE})\,;$

Rule 123. Exit substates

$$\text{exitSubstates}\left(\begin{array}{l}\text{states} : \mathbb{P}\,\text{State}, \text{exit} : \text{VarDecl}_{\text{pr}}, \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}},\\ \text{stnumber} : \text{int}, \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef},\\ \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$\textbf{if}\ \text{states} = \emptyset\ \textbf{then}$

$\quad (\text{stnumber}, \emptyset, \emptyset)$

$\textbf{else}$

$\quad (\underline{\text{othersret.1}}, \underline{\text{sret.2}} \cup \underline{\text{otherret.2}}, \underline{\text{sret.3}} \cup \underline{\text{otherret.3}})$

where

$\underline{\text{s}} \in \underline{\text{states}}$

$\underline{\text{sret}} = \text{exitSubstate}^1(\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{ctrlopdefs}, \text{namemap})$

$\underline{\text{othersret}} = \text{exitSubstates}^2(\underline{\text{states}} \setminus \{\text{s}\}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \underline{\text{sret.1}}, \text{ctrlopdefs}, \text{namemap})$

Rule 124. Exit a substate

$$\text{exitSubstate}\left(\begin{array}{l}\text{s} : \text{State}, \text{exit} : \text{VarDecl}_{\text{pr}}, \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}},\\ \text{stnumber} : \text{int}, \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef},\\ \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$\textbf{if}\ \text{isComposite}^2(\text{s})\ \textbf{then}$

$\quad \text{exitCompSubstate}^1(\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber})$

$\textbf{else}$

$\quad \text{exitSimpSubstate}^1(\text{s}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{ctrlopdefs}, \text{namemap})$

Rule 125. Exit a simple substate

$$\text{exitSimpSubstate}\left(\begin{array}{l} s : State, exit : VarDecl_{pr}, scpcname : Name, pcconstrs : \mathbb{P}\, BoolExpr_{pr}, \\ stnumber : int, ctrlopdefs : \mathbb{P}\, OperationDef, \\ namemap : (Event_{rc} \times Boolean) \nrightarrow String \end{array}\right)$$

$$: int \times \mathbb{P}\, Constant \times \mathbb{P}\, Command =$$

$$(exitret.1, exitret.2, \{cmd1\} \cup exitret.4)$$

where

$$exitret = \left[\!\left[\begin{array}{l} getExitAction^2(s), \emptyset, stnumber, scpcname, null, \\ \{\, exit.name' = \texttt{Exit\_Sub\_EXITED} \,\}, ctrlopdefs, namemap \end{array} \right]\!\right]_{\mathcal{A}^2}$$

$$cmd1 = [] \left(\begin{array}{l} andExprs^3(pcconstrs) \& \left(scpcname = id(s)\right) \& \\ (exit.name = Exit\_Sub\_ACT) \end{array} \right) \rightarrow 1.0 : exitret.3;$$

Rule 126. Exit a composite substate

$$\text{exitCompSubstate}\left(\begin{array}{l} s : State, exit : VarDecl_{pr}, scpcname : Name, \\ pcconstrs : \mathbb{P}\, BoolExpr_{pr}, stnumber : int \end{array}\right)$$

$$: int \times \mathbb{P}\, Constant \times \mathbb{P}\, Command =$$

$$(stnumber, \emptyset, \{cmd1, cmd2\})$$

where

$$exitreq = (scpcname = id(s)) \& (exit.name = Exit\_Sub\_ACT)$$

$$cmd1 = [] \left(andExprs^4(pcconstrs) \& exitreq\right) \rightarrow 1.0 : \left(\begin{array}{l} (exit.name' = Exit\_Sub\_ACT\_Waiting) \& \\ \left(id(s)\_exit' = Exit\_ACT\_Parent\right) \end{array} \right);$$

$$exitedcheck = \left(\begin{array}{l} (scpcname = id(s)) \& (id(s)\_exit = Exit\_EXITED) \& \\ (exit.name = Exit\_Sub\_ACT\_Waiting) \end{array} \right)$$

$$cmd2 = [] \left(andExprs^5(pcconstrs) \& exitedcheck\right) \rightarrow 1.0 : \left(\begin{array}{l} (exit.name' = Exit\_Sub\_EXITED) \& \\ \left(id(s)\_exit' = Exit\_NONE\right) \end{array} \right);$$

Rule 127. Enter substates (with entry actions)

$$\text{enterSubstates}\left(\begin{array}{l} states : \mathbb{P}\, State, stm : StateMachineDef, scpcname : Name, \\ pcconstrs : \mathbb{P}\, BoolExpr_{pr}, stnumber : int, ctrlopdefs : \mathbb{P}\, OperationDef, \\ namemap : (Event_{rc} \times Boolean) \nrightarrow String \end{array}\right)$$

$$: int \times \mathbb{P}\, Constant \times \mathbb{P}\, Command =$$

if $states = \emptyset$ **then**

$$(stnumber, \emptyset, \emptyset)$$

else

$$(othersret.1, sret.2 \cup otherret.2, sret.3 \cup otherret.3)$$

where

$$s \in states$$

$$sret = enterSubstate^1(s, stm, scpcname, pcconstrs, stnumber, ctrlopdefs, namemap)$$

$$othersret = enterSubstates^2(states \setminus \{s\}, stm, scpcname, pcconstrs, sret.1, ctrlopdefs, namemap)$$

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

$$\text{enterSubstate}\left(\begin{array}{l} s : \text{State}, stm : \text{StateMachineDef}, scpcname : \text{Name}, \\ pcconstrs : \mathbb{P}\,\text{BoolExpr}_{pr}, stnumber : \text{int}, ctrlopdefs : \mathbb{P}\,\text{OperationDef}, \\ namemap : (\text{Event}_{rc} \times \text{Boolean}) \rightarrow\!\!\!\!\!+ \text{String} \end{array}\right)$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$$(entryret.1, \{enteringconst\} \cup entryret.2, entryret.4)$$

where

$$enteringconst = \texttt{const int}\ id(s)\texttt{\_entering} = stnumber;$$

$$extraassign = \left(\begin{array}{l} \textbf{if}\ \text{isComposite}^3(s)\ \textbf{then} \\ \quad id(s)\texttt{\_scpc}' = id\left(\text{getInitial}^2(s)\right) \\ \textbf{else} \\ \quad id(stm)\texttt{\_lock}' = id(stm)\texttt{\_LOCK\_FREE} \end{array}\right)$$

$$entryret = \left[\!\left[\begin{array}{l} \text{getEntryAction}^1(s), pcconstrs, stnumber + 1, scpcname, enteringconst, \\ \{scpcname' = id(s), extraassign\}, ctrlopdefs, namemap \end{array}\right]\!\right]_{\mathcal{A}^3}$$

Rule 129. Constants representing states for named elements

$$\text{constantsOfNamedElems}\left(\ elems : \mathbb{P}\,\text{NamedElement}, stnumber : \text{int}\ \right) : \text{int} \times \mathbb{P}\,\text{Constant} =$$

$$\textbf{if}\ elems = \emptyset\ \textbf{then}\ (stnumber, \emptyset)\ \textbf{else}\ (othersret.1, \{elconst\} \cup othersret.2)$$

where

$$el \in elems$$

$$elconst = \text{const int}\ id(el) = stnumber;$$

$$othersret = \text{constantsOfNamedElems}^2(elems \setminus \{el\}, stnumber + 1)$$

Rule 130. Composite substates

$$\left[\!\left[\begin{array}{l} cstates : \mathbb{P}\,\text{State}, stm : \text{StateMachineDef}, scpcname : \text{Name}, \\ pcconstrs : \mathbb{P}\,\text{BoolExpr}_{pr}, trnumber : \text{int}, ctrlopdefs : \mathbb{P}\,\text{OperationDef}, \\ namemap : (\text{Event}_{rc} \times \text{Boolean}) \rightarrow\!\!\!\!\!+ \text{String} \end{array}\right]\!\right]_{\mathcal{SS}}$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{VarDecl} \times \mathbb{P}\,\text{Command} =$$

$$\textbf{if}\ cstates = \emptyset\ \textbf{then}$$

$$(trnumber, \emptyset, \emptyset, \emptyset)$$

$$\textbf{else}$$

$$(othersret.1, csret.2 \cup otherret.2, csret.3 \cup otherret.3, csret.4 \cup otherret.4)$$

where

$$cs \in cstates$$

$$csret = [\![cs, stm, pcconstrs \cup \{scpcname = id(cs)\}, trnumber, ctrlopdefs, namemap]\!]_{\mathcal{S}^2}$$

$$othersret = [\![cstates \setminus \{cs\}, stm, scpcname, pcconstrs, csret.1, ctrlopdefs, namemap]\!]_{\mathcal{SS}^2}$$

Transitions

Rule 131. Nodes (transitions)

$$\frac{\left[\!\left[\begin{array}{l}\text{nodes} : \mathbb{P}\,\text{Node}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{\text{pr}},\\ \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int},\\ \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef}, \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right]\!\right]_{\mathcal{TNS}} : \text{int} \times \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =}{\begin{array}{l}\textbf{if}\ \text{nodes} = \emptyset\ \textbf{then}\\ \quad (\text{stnumber}, \text{trnumber}, \emptyset, \emptyset)\\ \textbf{else}\\ \quad (\text{othersret}.1, \text{othersret}.2, \text{nret}.3 \cup \text{othersret}.3, \text{nret}.4 \cup \text{othersret}.4)\\ \textbf{where}\\ \quad \text{n} \in \text{nodes}\\ \quad \text{nret} = [\![\text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{TN}'}\\ \quad \text{othersret} = \left[\!\left[\begin{array}{l}\text{nodes} \setminus \{\text{n}\}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname},\\ \text{pcconstrs}, \text{nret}.1, \text{nret}.2, \text{ctrlopdefs}, \text{namemap}\end{array}\right]\!\right]_{\mathcal{TNS}^2}\end{array}}$$

Rule 132. Node (transitions)

$$\frac{\left[\!\left[\begin{array}{l}\text{n} : \text{Node}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{\text{pr}},\\ \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int},\\ \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef}, \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right]\!\right]_{\mathcal{TN}} : \text{int} \times \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =}{\begin{array}{l}\textbf{if}\ \text{n} \in \text{ProbJunc}\ \textbf{then}\\ \quad \dfrac{[\![\text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{TNprob}'}}{}\\ \textbf{else}\\ \quad \dfrac{[\![\text{n}, \text{cs}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{TNother}'}}{}\end{array}}$$

Rule 133. Probabilistic junction (transitions)

$$\frac{\left[\!\left[\begin{array}{l}\text{n} : \text{ProbJunc}, \text{cs} : \text{NodeContainer}, \text{stm} : \text{StateMachineDef}, \text{exit} : \text{VarDecl}_{\text{pr}},\\ \text{scpcname} : \text{Name}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{\text{pr}}, \text{stnumber} : \text{int}, \text{trnumber} : \text{int},\\ \text{ctrlopdefs} : \mathbb{P}\,\text{OperationDef}, \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right]\!\right]_{\mathcal{TNprob}} : \text{int} \times \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =}{\begin{array}{l}(\text{transret}.1, \text{transret}.2, \text{transret}.3, \{\text{cmd}\} \cup \text{transret}.5)\\ \textbf{where}\\ \quad \text{trans} = \{\text{t} : \text{cs.transitions} \mid \text{t.source} = \text{n}\}\\ \quad \text{transret} = [\![\text{trans}, \text{n}, \text{stm}, \text{exit}, \text{scpcname}, \text{pcconstrs}, \text{stnumber}, \text{trnumber}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{TS}'}\\ \quad \text{cmd} = [\,]\left(\text{andExprs}^6(\text{pcconstrs}) \,\&\, \left(\text{scpcname} = \text{id}(\text{n})\right)\right) \rightarrow \text{transret}.4;\end{array}}$$

Rule 134. Other nodes (initial, normal junction, and states) (transitions)

$$\frac{\left[\!\!\left[\begin{array}{l} n : Node, cs : NodeContainer, stm : StateMachineDef, exit : VarDecl_{pr}, \\ scpcname : Name, pcconstrs : \mathbb{P}\, BoolExpr_{pr}, stnumber : int, trnumber : int, \\ ctrlopdefs : \mathbb{P}\, OperationDef, namemap : (Event_{rc} \times Boolean) \rightarrow\!\!\!\!\!+ String \end{array}\right]\!\!\right]_{\mathcal{T}Nother} \\ : int \times int \times \mathbb{P}\, Constant \times \mathbb{P}\, Command = \\ \hline (transret.1, transret.2, transret.3, transret.5)}$$

where

$$trans = \{t : cs.transitions \mid t.source = n\}$$

$$\frac{transret = \left[\!\!\left[\begin{array}{l} trans, n, stm, exit, scpcname, pcconstrs, \\ stnumber, trnumber, ctrlopdefs, namemap \end{array}\right]\!\!\right]_{\mathcal{TS}^2}}{}$$

Rule 135. Transitions from a node

$$\frac{\left[\!\!\left[\begin{array}{l} trans : \mathbb{P}\, Transition, n : Node, stm : StateMachineDef, exit : VarDecl_{pr}, \\ scpcname : Name, pcconstrs : \mathbb{P}\, BoolExpr_{pr}, stnumber : int, trnumber : int, \\ ctrlopdefs : \mathbb{P}\, OperationDef, namemap : (Event_{rc} \times Boolean) \rightarrow\!\!\!\!\!+ String \end{array}\right]\!\!\right]_{\mathcal{TS}} \\ : int \times int \times \mathbb{P}\, Constant \times \mathbb{P}\, Update \times \mathbb{P}\, Command = \\ \hline \begin{array}{l} \textbf{if}\ trans = \emptyset\ \textbf{then} \\ \quad (stnumber, trnumber, \emptyset, \emptyset, \emptyset) \\ \textbf{else} \\ \quad (othersret.1, othersret.2, tret.3 \cup othersret.3, tret.4 \cup othersret.4, tret.5 \cup othersret.5) \end{array}}$$

where

$$t \in trans$$

$$\frac{tret = [\![t, n, stm, exit, scpcname, pcconstrs, stnumber, trnumber, ctrlopdefs, namemap]\!]_{\mathcal{T}'}}{}$$

$$\frac{othersret = [\![trans \setminus \{t\}, n, stm, exit, scpcname, pcconstrs, tret.1, tret.2, ctrlopdefs, namemap]\!]_{\mathcal{TS}^3}}{}$$

Rule 136. Action and target of a transition

$$\frac{actionTargetOfTran\left(\begin{array}{l} t : Transition, stm : StateMachineDef, scpcname : Name, \\ pcconstrs : \mathbb{P}\, BoolExpr_{pr}, stnumber : int, ctrlopdefs : \mathbb{P}\, OperationDef, \\ namemap : (Event_{rc} \times Boolean) \rightarrow\!\!\!\!\!+ String \end{array}\right) \\ : int \times \mathbb{P}\, Constant \times \mathbb{P}\, Assignment \times \mathbb{P}\, Command = \\ \hline tactionret}$$

where

$$target = \left(\begin{array}{l} \textbf{if}\ t.target \in State \wedge hasEntryAction^2(t.target)\ \textbf{then}\ id(t.target)\_\texttt{entering} \\ \textbf{else}\ id(t.target) \end{array}\right)$$

$$targetassigns = \left(\begin{array}{l} \textbf{if}\ t.target \in State \wedge \neg\, hasEntryAction^3(t.target)\ \textbf{then} \\ \quad \textbf{if}\ isComposite(t.target) \\ \quad \textbf{then}\ \{id(t.target)\_\texttt{scpc}' = id(getInitial^3(t.target))\} \\ \quad \textbf{else}\ \{id(stm)\_\texttt{lock}' = id(stm)\_\texttt{LOCK\_FREE}\} \\ \textbf{else}\ \emptyset \end{array}\right)$$

$$tactionret = \left[\!\!\left[\begin{array}{l} t.action, pcconstrs, stnumber, scpcname, null,, \\ \{scpcname' = target\} \cup targetassigns, ctrlopdefs, namemap \end{array}\right]\!\!\right]_{\mathcal{AST}'}$$

Rule 137. Transition from a node

$$\frac{\left[\!\left[\begin{array}{l} t : \mathrm{Transition}, n : \mathrm{Node}, stm : \mathrm{StateMachineDef}, exit : \mathrm{VarDecl}_{pr}, \\ scpcname : \mathrm{Name}, pcconstrs : \mathbb{P}\,\mathrm{BoolExpr}_{pr}, stnumber : \mathrm{int}, trnumber : \mathrm{int}, \\ ctrlopdefs : \mathbb{P}\,\mathrm{OperationDef}, namemap : (\mathrm{Event}_{rc} \times \mathrm{Boolean}) \nrightarrow \mathrm{String} \end{array}\right]\!\right]_{\mathcal{T}} : \mathrm{int} \times \mathrm{int} \times \mathbb{P}\,\mathrm{Constant} \times \mathbb{P}\,\mathrm{Update} \times \mathbb{P}\,\mathrm{Command} =}{\begin{array}{l} \textbf{if } n \in \mathrm{Initial} \textbf{ then} \\ \quad \dfrac{[\![t, n, stm, exit, scpcname, pcconstrs, stnumber, trnumber, ctrlopdefs, namemap]\!]}{\mathcal{T}_i'} \\ \textbf{else if } n \in \mathrm{ProbJunc} \textbf{ then} \\ \quad \dfrac{[\![t, n, stm, exit, scpcname, pcconstrs, stnumber, trnumber, ctrlopdefs, namemap]\!]}{\mathcal{T}_p'} \\ \textbf{else if } n \in \mathrm{State} \textbf{ then} \\ \quad \dfrac{[\![t, n, stm, exit, scpcname, pcconstrs, stnumber, trnumber, ctrlopdefs, namemap]\!]}{\mathcal{T}_s'} \\ \textbf{else} \\ \quad \dfrac{[\![t, n, stm, exit, scpcname, pcconstrs, stnumber, trnumber, ctrlopdefs, namemap]\!]}{\mathcal{T}_j'} \end{array}}$$

Rule 138. Transition from an initial junction

$$\frac{\left[\!\left[\begin{array}{l} t : \mathrm{Transition}, n : \mathrm{Initial}, stm : \mathrm{StateMachineDef}, exit : \mathrm{VarDecl}_{pr}, \\ scpcname : \mathrm{Name}, pcconstrs : \mathbb{P}\,\mathrm{BoolExpr}_{pr}, stnumber : \mathrm{int}, trnumber : \mathrm{int}, \\ ctrlopdefs : \mathbb{P}\,\mathrm{OperationDef}, namemap : (\mathrm{Event}_{rc} \times \mathrm{Boolean}) \nrightarrow \mathrm{String} \end{array}\right]\!\right]_{\mathcal{T}_i} : \mathrm{int} \times \mathrm{int} \times \mathbb{P}\,\mathrm{Constant} \times \mathbb{P}\,\mathrm{Update} \times \mathbb{P}\,\mathrm{Command} =}{(tactret.1, trnumber + 1, \{locktconst\} \cup tactret.2, \emptyset, tactret.4 \cup \{cmd1\})}$$

where

$locktconst = \texttt{const int}\ id(t) = trnumber;$

$tactret = actionTargetOfTran^{1}(t, stm, scpcname, pcconstrs, stnumber, ctrlopdefs, namemap)$

$lockassign = \textbf{if}\ tactret.4 \neq \emptyset\ \textbf{then}\ \{id(stm)\_lock' = locktconst.name\}\ \textbf{else}\ \emptyset$

$cmd1 = [\,](andExprs^{7}(pcconstrs) \& (scpcname = id(n))) \rightarrow 1.0 : tactret.3 \cup lockassign;$

Rule 139. Transition from a probabilistic junction

$$\frac{\left[\!\left[\begin{array}{l} t : \mathrm{Transition}, n : \mathrm{ProbJunc}, stm : \mathrm{StateMachineDef}, exit : \mathrm{VarDecl}_{pr}, \\ scpcname : \mathrm{Name}, pcconstrs : \mathbb{P}\,\mathrm{BoolExpr}_{pr}, stnumber : \mathrm{int}, trnumber : \mathrm{int}, \\ ctrlopdefs : \mathbb{P}\,\mathrm{OperationDef}, namemap : (\mathrm{Event}_{rc} \times \mathrm{Boolean}) \nrightarrow \mathrm{String} \end{array}\right]\!\right]_{\mathcal{T}_p} : \mathrm{int} \times \mathrm{int} \times \mathbb{P}\,\mathrm{Constant} \times \mathbb{P}\,\mathrm{Update} \times \mathbb{P}\,\mathrm{Command} =}{(tactionret.1, trnumber, tactionret.2, \{[\![t.probability]\!]_e : tactionret.3\}, tactionret.4)}$$

where

$tactret = actionTargetOfTran^{2}(t, stm, scpcname, pcconstrs, stnumber, ctrlopdefs, namemap)$

Rule 140. Transition from a normal junction

$$\frac{\left[\!\left[\begin{array}{l} t : \mathsf{Transition}, n : \mathsf{Junction}, stm : \mathsf{StateMachineDef}, exit : \mathsf{VarDecl}_{pr}, \\ scpcname : \mathsf{Name}, pcconstrs : \mathbb{P}\,\mathsf{BoolExpr}_{pr}, stnumber : \mathsf{int}, trnumber : \mathsf{int}, \\ ctrlopdefs : \mathbb{P}\,\mathsf{OperationDef}, namemap : (\mathsf{Event}_{rc} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\right]_{\mathcal{T}j} : \mathsf{int} \times \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Update} \times \mathbb{P}\,\mathsf{Command} =}{(tactret.1, trnumber, tactret.2, \emptyset, tactret.4 \cup \{cmd1\})}$$

where

$tactret = \mathsf{actionTargetOfTran}^3\,(t, stm, scpcname, pcconstrs, stnumber, ctrlopdefs, namemap)$

$scpcgrd = pcconstrs \cup \{scpcname = id(n)\}$

$condgrd = \textbf{if}\ t.condition \neq null\ \textbf{then}\ (scpcgrd \cup \{[\![t.condition]\!]_e\})\ \textbf{else}\ scpcgrd$

$cmd1 = []\,\mathsf{andExprs}^8(condgrd) \rightarrow 1.0 : tactret.3;$

Rule 141. Transition from a state

$$
\frac{\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \mathsf{t} : \mathsf{Transition}, \mathsf{n} : \mathsf{State}, \mathsf{stm} : \mathsf{StateMachineDef}, \mathsf{exit} : \mathsf{VarDecl}_{\mathsf{pr}}, \\ \mathsf{scpcname} : \mathsf{Name}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{trnumber} : \mathsf{int}, \\ \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\!\right]_{\mathcal{T}s} \\
\quad : \mathsf{int} \times \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Update} \times \mathbb{P}\,\mathsf{Command} =
\end{array}}{(\mathsf{trigret}.1, \mathsf{trnumber}+1, \{\mathsf{locktconst}\} \cup \mathsf{tactret}.2 \cup \mathsf{trigret}.2, \emptyset, \mathsf{tactret}.4 \cup \mathsf{exitsrcret}.2 \cup \mathsf{trigret}.3)}
$$

where

$\mathsf{locktconst} = \texttt{const int}\ \mathsf{id}(\mathsf{t}) = \mathsf{trnumber};$

$\mathsf{scpcgrd} = \mathsf{pcconstrs} \cup \{\mathsf{scpcname} = \mathsf{id}(\mathsf{n})\}$

$\mathsf{condgrd} = \textbf{if}\ \mathsf{t.condition} \neq \mathsf{null}\ \textbf{then}\ (\mathsf{scpcgrd} \cup \{[\![\mathsf{t.condition}]\!]_e\})\ \textbf{else}\ \mathsf{scpcgrd}$

$\mathsf{lockfreegrd} = \left(\mathsf{id}(\mathsf{stm})\_\mathsf{lock} = \mathsf{id}(\mathsf{stm})\_\mathsf{LOCK\_FREE}\right)$

$\mathsf{tactret} = \mathsf{actionTargetOfTran}^4(\mathsf{t}, \mathsf{stm}, \mathsf{scpcname}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{ctrlopdefs}, \mathsf{namemap})$

$\mathsf{locktgrd} = \left(\mathsf{id}(\mathsf{stm})\_\mathsf{lock} = \mathsf{locktconst.name}\right)$

$$
\mathsf{exitsrcret} = \left(\begin{array}{l} \textbf{if}\ \mathsf{isComposite}^4(\mathsf{n})\ \textbf{then}\ \mathsf{exitCompSrcState}^1(\mathsf{t}, \mathsf{scpcgrd} \cup \{\mathsf{locktgrd}\}, \mathsf{tactret}.3) \\ \textbf{else}\ \mathsf{exitSimpSrcState}^1(\mathsf{t}, \mathsf{exit}, \mathsf{scpcgrd}, \mathsf{tactret}.3) \end{array}\right)
$$

$$
\mathsf{trigvarassign} = \left(\begin{array}{l} \textbf{if}\ \mathsf{t.trigger} \neq \mathsf{null} \wedge \mathsf{t.trigger.type} = \mathsf{CommunicationType.INPUT} \\ \textbf{then}\ \{(\mathsf{id}(\mathsf{t.trigger.parameter})' = \texttt{EVT}\_\_\mathsf{namemap}(\mathsf{t.trigger.event} \mapsto \mathsf{false}))\} \\ \textbf{else}\ \emptyset \end{array}\right)
$$

$\mathsf{locktassign} = \mathsf{id}(\mathsf{stm})\_\texttt{lock}' = \mathsf{locktconst.name}$

$$
\mathsf{lockassigns} = \left(\begin{array}{l} \textbf{if} \left(\begin{array}{l} (\mathsf{tactret}.4 \cup \mathsf{exitsrcret}.2 \neq \emptyset) \vee \\ (\mathsf{t.target} \notin \mathsf{State}) \vee \\ \left(\mathsf{t.target} \in \mathsf{State} \wedge \left(\begin{array}{l} \mathsf{hasEntryAction}(\mathsf{t.target}) \vee \\ \mathsf{isComposite}(\mathsf{t.target}) \end{array}\right)\right) \end{array}\right) \\ \textbf{then}\ \{\mathsf{locktassign}\} \\ \textbf{else}\ \emptyset \end{array}\right)
$$

$$
\mathsf{trigret} = \left(\begin{array}{l}
\textbf{if}\ \mathsf{t.trigger.\_type} = \mathsf{INPUT}\ \textbf{then} \\
\quad \left[\!\!\left[\begin{array}{l} \mathsf{t.trigger}, \mathsf{t.condition}, \mathsf{pcconstrs}, \{\mathsf{lockfreegrd}\}, \mathsf{id}(\mathsf{n}), \\ \mathsf{scpcname}, \mathsf{tactret}.1, \{\mathsf{locktassign}\}, \mathsf{exitsrcret}.1, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{I}nputEvent'} \\
\textbf{else if}\ \mathsf{t.trigger.\_type} = \mathsf{OUTPUT} \vee \mathsf{t.trigger.\_type} = \mathsf{SYNC}\ \textbf{then} \\
\quad \left[\!\!\left[\begin{array}{l} \mathsf{t.trigger}, \mathsf{t.condition}, \mathsf{pcconstrs}, \{\mathsf{lockfreegrd}\}, \mathsf{id}(\mathsf{n}), \mathsf{scpcname}, \\ \mathsf{tactret}.1, \left(\begin{array}{l} \textbf{if}\ \mathsf{isEventToRP}(\mathsf{t.trigger.event}) \\ \textbf{then}\ \mathsf{lockassigns} \\ \textbf{else}\ \{\mathsf{locktassign}\} \end{array}\right), \\ \mathsf{exitsrcret}.1, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{O}utputEvent'} \\
\textbf{else if}\ \mathsf{t.trigger.\_type} = \mathsf{SIMPLE}\ \textbf{then} \\
\quad \left[\!\!\left[\begin{array}{l} \mathsf{t.trigger}, \mathsf{t.condition}, \mathsf{pcconstrs}, \{\mathsf{lockfreegrd}\}, \mathsf{id}(\mathsf{n}), \mathsf{scpcname}, \\ \mathsf{tactret}.1, \mathsf{lockassigns} \cup \mathsf{exitsrcret}.1, \mathsf{false}, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{S}impEvent'} \\
\textbf{else} \\
\quad \left[\!\!\left[\begin{array}{l} \mathsf{t.condition}, \mathsf{pcconstrs}, \{\mathsf{lockfreegrd}\}, \mathsf{id}(\mathsf{n}), \mathsf{scpcname}, \\ \mathsf{tactret}.1, \mathsf{lockassigns} \cup \mathsf{exitsrcret}.1, \mathsf{false} \end{array}\right]\!\!\right]_{\mathcal{E}mptyEvent'}
\end{array}\right)
$$

Rule 142. Exit composite source state

$$\frac{\begin{array}{l}\text{exitCompSrcState}\left(\ t : \text{Transition}, \text{pcconstrs} : \text{BoolExpr}_{pr}, \text{curassgns} : \mathbb{P}\ \text{Assignment}_{pr}\ \right) \\ \quad : \mathbb{P}\ \text{Assignment} \times \mathbb{P}\ \text{Command} = \\ \quad (\ \{\ \text{id}(t.\text{source})\_\text{exit}' = \text{Exit\_ACT\_Trans}\ \}, \{\text{exitcmd}\}\ )\end{array}}{}$$

where

$$\text{exitcmd} = [\,]\left(\text{andExprs}^{9}(\text{pcconstrs})\,\&\left(\text{id}(t.\text{source})\_\text{exit} = \text{Exit\_EXITED}\right)\right) \rightarrow 1.0 : \text{curassgns};$$

Rule 143. Exit simple source state

$$\frac{\begin{array}{l}\text{exitSimpSrcState}\,(t : \text{Transition}, \text{exit} : \text{VarDecl}_{pr}, \text{pcconstrs} : \text{BoolExpr}_{pr}, \text{curassgns} : \mathbb{P}\ \text{Assignment}_{pr}) \\ \quad : \mathbb{P}\ \text{Assignment} \times \mathbb{P}\ \text{Command} = \\ \quad \textbf{if}\ \neg\ \text{hasExitAction}^{1}(t.\text{source})\ \ \textbf{then}\ \ (\text{curassgns}, \emptyset) \\ \quad \textbf{else}\ \ (\ \{\ \text{exit.name} = \text{Exit\_Sub\_ACT}\ \}, \{\text{exitcmd}\}\ )\end{array}}{}$$

where

$$\text{exitcmd} = [\,]\left(\text{andExprs}^{10}(\text{pcconstrs})\,\&\,(\text{exit.name} = \text{Exit\_Sub\_EXITED})\right) \rightarrow$$
$$1.0 : \text{curassgns}\,\&\,(\text{exit.name}' = \texttt{EXIT\_NONE});$$

Actions

Rule 144. Actions

$$\frac{\begin{array}{l}\left[\!\!\left[\begin{array}{l}\text{act} : \text{Action}, \text{pcconstrs} : \mathbb{P}\ \text{BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{scpcname} : \text{Name}, \\ \text{curstate} : \text{Constant}_{pr}, \text{assigns} : \mathbb{P}\ \text{Assignment}_{pr}, \text{ctrlopdefs} : \mathbb{P}\ \text{OperationDef}, \\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right]\!\!\right]_{\mathcal{A}} \\ \quad : \text{int} \times \mathbb{P}\ \text{Constant} \times \mathbb{P}\ \text{Assignment} \times \mathbb{P}\ \text{Command} = \\ \quad \textbf{if}\ \ \text{act} = \text{null}\ \textbf{then} \\ \quad\quad (\text{stnumber}, \emptyset, \text{assigns}, \emptyset) \\ \quad \textbf{else} \\ \quad\quad [\![\text{act.action}, \text{pcconstrs}, \text{stnumber}, \text{scpcname}, \text{curstate}, \text{assigns}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{AST}^{2}}\end{array}}{}$$

The function $[\![\_]\!]_{\mathcal{AST}}$ is the entry to translate a statement stmt, and the function $[\![\_]\!]_{\mathcal{ST}}$ should not be called directly except inside this function $[\![\_]\!]_{\mathcal{AST}}$.

Rule 145. Statements

$$\frac{\begin{array}{l}\left[\!\!\left[\begin{array}{l}\text{stmt} : \text{Statement}, \text{pcconstrs} : \mathbb{P}\ \text{BoolExpr}_{pr}, \text{stnumber} : \text{int}, \text{scpcname} : \text{Name}, \\ \text{curstate} : \text{Constant}_{pr}, \text{assigns} : \mathbb{P}\ \text{Assignment}_{pr}, \text{ctrlopdefs} : \mathbb{P}\ \text{OperationDef}, \\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right]\!\!\right]_{\mathcal{AST}} \\ \quad : \text{int} \times \mathbb{P}\ \text{Constant} \times \mathbb{P}\ \text{Assignment} \times \mathbb{P}\ \text{Command} = \\ \quad \textbf{if}\ \ \text{stmt} = \text{null}\ \ \textbf{then} \\ \quad\quad (\text{stnumber}, \emptyset, \text{assigns}, \emptyset) \\ \quad \textbf{else} \\ \quad\quad [\![\text{stmt}, \text{pcconstrs}, \text{stnumber}, \text{scpcname}, \text{curstate}, \text{assigns}, \text{ctrlopdefs}, \text{namemap}]\!]_{\mathcal{ST}^{1}}\end{array}}{}$$

Rule 146. Statement

$$\left[\!\!\left[\begin{array}{l}\mathsf{stmt} : \mathsf{Statement}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name},\\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef},\\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\!\right]_{\mathcal{ST}}$$
$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

This rule is split in multiple rules from Rule 46 to Rule 52 according to the subtype of the statement.

Rule 147. Statement (Skip)

$$\left[\!\!\left[\begin{array}{l}\mathsf{stmt} : \mathsf{Skip}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name},\\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef},\\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\!\right]_{\mathcal{ST}}$$
$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

$$(\mathsf{stnumber}, \emptyset, \mathsf{assigns}, \emptyset)$$

Rule 148. Statement (ParStmt)

$$\left[\!\!\left[\begin{array}{l}\mathsf{stmt} : \mathsf{ParStmt}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name},\\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef},\\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\!\right]_{\mathcal{ST}}$$
$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

$$[\![\mathsf{stmt.stmt}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{scpcname}, \mathsf{curstate}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{AST}^3}$$

Rule 149. Statement (Assignment)

$$\left[\!\!\left[\begin{array}{l}\mathsf{stmt} : \mathsf{Assignment}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name},\\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef},\\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}\end{array}\right]\!\!\right]_{\mathcal{ST}}$$
$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

$$\left(\begin{array}{l}\textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{stnumber}\ \textbf{else}\ \mathsf{stnumber} + 1,\\ \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \emptyset\ \textbf{else}\ \{\mathsf{const}\},\\ \mathsf{curassgns}, \{\mathsf{cmd}\}\end{array}\right)$$

where

$\mathsf{const} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{curstate}\ \textbf{else}\ \texttt{const int}\ \mathsf{uname}(\mathsf{stmt}, \mathsf{stnumber}) = \mathsf{stnumber};$

$\mathsf{curassgns} = \{\mathsf{scpcname}' = \mathsf{const.name}\}$

$$\mathsf{cmd} = \left(\begin{array}{l}[\,](\mathsf{andExprs}^{11}(\mathsf{pcconstrs})\,\&\,(\mathsf{scpcname} = \mathsf{const.name}))\\ \quad \rightarrow 1.0 : \mathsf{assigns} \cup \{\mathsf{id}(\mathsf{stmt.left})' = [\![\mathsf{stmt.right}]\!]_e\};\end{array}\right)$$

A CommunicationStmt does not allow an empty event and so the else branch in the definition of trigret should not be reached, which is guaranteed by the corresponding well-formedness condition.

Rule 150. Statement (CommunicationStmt)

$$\left[\!\!\left[\begin{array}{l} \mathsf{stmt} : \mathsf{CommunicationStmt}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{pr}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\ \mathsf{curstate} : \mathsf{Constant}_{pr}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{pr}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \\ \mathsf{namemap} : (\mathsf{Event}_{rc} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\!\right]_{\mathcal{ST}}$$

$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

$$\left(\mathsf{commret.1}, \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \emptyset\ \textbf{else}\ \{\mathsf{const}\} \cup \mathsf{commret.2}, \{\mathsf{curassgn}\}, \mathsf{commret.3} \right)$$

where

$$\mathsf{const} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{curstate}\ \textbf{else}\ \texttt{const int}\ \mathsf{uname}(\mathsf{stmt}, \mathsf{stnumber}) = \mathsf{stnumber};$$

$$\mathsf{stnumber1} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{stnumber}\ \textbf{else}\ \mathsf{stnumber} + 1$$

$$\mathsf{curassgn} = \mathsf{scpcname}' = \mathsf{const.name}$$

$$\mathsf{commret} = \left(\begin{array}{l} \textbf{if}\ \mathsf{stmt.communication.\_type} = \mathsf{CommunicationType.INPUT}\ \textbf{then} \\ \quad \left[\!\!\left[\begin{array}{l} \mathsf{stmt.communication}, \mathsf{null}, \mathsf{pcconstrs}, \emptyset, \mathsf{const.name}, \\ \mathsf{scpcname}, \mathsf{stnumber1}, \emptyset, \mathsf{assigns}, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{InputEvent}^2} \\ \textbf{else if} \left(\begin{array}{l} \mathsf{stmt.communication.\_type} = \mathsf{CommunicationType.OUTPUT} \vee \\ \mathsf{stmt.communication.\_type} = \mathsf{CommunicationType.SYNC} \end{array} \right) \textbf{then} \\ \quad \left[\!\!\left[\begin{array}{l} \mathsf{stmt.communication}, \mathsf{null}, \mathsf{pcconstrs}, \emptyset, \mathsf{const.name}, \\ \mathsf{scpcname}, \mathsf{stnumber1}, \emptyset, \mathsf{assigns}, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{OutputEvent}^2} \\ \textbf{else if}\ \mathsf{stmt.communication.\_type} = \mathsf{CommunicationType.SIMPLE}\ \textbf{then} \\ \quad \left[\!\!\left[\begin{array}{l} \mathsf{stmt.communication}, \mathsf{null}, \mathsf{pcconstrs}, \emptyset, \mathsf{const.name}, \\ \mathsf{scpcname}, \mathsf{stnumber1}, \mathsf{assigns}, \mathsf{true}, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{SimpEvent}^2} \\ \textbf{else} \end{array} \right)$$

Rule 151. Statment (If Statements)

$$\left[\!\!\left[\begin{array}{l} \mathsf{stmt} : \mathsf{IfStmt}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{pr}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\ \mathsf{curstate} : \mathsf{Constant}_{pr}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{pr}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \\ \mathsf{namemap} : (\mathsf{Event}_{rc} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\!\right]_{\mathcal{ST}}$$

$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =$$

$$\left(\begin{array}{l} \mathsf{elseret.1}, (\textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \emptyset\ \textbf{else}\ \{\mathsf{const}\}) \cup \mathsf{thenret.2} \cup \mathsf{elseret.2}, \mathsf{curassgns}, \\ \{\mathsf{thencmd}, \mathsf{elsecmd}\} \cup \mathsf{thenret.4} \cup \mathsf{elseret.4} \end{array} \right)$$

where

$$\mathsf{const} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{curstate}\ \textbf{else}\ \texttt{const int}\ \mathsf{uname}(\mathsf{stmt}, \mathsf{stnumber}) = \mathsf{stnumber};$$

$$\mathsf{curassgns} = \{\mathsf{scpcname}' = \mathsf{const.name}\}$$

$$\mathsf{thenret} = \left[\!\!\left[\begin{array}{l} \mathsf{stmt.then}, \mathsf{pcconstrs}, \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{stnumber}\ \textbf{else}\ \mathsf{stnumber} + 1, \\ \mathsf{scpcname}, \mathsf{null}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap} \end{array}\right]\!\!\right]_{\mathcal{AST}^4}$$

$$\mathsf{thencmd} = [] \left(\begin{array}{l} \mathsf{andExprs}^{12}(\mathsf{pcconstrs})\& \\ (\mathsf{scpcname} = \mathsf{const.name})\& \\ [\![\mathsf{stmt.expression}]\!]_e \end{array} \right) \rightarrow 1.0 : \mathsf{thenret.3};$$

$$\mathsf{elseret} = [\mathsf{stmt.else}, \mathsf{pcconstrs}, \mathsf{thenret.1}, \mathsf{scpcname}, \mathsf{null}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap}]_{\mathcal{AST}^5}$$

$$\mathsf{elsecmd} = [] \left(\begin{array}{l} \mathsf{andExprs}^{13}(\mathsf{pcconstrs})\& \\ (\mathsf{scpcname} = \mathsf{const.name})\& \\ ![\![\mathsf{stmt.expression}]\!]_e \end{array} \right) \rightarrow 1.0 : \mathsf{elseret.3};$$

Rule 152. Statement (Call)

$$\frac{\left[\!\left[\begin{array}{l} \mathsf{stmt} : \mathsf{Call}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{curupdates} : \mathbb{P}\,\mathsf{Update}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \\ \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\right]_{\mathcal{ST}} \\ : \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =}{\left(\begin{array}{l} \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{stnumber}\ \textbf{else}\ \mathsf{stnumber} + 1, \\ \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \emptyset\ \textbf{else}\ \{\mathsf{const}\}, \\ \mathsf{curassgns}, \mathsf{cmds} \end{array}\right)}$$

$$\begin{array}{l} \textbf{where} \\ \quad \mathsf{const} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \textbf{then}\ \mathsf{curstate}\ \textbf{else}\ \texttt{const int}\ \mathsf{uname}(\mathsf{stmt}, \mathsf{stnumber}) = \mathsf{stnumber}; \\ \quad \mathsf{curassgns} = \{\mathsf{scpcname}' = \mathsf{const.name}\} \\ \quad \mathsf{opvarassigns} = \left\{\begin{array}{l} \mathsf{p} : \mathsf{getParaArgPairs}^{1}(\mathsf{stmt.operation.parameters}, \mathsf{stmt.args}) \\ \quad \bullet\ \mathsf{id}(\mathsf{stmt.operation}, \mathsf{p.1})' = [\![\mathsf{p.2}]\!]_{e} \end{array}\right\} \\ \quad \mathsf{opdef} = \mathsf{getOperationDef}^{1}(\mathsf{stmt.operation}, \mathsf{ctrlopdefs}) \\ \quad \mathsf{grd} = \mathsf{pcconstrs} \cup \{\mathsf{scpcname} = \mathsf{const.name}\} \\ \quad \mathsf{cmds} = \left(\begin{array}{l} \textbf{if}\ \mathsf{opdef} \neq \mathsf{null}\ \textbf{then} \\ \quad \mathsf{callOpdef}^{1}\left(\ \mathsf{opdef}, \mathsf{grd}, \mathsf{opvarassigns}, \mathsf{assigns}\ \right) \\ \textbf{else} \\ \quad \left\{\ [\mathsf{id}(\mathsf{stmt.operation})]\,(\mathsf{andExprs}^{14}(\mathsf{grd}) \rightarrow \texttt{1.0} : \mathsf{assigns} \cup \mathsf{opvarassigns};\ \right\} \end{array}\right) \end{array}$$

Rule 153. Statement (sequential composition)

$$\frac{\left[\!\left[\begin{array}{l} \mathsf{stmt} : \mathsf{SeqStatement}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\ \mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\right]_{\mathcal{ST}} \\ : \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Assignment} \times \mathbb{P}\,\mathsf{Command} =}{[\![\mathsf{stmt.statements}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{scpcname}, \mathsf{curstate}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{STS}'}}$$

Rule 154. Statement (sequence of statements)

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l}
\mathsf{sts} : \mathbf{seq}\ \mathsf{Statement}, \mathsf{pcconstrs} : \mathbb{P}\ \mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\
\mathsf{curstate} : \mathsf{Constant}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\ \mathsf{Assignment}_{\mathsf{pr}}, \mathsf{ctrlopdefs} : \mathbb{P}\ \mathsf{OperationDef}, \\
\mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \rightarrow\!\!\!\!\!+ \mathsf{String}
\end{array}\right]\!\!\right]_{\mathcal{STS}} \\
\quad : \mathsf{int} \times \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{Assignment} \times \mathbb{P}\ \mathsf{Command} = \\
\hline
\quad \mathbf{if}\ \mathsf{sts} = \langle\rangle\ \mathbf{then} \\
\quad\quad (\mathsf{stnumber}, \emptyset, \mathsf{assigns}, \emptyset) \\
\quad \mathbf{else\ if}\ \#\mathsf{sts} = 1\ \mathbf{then} \\
\quad\quad [\![\mathsf{head}\,\mathsf{sts}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{scpcname}, \mathsf{curstate}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{AST}^6} \\
\quad \mathbf{else} \\
\quad\quad \mathbf{let}\ \mathsf{lres} == [\![\mathsf{last}\,\mathsf{sts}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{scpcname}, \mathsf{null}, \mathsf{assigns}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{AST}^7}\ ; \\
\quad\quad\quad \mathsf{fsres} == [\![\mathsf{front}\,\mathsf{sts}, \mathsf{pcconstrs}, \mathsf{lres}.1, \mathsf{scpcname}, \mathsf{curstate}, \mathsf{lres}.3, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{STS}^2} \\
\quad\quad\quad \bullet\ (\mathsf{fsres}.1, \mathsf{lres}.2 \cup \mathsf{fsres}.2, \mathsf{fsres}.3, \mathsf{lres}.4 \cup \mathsf{fsres}.4)
\end{array}
$$

Connections

Rule 155. Asynchronous connections

$$
\begin{array}{l}
\mathsf{asyncConnectedEvents}\left(\begin{array}{l}\mathsf{parent} : \mathsf{NamedElement}, \mathsf{connections} : \mathbb{P}\ \mathsf{Connection}, \\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \rightarrow\!\!\!\!\!+ \mathsf{String}\end{array}\right) : \mathbb{P}\ \mathsf{Module} = \\
\hline
\quad \bigcup \left\{ \begin{array}{l}
\mathsf{c} : \mathsf{connections} \bullet \\
\left(\begin{array}{l}
\mathbf{if}\ \mathsf{isInternalAsyncConn}^1(\mathsf{c})\ \mathbf{then} \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \mathsf{c.bidirec}\ \mathbf{then} \\
\quad \left(\begin{array}{l}
\left\{ \begin{array}{l}
\mathsf{asyncConnectedEvent}^1 \left(\begin{array}{l}\mathsf{parent}, \mathsf{c.from}, \mathsf{c.to}, \\ \mathsf{c.efrom}, \mathsf{c.eto}, \mathsf{namemap}\end{array} \right), \\
\mathsf{asyncConnectedEvent}^2 \left(\begin{array}{l}\mathsf{parent}, \mathsf{c.to}, \mathsf{c.from}, \\ \mathsf{c.eto}, \mathsf{c.efrom}, \mathsf{namemap}\end{array} \right)
\end{array} \right\} \\
\mathbf{else} \left\{ \mathsf{asyncConnectedEvent}^3 \left(\begin{array}{l}\mathsf{parent}, \mathsf{c.from}, \mathsf{c.to}, \\ \mathsf{c.efrom}, \mathsf{c.eto}, \mathsf{namemap}\end{array} \right) \right\}
\end{array} \right)
\end{array} \right) \\
\mathbf{else}\ \emptyset
\end{array} \right)
\end{array} \right\}
\end{array}
$$

For connections between a controller and its contained state machine. If it is from the controller to a state machine, its event should be regarded as INPUT for the state machine. Otherwise, it should be regarded as OUTPUT.

Rule 156. Asynchronous connection

$$\text{asyncConnectedEvent}\left(\begin{array}{l}\text{parent}:\text{NamedElement},\text{cfrom}:\text{ConnectionNode},\\ \text{cto}:\text{ConnectionNode},\text{efrom}:\text{Event},\text{eto}:\text{Event},\\ \text{namemap}:(\text{Event}_{\text{rc}}\times\text{Boolean})\nrightarrow\text{String}\end{array}\right):\text{Module}=$$

$$\begin{array}{l}\quad\textbf{if}\ \text{efrom.type}\neq\text{null}\ \textbf{then}\\ \qquad\text{buffer}^1\,(\text{namemap}(\text{eout}),\text{namemap}(\text{ein}),\text{efrom.type},)\\ \quad\textbf{else}\\ \qquad\text{simpleBuffer}^1\,(\text{namemap}(\text{eout}),\text{namemap}(\text{ein}))\\ \textbf{where}\\ \quad\text{fromctrl}=\text{parent}\in\text{Controller}\wedge\text{cfrom}=\text{parent}\\ \quad\text{toctrl}=\text{parent}\in\text{Controller}\wedge\text{cto}=\text{parent}\\ \quad\text{eout}=\textbf{if}\ \text{fromctrl}\ \textbf{then}\ (\text{efrom},\text{false})\ \textbf{else}\ (\text{efrom},\text{true})\\ \quad\text{ein}=\textbf{if}\ \text{toctrl}\ \textbf{then}\ (\text{eto},\text{true})\ \textbf{else}\ (\text{eto},\text{false})\end{array}$$

A connection is called an internal asynchronous connection (isInternalAsyncConn^2) if the connection is asynchronous and not from or to a robotic platform. For the asynchronous connection between a controller and a platform, we do not need a buffer to connect them because the platform is the environment of the controllers.

Rule 157. Asynchronous connection

$$\begin{array}{l}\text{isInternalAsyncConn}\,(\text{c}:\text{Connection}):\text{Boolean}=\\ \quad\textbf{if}\ \text{c.async}\wedge\text{c.from}\notin\text{RoboticPlatform}\wedge\text{c.to}\notin\text{RoboticPlatform}\ \textbf{then}\ \text{true}\ \textbf{else}\ \text{false}\end{array}$$

In Rule 57, namemap is a function from an event and its direction (for input or output) to its name in PRISM. Here we use true to denote the event is used for output and false for input.

Rule 158. Connections: events renaming

$$\text{renameEventsConns}\left(\begin{array}{l}\text{parent}:\text{NamedElement},\text{connections}:\mathbb{P}\,\text{Connection},\\ \text{namemap}:(\text{Event}_{\text{rc}}\times\text{Boolean})\nrightarrow\text{String}\end{array}\right)$$

$$:(\text{Event}_{\text{rc}}\times\text{Boolean})\nrightarrow\text{String}=$$

$$\begin{array}{l}\textbf{if}\ \text{connections}=\emptyset\ \textbf{then}\ \text{namemap}\ \textbf{else}\ \text{osnamemap}\\ \textbf{where}\\ \quad\text{c}\in\text{connections}\\ \quad\text{cnamemap}=\text{renameEventsConn}^1\,(\text{parent},\text{c},\text{namemap})\\ \quad\text{osnamemap}=\text{renameEventsConns}^3\,(\text{parent},\text{connections}\setminus\{\text{c}\},\text{cnamemap})\end{array}$$

Rule 159. Connection: events renaming

$$\text{renameEventsConn}\left(\begin{array}{l}\text{parent} : \text{NamedElement}, \text{c} : \text{Connection},\\ \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$
$$: (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String} =$$

$$\left(\begin{array}{l}\textbf{if } \text{isInternalAsyncConn}^3(\text{c}) \textbf{ then}\\ \quad\left(\begin{array}{l}\textbf{if } \text{c.bidirec} \textbf{ then}\\ \quad\left(\begin{array}{l}\text{renameAsyncEvents}^1(\text{parent}, \text{c.from}, \text{c.to}, \text{c.efrom}, \text{c.eto}, \text{namemap})\\ \quad\oplus\\ \quad\text{renameAsyncEvents}^2(\text{parent}, \text{c.to}, \text{c.from}, \text{c.eto}, \text{c.efrom}, \text{namemap})\\ \quad\textbf{else}\\ \quad\text{renameAsyncEvents}^3(\text{parent}, \text{c.from}, \text{c.to}, \text{c.efrom}, \text{c.eto}, \text{namemap})\end{array}\right)\end{array}\right)\\ \textbf{else}\\ \quad\left(\begin{array}{l}\textbf{if } \text{c.bidirec} \textbf{ then}\\ \quad\left(\begin{array}{l}\text{renameSyncEvents}^1(\text{parent}, \text{c.from}, \text{c.to}, \text{c.efrom}, \text{c.eto}, \text{namemap})\\ \quad\oplus\\ \quad\text{renameSyncEvents}^2(\text{c.to}, \text{c.from}, \text{c.eto}, \text{c.efrom}, \text{namemap})\\ \quad\textbf{else}\\ \quad\text{renameSyncEvents}^3(\text{parent}, \text{c.from}, \text{c.to}, \text{c.efrom}, \text{c.eto}, \text{namemap})\end{array}\right)\end{array}\right)\end{array}\right)$$

If the connection is within a controller and between the controller and a state machine, the event direction considered will be different because the event on the controller acts as a relay and it is shared by two connections: one inside the controller and one outside the controller (inside a module). The event is used for opposite directions in both connections. We give names to events in connections of a module first, and then propagate to connections inside controllers. Therefore, for a connection that is inside a controller and between the controller and a state machine, it direction should be opposite. In Rule 59 and Rule 60, the definitions of eout and ein take this into account.

Rule 160. Connection: events renaming

$$\text{renameSyncEvents}\left(\begin{array}{l}\text{parent} : \text{NamedElement}, \text{cfrom} : \text{ConnectionNode},\\ \text{cto} : \text{ConnectionNode}, \text{efrom} : \text{Event}, \text{eto} : \text{Event},\\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String} =$$

$$\begin{array}{l}
\quad \textbf{if } \text{eout} \notin \text{mapdom} \wedge \text{ein} \notin \text{mapdom} \textbf{ then}\\
\qquad \textbf{let } \text{ename} == \text{id}(\text{eout}.1, \text{eout}.2) \bullet \text{namemap} \oplus \{\text{eout} \mapsto \text{ename}, \text{ein} \mapsto \text{ename}\}\\
\quad \textbf{else if } \text{eout} \in \text{mapdom} \wedge \text{ein} \notin \text{mapdom} \textbf{ then}\\
\qquad \text{namemap} \oplus \{\text{ein} \mapsto \text{namemap}(\text{eout})\}\\
\quad \textbf{else if } \text{eout} \notin \text{mapdom} \wedge \text{ein} \in \text{mapdom} \textbf{ then}\\
\qquad \text{namemap} \oplus \{\text{eout} \mapsto \text{namemap}(\text{ein})\}\\
\quad \textbf{else}\\
\qquad \text{namemap}\\
\textbf{where}\\
\quad \text{mapdom} = \mathrm{dom}\ \text{namemap}\\
\quad \text{fromctrl} = \text{parent} \in \text{Controller} \wedge \text{cfrom} = \text{parent}\\
\quad \text{toctrl} = \text{parent} \in \text{Controller} \wedge \text{cto} = \text{parent}\\
\quad \text{eout} = \textbf{if } \text{fromctrl} \textbf{ then } (\text{efrom}, \text{false}) \textbf{ else } (\text{efrom}, \text{true})\\
\quad \text{ein} = \textbf{if } \text{toctrl} \textbf{ then } (\text{eto}, \text{true}) \textbf{ else } (\text{eto}, \text{false})
\end{array}$$

For an asynchronous connection between controllers or between state machines, we will add one buffer module and so the translated names for the two events in both ends of the connection are different.

Rule 161. Asynchronous connection: events renaming

$$\text{renameAsyncEvents}\left(\begin{array}{l}\text{parent} : \text{NamedElement}, \text{cfrom} : \text{ConnectionNode},\\ \text{cto} : \text{ConnectionNode}, \text{efrom} : \text{Event}, \text{eto} : \text{Event},\\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String} =$$

$$\begin{array}{l}
\quad \textbf{if } \text{eout} \notin \text{mapdom} \wedge \text{ein} \notin \text{mapdom} \textbf{ then}\\
\qquad \textbf{let } \text{efname} == \text{id}(\text{eout}.1, \text{eout}.2)\,;\ \text{etname} == \text{id}(\text{ein}.1, \text{ein}.2)\\
\qquad\quad \bullet\ \text{namemap} \oplus \{\text{eout} \mapsto \text{efname}, \text{ein} \mapsto \text{etname}\}\\
\quad \textbf{else if } \text{eout} \in \text{mapdom} \wedge \text{ein} \notin \text{mapdom} \textbf{ then}\\
\qquad \textbf{let } \text{etname} == \text{id}(\text{ein}.1, \text{ein}.2) \bullet \text{namemap} \oplus \{\text{ein} \mapsto \text{etname}\}\\
\quad \textbf{else if } \text{eout} \notin \text{mapdom} \wedge \text{ein} \in \text{mapdom} \textbf{ then}\\
\qquad \textbf{let } \text{efname} == \text{id}(\text{eout}.1, \text{eout}.2) \bullet \text{namemap} \oplus \{\text{eout} \mapsto \text{efname}\}\\
\quad \textbf{else}\\
\qquad \text{namemap}\\
\textbf{where}\\
\quad \text{mapdom} = \mathrm{dom}\ \text{namemap}\\
\quad \text{fromctrl} = \text{parent} \in \text{Controller} \wedge \text{cfrom} = \text{parent}\\
\quad \text{toctrl} = \text{parent} \in \text{Controller} \wedge \text{cto} = \text{parent}\\
\quad \text{eout} = \textbf{if } \text{fromctrl} \textbf{ then } (\text{efrom}, \text{false}) \textbf{ else } (\text{efrom}, \text{true})\\
\quad \text{ein} = \textbf{if } \text{toctrl} \textbf{ then } (\text{eto}, \text{true}) \textbf{ else } (\text{eto}, \text{false})
\end{array}$$

Buffer

This buffer module simulates a buffer of size 1 for an asynchronous connection on an input event and an output event. The fvar denotes whether a buffer is full or not. The vvar is the element stored in the buffer. The finvar denotes whether a communication is finished or not because a communication takes two steps in PRISM: 1) cmd1 synchronises and 2) cmd2 updates the buffer.

Rule 162. Buffer

$$\mathsf{buffer}(\mathsf{ein} : \mathsf{String}, \mathsf{eout} : \mathsf{String}, \mathsf{etype} : \mathsf{Type}) : \mathsf{Module} =$$

$$\begin{array}{l}
\quad \mathsf{module} \\
\textbf{where} \\
\quad \mathsf{fvar} = \mathsf{ein_f} : \texttt{bool init false}; \\
\quad \mathsf{vvar} = \texttt{EVT__}\mathsf{ein} : [\![\mathsf{etype}, \mathsf{false}]\!]_t; \\
\quad \mathsf{finvar} = \texttt{FIN__}\mathsf{ein} : \texttt{bool init true}; \\
\quad \mathsf{cmd1} = \left(\begin{array}{l} [\mathsf{ein}]\,(\mathsf{FIN__eout} = \mathsf{true})\,\&\,(\mathsf{finvar.name} = \mathsf{true}) \\ \quad \rightarrow 1.0 : (\mathsf{finvar.name}' = \mathsf{false} \& \mathsf{fvar.name}' = \mathsf{false}); \end{array} \right) \\
\quad \mathsf{cmd2} = \left(\begin{array}{l} []\,(\mathsf{fvar.name} = \mathsf{false})\,\&\,(\mathsf{finvar.name} = \mathsf{true}) \\ \quad \rightarrow 1.0 : \left(\begin{array}{l} \mathsf{finvar.name}' = \mathsf{true}\,\&\,\mathsf{fvar.name}' = \mathsf{true}\,\& \\ \mathsf{vvar.name}' = \mathsf{EVT__OUT_BUF__ename} \end{array} \right); \end{array} \right) \\
\quad \mathsf{cmd3} = [\mathsf{eout}]\,(\mathsf{fvar.name} = \mathsf{true}) \rightarrow 1.0 : (\mathsf{fvar.name}' = \mathsf{false}); \\
\quad \mathsf{module} = \left(\begin{array}{l} \texttt{module}\,\mathsf{BUF_ein} \\ \quad \mathsf{vars} \rightsquigarrow \{\mathsf{fvar}, \mathsf{vvar}, \mathsf{finvar}\}, \mathsf{commands} \rightsquigarrow \{\mathsf{cmd1}, \mathsf{cmd2}, \mathsf{cmd3}\} \\ \texttt{endmodule} \end{array} \right)
\end{array}$$

A simple buffer simulates an asynchronous connection on simple events.

Rule 163. Simple Buffer

$$\mathsf{simpleBuffer}(\mathsf{ein} : \mathsf{String}, \mathsf{eout} : \mathsf{String}) : \mathsf{Module} =$$

$$\begin{array}{l}
\quad \mathsf{module} \\
\textbf{where} \\
\quad \mathsf{fvar} = \mathsf{ein_f} : \texttt{bool init false}; \\
\quad \mathsf{cmd1} = [\mathsf{ein}]\,(\texttt{true}) \rightarrow 1.0 : (\mathsf{fvar.name}' = \texttt{true}); \\
\quad \mathsf{cmd2} = [\mathsf{eout}]\,(\mathsf{fvar.name} = \mathsf{true}) \rightarrow 1.0 : (\mathsf{fvar.name}' = \mathsf{false}); \\
\quad \mathsf{module} = \left(\begin{array}{l} \texttt{module}\,\mathsf{BUF_ein} \\ \quad \mathsf{vars} \rightsquigarrow \{\mathsf{fvar}\}, \mathsf{commands} \rightsquigarrow \{\mathsf{cmd1}, \mathsf{cmd2}\} \\ \texttt{endmodule} \end{array} \right)
\end{array}$$

Events

The function isStandaloneEvent returns if an event used for input or output is a standalone event, that is, an event of a state machine that is not connected to the events of other state machines. The functions isEventToBuffer and isEventToRP are similar, and they return if an event for output is connected to an asynchronous connection or to a robotic platform respectively. These functions are implementation dependent and one possible solution is to use different name prefixing to distinguish

these events in PRISM.

Rule 164. Input Events

$$
\begin{array}{l}
\left[\!\left[\begin{array}{l}
\mathsf{trigger} : \mathsf{Communication}, \mathsf{condition} : \mathsf{Expression}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \\
\mathsf{lockfreegrd} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{curstate} : \mathsf{String}, \mathsf{scpcname} : \mathsf{Name}, \mathsf{stnumber} : \mathsf{int}, \\
\mathsf{lockassign} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}}, \\
\mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String}
\end{array}\right]\!\right]_{\mathcal{I}nput\mathcal{E}vent} \\
\quad : \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Command} = \\
\hline
\quad (\mathsf{stnumber} + 1, \{\mathsf{stconst}\}, \{\mathsf{cmd1}, \mathsf{cmd2}\})
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{ename} = \mathsf{namemap}\,(\mathsf{trigger.event} \mapsto \mathsf{false}) \\
\mathsf{scpcgrd} = \mathsf{pcconstrs} \cup \{\mathsf{scpcname} = \mathsf{curstate}\} \\
\mathsf{condgrd} = \left(\begin{array}{l}
\textbf{if}\ \mathsf{condition} \neq \mathsf{null} \\
\textbf{then}\ \mathsf{substVarName}\left(\begin{array}{l} [\![\mathsf{condition}]\!]_e, \mathsf{id}(\mathsf{trigger.parameter}), \\ \mathtt{EVT\_\_}\mathsf{ename} \end{array}\right) \\
\textbf{else}\ \mathtt{true}
\end{array}\right) \\
\mathsf{stconst} = \mathtt{const\ int}\ \mathsf{uname}(\mathsf{trigger.event}, \mathsf{stnumber}) = \mathsf{stnumber}; \\
\mathsf{notupdatinggrd} = \left(\begin{array}{l} \textbf{if}\ \mathsf{isStandaloneEvent}(\mathsf{trigger.event}, \mathsf{false})\ \textbf{then}\ \mathtt{true} \\ \textbf{else}\ \mathtt{FIN\_\_}\mathsf{ename} = \mathtt{true} \end{array}\right) \\
\mathsf{finassign} = \mathtt{FIN\_\_}\mathsf{ename}' = \mathtt{false} \\
\mathsf{scpcassign} = \mathtt{scpcname}' = \mathtt{stconst.name} \\
\mathsf{cmd1} = \left(\begin{array}{l}
[\mathsf{ename}]\left(\mathsf{andExprs}^{15}\left(\begin{array}{l} \mathsf{lockfreegrd} \cup \mathsf{scpcgrd} \cup \\ \{\mathsf{condgrd}, \mathsf{notupdatinggrd}\} \end{array}\right)\right) \\
\quad \rightarrow \mathtt{1.0} : \mathsf{lockassign} \cup \{\mathsf{scpcassign}, \mathsf{finassign}\};
\end{array}\right) \\
\mathsf{scpcgrd2} = \mathsf{pcconstrs} \cup \{\mathsf{scpcname} = \mathsf{stconst.name}\} \\
\mathsf{trigvarassign} = \mathsf{id}(\mathsf{trigger.parameter})' = \mathtt{EVT\_\_}\mathsf{ename} \\
\mathsf{finassign2} = \mathtt{FIN\_\_}\mathsf{ename}' = \mathtt{true} \\
\mathsf{cmd2} = []\,\mathsf{andExprs}^{16}(\mathsf{scpcgrd2}) \rightarrow \mathtt{1.0} : \mathsf{assigns} \cup \{\mathsf{trigvarassign}, \mathsf{finassign2}\};
\end{array}
$$

Rule 165. Output Events

$$\left[\!\!\left[\begin{array}{l} \text{trigger} : \text{Communication}, \text{condition} : \text{Expression}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{pr}, \\ \text{lockfreegrd} : \mathbb{P}\,\text{BoolExpr}_{pr}, \text{curstate} : \text{String}, \text{scpcname} : \text{Name}, \text{stnumber} : \text{int}, \\ \text{lockassign} : \mathbb{P}\,\text{Assignment}_{pr}, \text{assigns} : \mathbb{P}\,\text{Assignment}_{pr}, \\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String} \end{array}\right]\!\!\right]_{\mathcal{O}utput\mathcal{E}vent}$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$$\begin{array}{l}
\textbf{if}\ \text{isEventToRP}(\text{trigger.event})\ \textbf{then} \\
\quad (\text{stnumber}, \emptyset, \{\text{cmd}\}) \\
\quad \textbf{where} \\
\qquad \text{cmd} = \left(\begin{array}{l} [\text{ename}]\left(\text{andExprs}^{17}\,(\text{lockfreegrd} \cup \text{scpcgrd} \cup \{\text{condgrd}\})\right) \\ \quad \rightarrow \texttt{1.0} : \text{assigns} \cup \text{lockassign} \cup \{\text{trigvarassign}\}; \end{array} \right) \\
\textbf{else} \\
\quad (\text{stnumber} + 1, \{\text{stconst}\}, \{\text{cmd1}, \text{cmd2}\}) \\
\quad \textbf{where} \\
\qquad \text{stconst} = \texttt{const int}\ \text{uname}(\text{trigger.event}, \text{stnumber}) = \text{stnumber}; \\
\qquad \text{notupdatinggrd} = \left(\begin{array}{l} \textbf{if} \left(\begin{array}{l} \text{isStandaloneEvent}(\text{trigger.event}, \text{false}) \vee \\ \text{isEventToBuffer}(\text{trigger.event}) \end{array} \right) \textbf{then}\ \texttt{true} \\ \hline \textbf{else}\ \texttt{FIN__}\text{ename} = \texttt{true} \end{array} \right) \\
\qquad \text{cmd1} = \left(\begin{array}{l} [\text{ename}]\left(\text{andExprs}^{18}\left(\begin{array}{l} \text{lockfreegrd} \cup \text{scpcgrd} \cup \\ \{\text{condgrd}, \text{notupdatinggrd}\} \end{array} \right)\right) \\ \quad \rightarrow \texttt{1.0} : \text{lockassign} \cup \{\text{scpcassign}, \text{trigvarassign}\}; \end{array} \right) \\
\qquad \text{scpcgrd2} = \text{pcconstrs} \cup \{\text{scpcname} = \text{stconst.name}\} \\
\qquad \text{cmd2} = [\,]\,\text{andExprs}^{19}\,(\text{scpcgrd2}) \rightarrow \texttt{1.0} : \text{assigns}; \\
\textbf{where} \\
\quad \text{ename} = \text{namemap}\,(\text{trigger.event} \mapsto \text{true}) \\
\quad \text{scpcgrd} = \text{pcconstrs} \cup \{\text{scpcname} = \text{curstate}\} \\
\quad \text{condgrd} = \textbf{if}\ \text{condition} \neq \text{null}\ \textbf{then}\ [\![\text{condition}]\!]_e\ \textbf{else}\ \texttt{true} \\
\quad \text{trigvarassign} = \texttt{EVT__}\text{ename}' = [\![\text{trigger.value}]\!]_e
\end{array}$$

Rule 166. Simple Events

$$\left[\!\!\left[\begin{array}{l} \text{trigger} : \text{Communication}, \text{condition} : \text{Expression}, \text{pcconstrs} : \mathbb{P}\,\text{BoolExpr}_{pr}, \\ \text{lockfreegrd} : \mathbb{P}\,\text{BoolExpr}_{pr}, \text{curstate} : \text{String}, \text{scpcname} : \text{Name}, \text{stnumber} : \text{int}, \\ \text{assigns} : \mathbb{P}\,\text{Assignment}_{pr}, \text{output} : \text{Boolean}, \\ \text{namemap} : (\text{Event}_{rc} \times \text{Boolean}) \nrightarrow \text{String} \end{array}\right]\!\!\right]_{\mathcal{S}imp\mathcal{E}vent}$$

$$: \text{int} \times \mathbb{P}\,\text{Constant} \times \mathbb{P}\,\text{Command} =$$

$$\begin{array}{l}
(\text{stnumber}, \emptyset, \{\text{cmd}\}) \\
\textbf{where} \\
\quad \text{ename} = \text{namemap}\,(\text{trigger.event} \mapsto \text{output}) \\
\quad \text{scpcgrd} = \text{pcconstrs} \cup \{\text{scpcname} = \text{curstate}\} \\
\quad \text{condgrd} = \textbf{if}\ \text{condition} \neq \text{null}\ \textbf{then}\ [\![\text{condition}]\!]_e\ \textbf{else}\ \texttt{true} \\
\quad \text{cmd} = [\text{ename}]\left(\text{andExprs}^{20}\left(\begin{array}{l} \text{lockfreegrd} \cup \text{scpcgrd} \cup \\ \{\text{condgrd}\} \end{array} \right)\right) \rightarrow \texttt{1.0} : \text{assigns};
\end{array}$$

Rule 167. Empty Events

$$\left[\!\!\left[\begin{array}{l} \mathsf{condition} : \mathsf{Expression}, \mathsf{pcconstrs} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \\ \mathsf{lockfreegrd} : \mathbb{P}\,\mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{curstate} : \mathsf{String}, \mathsf{scpcname} : \mathsf{Name}, \mathsf{stnumber} : \mathsf{int}, \\ \mathsf{assigns} : \mathbb{P}\,\mathsf{Assignment}_{\mathsf{pr}} \end{array}\right]\!\!\right]_{\mathcal{E}mpty\mathcal{E}vent}$$

$$: \mathsf{int} \times \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{Command} =$$

$$\underline{(\mathsf{stnumber}, \emptyset, \{\mathsf{cmd}\})}$$

where

$$\underline{\mathsf{scpcgrd} = \mathsf{pcconstrs} \cup \{\mathsf{scpcname} = \mathsf{curstate}\}}$$

$$\underline{\mathsf{condgrd} = \mathbf{if}\ \mathsf{condition} \neq \mathsf{null}\ \mathbf{then}\ [\![\mathsf{condition}]\!]_e\ \mathbf{else}\ \mathtt{true}}$$

$$\underline{\mathsf{cmd} = []\left(\mathsf{andExprs}^{21}\left(\ \mathsf{lockfreegrd} \cup \mathsf{scpcgrd} \cup \{\mathsf{condgrd}\}\ \right)\right)} \rightarrow \mathtt{1.0} : \underline{\mathsf{assigns}};$$

Operations

In RoboChart, an operation can be introduced in several ways:

OP-1 provided by a robotic platform, and its reference is OperationSig in a model;

OP-2 defined fully (using a state machine, and so the number of nodes is larger than 0), in the model. The definition is of type OperationDef, inherited from OperationSig; and

OP-3 defined partial (using a state machine but no detail is given, and so the number of nodes is 0), in the model. The definition is of type OperationDef.

The function isFullyDefinedOperation[1] determines if an operation is OP-2 or OP-3.

Rule 168. Is an operation fully defined

$$\underline{\mathsf{isFullyDefinedOperation}(\mathsf{opdef} : \mathsf{OperationDef}) : \mathsf{Boolean} =}$$

$$\underline{\mathbf{if}\ \#(\mathsf{opdef.nodes}) > 0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false}}$$

The function getAllCalledOperations returns all operations that are called in cs, a StateMachineBody, and provided by a robotic platform. If cs is

1. a StateMachineDef, recursively visit all required operations that are OP-2 or OP-3, but exclude all operations that are OP-1 because they might be not called in actions;
2. a OperationDef and OP-3, just all required operations in cs that are OP-1;
3. a OperationDef and OP-2, just all operations that are called in cs and provided by a robotic platform;

Rule 169. Get all called operations (that are provided by a robotic platform) in a StateMachineBody

$\text{getAllCalledOperations}(\text{reqallops} : \mathbb{P}\,\text{OperationSig}) : \text{cs} : \text{StateMachineBody} \rightarrow \mathbb{P}\,\text{OperationSig} =$

$\textbf{if}\ \text{cs} \in \text{StateMachineDef}\ \textbf{then}$

$\quad \bigcup \left(\text{getAllCalledOperations}^2(\text{reqallops})(\!|\ \text{reqopdefs}\ |\!) \right) \cup \text{getCalledOperations}^1(\text{cs}, \text{reqallops})$

$\quad \textbf{where}$

$\qquad \text{reqops} = \text{getRequiredOperations}^5(\text{cs})$

$\qquad \text{reqopdefs} = \{\text{op} : \text{reqallops} \mid (\exists \text{rop} : \text{reqops} \bullet \text{rop.name} = \text{op.name}) \wedge \text{op} \in \text{OperationDef}\}$

$\textbf{else if}\ \text{cs} \in \text{OperationDef} \wedge \neg\ \text{isFullyDefinedOperation}^2(\text{cs})\ \textbf{then}$

$\quad \{\text{rop} : \text{reqops} \mid (\exists \text{op} : \text{reqallops} \bullet \text{rop.name} = \text{op.name} \wedge \text{op} \notin \text{OperationDef})\}$

$\quad \textbf{where}$

$\qquad \text{reqops} = \text{getRequiredOperations}^6(\text{cs})$

$\textbf{else if}\ \text{cs} \in \text{OperationDef} \wedge \text{isFullyDefinedOperation}^3(\text{cs})\ \textbf{then}$

$\quad \text{getCalledOperations}^2(\text{cs}, \text{reqallops})$

$\textbf{else}$

$\quad \emptyset$

Rule 170. Get called operations (that are provided by a robotic platform) in a NodeContainer

$\text{getCalledOperations}(\text{cs} : \text{NodeContainer}, \text{reqallops} : \mathbb{P}\,\text{OperationSig}) : \mathbb{P}\,\text{OperationSig} =$

$\quad \text{calledops}$

$\textbf{where}$

$\quad \text{actions1} = \bigcup \{\text{n} : \text{cs.nodes} \mid \text{n} \in \text{State} \bullet \{\text{act} : \text{n.actions} \mid \text{act} \in \text{EntryAction} \vee \text{act} \in \text{ExitAction}\}\}$

$\quad \text{actions2} = \{\text{t} : \text{cs.transitions} \mid \text{t.action} \neq \text{null} \bullet \text{t.action}\}$

$\quad \text{calledops} = \bigcup \left(\begin{array}{l} \text{getCalledOperationsInStmt}^1(\text{reqallops}) \\ (\!|\ \{\text{act} : \text{actions1} \cup \text{actions2} \bullet \text{act.action}\}\ |\!) \end{array} \right)$

Rule 171. Get called operations (that are provided by a robotic platform) in a statement

$\text{getCalledOperationsInStmt}(\text{reqallops} : \mathbb{P}\,\text{OperationSig}) : \text{stmt} : \text{Statement}_{\text{rc}} \rightarrow \mathbb{P}\,\text{OperationSig} =$

$\textbf{if}\ \text{stmt} \in \text{IfStmt}\ \textbf{then}$

$\quad \text{thenops} \cup \text{elseops}$

$\quad \textbf{where}$

$\qquad \text{thenops} = \text{getCalledOperationsInStmt}^2(\text{stmt.then}, \text{reqallops})$

$\qquad \text{elseops} = \text{getCalledOperationsInStmt}^3(\text{stmt.else}, \text{reqallops})$

$\textbf{else if}\ \text{stmt} \in \text{SeqStatement}\ \textbf{then}$

$\quad \bigcup \left(\text{getCalledOperationsInStmt}^4(\text{reqallops})(\!|\ \text{stmt.statements}\ |\!) \right)$

$\textbf{else if}\ \text{stmt} \in \text{Call}\ \textbf{then}$

$\quad \{\text{op} : \text{reqallops} \mid \text{op.name} = \text{stmt.operation.name} \wedge \text{op} \notin \text{OperationDef}\}$

$\textbf{else}$

$\quad \emptyset$

Rule 172. Operation definitions

$$[\![\mathsf{opdef} : \mathsf{OperationDef}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \rightarrow \mathsf{String}]\!]_{\mathcal{OP}}$$
$$: \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathbb{P}\,\mathsf{Command} =$$

$$\textbf{if}\ \mathsf{isFullyDefinedOperation}^{4}(\mathsf{opdef})\ \textbf{then}$$
$$[\![\mathsf{opdef}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{STB}^{2}}$$
$$\textbf{else}$$
$$[\![\mathsf{opdef}, \mathsf{ctrlopdefs}, \mathsf{namemap}]\!]_{\mathcal{OPpar}^{1}}$$

Rule 173. Partially defined operation definitions

$$\left[\!\!\left[\begin{array}{l} \mathsf{opdef} : \mathsf{OperationDef}, \mathsf{ctrlopdefs} : \mathbb{P}\,\mathsf{OperationDef}, \\ \mathsf{namemap} : (\mathsf{Event}_{\mathsf{rc}} \times \mathsf{Boolean}) \nrightarrow \mathsf{String} \end{array}\right]\!\!\right]_{\mathcal{OP}par}$$

$$: \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} \times \mathbb{P}\,\mathsf{Command} =$$

$$\left(\begin{array}{l} \bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.1}\} \cup \left\{ \begin{array}{l} \mathsf{const0}, \mathsf{constterm}, \mathsf{construnning}, \mathsf{conststop}, \mathsf{constnotcomp} \end{array} \right\}, \\ \bigcup\{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.2}\} \cup (\textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then}\ \{\mathsf{bounditer}\}\ \textbf{else}\ \emptyset) \cup \{\mathsf{scpc}\}, \\ \mathsf{varcmds} \cup \mathsf{opcmds} \cup \mathsf{eventcmds} \cup \{\mathsf{stopcmd}, \mathsf{termcmd}\} \cup \mathsf{termcmds} \end{array}\right)$$

where

$$\mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}^5}\ (\!|\ \mathsf{getVariableLists}^5(\mathsf{opdef})\ |\!)$$

$$\mathsf{bound} = \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then}\ \mathsf{OP\_BOUND}\ \textbf{else}\ 0$$

$$\mathsf{bounditer} = \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then}\ \mathsf{id}(\mathsf{opdef})\texttt{\_ITER} : [0..\mathsf{bound}]\ \texttt{init}\ 0;\ \textbf{else}\ \mathsf{null}$$

$$\mathsf{const0} = \texttt{const int}\ \mathsf{id}(\mathsf{opdef})\texttt{\_INACTIVE} = 0;$$

$$\mathsf{constterm} = \texttt{const int}\ \mathsf{id}(\mathsf{opdef})\texttt{\_TERMINATED} = 1;$$

$$\mathsf{construnning} = \texttt{const int}\ \mathsf{id}(\mathsf{opdef})\texttt{\_RUNNING} = 2;$$

$$\mathsf{conststop} = \texttt{const int}\ \mathsf{id}(\mathsf{opdef})\texttt{\_STOP} = 3;$$

$$\mathsf{constnotcomp} = \texttt{const int}\ \mathsf{id}(\mathsf{opdef})\texttt{\_NOTCOMPLETED} = 4;$$

$$\mathsf{scpc} = \mathsf{id}(\mathsf{opdef})\texttt{\_scpc} : [0..4] = \mathsf{const0.name};$$

$$\mathsf{inactivegrd} = \mathsf{scpc.name} = \mathsf{const0.name}$$

$$\mathsf{inactiveassign} = \mathsf{scpc.name}' = \mathsf{const0.name}$$

$$\mathsf{runninggrd0} = \mathsf{scpc.name} = \mathsf{construnning.name}$$

$$\mathsf{runningassign} = \mathsf{scpc.name}' = \mathsf{construnning.name}$$

$$\mathsf{notcompgrd} = \mathsf{scpc.name} = \mathsf{constnotcomp.name}$$

$$\mathsf{notcompassign} = \mathsf{scpc.name}' = \mathsf{constnotcomp.name}$$

$$\mathsf{stopassign} = \mathsf{scpc.name}' = \mathsf{conststop.name}$$

$$\mathsf{termassign} = \mathsf{scpc.name}' = \mathsf{constterm.name}$$

$$\mathsf{runninggrdbound} = \left(\begin{array}{l} \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then} \\ \quad \mathsf{runninggrd0}\ \&\ (\mathsf{bounditer.name} = \mathsf{bound}) \\ \textbf{else}\ \texttt{true} \end{array}\right)$$

$$\mathsf{runninggrd} = \left(\begin{array}{l} \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then} \\ \quad \mathsf{runninggrd0}\ \&\ (\mathsf{bounditer.name} < \mathsf{bound}) \\ \textbf{else}\ \mathsf{runninggrd} \end{array}\right)$$

$$\mathsf{assigns} = \left(\ \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then}\ \ \{\mathsf{bounditer.name}' = \mathsf{bounditer.name} + 1\}\ \textbf{else}\ \emptyset\ \right)$$

$$\mathsf{varsinscope} = \left\{\mathsf{vl} : \mathsf{getVariableLists}^6(\mathsf{opdef}) \cup \mathsf{getRequiredVariableLists}^1(\mathsf{opdef}) \bullet \mathsf{vl.vars}\right\}$$

$$\mathsf{varcmds} = \left\{\begin{array}{l} \mathsf{v} : \mathsf{varsinscope} \mid \neg\,\mathsf{isConstant}^1(\mathsf{v}) \bullet \\ \quad \left\{\begin{array}{l} \mathsf{val} : \mathsf{enumValuesInType}^1([\![\mathsf{v.type}]\!]_e) \bullet \\ \quad [\,]\ \mathsf{runninggrd} \rightarrow \texttt{1.0} : \mathsf{assigns}\ \&\ \left(\mathsf{id}(\mathsf{v})' = \texttt{val}\right); \end{array}\right\} \end{array}\right\}$$

$$\mathsf{rops} = \mathsf{getRequiredOperations}^7(\mathsf{opdef})$$

$$\mathsf{opcmds} = \bigcup\left\{\mathsf{op} : \mathsf{rops} \bullet \mathsf{callOpNondeter}^1\left(\begin{array}{l} \mathsf{op}, \mathsf{opdef}, \mathsf{ctrlopdefs}, \mathsf{runninggrd}, \\ \mathsf{runningassign}, \mathsf{notcompassign}, \mathsf{assigns} \end{array}\right)\right\}$$

$$\mathsf{events} = \mathsf{getEvents}^2(\mathsf{opdef})$$

$$\mathsf{eventcmds} = \bigcup\left\{\mathsf{e} : \mathsf{event} \bullet \mathsf{occurEventNondeter}^1\left(\begin{array}{l} \mathsf{e}, \mathsf{opdef}, \mathsf{runninggrd}, \mathsf{runningassign}, \\ \mathsf{notcompassign}, \mathsf{assigns}, \mathsf{namemap} \end{array}\right)\right\}$$

$$\mathsf{stopcmd} = [\,]\ \mathsf{runninggrd} \rightarrow \texttt{1.0} : \mathsf{assigns} \cup \{\mathsf{stopassign}\};$$

$$\mathsf{termcmd} = [\,]\ \mathsf{runninggrd} \rightarrow \texttt{1.0} : \mathsf{assigns} \cup \{\mathsf{termassign}\};$$

$$\mathsf{termcmds} = \left(\ \textbf{if}\ \mathsf{opdef.terminates}\ \textbf{then}\ \ [\,]\ \mathsf{runninggrdbound} \rightarrow \texttt{1.0} : \mathsf{termassign};\ \textbf{else}\ \emptyset\ \right)$$

Rule 174. Call Operations nondeterministically

$$\text{callOpNondeter}\left(\begin{array}{l}\text{rop}:\text{OperationSig},\text{opdef}:\text{OperationDef},\text{ctrlopdefs}:\mathbb{P}\,\text{OperationDef},\\ \text{runninggrd}:\text{BoolExpr}_{pr},\text{runningassign}:\text{Assignment}_{pr},\\ \text{notcompassign}:\text{Assignment}_{pr},\text{assigns}:\mathbb{P}\,\text{Assignment}_{pr}\end{array}\right)$$

$$:\mathbb{P}\,\text{Command}=$$

$$\begin{array}{l}
\textbf{if}\ \text{ropdef}\neq\text{null}\ \textbf{then}\\
\quad\textbf{if}\ \text{allopvarassigns}=\emptyset\ \textbf{then}\\
\qquad\text{callOpdef}^{2}(\text{ropdef},\text{runninggrd},\{\text{notcompassign}\},\text{assigns}\cup\{\text{runningassign}\})\\
\quad\textbf{else}\\
\qquad\bigcup\left\{\ \text{opassigns}:\text{allopvarassigns}\bullet\text{callOpdef}^{3}\left(\begin{array}{l}\text{ropdef},\text{runninggrd},\\ \text{opassigns}\cup\{\text{notcompassign}\},\\ \text{assigns}\cup\{\text{runningassign}\}\end{array}\right)\right\}\\
\textbf{else}\\
\quad\textbf{if}\ \text{allopvarassigns}=\emptyset\ \textbf{then}\\
\qquad\left\{[\text{id}(\text{op})]\,\text{runninggrd}\rightarrow\textbf{if}\ \text{opdef.terminates}\ \textbf{then}\ 1.0:\text{assigns}\ \textbf{else}\ \texttt{true};\right\}\\
\quad\textbf{else}\\
\qquad\bigcup\left\{\ \text{opassigns}:\text{allopvarassigns}\bullet[\text{id}(\text{op})]\,\text{runninggrd}\rightarrow 1.0:\text{assigns}\cup\text{opassigns};\ \right\}\\
\textbf{where}\\
\quad\text{ropdef}=\text{getOperationDef}^{2}(\text{rop},\text{ctrlopdefs})\\
\quad\text{allopvarassigns}=\text{permOPParameterAssigns}^{1}(\text{op},\text{op.parameters})
\end{array}$$

Rule 175. Get operation definition

$$\text{getOperationDef}(\text{op}:\text{OperationSig},\text{ctrlopdefs}:\mathbb{P}\,\text{OperationDef}):\text{OperationDef}=$$

$$\begin{array}{l}
\textbf{if}\ (\text{opdefs}=\emptyset)\ \textbf{then}\\
\quad\text{null}\\
\textbf{else}\\
\quad\mu\,\text{o}:\text{ctrlopdefs}\mid\text{o.name}=\text{op.name}\\
\textbf{where}\\
\quad\text{opdefs}=\{\text{o}:\text{ctrlopdefs}\mid\text{o.name}=\text{op.name}\}
\end{array}$$

Rule 176. Call Operation definitions

$$\text{callOpdef}\left(\begin{array}{l}\text{opdef}:\text{OperationDef},\text{pcconstrs}:\mathbb{P}\,\text{BoolExpr}_{pr},\\ \text{opvarassigns}:\mathbb{P}\,\text{Assignment}_{pr},\text{assigns}:\mathbb{P}\,\text{Assignment}_{pr}\end{array}\right):\mathbb{P}\,\text{Command}=$$

$$\begin{array}{l}
\{\text{cmd1},\text{cmd2}\}\\
\textbf{where}\\
\quad\text{ass1}=\text{id}(\text{opdef})_\texttt{scpc}'=\left(\begin{array}{l}\textbf{if}\ \text{isFullyDefinedOperation}^{5}(\text{opdef})\ \textbf{then}\\ \quad\text{id}(\text{opdef})_\text{getInitial}^{4}(\text{opdef}).\text{name}\\ \textbf{else}\ \text{id}(\text{opdef})_\text{RUNNING}\end{array}\right)\\
\quad\text{opinactive}=\text{id}(\text{opdef})_\texttt{scpc}=\text{id}(\text{opdef})_\texttt{INACTIVE}\\
\quad\text{cmd1}=\ [\,](\text{andExprs}^{22}(\text{pcconstrs}\cup\{\text{opinactive}\})\rightarrow 1.0:\text{opvarassigns}\cup\{\text{ass1}\};\\
\quad\text{ass2}=\text{id}(\text{opdef})_\texttt{scpc}'=\text{id}(\text{opdef})_\texttt{INACTIVE}\\
\quad\text{opterminated}=\text{id}(\text{opdef})_\texttt{scpc}=\text{id}(\text{opdef})_\texttt{TERMINATED}\\
\quad\text{cmd2}=\ [\,](\text{andExprs}^{23}(\text{pcconstrs}\cup\{\text{opterminated}\})\rightarrow 1.0:\text{assigns}\cup\{\text{ass2}\};
\end{array}$$

Rule 177. Events occur nondeterministically

$$\text{occurEventNondeter}\left(\begin{array}{l}\text{event} : \text{Event}, \text{opdef} : \text{OperationDef}, \text{runninggrd} : \text{BoolExpr}_{\text{pr}},\\ \text{runningassign} : \text{Assignment}_{\text{pr}}, \text{notcompassign} : \text{Assignment}_{\text{pr}},\\ \text{assigns} : \mathbb{P}\,\text{Assignment}_{\text{pr}}, \text{namemap} : (\text{Event}_{\text{rc}} \times \text{Boolean}) \nrightarrow \text{String}\end{array}\right)$$

$$: \mathbb{P}\,\text{Command} =$$

$$\text{inputcmds} \cup \text{outputcmds}$$

where

$$\text{forinput} = (\text{event}, \text{false}) \in \text{dom namemap}$$

$$\text{foroutput} = (\text{event}, \text{true}) \in \text{dom namemap}$$

$$\text{einame} = \text{namemap}(\text{event} \mapsto \text{false})$$

$$\text{eoname} = \text{namemap}(\text{event} \mapsto \text{true})$$

$$\text{inputcmds} = \left(\begin{array}{l}\textbf{if}\ \text{forinput}\ \textbf{then}\\ \quad \left\{\begin{array}{l}[\text{einame}]\,\text{runninggrd}\\ \quad \rightarrow \textbf{if}\ \text{opdef.terminates}\ \textbf{then}\ \mathtt{1.0} : \text{assigns}\ \textbf{else}\ \mathtt{true}\,;\end{array}\right\}\\ \textbf{else}\ \emptyset\end{array}\right)$$

$$\text{outputcmds} = \left(\begin{array}{l}\textbf{if}\ \text{foroutput} \wedge \text{event.type} = \text{null}\ \textbf{then}\\ \quad \left\{\begin{array}{l}[\text{eoname}]\,\text{runninggrd}\\ \quad \rightarrow \textbf{if}\ \text{opdef.terminates}\ \textbf{then}\ \mathtt{1.0} : \text{assigns}\ \textbf{else}\ \mathtt{true}\,;\end{array}\right\}\\ \textbf{else if}\ \text{foroutput} \wedge \text{event.type} \neq \text{null}\ \textbf{then}\\ \quad \left\{\begin{array}{l}\text{v} : \text{enumValuesInType}^2\,(\llbracket \text{event.type} \rrbracket_t) \bullet\\ \quad \left(\begin{array}{l}[\text{eoname}]\,\text{runninggrd}\\ \quad \rightarrow \mathtt{1.0} : \text{assigns} \,\&\, (\mathtt{EVT\_\_}\text{eoname}' = \mathtt{v})\,;\end{array}\right)\end{array}\right\}\\ \textbf{else}\ \emptyset\end{array}\right)$$

Auxiliary functions

The function getVariableLists, defined in Rule 77, returns a set of VariableList provided or defined in a Context.

Rule 178. Get variable lists

$$\text{getVariableLists}\,(\text{c} : \text{Context}) : \mathbb{P}\,\text{VariableList} =$$

$$(\text{c.variableList} \cup \text{vl}_{\text{p}} \cup \text{vl}_{\text{d}})$$

where

$$\text{vl}_{\text{p}} = \{\text{i} : \text{c.pInterfaces} \bullet \text{i.variableList}\}$$

$$\text{vl}_{\text{d}} = \{\text{i} : \text{c.interfaces} \bullet \text{i.variableList}\}$$

Rule 179. Get required variable lists

$$\text{getRequiredVariableLists}\,(\text{c} : \text{Context}) : \mathbb{P}\,\text{VariableList} =$$

$$\{\text{i} : \text{c.rInterfaces} \bullet \text{i.variableList}\}$$

Rule 180. Is a constant variable

isConstant (v : Variable) : Boolean =

if (v.modifier = CONST) **then** true **else** false

Rule 181. Get events

getEvents (c : Context) : $\mathbb{P}$ Event =

(c.events $\cup$ ev_d)

where

ev_d = $\bigcup$ {i : c.interfaces • i.events}

Rule 182. Get required operations

getRequiredOperations (c : Context) : $\mathbb{P}$ OperationSig =

$\bigcup$ {i : c.rInterfaces • i.operations}

Rule 183. Get the exit action

getExitAction (s : State) : Action =

if (exitactions = $\emptyset$) **then**

null

else

μ a : s.actions | a $\in$ ExitAction

where

exitactions = {a : s.actions | a $\in$ ExitAction}

Rule 184. Has the exit action

hasExitAction (s : State) : Boolean =

if (getExitAction^3(s) = null) **then**

false

else

true

Rule 185. Get the entry action

getEntryAction (s : State) : Action =

if (entryActions = ∅) **then**
null
else
μ a : s.actions | a ∈ EntryAction
where
entryactions = {a : s.actions | a ∈ EntryAction}

Rule 186. Has the entry action

hasEntryAction (s : State) : Boolean =

if (getEntryAction^{2}(s) = null) **then**
false
else
true

Rule 187. Conjunction of boolean expressions

andExprs (exprs : ℙ BoolExpr$_{pr}$) : BoolExpr$_{pr}$ =

if (exprs = ∅) **then**
`true`
else
expr & andExprs^{24}(exprs \ {expr})
where
expr ∈ exprs

Rule 188. Is a state composite

isComposite (s : State) : boolean =

if (s.nodes ≠ ∅) **then** true **else** false

Rule 189. Get the initial junction of a node container

getInitial (s : NodeContainer) : Initial =

(μ n : s.nodes | n ∈ Initial)

Rule 190. Get the final state of a node container

getFinal (s : NodeContainer) : Final =

(μ n : s.nodes | n ∈ Final)

The function getParaArgPairs in Rule 90 assembles a set of pairs from a sequence of parameters and a sequence of arguments, provided they have the same length.

Rule 191. Get parameters and arguments pair

$\mathsf{getParaArgPairs}(\mathsf{paras} : \mathrm{seq}\ \mathsf{Parameter}, \mathsf{args} : \mathrm{seq}\ \mathsf{Expression}) : \mathsf{Parameter} \leftrightarrow \mathsf{Expression} =$

$\quad \textbf{if}\ \#\mathsf{paras} = 0\ \textbf{then}\ \emptyset[\mathsf{Parameter} \times \mathsf{Expression}]$

$\quad \textbf{else}\ \mathsf{getParaArgPairs}^2(\mathsf{tail\,paras}, \mathsf{tail\,args}) \oplus \{\mathsf{head\,paras} \mapsto \mathsf{head\,args}\}$

The function enumValuesInType, defined in Rule 91, lists all possible values of a PRISM type. In the rule, vt. NAT_MIN, NAT_MAX, INT_MIN, INT_MAX, REAL_MIN, and REAL_MAX are the parameters to the translation.

Rule 192. Enumerate all values in a type

$\mathsf{enumValuesInType}(\mathsf{vt} : \mathsf{Type}_{\mathsf{pr}}) : \mathbb{P}\ \mathsf{Expr} =$

$\quad \textbf{if}\ \mathsf{vt} \in \mathsf{IntType}_{\mathsf{pr}}\ \textbf{then}$

$\quad\quad \textbf{if}\ \mathsf{vt.nat}\ \textbf{then}\ \left\{\mathsf{v} : \mathsf{NAT\_MIN} \mathinner{..} \mathsf{NAT\_MAX} \bullet \lfloor \mathsf{value} \rightsquigarrow \mathsf{v} \rfloor_{\mathsf{IntLitExpr}}\right\}$

$\quad\quad \textbf{else}\ \left\{\mathsf{v} : \mathsf{INT\_MIN} \mathinner{..} \mathsf{INT\_MAX} \bullet \lfloor \mathsf{value} \rightsquigarrow \mathsf{v} \rfloor_{\mathsf{IntLitExpr}}\right\}$

$\quad \textbf{else if}\ \mathsf{vt} \in \mathsf{DoubleType}_{\mathsf{pr}}\ \textbf{then}\ \left\{\mathsf{v} : \mathsf{REAL\_MIN} \mathinner{..} \mathsf{REAL\_MAX} \bullet \lfloor \mathsf{value} \rightsquigarrow \mathsf{v} \rfloor_{\mathsf{IntLitExpr}}\right\}$

$\quad \textbf{else if}\ \mathsf{vt} \in \mathsf{RangeType}_{\mathsf{pr}}\ \textbf{then}\ \left\{\mathsf{v} : \mathsf{vt.bottom} \mathinner{..} \mathsf{vt.top} \bullet \lfloor \mathsf{value} \rightsquigarrow \mathsf{v} \rfloor_{\mathsf{IntLitExpr}}\right\}$

$\quad \textbf{else if}\ \mathsf{vt} \in \mathsf{BoolType}_{\mathsf{pr}}\ \textbf{then}\ \left\{\lfloor \mathsf{value} \rightsquigarrow \mathsf{true} \rfloor_{\mathsf{BoolLitExpr}}, \lfloor \mathsf{value} \rightsquigarrow \mathsf{false} \rfloor_{\mathsf{BoolLitExpr}}\right\}$

$\quad \textbf{else}\ \mathrm{Error:\ Not\ yet\ supported!}$

Rule 193. All possible combinations of assignments to parameters of an operation

$\mathsf{permOPParameterAssigns}(\mathsf{op} : \mathsf{OperationSig}, \mathsf{paras} : \mathrm{seq}\ \mathsf{Parameter}) : \mathbb{P}\ (\mathbb{P}\ \mathsf{Assignment}) =$

$\quad \textbf{if}\ \#\mathsf{paras} = 0\ \textbf{then}\ \emptyset$

$\quad \textbf{else}$

$\quad\quad \textbf{let}\ \mathsf{tassigns} == \mathsf{permParameterAssigns}(\mathsf{op}, \mathsf{tail\,paras});$

$\quad\quad\quad \mathsf{para} == \mathsf{head\,paras};$

$\quad\quad\quad \mathsf{values} == \mathsf{enumValuesInType}^3([\![\mathsf{para.type}]\!]_t);$

$\quad\quad\quad \mathsf{hassigns} == \left\{\mathsf{v} : \mathsf{values} \bullet \mathsf{id}(\mathsf{op}, \mathsf{para})' = \mathsf{v}\right\};$

$\quad\quad\quad \bullet \{\mathsf{ta} : \mathsf{tassigns};\ \mathsf{ha} : \mathsf{hassigns} \bullet \mathsf{ta} \cup \{\mathsf{ha}\}\}$

Variables

Rule 194. Variable lists

$[\![\mathsf{vl} : \mathsf{VariableList}]\!]_{\mathcal{VL}} : \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{VarDecl} =$

$\quad \left(\bigcup \mathsf{retvars.1}, \bigcup \mathsf{retvars.2}\right)$

$\textbf{where}$

$\quad \mathsf{retvars} = \{\mathsf{v} : \mathsf{vl.vars} \bullet [\![\mathsf{v}]\!]_{\mathcal{V}'}\}$

Rule 195. Variables

$\llbracket \mathsf{v} : \mathsf{Variable} \rrbracket_{\mathcal{V}} : \mathbb{P}\,\mathsf{Constant} \times \mathbb{P}\,\mathsf{VarDecl} =$

$\quad$ **if** $\mathsf{v.modifier} = \text{“const”}$

$\quad\quad \left(\left\{ \texttt{const}\ \llbracket \mathsf{v.type} \rrbracket_t\ \texttt{id(v)} = \llbracket \mathsf{v.initial} \rrbracket_e\ \texttt{;} \right\}, \emptyset \right)$

$\quad$ **else if** $\mathsf{v.modifier} = \text{“var”}$

$\quad\quad \left(\emptyset, \left\{ \texttt{id(v)} : \llbracket \mathsf{v.type} \rrbracket_t = \llbracket \mathsf{v.initial} \rrbracket_e\ \texttt{;} \right\} \right)$

Expressions

The translation of expressions (including types) is standard. We note, however, that the expression language of Z is much richer than that of PRISM. So, in some cases, we require user input to translate the expressions.

II Assertions

6. Assertion DSL Syntax

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

6.1 Standard Assertions

Syntax — Assertions.

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

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

Syntax — Specification.

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

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

Syntax — CSP Models.

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

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

Syntax — Constant Definitions.

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

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

Syntax — CSP specification.

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

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

6.2 Probabilistic Properties

Syntax — Probabilistic Statements.

```
ProbStatements   ::=  ProbStatement*
ProbStatement    ::=  Constant
                   |  Constants
                   |  Label
                   |  Formula
                   |  Rewards
                   |  Definitions
                   |  pModules
                   |  ProbProperty
```

The syntax of probabilistic properties consists of various probabilistic statements:

- constant declarations (Constant in Syntax 6.2.2),
- constant configurations (Constants in Syntax 6.2.2),
- labels (Label in Syntax 6.2.3),
- formulas (Formula in Syntax 6.2.3),
- rewards (Rewards in Syntax 6.2.4),
- function and operation definitions (Definitions in Syntax 6.2.5),
- probabilistic PRISM modules (pModules in Syntax 6.2.6),
- probabilistic properties (ProbProperty in Syntax 6.2.1).

6.2.1 Probability Properties

Syntax — Probabilistic Properties.

```
ProbProperty  ::=  'prob' 'property' N ':' pExpr
                   ('with' 'constants' (ConstConfig+ | N))?
                   ('with' 'definitions' ((pFunction|pOperation)+ | N))?
                   ('with' 'modules' (pModule+ | N))?
                   ('with' 'cmdoptions' STRING)?
```

A probabilistic property starts with prob. And a name (N denotes the name category, actually it is ID in Syntax 6.2.11) is associated with the property. The body of the property is an expression (pExpr in Syntax 6.2.7) that denotes the property to be verified. This expression must be a boolean expression (see Section 6.2.7). In addition, a property optionally has constant configurations (given

in ConstConfigs or a reference to a constants configuration name N in Syntax 6.2.2), function and operations definitions (given in pFunction or pOperation or a reference to definitions N in Syntax 6.2.5), PRISM modules (given in pModule or a reference to PRISM modules N in Syntax 6.2.6), and customised PRISM command line options. These options won't be parsed and processed. They are just passed to the PRISM command line tool to provide a flexible way for users to specify command line options.

6.2.2 Constants

Syntax — Constant Declaration.

```
Constant  ::=  'const' N ':' Type
```

A constant variable declared with a type (Type is the RoboChart Type class, see Appendix B, but in probabilistic properties only primitive types nat, int, bool, and real are allowed, which is enforced in scoping and validation rules of RoboTool) will be used in probabilistic formulas.

Syntax — Constants.

```
Constants     ::=  'constants' N ':' ConstConfigs
ConstConfigs  ::=  ConstConfig (','ConstConfig)*
                   ((',')? 'and' ConstConfig)?
```

Constants associates a name with configurations of various constants.

Syntax — Constant Configuration.

```
ConstConfig  ::=  QualifiedNameToElement
                  (('set' 'to' | 'assigned' | 'with' 'value') pExpr
                  | 'from' 'set' pExpr)
```

The configuration of each constant by ConstConfig can be a single value (pExpr) or a set of values (either a set extension or a set range expression in Syntax 6.2.7 and Section 6.2.7).

6.2.3 Labels and Formulas

Syntax — Labels.

```
Label  ::=  'label' N '=' pExpr
```

Labels provide a way to identify a set of states that are of particular interest. A label has its name associated with a boolean expression (see Section 6.2.7) and could be referred to later by '#' N in properties (see Syntax 6.2.7).

Syntax — Formulas.

```
Formula  ::=  'formula' N '=' pExpr
```

Formulas define expressions for reuse. The name given in a formula definition likes a shorthand to the defined expression. A formula can be referred to by '$' N in properties (see Syntax 6.2.7).

6.2.4 Rewards

Syntax — Rewards.

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

A collection of rewards can be assigned a name and this name will be used in RFormula in Syntax 6.2.7.

Syntax — Reward.

```
Reward  ::=  ('[' pEvent ']')? pExpr ':' pExpr ';'
```

A reward composes an optional event (pEvent, in Syntax 6.2.8), a guard condition (a boolean expression, see Section 6.2.7), and a reward (a numeric expression, see Section 6.2.7).

6.2.5 Function and Operation Definitions

Syntax — Function and Operation Definitions.

```
Definitions   ::=  'defs' N ':' (pFunction | pOperation)+
pFunction     ::=  'pfunction' N '(' N* ')' '=' '{' 'return' pExpr '}'
pOperation    ::=  'poperation' N '(' N* ')' '='
                   '{' pAssignment ('and' pAssignment)* '}'
pAssignment   ::=  '(' QualifiedNameToElement '=' pExpr')'
```

A Definitions identifies a set of function definitions (pFunction) and operation definitions (pOperation) by a name N. It can be referred to by its name in ProbProperty (see Syntax 6.2.1).

A pFunction defines a function with a name N, zero or more parameters (only their names N and no types), and a return expression pExpr. A pOperation, however, is defined as a collection of assignments pAssignment. A pAssignment assigns an expression to a RoboChart variable denoted by QualifiedNameToElement.

6.2.6 Probabilistic PRISM Modules

We can define PRISM modules pModules in properties. These modules will be added to the generated PRISM model, and used to specify properties for reactive systems that involve events. The need of these additional modules is due to the fact that the property language of PRISM is state-based. And so we cannot specify events directly.

Syntax — PRISM Modules.

```
pModules    ::=  'pmodules' N ':' pModule+
pModule     ::=  'pmodule' N '{' pVariable* pCommand+ '}'
pCommand    ::=  '[' pEvent? ']' pExpr '->'
                 ((pUpdate ('&' pUpdate)*) | 'skip') ';'
pUpdate     ::=  '(' (pExpr ':')? '@' N '=' pExpr ')'
pVariable   ::=  N ':' pType ('init' pExpr)? ';'
pType       ::=  'bool'
              |  '[' pExpr 'to' pExpr ']'
```

With pModules, a name N gives one or more PRISM modules. A such module is given by pModule which includes zero or module variables pVariable and one or more commands pComman. A pVariable declaration has a name N, a type pType, and an optionally initial expression pExpr. A pType can be a boolean or an integer range. A pCommand has an optional event pEvent, a guard condition pExpr, and a set of updates or a skip. An update is composed of an optional probability value pExpr, a reference to a pVariable variable by its name N, and an expression assigned to the variable.

6.2.7 Expressions

Expressions used in probabilistic properties are different from those in RoboChart. On the one hand, expressions need to cover state and path formulas in temporal logic, but these formulas are not part of RoboChart. On the other hand, many expressions in RoboChart could not be in properties, such as set and sequence expressions. For this season, we cannot simply reuse RoboChart expressions. We, therefore, define expressions used in probabilistic properties here. In principle, if an expression has a counterpart in RoboChart, then we use the same syntax. For example, the syntax of conditional, logical, arithmetic, comparison, array, set extension, and set range expressions is the same as that of RoboChart.

Furthermore, expressions are categorised into boolean or non-boolean expressions, numeric or non-numeric expressions, and set or non-set expression.. A boolean expression can be used in the place where a boolean expression is expected, such as pExpr in ProbProperty and Label. If a non-boolean is given in such a place, it results in a grammar error. Numeric and non-numeric expressions are similar.

Syntax — Expressions.

```
pExpr  ::=  INT                                  – integer
         |  FLOAT                                – real number
         |  BOOLEAN                              – boolean
         |  pExpr 'iff' pExpr                    – if and only if
         |  pExpr '=>' pExpr                     – implication
         |  pExpr '\/' pExpr                     – disjunction
         |  pExpr '/\' pExpr                     – conjunction
         |  'not' pExpr                          – logical negation
         |  pExpr '==' pExpr                     – equal
         |  pExpr '!=' pExpr                     – inequal
         |  pExpr '>' pExpr                      – larger than
         |  pExpr '>=' pExpr                     – larger than or equal
         |  pExpr '<' pExpr                      – less than
         |  pExpr '<=' pExpr                     – less than or equal
         |  pExpr '+' pExpr                      – addition
         |  pExpr '-' pExpr                      – subtraction
         |  pExpr '*' pExpr                      – multiplication
         |  pExpr '/' pExpr                      – division
         |  pExpr '%' pExpr                      – modulus
         |  '-' pExpr                            – negation
         |  'if' pExpr 'then' pExpr
            'else' pExpr 'end'                   – conditional
         |  pExpr '[' pExpr (',' pExpr)* ']'     – array
         |  '{' pExpr (',' pExpr)* '}'           – set extension
         |  '{' pExpr 'to' pExpr
            ('by' 'step' pExpr)? '}'             – set range
         |  StateFormula                         – state formula
         |  PathFormula                          – path formula
         |  RPathFormula                         – rewards path formula
         |  QualifiedNameToElement               – refer to RoboChart elements
         |  '@' (N '::' N '::')? N               – refer to pModule variables
         |  pEventVal                            – refer to data on event
         |  QualifiedNameToElement 'is' 'in'
            QualifiedNameToElement               – current state of a composite state
         |  '$' N                                – refer to a Formula
         |  LabelRef                             – refer to a Label
         |  '$$' N                               – refer to a parameter of pFunction
         |  '&' N '(' pExpr (',' pExpr)* ')'     – Call of a pFunction
         |  '(' pExpr ')'                        – parenthesised expression
```

Boolean Expressions

Boolean expressions includes:

- Boolean literals: true or false;
- Logical expressions: iff, =>, \/, /\, not;
- Comparison expressions: ==, !=, >, >=, <, <=;
- Conditional: if ... then ... else ... end, if the expressions of both if and else branches are boolean;
- Array expressions: pExpr (...), if the basic type of pExpr is boolean;
- State formulas: see Syntax 6.2.7 if a Query is not used;
- Path formulas;
- Rewards path formulas;
- A reference to a variable in a RoboChart model: QualifiedNameToElement, if the variable is boolean;
- A reference to a variable in pModule: @, if the variable is of the boolean type bool;
- A reference to the data on an event: pEventVal, if the event is of the boolean type;
- Current state of a composite state: is in;
- A reference to a Formula: $, if the expression of the formula (see Syntax 6.2.3) is boolean;
- A reference to a Label: LabelRef;
- Call to a pFunction if the result expression of the function definition is boolean.

Numeric Expressions

All boolean expressions are non-numeric. Numeric expressions includes:

- Integer and real numbers: INT and FLOAT;
- Arithmetic expressions: +, -, *, /, %, -;
- Conditional: if ... then ... else ... end, if the expressions of both if and else branches are numeric;
- Array expressions: pExpr (...), if the basic type of pExpr is numeric;
- State formulas: see Syntax 6.2.7 if a Query is used;
- A reference to a variable in a RoboChart model: QualifiedNameToElement, if the variable is numeric;
- A reference to a variable in pModule: @, if the variable is of the integer range type;
- A reference to the data on an event: pEventVal, if the event is of a numeric type;
- A reference to a Formula: $, if the expression of the formula (see Syntax 6.2.3) is numeric;
- Call to a pFunction if the result expression of the function definition is numeric.

Set Expressions

Set expressions includes:

- Set extension expressions;
- Set range expressions.

State Formulas

Syntax — State Formulas.

```
StateFormula  ::=  PFormula | RFormula | AFormula | EFormula
PFormula      ::=  'Prob' (Bound | Query) 'of' '[' pExpr ']'
                   (UseMethod)?
RFormula      ::=  'Reward' ('{' N '}')? (Bound | Query)
                   'of' '[' RPathFormula ']' (UseMethod)?
AFormula      ::=  'Forall' '[' pExpr ']'
EFormula      ::=  'Exists' '[' pExpr ']'
```

A state formula could be one of the followings:

- a probability formula (PFormula) starting with Prob and a bound (Bound) or a query (Query), followed by a boolean expression and an optional simulation method (UseMethod);
- a reward formula (RFormula) starting with Reward and an optional rewards name, and a bound (Bound) or a query (Query), followed by a reward path formula (see Syntax 6.2.7) and an optional simulation method (UseMethod);
- a non-probability property by a Forall operator followed by a boolean expression;
- a non-probability property by an Exists operator followed by a boolean expression.

The syntax of Bound and Query is shown below.

Syntax — Bound.

```
Bound  ::=  ('>' | '>=' | '<' | '<=') pExpr
```

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

Syntax — Query.

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

A query has three formats: what is the probability, the minimum probability, and the maximum probability.

Path Formulas

Syntax — Path Formulas.

```
PathFormula  ::=  'Next' pExpr
               |  pExpr 'Until' (Bound)? pExpr
               |  'Finally' (Bound)? pExpr
               |  'Globally' (Bound)? pExpr
               |  'Weak' 'Until' (Bound)? pExpr
               |  'Release' (Bound)? pExpr
```

Path formulas could be constructed from one of six common CTL and LTL operators. Expressions in these formulas must be boolean and could be path formulas too.

Syntax — Rewards Path Formulas.

```
RPathFormula  ::=  'Reachable' pExpr
                |  'LTL' pExpr
                |  'Cumul' pExpr
                |  'Total'
```

Reward path formulas could specify reachability, cumulative, total and instantaneous rewards. Expressions in these formulas also must be boolean. In a reachability reward formula (Reachable), the expression should not include any path formulas. In a LTL reward formula (LTL), the expression is a path formula, but restricted to only Next, Finally, and Until operators[1]. In a cumulative formula, the expression must be numeric.

Label Reference

Syntax — Label Reference.

```
LabelRef  ::=  '#' N | 'deadlock' | 'init'
```

Defined labels (Label, see Syntax 6.2.3) can be referred to by its name with a prefix #. Two built-in labels deadlock and init can be specified directly without a prefix.

6.2.8 Events

An event on a state machine or robotic platform cannot be identified simply by its name because its role in a connection could be for input or for output. Its direction in the connection, therefore, should be taken into account.

[1] In PRISM, it is called Co-Safe LTL reward properties. See http://www.prismmodelchecker.org/manual/PropertySpecification/Reward-basedProperties for more information.

Syntax — Events.

```
pEvent      ::=  QualifiedNameToElement '.' pEventDir
pEventDir   ::=  'in' | 'out'
pEventVal   ::=  pEvent '.' 'val'
```

We use pEvent to identify an event by its name (QualifiedNameToElement) and its role (pEventDir, either in or out) in a connection. For a typed event, we use pEventVal to refer to the data carried on the event.

6.2.9 Simulations

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

Syntax — Use Simulation Method.

```
UseMethod   ::=  'using' 'sim' 'with' SimMethod
                 (',' 'and' 'pathlen' '=' Expr)?
SimMethod   ::=  'CI' ('at' CiMethod)?
              |  'ACI' ('at' CiMethod)?
              |  'APMC' ('at' APMCMethod)?
              |  'SPRT' ('at' SPRTMethod)?
```

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

Syntax — CI Simulation Method.

```
CiMethod   ::=  ((',')? 'w' '=' Expr)?
                & ((',')? 'alpha' '=' alpha=Expr)?
                & ((',')? 'n' '=' n=Expr)?
```

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

Syntax — APMC Simulation Method.

```
APMCMethod   ::=  ((',')? 'epsilon' '=' Expr)?
                  & ((',')? 'delta' '=' Expr)?
                  & ((',')? 'n' '=' n=Expr)?
```

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

Syntax — SPRT Simulation Method.

```
SPRTMethod  ::=  ((',')? 'alpha' '=' Expr)?
                 & ((',')? 'delta' '=' Expr)?
```

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

6.2.10 Qualified Names

Syntax — Qualified Name To Element.

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

A fully qualified name annotated with :: provides a way to uniquely identify each instance of named elements (NamedElement) in RoboChart. Here instances for controllers and state machines denote every reference to them. For instance, there is a state machine definition (named m) in the RoboChart model and three state machine references to this definition: r1, r2, and r3. In order to refer to every element i of m, we use qualified names such as ...::r1::i. In particular, each qualified name shall start with a RoboChart module name, except that the constants declared in probabilistic properties by Constant (see Syntax 6.2.2).

6.2.11 Terminals

Syntax — Terminal rules.

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

7. Assertions DSL Usage

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

7.1 Standard Assertions

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

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

background, and RoboTool summarises the result in the form of a report, which is automatically opened upon completion of the checks.

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

7.2 Probabilistic Assertions

7.2.1 Instructions

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

7.2.2 Examples

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

■ **Example 7.1 — Constants configuration.** This example defines a constants configuration C1 that sets the constants batteryCapacity and chargeStep from the robotic platform reference rp_ref0 (that is from the module deliverMOD) to 20 and 4 respectively. Then the configuration C1 is referred in the property P_deadlock_free.

```
constants C1:
  deliverMOD::rp_ref0::batteryCapacity set to 20,
  and deliverMOD::rp_ref0::chargeStep set to 4
prob property P_deadlock_free:
  not Exists [Finally deadlock]
  with constants C1
```

■

■ **Example 7.2 — Constant declaration.** A constant l of type nat is declared, then it is used in the property P_stuck_loc. In addition, l is configured in the constants configuration to have values from a set {0,2,4,6,8}. This property can be regarded as five properties that have l equal to each element in the set separately. This example also shows the usage of labels. A label l_stuck, associated with a boolean expression, is declared and used in the property.

```
const l : core::nat
constants C2:
  deliverMOD::rp_ref0::batteryCapacity set to 20,
  deliverMOD::rp_ref0::chargeStep set to 4,
  and l from set {0 to 8 by step 2}
label l_stuck =
  deliverMOD::ctrl_ref0::stm_ref0 is in
  deliverMOD::ctrl_ref0::stm_ref0::Stuck
prob property P_stuck_loc:
  Prob=? of [Finally deliverMOD::rp_ref0::p==l /\ #l_stuck]
  with constant C2
```

■

■ **Example 7.3 — Formula.** A formula is just a shorthand of an expression and it is referred by putting a prefix $ in front of its name.

```
formula f_c = deliverMOD::rp_ref0::c
prob property P_stuck_loc1:
  Prob=? of [Finally deliverMOD::rp_ref0::p==1 /\ $f_c==0]
  with constant C1
```

■

■ **Example 7.4 — Rewards.** The reward nbmove assigns 1 to each synchronisation over the move event that is from the state machine reference stm_ref0 and used for input. Then a reward operator Reward uses this defined reward nbmove to check the average number of synchronisation over move when finally the robot gets stuck.

```
rewards nbmove =
  [deliverMOD::ctrl_ref0::stm_ref0::move.in] true : 1;
endrewards
prob property R_outofpower_moves:
  Reward {nbmove} =? of [ Reachable #l_stuck ]
  with constants C1
```

■

■ **Example 7.5 — Forall.** This assertion checks if the robot won't always finally get stuck. A cmdoptions is also supplied to the PRISM tool to set its maximum CUDD memory to 32 gigabytes.

```
prob property A_stuck:
  not Forall [Finally #l_stuck]
  with constant C1
  with cmdoptions "-cuddmaxmem 32g"
```

■

■ **Example 7.6 — Simulation.** This example applies statistic model checking to verify the property using the CI method with supplied parameters.

```
prob property P_stuck_loc_sim:
  Prob=? of [Finally deliverMOD::rp_ref0::p==1 /\ #l_stuck]
  using sim with CI at alpha=0.01,n=1000, and pathlen=100000
  with constant C1
```

■

8. Checking core assertions

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

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

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

about termination of the `move` operation.

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

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

In order to establish termination, we create two assertions. For example, in order to establish that the operation `move` terminates, we must show that the process `P_move` is not deadlock free, and that the process `P_move` followed by the process `RUN(r__)` is deadlock free.

III Collections

9. Collections

This chapter describe the RoboChart extensions designed to support modelling, analysis and simulation of collections of robots. Section 9.1 describes the extensions of the metamodel of RoboChart, Section 9.2 describes the conditions that characterise well-formed RoboChart collections, and Section 9.3 specifies the semantics of collections based on the untimed and timed semantics in Section 4.

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

9.1 Metamodel

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

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

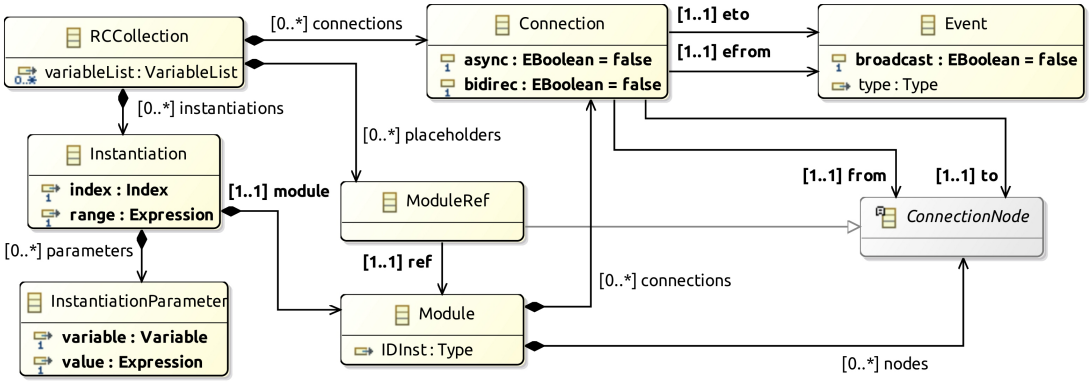

Figure 9.1: Metamodel for collections: RCCollection and Event

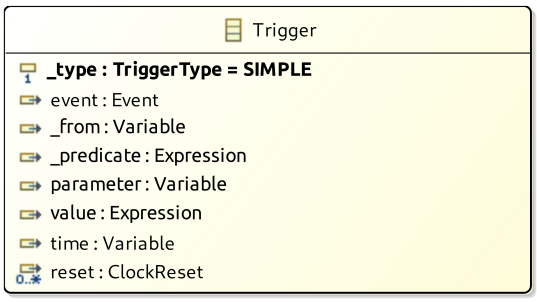

Figure 9.2: Metamodel for collections: Trigger

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

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

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

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

Finally, Triggers (shown in Figure 9.2) are extended with two new attributes:

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

9.2 Well-formedness Conditions

9.2.1 RCCollection

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

9.2.2 Instantiation

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

9.2.3 Event

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

9.2.4 Trigger

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

9.3 Semantics

Rule 196. Semantics of Collections

$$
\begin{array}{l}
[\![\mathsf{c} : \mathsf{RCCollection}]\!]_{\mathcal{C}ol} : \mathsf{CSPProcess} = \\
\quad \mathop{|||}\, \mathsf{inst} : \mathsf{c.instantiations} \bullet \mathop{|||}\, i : \mathsf{inst.range} \bullet [\![\mathsf{inst.module}]\!]_{\mathcal{M}'}(i) \\
\quad\quad [\![\{e_1, e_2 \mid (e_1, e_2) \leftarrow \mathsf{connectedEvents}(\mathsf{c})\}]\!] \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} \mathop{|||}\, \mathsf{conn} : \mathsf{c.connections} \bullet \mathop{|||}\, (i,j) : \mathsf{inds}(\mathsf{conn}, \mathsf{c}) \bullet \\ \quad \mathsf{BBuffer}^1(\mathsf{eventId}(\mathsf{conn.efrom}), \mathsf{i}, \mathsf{eventId}(\mathsf{conn.eto}), \mathsf{j}) \end{array} \right) \\
\quad ||| \\
\left(\begin{array}{l} \mathop{|||}\, \mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \wedge \mathsf{heterogeneous} \bullet \mathop{|||}\, (i,j) : \mathsf{inds}(\mathsf{conn}, \mathsf{c}) \bullet \\ \quad \mathsf{BBuffer}^2(\mathsf{eventId}(\mathsf{conn.eto}), \mathsf{j}, \mathsf{eventId}(\mathsf{conn.efrom}), \mathsf{i}) \end{array} \right)
\end{array} \right) \\
\mathsf{where} \\
\quad \mathsf{connectedEvents}(\mathsf{c} : \mathsf{Collection}) : \mathbb{P}(\mathsf{Event} \times \mathsf{Event}) = \\
\quad\quad \{\mathsf{conn} : \mathsf{c.connections} \bullet (\mathsf{eventId}(\mathsf{conn.efrom}), \mathsf{eventId}(\mathsf{conn.eto}))\} \\
\quad\quad \cup \\
\quad\quad \{\mathsf{conn} : \mathsf{c.connections} \mid \mathsf{conn.bidirec} \bullet (\mathsf{eventId}(\mathsf{conn.eto}), \mathsf{eventId}(\mathsf{conn.efrom}))\} \\
\quad \mathsf{inds}(\mathsf{conn} : \mathsf{Connection}, \mathsf{c} : \mathsf{Collection}) : \mathbb{P}(\mathsf{ID} \times \mathsf{ID}) = \\
\quad\quad \mathsf{if}\ \mathsf{conn.from.ref} = \mathsf{conn.to.ref}\ \mathsf{then} \\
\quad\quad\quad \mathsf{range}(\mathsf{conn.to}, \mathsf{c}) \times \mathsf{range}(\mathsf{conn.from}, \mathsf{c}) \setminus \{i : \mathsf{range}(\mathsf{conn.to}, \mathsf{c}) \bullet (i, i)\} \\
\quad\quad \mathsf{else} \\
\quad\quad\quad \mathsf{range}(\mathsf{conn.to}, \mathsf{c}) \times \mathsf{range}(\mathsf{conn.from}, \mathsf{c}) \\
\quad \mathsf{range}(\mathsf{m} : \mathsf{Module}, \mathsf{c} : \mathsf{Collection}) : \mathbb{P}\,\mathsf{ID} = (\imath \mathsf{i} : \mathsf{c.instantiations} \mid \mathsf{i.module} = \mathsf{m}).\mathsf{range} \\
\quad \mathsf{heterogeneous} = (\mathsf{c.from.ref} \neq \mathsf{c.to.ref})
\end{array}
$$

Rule 197. Broadcast Buffer

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

$$
\begin{array}{l}
\quad let \\
\qquad BufferEmpty = \text{prefixIn} \rightarrow BufferFull(x) \\
\qquad BufferFull(v) = \text{prefixIn} \rightarrow BufferFull(x) \,\Box\, \text{prefixOut} \rightarrow BufferEmpty \\
\quad within \\
\qquad BufferEmpty
\end{array}
$$

where

e_{in}.broadcast

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

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

Rule 198. Semantics of triggers

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

if t.type = INPUT then

eventId(t.event).$tid?f__!id_?x__ \rightarrow set_$vid(t._from)$!f__ \rightarrow set_$vid(t.parameter)$!x\_\_ \rightarrow Skip$

else if t.type = SIMPLE then

eventId(t.event).$tid?f__!id_ \rightarrow set_$vid(t._from)$!f\_\_ \rightarrow Skip$

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

where

t.event.broadcast

Rule 199. Semantics of send event statements

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

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

$\|\| i : \{x : ID \mid [\![\text{t._predicate}]\!]_{\text{Expr}}\} \bullet$ eventId(t.event)$!id\_!i!$[$\![$t.value$]\!]_{\text{Expr}} \rightarrow Skip$

else if t.type = SIMPLE then

$\|\| i : \{x : ID \mid [\![\text{t._predicate}]\!]_{\text{Expr}}\} \bullet$ eventId(t.event)$!id\_!i \rightarrow Skip$

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

where

s.trigger.broadcast

10. 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* [3], a process algebra that combines Z [16] and CSP, and includes time constructs [6]. Use of *Circus* and its UTP foundation will enable use of theorem proving as well as model checking.

An approach for writing object-oriented simulations of RoboChart diagrams has also been defined. Automatic generation of simulations is possible and part of our future work. Verification of correctness of simulations will use the object-oriented version of *Circus* [4], with a semantics given by the UTP theory in [12].

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 [30] to extend the notation, and from the UTP model of continuous variables [28] to define the semantics.

IV Appendices

A. RoboChart diagrams - an informal overview

To illustrate the concepts, we present the model of a robot for chemical detection based on that in [15] [1]. In our example, the robot employs a random walk and, upon detection of a chemical source, it turns on a light and drops a flag.

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

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

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

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

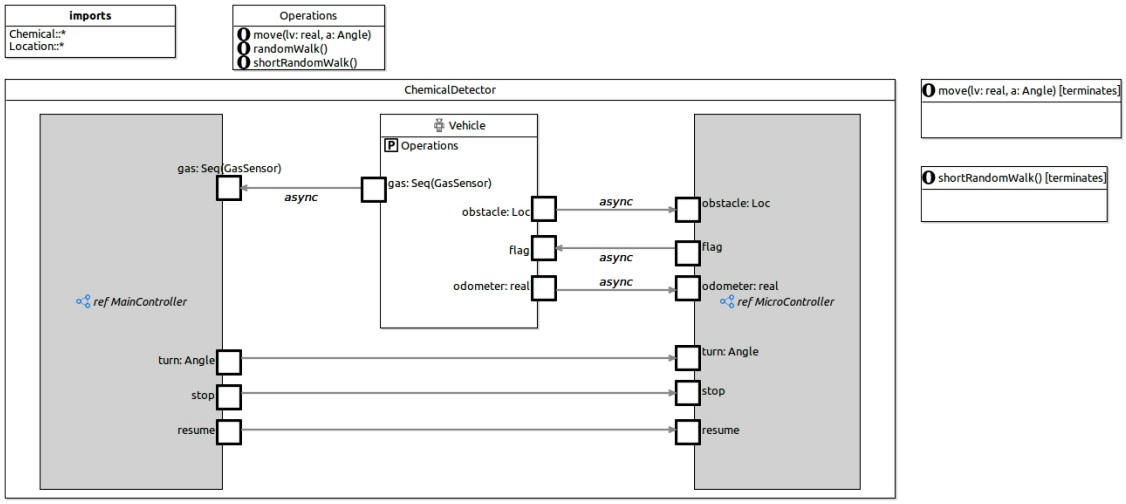

Figure A.1: Chemical Detector

random walk that is guaranteed to terminate.

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

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

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

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

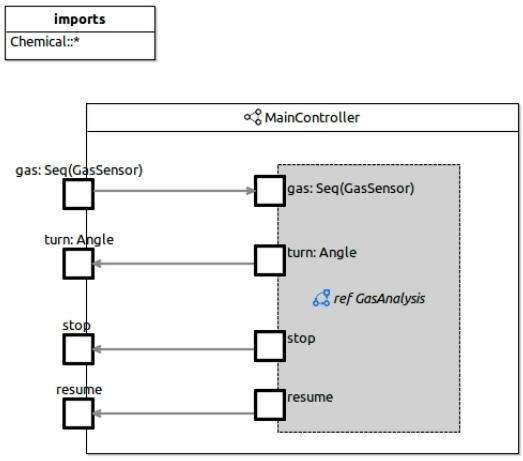

Figure A.2: MainController

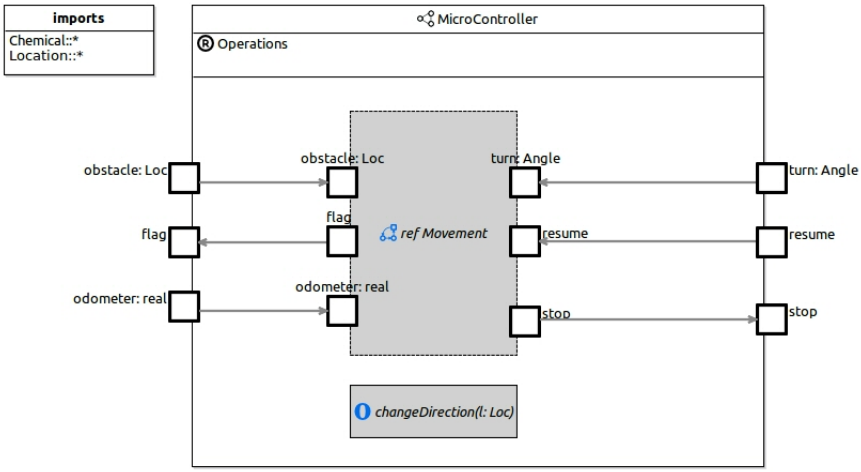

Figure A.3: MicroController

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

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

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

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

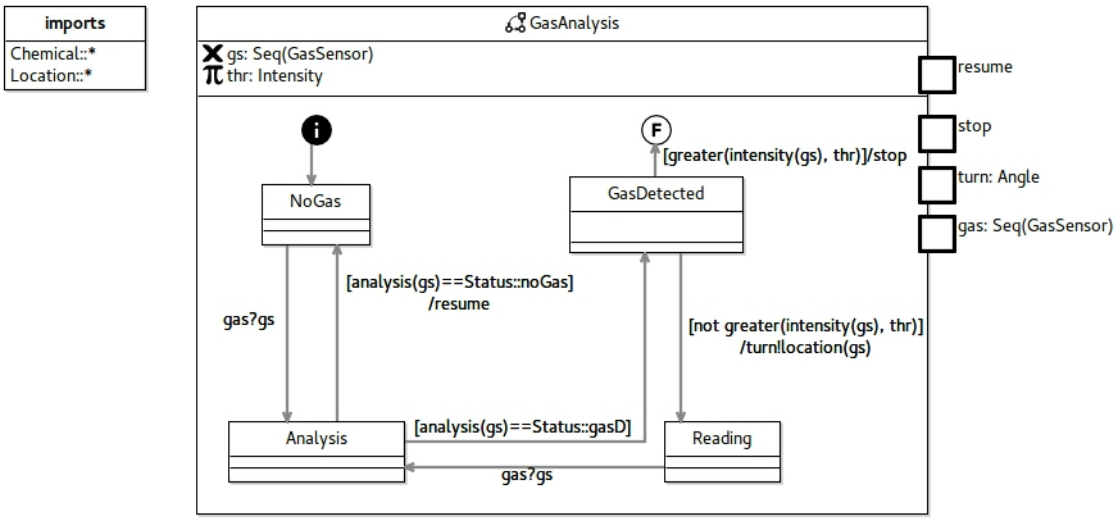

Figure A.4: GasAnalysis state machine

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

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

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

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

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

The strategy can be summarised as follows. When an obstacle is first detected, a clock is reset and

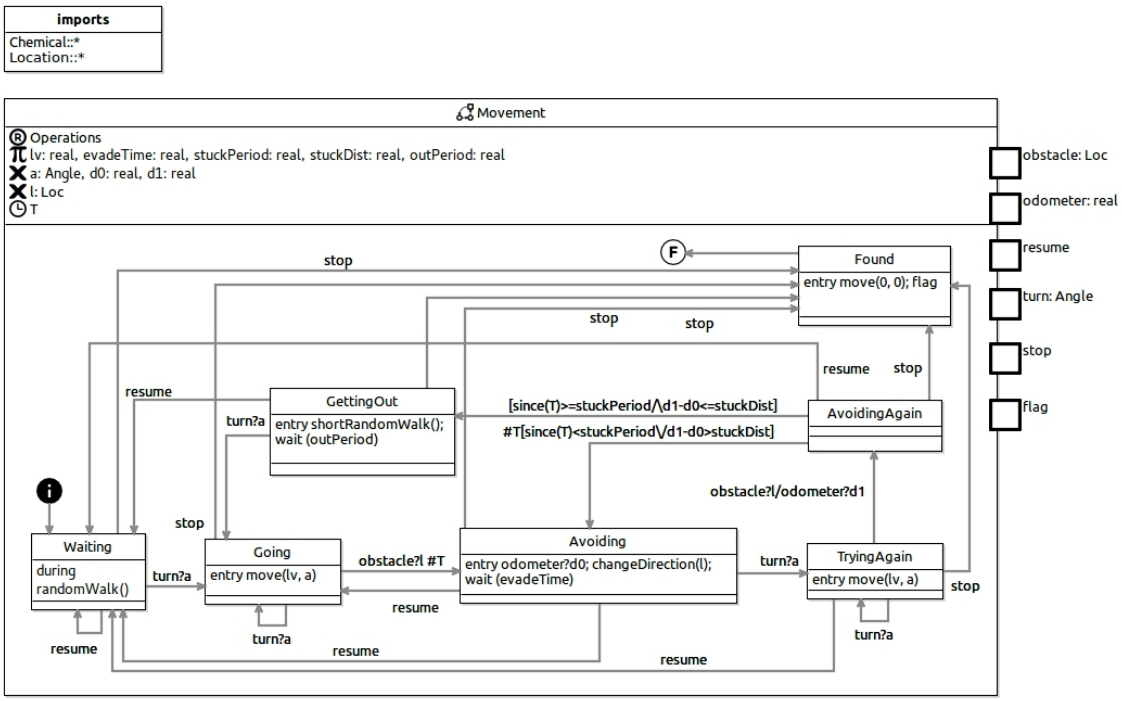

Figure A.5: Movement state machine

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

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

RoboChart state machines are standard, but restricted and with a well defined semantics. They can have composed states, junctions, and entry, during, exit, and transition actions defined using a well defined action language. Features of UML state machines [24] deemed not essential for robotics are not included, resulting in a streamlined semantics.

Definitions of a model can be organised in packages. Like in UML, they are just containers. They do not correspond to a concept or abstraction, and so do not have an interface. An imports mechanism controls scope of the definitions. All packages that do not have a package name conceptually compose the same package. Elements defined in the unnamed package are available in all other packages. Elements defined in a package with a name can only be used if they are explicitly imported. A model is identified by one module and all the other elements there. Figures A.6 and A.7 define two packages used in our example.

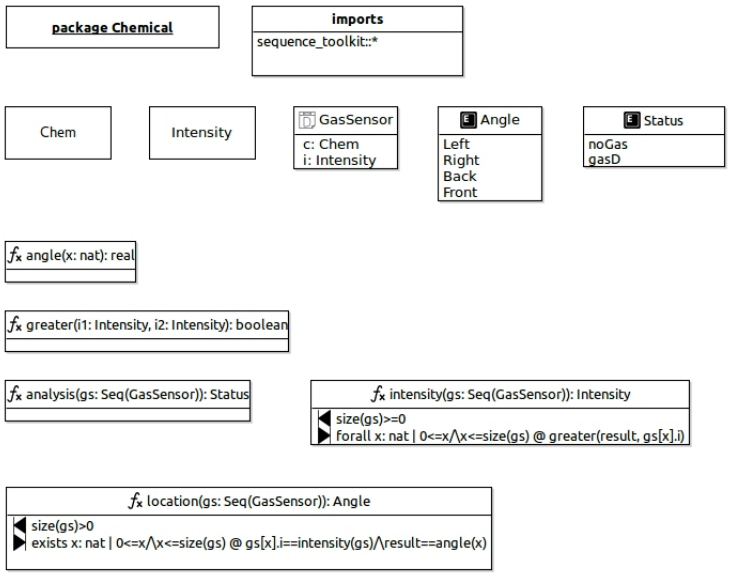

Figure A.6: Chemical package

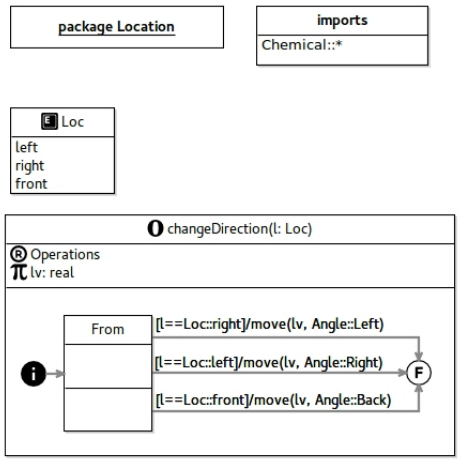

Figure A.7: Location package

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

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

A.1 Time primitives

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

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

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

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

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

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

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

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

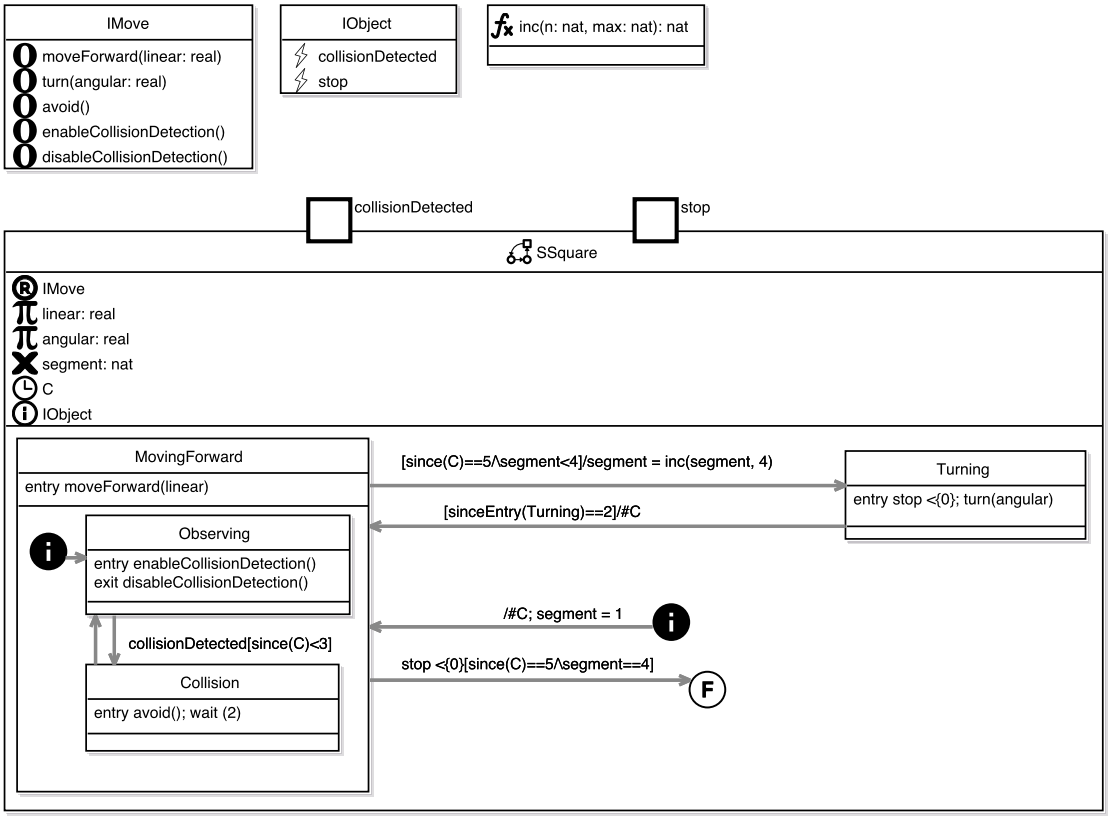

Figure A.8: Square-trajectory robot

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

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

Before MovingForward is actually entered, its entry action executes, followed by that of its substate Observing, enabling the collision detection capability. Once a collision is detected, the event collisionDetected is raised by the robotic platform: the transition from Observing to the state Collision is then triggered, executing the exit action of Observing and subsequently the avoid operation that performs the actual collision avoidance. Here we do not specify this operation, but record its budget of 2 time units by sequentially composing it with the timed primitive wait(2). In RoboChart time elapses explicitly via budgets, unless a state has been entered and no transitions are enabled, or, every enabled transition is associated with an external event. Once the collision is resolved, a transition back to Observing is taken. Transitions are triggered once the guard is true

and the associated event is raised, or, if there is no event or guard associated, immediately, as in this example.

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

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

B. Complete Metamodel

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

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

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

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

package robochart : robochart = 'http://www.robocalc.circus/RoboChart'
{
  class BasicPackage
  {
    attribute name : String[?];
    property imports : Import[*] { composes };
  }
  class RCPackage extends BasicPackage
  {
    property interfaces : Interface[*] { composes };
    property robots : RoboticPlatformDef[*] { composes };
    property types : TypeDecl[*] { composes };
    property machines : StateMachineDef[*] { composes };
```

```
  property controllers : ControllerDef[*] { composes };
  property modules : RCModule[*] { composes };
  property operations : OperationDef[*] { composes };
  property functions : Function[*] { composes };
}
class Import
{
  attribute importedNamespace : String[1];
}
abstract class NamedElement
{
  attribute name : String[1];
}

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

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

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

/*
* Enumeration Types
*/
class Enumeration extends TypeDecl
{
  property literals : Literal[*|1] { composes };
}
class Literal extends TypeDecl,NamedExpression
{
  property types : Expression[*] { composes };
}
```

```
/*
* Named Types
*/
class NamedType extends TypeDecl
{
  property type : Expression[1] { composes };
}

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

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

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

/*
* Operations
```

```
*/
class OperationSig extends NamedElement
{
  attribute terminates : Boolean[1];
  property parameters : Parameter[*] { ordered composes };
  property preconditions : Expression[*] { composes };
  property postconditions : Expression[*] { composes };
}
abstract class Operation extends NamedElement,ConnectionNode;
class OperationDef extends Operation,OperationSig,StateMachineBody;

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

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

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

/*
* State Machines
```

```
*/
abstract class StateMachine extends NamedElement,ConnectionNode;
class StateMachineDef extends StateMachineBody,StateMachine;
class StateMachineRef extends StateMachine,Reference
{
  property ref : StateMachineDef[1];
}
class StateMachineBody extends Context,NodeContainer;
class Clock extends NamedElement;
abstract class NodeContainer
{
  property nodes : Node[*] { composes };
  property transitions : Transition[*] { composes };
}
abstract class Node extends NamedElement;
class Junction extends Node;
class Initial extends Junction;
class State extends Node,NodeContainer
{
  property actions : Action[*] { composes };
}

class Final extends State;
class ProbabilisticJunction extends Junction;
class Transition extends NamedElement
{
  property source : Node[1];
  property target : Node[1];
  property trigger : Communication[?] { composes };
  property deadline : Expression[?] { composes };
  property condition : Expression[?] { composes };
  property action : Statement[?] { composes };
  property probability : Expression[?] { composes };
  property reset : ClockReset[*] { ordered composes };
}
class Communication
{
  property event : Event[?];
  property _from : Variable[?];
  property _predicate : Expression[?] { composes };
  property parameter : Variable[?];
  property value : Expression[?] { composes };
  attribute _type : CommunicationType[1];
}
enum CommunicationType { serializable }
{
  literal SIMPLE;
  literal INPUT;
  literal OUTPUT;
  literal SYNC;
```

```
}

/*
* Actions
*/
abstract class Action
{
  property action : Statement[1] { composes };
}
class EntryAction extends Action;
class DuringAction extends Action;
class ExitAction extends Action;

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

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

/*
* Modules
*/
class RCModule extends NamedElement
{
  property connections : Connection[*] { composes };
  property nodes : ConnectionNode[*] { composes };
}
```

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

/*
* Templates
```

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

/*
* Expression
*/
abstract class Expression;

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

/*
* Timed Expression
*/

class ClockExp extends Expression
{
  property clock : Clock[1];
}
class StateClockExp extends Expression
{
  property state : State[1];
}
```

```
  /*
  * Assignable
  */
  abstract class Assignable;
  class VarSelection extends Assignable
  {
    property receiver : Assignable[1] { composes };
    property member : Member[1];
  }
  class ArrayAssignable extends Assignable
  {
    property value : Assignable[1] { composes };
    property parameters : Expression[+] { composes };
  }
  class VarRef extends Assignable
  {
    property name : Mutable[1];
  }
  abstract class ConnectionNode;
  abstract class NamedExpression;
  class WaitingCondition extends NamedElement
  {
    property expression : Expression[?];
    property transitions : Transition[*] { ordered };
  }
  class WaitingConditionRef extends Expression
  {
    property ref : WaitingCondition[?];
  }
  class FieldDefinition
  {
    property field : Field[1];
    property value : Expression[1] { composes };
  }
  abstract class Mutable extends NamedElement,NamedExpression;
}
```

C. ✓-*tock* and *share*-CSP

In this chapter we describe additional operators for ✓-*tock*-CSP [9] that are used in the semantics of RoboChart, and the use of *share* to model state in a distributed way, as required, for example, to give a compositional semantics to software operations.

C.1 ✓-*tock*

C.1.1 Timeout operator

The timeout operator can be defined in terms of core ✓-*tock* operators using: parallel composition, strict timed interrupt, external choice, and hiding.

$$P \rhd_d Q = (P \,[\![\Sigma]\!]_{\rhd}\, (((\Box\, e : \Sigma \bullet e \rightarrow Skip) \triangle_d Stop);\, RUN(\Sigma))) \,\Box\, (Wait(d);\, Q)$$

The process initially behaves as P, and after exactly d time units, if no event of P is taken up by the environment, then the process behaves as Q. The timed constraint on P is imposed by controlling the events that it can perform via the biased terminating parallel composition ($[\![cs]\!]_{\rhd}$) with a controller process that can perform any event $e : \Sigma$ before d time units, followed by $RUN(\Sigma)$, or if d time units pass before P engages in an event in Σ, then the controller process deadlocks, effectively blocking P from performing any event. After d time units have passed Q is offered via the external choice if P has not engaged in any event.

C.1.2 Biased terminating parallel composition

The following is a biased parallel composition operator where termination of the P leads to the interruption of Q and termination overall.

$$P \,[\![cs]\!]_{\rhd}\, Q = ((P; f \rightarrow Skip) \,[\![cs \cup \{f\}]\!]\, (Q \triangle f \rightarrow Skip)) \setminus \{f\}$$

It is defined by considering a fresh event f, where after termination of P there is a prefixing on f. Upon synchronisation on f this interrupts the behaviour of Q leading to overall termination of the parallel composition. Finally the event f is hidden.

C.2 *share*-CSP

To define a compositional semantics with shared state, where assignments are encoded via *set* events, and reads via *get* events, we consider a special event *share* to generically encode any changes in state. Processes that employ *share* typically follow a synchronisation on *share* by updating their copy of the shared state via synchronisation on *get* events. Processes offer to synchronise on *share* events whenever there is the possibility for interference, that is, for other components to update a shared variable, while refusal of *share* events can be used to ensure atomicity of state updates.

Two basic processes that allow *share* events are *SStop* and *SSkip*, versions of *Stop* and *Skip*, respectively.

Definition C.2.1

$$SStop \mathrel{\widehat{=}} share \rightarrow SStop$$
$$SSkip \mathrel{\widehat{=}} SStop \triangle Skip$$

SStop always offers *share*, while *SSkip* offers, in addition, the possibility to terminate. Similarly to the *tock*-CSP encoding of *Skip* in FDR, for example, *Skip* can nondeterministically terminate.

C.2.1 Parallel composition

In order to enable processes to respond synchronously to updates to the shared state, below we introduce a *meta* alphabetised parallel operator, defined in terms of sets of NamedElements. This is the case, for example, in the semantics of composite states, where an assignment performed as part of the control flow of the state, or its during action, can directly influence the availability of its outgoing transitions.

The following meta operator enables the alphabetised parallel composition of processes P and Q, synchronising on a set cs of events, to be updated synchronously whenever a RoboChart 'variable' is updated. The set p_{write} is the set of variables whose values may be changed by P, and p_{read} is

the set of variables read by P. Similarly, for Q there are two such sets q_{write}, q_{read}. Writes by P, via events set_x to a variable x, which is in the intersection $p_{write} \cap q_{read}$ are propagated to Q, and writes by Q, to events in $q_{write} \cap p_{read}$ are similarly propagated to P.

Definition C.2.2

$$
\begin{array}{l}
P \llbracket p_{read} : p_{write} \mid cs \mid q_{read} : q_{write} \rrbracket Q \\
\widehat{=} \left(\begin{array}{l}
P \left[\!\left[\begin{array}{l} share \leftarrow share, \\ \underline{\mathsf{v} : \mathsf{p}_{\mathsf{read}} \cap \mathsf{q}_{\mathsf{write}}} \bullet share \leftarrow \underline{\mathsf{setR}^1(\mathsf{v})}, \\ \underline{\mathsf{v} : \mathsf{p}_{\mathsf{write}} \cap \mathsf{q}_{\mathsf{read}}} \bullet \underline{\mathsf{set}^1(\mathsf{v})} \leftarrow \underline{\mathsf{setL}^1(\mathsf{v})} \end{array} \right]\!\right] \\
\llbracket cs \cup \underline{\mathsf{setLC}^1(\mathsf{p}_{\mathsf{write}} \cap \mathsf{q}_{\mathsf{read}})} \cup \underline{\mathsf{setRC}^1(\mathsf{q}_{\mathsf{write}} \cap \mathsf{p}_{\mathsf{read}})} \cup \{share\} \rbrace \rrbracket \\
Q \left[\!\left[\begin{array}{l} share \leftarrow share, \\ \underline{\mathsf{v} : \mathsf{q}_{\mathsf{read}} \cap \mathsf{p}_{\mathsf{write}}} \bullet share \leftarrow \underline{\mathsf{setL}^2(\mathsf{v})}, \\ \underline{\mathsf{v} : \mathsf{q}_{\mathsf{write}} \cap \mathsf{p}_{\mathsf{read}}} \bullet \underline{\mathsf{set}^2(\mathsf{v})} \leftarrow \underline{\mathsf{setR}^2(\mathsf{v})} \end{array} \right]\!\right]
\end{array} \right) \\
\left[\!\left[\begin{array}{l} \underline{\mathsf{v} : \mathsf{p}_{\mathsf{read}} \cap \mathsf{q}_{\mathsf{write}}} \bullet \underline{\mathsf{setR}^3(\mathsf{v})} \leftarrow \underline{\mathsf{set}^3(\mathsf{v})}, \\ \underline{\mathsf{v} : \mathsf{q}_{\mathsf{read}} \cap \mathsf{p}_{\mathsf{write}}} \bullet \underline{\mathsf{setL}^3(\mathsf{v})} \leftarrow \underline{\mathsf{set}^4(\mathsf{v})} \end{array} \right]\!\right]
\end{array}
$$

The idea here is that P or Q, can engage in *share* events to participate in synchronous updates to state variables, subsequently refreshing the value of variables via *get* events. If the p_{read} and p_{write} sets are the same, we omit the delimiter : in $p_{read} : p_{write}$.

Because in RoboChart there is a need to take into account synchronous updates of other components, like reset of clocks, and entering of states, the operator is defined in terms of set of NamedElements, instead of Variables.

Rule 200. Setter channel
$\underline{\mathsf{set}(\mathsf{n} : \mathsf{NamedElement}) : \mathsf{CSPChannel} =}$

Split into several definitions according to the type of $\underline{\mathsf{n}}$.

Rule 201. Setter channel
$\underline{\mathsf{set}(\mathsf{v} : \mathsf{Variable}) : \mathsf{CSPChannel} =}$

$set\_\underline{\mathrm{id}(\mathsf{v})}$

Rule 202. Setter channel
$\underline{\mathsf{set}(\mathsf{c} : \mathsf{Clock}) : \mathsf{CSPChannel} =}$

$clockReset.\underline{\mathrm{id}(\mathsf{c})}$

Rule 203. Setter channels

setC(ns : Set(NamedElement)) : CSPChannelSet =

$\{|n : ns \bullet set^5(n)|\}$

Rule 204. Left setter channel

setL(v : Variable) : CSPChannel =

$setL\_id(v)$

Rule 205. Left setter channel

setL(c : Clock) : CSPChannel =

$clockResetL.id(c)$

Rule 206. Left setter channels

setLC(ns : Set(NamedElement)) : CSPChannelSet =

$\{|n : ns \bullet setL^4(n)|\}$

Rule 207. Right setter channel

setR(v : Variable) : CSPChannel =

$setR\_id(v)$

Rule 208. Right setter channel

setR(c : Clock) : CSPChannel =

$clockResetR.id(c)$

Rule 209. Right setter channels

setRC(ns : Set(NamedElement)) : CSPChannelSet =

$\{|n : ns \bullet setR^4(n)|\}$

D. Mathematical Toolkit

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

A package named core contains a number of primitive types and is imported by default in every RoboChart model. They are shown in Table 4.1.

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

Table 4.1 : **Primitive types in** core **package..**

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

package set_toolkit

```
Union(A: Set(Set(?X))): Set(?X)
  result=={ x: ?X | exists a: Set(?X) | a in A @ x in a}

notin(m: ?X, s: Set(?X)): boolean
  result<=>not m in s

Inter(A: Set(Set(?X))): Set(?X)
  result=={ x: ?X | forall a: Set(?X) | a in A @ x in a}

diff(s1: Set(?X), s2: Set(?X)): Set(?X)
  result=={ x: ?X | x in s1/\not x in s2}

symetric_diff(s1: Set(?X), s2: Set(?X)): Set(?X)
  result=={ x: ?X | not (x in s1<=>x in s2)}

subseteq(ss: Set(?X), s: Set(?X)): boolean
  result<=>(forall x: ?X @ x in s)

subset(ss: Set(?X), s: Set(?X)): boolean
  result<=>subseteq(ss, s)/\ss!=s

union(s1: Set(?X), s2: Set(?X)): Set(?X)
  result=={ x: ?X | x in s1\/x in s2}

inter(s1: Set(?X), s2: Set(?X)): Set(?X)
  result=={ x: ?X | x in s1/\x in s2}
```

package relation_toolkit

imports

set_toolkit::*

f_x relcomp(r: ?X<->?Y, s: ?Y<->?Z): ?X<->?Z

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

f_x tr_closure(r: ?X<->?X): ?X<->?X

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

f_x dom(r: ?X<->?Y): Set(?X)

result=={ p: ?X*?Y | p in r @ p[1]}

f_x dres(a: Set(?X), r: ?X<->?Y): ?X<->?Y

result=={ p: ?X*?Y | p in r/\p[1] in a}

f_x rimage(r: ?X<->?Y, a: Set(?X)): Set(?Y)

result=={ p: ?X*?Y | p in r/\p[1] in a @ p[2]}

f_x override(r: ?X<->?Y, s: ?X<->?Y): ?X<->?Y

result==union(dsub(dom(s), r), s)

f_x first(p: ?X*?Y): ?X

result==p[1]

f_x rinv(r: ?X<->?Y): ?Y<->?X

result=={ p: ?X*?Y | p in r @ maplet(p[2], p[1])}

f_x refl_tr_closure(r: ?X<->?X): ?X<->?X

result==union(tr_closure(r), id())

f_x second(p: ?X*?Y): ?Y

result==p[2]

f_x rres(r: ?X<->?Y, b: Set(?Y)): ?X<->?Y

result=={ p: ?X*?Y | p in r/\p[2] in b}

f_x dsub(a: Set(?X), r: ?X<->?Y): ?X<->?Y

result=={ p: ?X*?Y | p in r/\not p[1] in a}

f_x maplet(x: ?X, y: ?Y): ?X*?Y

result==(|x, y|)

f_x funcomp(s: ?Y->?Z, r: ?X->?Y): ?X<->?Z

result==relcomp(r, s)

f_x ran(r: ?X<->?Y): Set(?Y)

result=={ p: ?X*?Y | p in r @ p[2]}

f_x id(): ?X<->?X

result=={ x: ?X | maplet(x, x)}

f_x rsub(r: ?X<->?Y, b: Set(?Y)): ?X<->?Y

result=={ p: ?X*?Y | p in r/\not p[2] in b}

package function_toolkit

imports

relation_toolkit::*
set_toolkit::*

f_x isFinite(s: Set(?X)): boolean

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

f_x isInjection(f: ?X->?Y): boolean

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

f_x disjoint(f: ?L<->Set(?X)): boolean

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

f_x isTotal(f: ?X->?Y): boolean

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

f_x isFiniteFunction(f: ?X->?Y): boolean

result<=>isFinite(f)

f_x isFiniteInjection(f: ?X->?Y): boolean

result<=>isFinite(f)/\isInjection(f)

f_x isBijection(f: ?X->?Y): boolean

result<=>isTotalInjection(f)/\isTotalSurjection(f)

f_x isTotalSurjection(f: ?X->?Y): boolean

result<=>isTotal(f)/\isSurjection(f)

f_x isTotalInjection(f: ?X->?Y): boolean

result<=>isTotal(f)/\isInjection(f)

f_x partitions(f: ?L<->Set(?X), a: Set(?X)): boolean

result<=>(disjoint(f)/\Union(ran(f))==a)

f_x isSurjection(f: ?X->?Y): boolean

result<=>(ran(f)={ y: ?Y})

package sequence_toolkit

imports

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

f_x dconcat(s: Seq(Seq(?X))): Seq(?X)

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

f_x iter(n: int, r: ?X<->?X): ?X<->?X

result==if n==0 then id() else if n>0 then relcomp(r, iter(n-1, r)) else iter(-n, rinv(r)) end end

f_x size(a: Set(?X)): nat

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

f_x squash(f: int->?X): Seq(?X)

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

f_x max(a: Set(int)): int

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

f_x concat(s: Seq(?X), t: Seq(?X)): Seq(?X)

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

f_x reverse(s: Seq(?X)): Seq(?X)

result==(lambda n: int | n in dom(s) @ s[size(s)-n+1])

f_x infix(s: Seq(?X), t: Seq(?X)): boolean

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

f_x suffix(s: Seq(?X), t: Seq(?X)): boolean

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

f_x hasMax(a: Set(int)): boolean

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

f_x front(s: Seq(?X)): Seq(?X)

result==dsub({size(s)}, s)

f_x range(x: int, y: int): Seq(int)

result=={ i: int | x<=i/\i<=y}

f_x hasMin(a: Set(int)): boolean

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

f_x filter(s: Seq(?X), a: Set(?X)): Seq(?X)

result==squash(rres(s, a))

f_x isNonEmpty(s: Seq(?X)): boolean

result<=>size(s)>0

f_x tail(s: Seq(?X)): Seq(?X)

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

f_x extract(a: Set(int), s: Seq(?X)): Seq(?X)

result==squash(dres(a, s))

f_x prefix(s: Seq(?X), t: Seq(?X)): boolean

result<=>subseteq(s, t)

f_x min(a: Set(int)): int

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

f_x head(s: Seq(?X)): ?X

isNonEmpty(s)
s[1]

f_x last(s: Seq(?X)): ?X

isNonEmpty(s)
s[size(s)]

f_x isInjectiveSequence(s: Seq(?X)): boolean

result<=>isInjection(s)

E. OCL Well-formedness Conditions

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

package robochart

-- Robotic Platforms well-formedness conditions (RoboChart reference 3.1.1)
context RoboticPlatformDef
  -- We note that variables and operations declared directly in the
     platform,
  -- outside an interface, are considered as if declared in a provided
  -- interface, for the reasons already explained above. Events declared
  -- directly in the platform, on the other hand, are defined.
  def: rpProvidedVars() : Bag(Variable) =
    self.pInterfaces.variableList.vars->union(self.variableList.vars)
  def: rpProvidedOps() : Bag(OperationSig) = self.pInterfaces.operations->
     union(self.operations)
  def: rpDefinedEvents() : Bag(Event) = self.interfaces.events->union(self.
     events)

  -- RP1: Robotic platforms cannot require interfaces
  inv RP1: self.rInterfaces->isEmpty()
  -- RP2: Defined interfaces can only have events
  inv RP2: self.interfaces->forAll(i |
    i.variableList->isEmpty() and i.operations->isEmpty()
  )
  -- RP3: The names of variables, operations, and events are unique to the
     platform
  inv RP3: self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
     pInterfaces))
```

```
      .variableList->union(self.variableList).vars.name
    ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
        pInterfaces))
      .operations->union(self.operations).name)
    ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
        pInterfaces))
      .events->union(self.events).name)->isUnique(i | i)


-- Interfaces well-formedness conditions (RoboChart reference 3.1.2)
context Interface
  -- I1: Provided and required interfaces contain only variables and
      operations
  inv I1: Context.allInstances()->collect(c | c.pInterfaces->union(c.
      rInterfaces))->includes(self)
    implies self.events->isEmpty()
  -- I2: Defined interfaces contain only variables and events
  inv I2: Context.allInstances().interfaces->includes(self)
    implies self.operations->isEmpty()
  -- I3: Names of variables, events and operations are unique
  inv I3: self.variableList.vars.name
    ->union(self.operations->asBag().name)
    ->union(self.events->asBag().name)->isUnique(i | i)


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

  -- M1: A module must contain exactly one robotic platform, at least one
      controller, and not state machines
  inv M1: self.nodes->selectByKind(RoboticPlatform)->one(true)
    and self.nodes->selectByKind(Controller)->exists(true)
    and self.nodes->selectByKind(StateMachine)->isEmpty()
  -- M2: All variables and operations required by the module's controllers
      must be provided by the platform
  inv M2: self.nodes->selectByKind(RoboticPlatform)->exists(true) implies
    (self.moduleRP().rpDef().rpProvidedVars()
      ->includesAll(self.moduleControllers().controllerDef().
          controllerRequiredVars())
    and self.moduleRP().rpDef().rpProvidedOps()
      ->includesAll(self.moduleControllers().controllerDef().
          controllerRequiredOps()))
  -- M3: Each event on the robotic platform and controllers of a module
      must have at most one connection to or from it within the module
  inv M3: self.nodes->selectByKind(RoboticPlatform)->exists(true) implies
    self.moduleRP().rpDef().rpDefinedEvents()
```

```
      ->forAll(e | self.connections->select(c | c.efrom = e or c.eto = e)->
          size() <= 1)
    and self.moduleControllers().controllerDef().controllerDefinedEvents()
      ->forAll(e | self.connections->select(c | c.efrom = e or c.eto = e)->
          size() <= 1)


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

```
    and self.to.oclIsKindOf(Controller)
  ) implies self.to.oclAsType(Controller).controllerDef().connections
      ->select(c | (c.from = self.to and c.efrom = self.eto)
            or (c.to = self.to and c.eto = self.eto))
      ->forAll(c | not c.bidirec and c.from = self.to)
  -- Cn7: Non-bidirectional connections of a module may only connect from
      events of a controller which have a non-bidirectional connection to
      them within the controller
  inv Cn7: (not self.bidirec
    and RCModule.allInstances()->exists(m | m.connections->includes(self))
    and self.from.oclIsKindOf(Controller)
  ) implies self.from.oclAsType(Controller).controllerDef().connections
      ->select(c | (c.from = self.from and c.efrom = self.efrom)
            or (c.to = self.from and c.eto = self.efrom))
      ->forAll(c | not c.bidirec and c.to = self.from)
  -- Cn8: Non-bidirectional connections of a controller must not connect to
       events that a state machine uses as an output.
  inv Cn8: (not self.bidirec
    and ControllerDef.allInstances()->exists(c | c.connections->includes(
        self))
    and self.to.oclIsKindOf(StateMachine)
  ) implies self.to.oclAsType(StateMachine).stmDef().ncOutputEvents()->
      excludes(self.eto)
  -- Cn9: Non-bidirectional connections of a controller must not connect
      from events that a state machine uses as an input
  inv Cn9: (not self.bidirec
    and ControllerDef.allInstances()->exists(c | c.connections->includes(
        self))
    and self.from.oclIsKindOf(StateMachine)
  ) implies self.from.oclAsType(StateMachine).stmDef().ncInputEvents()->
      excludes(self.efrom)


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

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

```
    controllers
  inv C2: self.pInterfaces->collect(i | i.variableList.vars->union(i.
      operations))->isEmpty()
  -- C3: All variables required by the controller's state-machines must be
      defined or required by the controller
  inv C3: self.controllerRequiredVars()->union(self.controllerDefinedVars()
      )
    ->includesAll(self.machines.stmDef().stmRequiredVars())
  -- C4: All operations required by the controller's state-machines must be
       required or defined by the controller
  inv C4: self.controllerRequiredOps()->union(self.lOperations)
    ->includesAll(self.machines.stmDef().stmRequiredOps())
  -- C5: The names of variables, operations, and events are unique to the
      controller
  -- operations declared directly in the controller are ruled out by C7
  inv C5: self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
      pInterfaces))
      .variableList->union(self.variableList).vars.name
    ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
        pInterfaces))
      .operations->union(self.lOperations).name)
    ->union(self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
        pInterfaces))
      .events->union(self.events).name)->isUnique(i | i)
  -- C6: Each event on state machines and boundary of a controller must
      have at most one connection to or from it within the controller
  inv C6: self.machines->forAll(m | m.stmDef().stmDefinedEvents()->forAll(e
       | self.connections
    ->select(c | (c.from = m and c.efrom = e) or (c.to = m and c.eto = e))
        ->size() <= 1
  )) and self.controllerDefinedEvents()->forAll(e | self.connections
    ->select(c | (c.from = self and c.efrom = e) or (c.to = self and c.eto
        = e))->size() <= 1
  )
  -- C7: Operations must not be declared directly in a controller, but may
      be defined in the controller
  inv C7: self.operations->isEmpty()


-- State Machines well-formedness conditions (RoboChart reference 3.1.6)
context StateMachineDef
  def: stmDefinedVars() : Bag(Variable) =
    self.interfaces.variableList.vars->union(self.variableList.vars)
  def: stmDefinedEvents() : Bag(Event) = self.interfaces.events->union(self
      .events)
  def: stmRequiredVars() : Bag(Variable) = self.rInterfaces.variableList.
      vars
  def: stmRequiredOps() : Bag(OperationSig) = self.rInterfaces.operations

  -- STM1: State machines cannot have provided interfaces
```

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


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


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

```
  -- this is implicit since Initial inherits from Junction


-- Junction well-formedness conditions (RoboChart reference 3.1.9)
context Junction
  -- J1: A junction must contain at least one outgoing transition
  inv J1: NodeContainer.allInstances()->select(nc | nc.nodes->includes(self
      )).transitions
    ->exists(t | t.source = self)
  -- J2: The guards of the transitions out of a junction must form a cover
  -- NOTE: cannot be checked in general, but does not prevent generation of
       semantics
--  inv J2: NodeContainer.allInstances()->select(nc | nc.nodes->includes(
    self)).transitions
--     ->select(t | t.source = self and t.condition <> null).condition
  -- J3: Transitions starting in junctions cannot have triggers
  inv J3: NodeContainer.allInstances()->select(nc | nc.nodes->includes(self
      )).transitions
    ->select(t | t.source = self)->forAll(t | t.trigger = null or t.trigger
        ._type = TriggerType::EMPTY)


-- Final states well-formedness conditions (RoboChart reference 3.1.10)
context Final
  -- FS1: Final states cannot be the source of transitions
  inv FS1: NodeContainer.allInstances()->select(nc | nc.nodes->includes(
      self)).transitions
    ->select(t | t.source = self)->isEmpty()


-- Triggers well-formedness conditions (RoboChart reference 3.1.11)
context Trigger
  -- Tg1: A trigger of type SIMPLE has neither the parameter attribute not
      the value attribute set. This is a pure synchronisations and does not
       involve exchange of values
  inv Tg1: self._type = TriggerType::SIMPLE implies (self.parameter = null
      and self.value = null)
  -- Tg2: A trigger of type SIMPLE must use a typeless event. This is a
      pure synchronisations and does not involve exchange of values
  inv Tg2: self._type = TriggerType::SIMPLE implies (self.event <> null and
       self.event.type = null)
  -- Tg3: A trigger of type INPUT must have a parameter attribute and
      cannot have its value attribute set
  inv Tg3: self._type = TriggerType::INPUT implies (self.parameter <> null
      and self.value = null)
  -- Tg4: A trigger of type OUPUT or SYNC must have a value attribute and
      cannot have its parameter attribute set
  inv Tg4: (self._type = TriggerType::OUTPUT or self._type = TriggerType::
      SYNC)
    implies (self.value <> null and self.parameter = null)
```

```
  -- Tg5: A trigger of type empty must not have its attributes event,
      parameter and value set
  inv Tg5: self._type = TriggerType::EMPTY
    implies (self.event = null and self.parameter = null and self.value =
        null)


-- Transitions well-formedness conditions (RoboChart reference 3.1.12)
context Transition
  -- T1: The source and target of a transition must belong to the same
      container
  inv T1: NodeContainer.allInstances()
    ->one(nc | nc.nodes->includes(self.source) and nc.nodes->includes(self.
        target))
  -- T2: If a transition has a trigger, it must be of type INPUT or SIMPLE
  inv T2: self.trigger <> null
    implies (self.trigger._type = TriggerType::INPUT
      or self.trigger._type = TriggerType::SIMPLE
      or self.trigger._type = TriggerType::EMPTY
    )


-- Operations well-formedness conditions (RoboChart reference 3.1.13)
context OperationDef
  -- O1: All state-machine conditions apply to operation definitions
  inv O1:
    -- STM1: State machines cannot have provided interfaces
    self.pInterfaces->isEmpty()
    -- STM2: Operations in state machines can only be required, not defined
    -- i.e. operations must not be declared directly in the state machine (
        defined interfaces can't have operations anyway by I2)
    and self.operations->isEmpty()
    -- STM3: Every state machine must have exactly one initial junction
    and self.nodes->selectByKind(Initial)->one(true)
    -- STM4: State machines must contain at least one state
    and self.nodes->selectByKind(State)->exists(true)
    -- STM5: The names of variables, operations, and events are unique to
        the machine
    and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
        pInterfaces))
        .variableList->union(self.variableList).vars->isUnique(i | i)
      and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
          pInterfaces))
        .operations->isUnique(i | i)
      and self->collect(c | c.interfaces->union(c.rInterfaces)->union(c.
          pInterfaces))
        .events->union(self.events)->isUnique(i | i)
    -- STM6: State machines must not have operations declared directly
        within them
    and self.operations->isEmpty()
```

```
-- Variables well-formedness conditions (RoboChart reference 3.1.14)
context Variable
  -- V1: If the initial value of a required variable or constant of a state
       machine or controller is defined, it must be consistent with the
      initial value of any (complementing) variable provided or required by
       the contexts (controllers or modules) where the state machine or
      controller is used
  -- NOTE: this requires expression evaluation in order to be properly
      defined


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


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

-- Clock expression well-formedness conditions
context ClockExp
  -- CE1: An expression since(C) may only occur as part of a transition
      guard
  inv CE1: self.parentIsTransition()
```

```
  -- CE2: An expression since(C) may only occur in a branch of a comparison
       expression in which the other branch is an integer or float
     expression
  inv CE2: Expression::ComparisonExpression()->exists(comp |
  (comp.left = self
      and (comp.right.oclIsKindOf(IntegerExp) or comp.right.oclIsKindOf(
          FloatExp)))
    or (comp.right = self
      and (comp.left.oclIsKindOf(IntegerExp) or comp.left.oclIsKindOf(
          FloatExp)))
  )
  -- CE3: The clock C in an expression since(C) may only reference a clock
     declared within the expression's containing state-machine
  inv CE3: self.parentIsTransition() implies
    self.parentTransition().containingStateMachine().clocks->includes(self.
        clock)

-- State Clock expression well-formedness conditions
context StateClockExp
  -- SCE1: An expression sinceEntry(S) may only occur as part of a
     transition guard
  inv SCE1: self.parentIsTransition()
  -- SCE2: An expression sinceEntry(S) may only occur in a branch of a
     comparison expression in which the other branch is an integer or
     float expression
  inv SCE2: ComparisonExpression()->exists(comp |
    (comp.left = self
      and (comp.right->oclIsKindOf(IntegerExp) or comp.right->oclIsKindOf(
          FloatExp)))
    or (comp.right = self
      and (comp.left->oclIsKindOf(IntegerExp) or comp.left->oclIsKindOf(
          FloatExp)))
  )
  -- SCE3: The state S in an expression sinceEntry(S) may only reference a
     state within the containing expression's state-machine. When the name
      S is ambiguous, because, for instance, there is a state and a
     substate with the same name in the state machine, the fully qualified
      name of the state S must be used.
  -- the state is referenced, not named in the metamodel, so the well-
     formedness conditions here are not concerned with resolving the
     ambiguity
  inv SCE3: self.parentIsTransition() implies
    self.parentTransition().containingStateMachine().nestedStates()->
        includes(self.state)


-- Timed Statements well-formedness conditions (RoboChart reference 3.2.2)
context ClockReset
  -- TS1: A clock reset #C may only reference a clock declared within the
     action's containing state-machine, or in the case of a trigger,
```

```
      within the trigger's containing state-machine
  inv TS1: self.containingStateMachine().clocks->includes(self.clock)


-- Auxiliary definitions

-- function to extract RoboticPlatformDef from a RoboticPlatform (which may
     be a ref)
context RoboticPlatform
  def: rpDef() : RoboticPlatformDef =
    if self.oclIsKindOf(RoboticPlatformDef) then
      self.oclAsType(RoboticPlatformDef)
    else
      self.oclAsType(RoboticPlatformRef).ref
    endif
-- function to extract ControllerDef from a Controller (which may be a ref)
context Controller
  def: controllerDef() : ControllerDef =
    if self.oclIsKindOf(ControllerDef) then
      self.oclAsType(ControllerDef)
    else
      self.oclAsType(ControllerRef).ref
    endif
-- function to extract StateMachineDef from a StateMachine (which may be a
    ref)
context StateMachine
  def: stmDef() : StateMachineDef =
    if self.oclIsKindOf(StateMachineDef) then
      self.oclAsType(StateMachineDef)
    else
      self.oclAsType(StateMachineRef).ref
    endif

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

```
        statementInputEvents())
  def: ncOutputEvents() : Bag(Event) =
    self.transitions->select(t | t.trigger <> null and
      (t.trigger._type = TriggerType::OUTPUT or t.trigger._type =
          TriggerType::SYNC)
    ).trigger.event
    ->union(self.transitions->select(t | t.action <> null).action.
        statementOutputEvents())
    ->union(self.nodes->selectByKind(NodeContainer).ncOutputEvents())
    ->union(self.nodes->selectByKind(State).actions.action.
        statementOutputEvents())
context Statement
  def: statementInputEvents() : Bag(Event) = Bag{}
  def: statementOutputEvents() : Bag(Event) = Bag{}
context SendEvent
  def: statementInputEvents() : Bag(Event) =
    Set{self.trigger}->select(t | t._type = TriggerType::INPUT).event
  def: statementOutputEvents() : Bag(Event) =
    Set{self.trigger}->select(t | Set{TriggerType::OUTPUT, TriggerType::
        SYNC, TriggerType::SIMPLE}->includes(t._type)).event
context SeqStatement
  def: statementInputEvents() : Bag(Event) =
    self.statements.statementInputEvents()->asBag()
  def: statementOutputEvents() : Bag(Event) =
    self.statements.statementOutputEvents()->asBag()
context IfStmt
  def: statementInputEvents() : Bag(Event) =
    self._'then'.statementInputEvents()->union(self._'else'.
        statementInputEvents())
  def: statementOutputEvents() : Bag(Event) =
    self._'then'.statementOutputEvents()->union(self._'else'.
        statementOutputEvents())

-- functions on expressions to support timed expressions well-formedness
    conditions
context Expression
  def: parentIsTransition() : Boolean =
    if self.oclContainer().oclIsKindOf(Expression) then
      self.oclContainer().oclAsType(Expression).parentIsTransition()
    else
      Transition.allInstances()->exists(t | t.condition = self)
    endif
  def: parentTransition() : Transition =
    if self.oclContainer().oclIsKindOf(Expression) then
      self.oclContainer().oclAsType(Expression).parentTransition()
    else
      self.oclContainer().oclAsType(Transition)
    endif

  def: ComparisonExpression() : Set(BinaryExpression) =
```

```
    BinaryExpression.allInstances()->select(x |
      x.oclIsKindOf(Equals) or x.oclIsKindOf(Different)
      or x.oclIsKindOf(GreaterThan) or x.oclIsKindOf(LessThan)
      or x.oclIsKindOf(GreaterOrEqual) or x.oclIsKindOf(LessOrEqual)
    )

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

    -- functions to find the containing state machine for various
        constructs,
    -- particularly ClockReset to define the timed statement well-
        formedness conditions
context NodeContainer
  def: containingStateMachine() : StateMachineBody =
    if self.oclIsKindOf(StateMachineBody) then
      self.oclAsType(StateMachineBody)
    else
      self.oclContainer().oclAsType(NodeContainer).containingStateMachine()
    endif
context Transition
  def: containingStateMachine() : StateMachineBody =
    self.oclContainer().oclAsType(NodeContainer).containingStateMachine()
context Action
  def: containingStateMachine() : StateMachineBody =
      self.oclContainer().oclAsType(State).containingStateMachine()
context Statement
  def: containingStateMachine() : StateMachineBody =
    if self.oclContainer().oclIsKindOf(Action) then
      self.oclContainer().oclAsType(Action).containingStateMachine()
    elseif self.oclContainer().oclIsKindOf(Transition) then
      self.oclContainer().oclAsType(Transition).containingStateMachine()
    else
      self.oclContainer().oclAsType(Statement).containingStateMachine()
    endif
context Trigger
  def: containingStateMachine() : StateMachineBody =
    if self.oclContainer().oclIsKindOf(Transition) then
      self.oclContainer().oclAsType(Transition).containingStateMachine()
    else
      self.oclContainer().oclAsType(Statement).containingStateMachine()
    endif
context ClockReset
  def: containingStateMachine() : StateMachineBody =
    if self.oclContainer().oclIsKindOf(Action) then
      self.oclContainer().oclAsType(Action).containingStateMachine()
    elseif self.oclContainer().oclIsKindOf(Trigger) then
```

```
      self.oclContainer().oclAsType(Trigger).containingStateMachine()
    else
      self.oclContainer().oclAsType(Statement).containingStateMachine()
    endif

endpackage
```

Credits

🤖 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
D Icon made by Revicon from www.flaticon.com is licensed by CC 3.0 BY
F Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
O Icon made by Icomoon from www.flaticon.com is licensed by CC 3.0 BY
⇥ Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
⇥ Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
π Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
🅿 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
® Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
🕒 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
Ⓟ Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
🅰 Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
⬡ Icon made by Freepik from www.flaticon.com is licensed by CC 3.0 BY
ⓘ Icon made by Popcic from www.flaticon.com is licensed by CC 3.0 BY

Bibliography

[1] ISO/IEC 13568:2002. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*. International Standard (cited on pages 22, 23, 153).

[2] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. "A Refinement Strategy for *Circus*". In: *Formal Aspects of Computing* 15.2 - 3 (2003), pages 146–181 (cited on page 69).

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. "A Refinement Strategy for *Circus*". In: *Formal Aspects of Computing* 15.2 - 3 (2003), pages 146–181 (cited on page 139).

[4] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. "Unifying Classes and Processes". In: *Software and System Modelling* 4.3 (2005), pages 277–296 (cited on page 139).

[5] A. Miyazawa et al. "RoboChart: modelling and verification of the functional behaviour of robotic applications". In: *Software & Systems Modeling* (2019), pages 1–53 (cited on page 35).

[6] A. Sherif et al. "A process algebraic framework for specification and validation of real-time systems". In: *Formal Aspects of Computing* 22.2 (2010), pages 153–191 (cited on pages 14, 139).

[7] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011 (cited on page 14).

[8] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011 (cited on page 66).

[9] James Baxter, Pedro Ribeiro, and Ana Cavalcanti. "Sound reasoning in tock-CSP". In: *Acta Informatica* (2021) (cited on page 163).

[10] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998 (cited on page 14).

[11] Ana Cavalcanti et al. "Verified simulation for robotics". In: *Science of Computer Programming* 174 (2019), pages 1–37. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2019.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0167642318301655.

[12] F. Zeyda et al. "A modular theory of object orientation in higher-order UTP". In: *Formal Methods*. Volume 8442. Lecture Notes in Computer Science. Springer, 2014, pages 627–642 (cited on page 139).

[13] H. Zhu et al. "Denotational Semantics for a Probabilistic Timed Shared-Variable Language". In: *Unifying Theories of Programming*. Edited by B. Wolff, M.-C. Gaudel, and A. Feliachi. Volume 7681. Lecture Notes in Computer Science. Springer, 2013, pages 224–247 (cited on page 14).

[14] Jifeng He, Carroll Morgan, and Annabelle McIver. "Deriving Probabilistic Semantics Via the 'Weakest Completion'". In: *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*. Edited by Jim Davies, Wolfram Schulte, and Michael Barnett. Volume 3308. Lecture Notes in Computer Science. Springer, 2004, pages 131–145. DOI: 10.1007/978-3-540-30482-1_17. URL: https://doi.org/10.1007/978-3-540-30482-1_17 (cited on page 67).

[15] J. A. Hilder et al. "Chemical Detection Using the Receptor Density Algorithm". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pages 1730–1741 (cited on page 143).

[16] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996 (cited on pages 22, 27, 69, 139).

[17] H Jensen. "Model checking probabilistic real time systems". In: *Proc. 7th Nordic Workshop on Programming Theory*. Citeseer. 1996, pages 247–261 (cited on page 68).

[18] Mark Kattenbelt et al. "A game-based abstraction-refinement framework for Markov decision processes". In: *Formal Methods in System Design* 36.3 (2010), pages 246–280 (cited on page 68).

[19] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In: *CAV*. Edited by Ganesh Gopalakrishnan and Shaz Qadeer. Volume 6806. Lecture Notes in Computer Science. Springer, 2011, pages 585–591. ISBN: 978-3-642-22109-5. URL: http://dblp.uni-trier.de/db/conf/cav/cav2011.html#KwiatkowskaNP11 (cited on page 65).

[20] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of probabilistic real-time systems". In: *International conference on computer aided verification*. Springer. 2011, pages 585–591 (cited on page 68).

[21] Marta Kwiatkowska et al. "Automatic verification of real-time systems with discrete probability distributions". In: *Theoretical Computer Science* 282.1 (2002), pages 101–150 (cited on page 68).

[22] Marta Kwiatkowska et al. "Performance analysis of probabilistic timed automata using digital clocks". In: *Formal Methods in System Design* 29.1 (2006), pages 33–78 (cited on page 68).

[23] Alvaro Miyazawa et al. "RoboChart: Modelling and verification of the functional behaviour of robotic applications". In: *Software & Systems Modeling* (Jan. 2019). ISSN: 1619-1374. DOI: 10.1007/s10270-018-00710-z. URL: doi.org/10.1007/s10270-018-00710-z (cited on page 68).

[24] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1 (cited on page 147).

[25] *OMG Unified Modeling Language*. Object Management Group. Mar. 2015. URL: www.omg.org/spec/UML/2.5/ (cited on page 13).

[26] R. Alur and D. L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pages 183–235 (cited on page 13).

[27] S. Foster, F. Zeyda, and J. C. P. Woodcock. "Isabelle/UTP: A Mechanised Theory Engineering Framework". In: *Unifying Theories of Programming*. Edited by D. Naumann. Volume 8963. Lecture Notes in Computer Science. Springer, 2015, pages 21–41 (cited on page 66).

[28] S. Foster et al. "Towards a UTP semantics for Modelica". In: *Unifying Theories of Programming*. Lecture Notes in Computer Science. Springer, 2016 (cited on page 139).

[29] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000 (cited on page 13).

[30] T. A. Henzinger. "The theory of hybrid automata". In: *11th Annual IEEE Symposium on Logic in Computer Science*. 1996, pages 278–292 (cited on page 139).

[31] T. Gibson-Robinson et al. "FDR3 - A Modern Refinement Checker for CSP". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2014, pages 187–201 (cited on page 14).

[32] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002 (cited on page 66).

[33] Jim Woodcock et al. "Probabilistic Semantics for RoboChart - A Weakest Completion Approach". In: *Unifying Theories of Programming - 7th International Symposium, UTP 2019, Dedicated to Tony Hoare on the Occasion of His 85th Birthday, Porto, Portugal, October 8, 2019, Proceedings*. Edited by Pedro Ribeiro and Augusto Sampaio. Volume 11885. Lecture Notes in Computer Science. Springer, 2019, pages 80–105. DOI: 10.1007/978-3-030-31038-7_5. URL: https://doi.org/10.1007/978-3-030-31038-7_5 (cited on page 67).

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.

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.

Index