# *RoboSim* Reference Manual

Ana Cavalcanti Augusto Sampaio
Pedro Ribeiro Alvaro Miyazawa
Madiel Conserva Filho Andre Didier

WWW.CS.YORK.AC.UK/ROBOSTAR

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by-nc/3.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

# Contents


1 Introduction . . . . . 7

2 *RoboSim* metamodel . . . . . 9

3 Well-formedness Conditions . . . . . 13

3.1 Robotic Platforms 13

3.2 Modules 13

3.3 Controllers 14

3.4 State Machine 14

3.5 Transitions 15

3.6 Statements 15

3.7 Events 15

4 Semantics . . . . . 17

4.1 Module 18

4.2 Controllers 21

4.3 State Machines 23

4.4 NodeContainers 25

4.5 Transitions 26

4.6 Statements 28

5 Conformance . . . . . 29

5.1 A1 29

5.2 A2 30

5.3 A3 30
5.4 Assumptions 31
6 Conclusions 33
A *RoboSim* Metamodel 35
Bibliography 37
Index of Semantic Rules 39
Index of Calls to Semantic Rules 41

# List of rules


1 Semantics of modules . . . 18
2 Function Inputs for Module . . . 18
3 Cycle for modules . . . 19
4 Module or controller memory . . . 19
5 Composition of controllers . . . 20
6 Renaming controller . . . 20
7 Semantics of controllers . . . 21
8 Cycle for controllers . . . 22
9 Composition of machines . . . 22
10 Renaming machine . . . 22
11 Semantics of state machine . . . 23
12 Semantics of state machine . . . 23
13 Get and set channels . . . 23
14 Cycle for machines . . . 23
15 Constants Initialisation for State Machines . . . 24
16 Behaviour of state machine . . . 24
17 Semantics of state machine . . . 24
18 State-machine memory . . . 25
19 Get and Set channels . . . 25
20 Semantics of defined operations . . . 25
21 Node container semantics . . . 25
22 Node container behaviour . . . 26
23 Semantics of node container . . . 26
24 Node container core behaviour . . . 26
25 Semantics of Transitions . . . 26
26 Semantics of a Transition . . . 27

| | | |
|---|---|---|
| 27 | Transition events | 27 |
| 28 | Semantics of a transition's trigger | 27 |
| 29 | Rename transition trigger events | 28 |
| 30 | Semantics of SimCall | 28 |
| 31 | Semantics of OutputCommunication | 28 |
| 32 | Semantics of Assignment | 28 |
| 33 | ExecStatement | 28 |
| 1 | A1 | 29 |
| 2 | A1Event | 29 |
| 3 | TA1 | 30 |
| 4 | TA2 | 30 |
| 5 | read | 30 |
| 6 | TA3 | 30 |
| 7 | Assumptions | 31 |
| 8 | Constrained | 31 |

# 1. Introduction

In this report, we present a state-machine based notation, called *RoboSim*, designed specifically for the modelling of verified simulations for robotic applications. As a graphical notation, *RoboSim* is intended to be more appealing than programming notations; furthermore, a model in *RoboSim* should be considered as an intermediate, implementation independent, representation that can be translated into languages used by several well-established simulators such as Netlogo [16], MASON [11], JASON [8], Argos [5], Enky (`home.gna.org/enki/`), Microsoft Robotics Developer Studio (`www.microsoft.com/robotics`), Webots, and Simulink + Stateflow [12, 13]. The graphical notation of *RoboSim* is inspired on *RoboChart* [9], a UML-like notation designed for robotics. However, as a simulation notation, it has the following

- A model in *RoboSim* specifies a cyclic mechanism, as is usual in languages like Netlogo and Stateflow, among others.
- The cycle itself is a separate mechanism from the executing machine; a *RoboSim* model is formed of the parallel execution of the cycle and the machine.
- Time information is restricted to the cycle; the model of the machine is untimed.
- The visible events are readings and writings in registers of sensors and actuators.
- These events are always available.

Additionally, *RoboSim* has some distinguishing features.

- The cycle step can be more flexibly defined than, for instance, in Stateflow. Particularly, a step is not constrained to entering a single state of the machine.
- A formal semantics for *RoboSim* is given in the same framework as that for *RoboChart*, which allows the definition of a notion of correct simulation with respect to the more abstract

*RoboChart* model.

- Other relevant properties of a simulation can also be mechanically checked like. For instance, the improper execution of the same operation more than once in a same cycle.

Like *RoboChart*, apart from the state-machine itself, *RoboSim* 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 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 is synchronous. Operations can be given pre and postconditions.

In this report, we formalise the semantics of *RoboSim* using CSP [3]. Importantly, CSP is a front end for a mathematical model that supports a number of analysis techniques that allow, for example, checking the validity of a simulation. For that we can use model-checking, which provides a high degree of automation, as well as more powerful (but not automatic) verification based on theorem proving. Use of CSP enables model checking with FDR [15]. On the other hand, CSP has an underlying model based Hoare and He's Unifying Theories of Programming [4] that makes our core semantics adequate for theorem proving and for extension to deal, for instance, with probability [6].

The remainder of this manual is structured as follows. Chapter 2 presents the *RoboSim* metamodel, and Chapter 3 defines their well-formedness conditions. The semantics is presented in Chapter 4. Finally, Chapter 6 concludes with a summary of the results and future work.

# 2. *RoboSim* metamodel

*RoboSim* models are structured using the elements sketched in Figure 2.1.The structure of a *RoboSim* package follows *RoboChart* structure, namely, modules, robotic platforms, controllers, simulation machines.

A simulation is defined by a Module, which includes a robotic platform and at least one Controller, which can each include one or more simulation machines. In the metamodel, RoboticPlatforms, Controllers, and StateMachines are ConnectionNodes. The restrictions on cardinality are well-formedness conditions. The ConnectionNodes are linked by Connections via their events efrom and eto. The connections can be asynchronous and bidirectional; further restrictions are well-formedness conditions.

Modules, controllers, and machines are self-contained: they declare the full Context (variables, operations, and events) used in their definitions, possibly via interfaces, to allow compositional reasoning. The main difference from *RoboChart* is that these elements include the cycle definition.

A variable cycle of type nat of natural numbers is implicitly declared in every model. Its value is defined by a boolean Expression given in each module, controller, and state machine, so that they are self-contained. Modules, controllers, and state machines define the valid sample times for their simulation by specifying restrictions on the value of cycle.

The cycle specifications may admit several valid periods. In general, however, an actual value needs to be defined just in association with a particular simulation execution. If a component (module, controller, or machine) does not explicitly specify it, a value for cycle that satisfies the specification can be identified, for example, when generating code for a simulator.

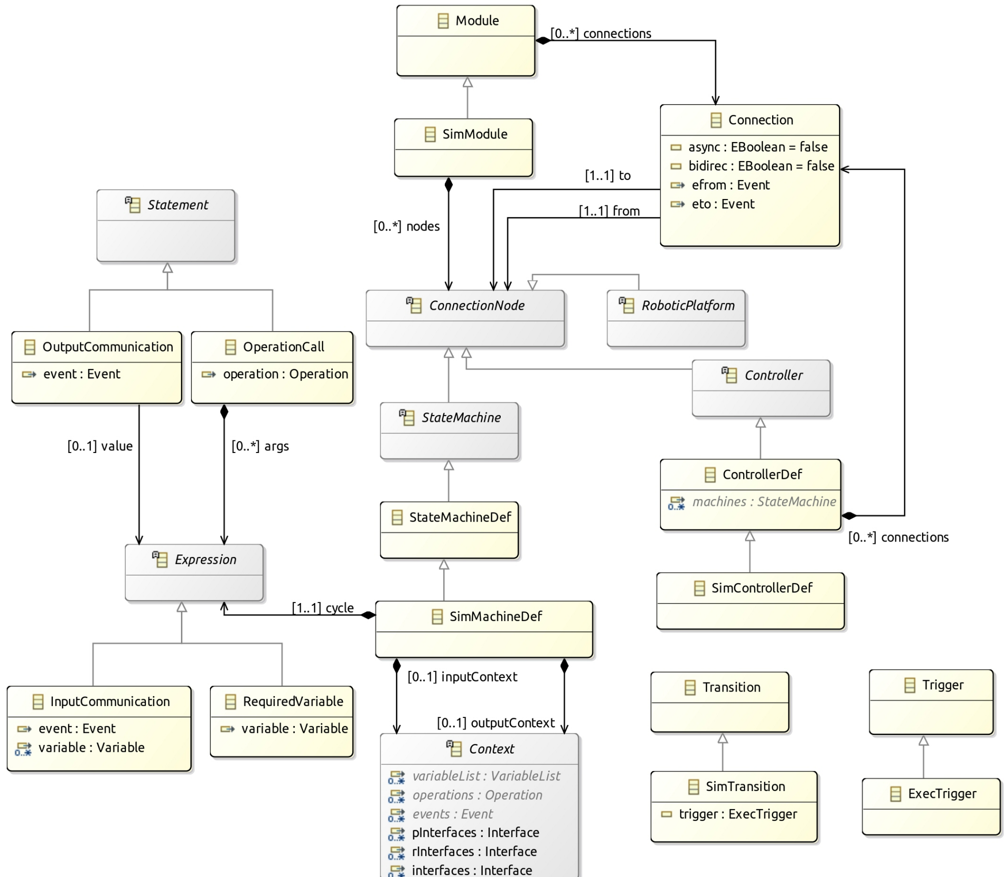


Figure 2.1: *RoboSim* metamodel

A module and each of its controllers may have different specifications for the value of cycle; similarly, a controller and each of its machines may also have different specifications. Ultimately, it is the module that defines the cycle of the simulation of a complete model. There has to be, therefore, at least one value that satisfies all the restrictions of the module and its controllers, or of a controller and its state machines.

A simulation-machine definition (SimMachineDef) is a StateMachine. Another form of StateMachine, is a reference to a state machine.

Declarations in a simulation machine are partitioned into three Contexts: local declarations, inputs and outputs. Input and output events have a semantics different from that in *RoboChart*. There, they are events raised by the platform or the software. In *RoboSim*, an input or output is represented by a boolean that indicates whether that event has occurred. If any values are communicated, they are meaningful only if the event occurs.

Calls to operations are treated as outputs in *RoboSim*. Programmatically, in a simulation cycle, operations may be called during the processing of inputs, but are only actually executed later, at the point of the cycle when the registers are written.

A required variable is also external to the machine, and so its value is read, in the case of an input variable, and written, for output variables, on a cyclic basis. A variable or event may be both an input and an output.

To make the semantic differences clear, references to inputs and outputs are all preceded by a $ in *RoboSim*. In the metamodel (see Figure 2.1), we have an extra form of Expression that allows the use of an input Event as a variable (InputCommunication). If the input communicates a value, that is recorded as part of that communication. An InputCommunication can take place only as part of a transition trigger, so that any variable assignments are an implicit part of the transition assignment. Expressions do not have side effects. Another new form of Expression, namely, RequiredVariable, allows references to a variable preceded by a $.

OutputCommunications and OperationCalls can be used in actions, and are, therefore, extra forms of Statement. Like *RoboChart*, *RoboSim* allows other types of Statement such as Assignment, which assigns a value to a variable, IfStmt, which conditionally executes one of two statements, and Clock Reset, which allows to reset a clock. A special form of Transition, namely, SimTransition, allows the use of the special event exec as a trigger; it does not need to be declared. No other triggers are possible in a *RoboSim* model.

Despite the semantic distinctions, we have interface declarations in *RoboSim* exactly like in *RoboChart*, since this facilitates the modularisation of declarations, and the transition from a *RoboChart* to a *RoboSim* model. Particularly, this allows the designer to reuse entirely the robotic platform in the simulation.

As opposed to *RoboChart* clocks, *RoboSim* clocks advance in each cycle by the number of time units of the period for the cycle. So, choice of that period needs to take into account the budgets and deadlines required for the model.

# 3. Well-formedness Conditions

We now define a number of well-formedness conditions that identify the valid models written using the *RoboSim* metamodel presented in the previous chapter. Many conditions are those expected of a standard state-machine notation, and are fully described in the definition of *RoboChart* [9]. In what follows, we present the well-formedness conditions that are specific to *RoboSim* or related to constructs specific to *RoboSim*.

## 3.1 Robotic Platforms

The well-formedness conditions associated with robotic platforms involve the declaration of elements of the provided interfaces.

**SRP1** Provided interfaces must declare just variables and operations. There is no notion of providing events in *RoboSim*, since events are taken to be elements of a component, which are used to establish a connection with another component for communication.

## 3.2 Modules

**SM1** The specification of cycle is a boolean expression. It is a predicate that specifies a restriction on the values (positive natural numbers) that the variable cycle may take.

**SM2** The conjunction of the cycle specification of all the controllers and of the module itself is not false. This means that it is possible to choose a valid positive value for cycle for use in a simulation of the module.

**SM3** The outputs of the controllers are disjoint, so that there is no conflict when writing to the actuator registers. In the case of events, this is ensured by the connections like in *RoboChart*: an event of the platform is linked to at most one controller. Required variables, however, may be required by several controllers, and should be an output just in one. An operation should be required just by one controller.

**SM4** The *RoboChart* facility for creating module instances (related to collections) is not allowed.

## 3.3 Controllers

**SC1** The specification of cycle is a boolean expression.

**SC2** The conjunction of the cycle specifications of all the simulation machines, of all the machines that define operations, and of the controller itself is not false. Besides machines, a controller may define operations used by its machines. These operations may be defined by simulation machines themselves, and their cycle specifications also need to be feasible in the context of the controller. This condition is similar to **SM2**, but refers to the specifications of cycle in the machines of a controller and in the controller itself, while **SM2** refers to the specifications of the controllers of a module and of the module itself. **SM2** is not concerned with restrictions in machines.

**SC3** Connections with a simulation machine must respect its input and output definitions. For instance, an input event of a machine must be either not connected or the connection must be an input to that machine. Similarly for outputs: they must be either not connected, or connected to an output from that machine.

**SC4** The outputs of the state machines are disjoint, to avoid conflicts as in the case of module.

## 3.4 State Machine

**SSTM1** The specification of cycle is a boolean expression.

**SSTM2** Input declarations can be only events and variables.

**SSTM3** Output declarations can be events, variables, or operations.

**SSTM4** Required variables must be inputs and outputs, since they can be read and written, like in *RoboChart*.

**SSTM5** Variables defined in a machine are there only for local use.

**SSTM6** All platform operations must be outputs. Calls to operations provided by the controller, however, do not become outputs of the system.

**SSTM7** Inputs, outputs, and required variables are referenced using a $. This emphasises the fact that they are handled in a cyclic pattern.

**SSTM8** Local event declarations are not allowed.

**SSTM9** The input and output contexts have only events and interfaces that can be defined or required.

## 3.5 → Transitions

**ST1** A transition cannot have deadline and probability junctions.

**ST2** The only possible trigger is exec.

## 3.6 Statements

**SS1** Timed and wait statements are not allowed.

## 3.7 Events

**SE1** There is no broadcast attribute.

**SE2** Exec is the only possible event that can be used as trigger.

# 4. Semantics

The semantics of *RoboSim* is based on the timed semantics of *RoboChart*, which is fully described in the *RoboChart* reference manual [9]. In what follows, we present the rules that are specific to *RoboSim*.

## 4.1 Module

**Rule 1. Semantics of modules**

$[\![ \mathsf{m : SimModule} ]\!]_{\mathcal{M}} : \mathsf{TimedCSPProcess} =$

$$\left(\left(\left(\left(\left(\begin{array}{l}\mathsf{cycleModule}^1(\mathsf{m}, \mathsf{ctrls}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{rvars}, \mathsf{evars})\\ [\![ \left\{\!\left| registerWriteC.\mathsf{data}(\mathsf{v}) \mid v : \mathrm{ran}(rvars \cup evars) \right|\!\right\} \cup \\ \{\!| registerWriteE |\!\} \cup \\ \left\{\!\left| registerReadC.\mathsf{data}(\mathsf{v}) \mid \mathsf{v} : \mathrm{dom}\,\mathsf{evars} \right|\!\right\} ]\!] \\ \mathsf{memoryComp}^1(\mathsf{rvars}, \mathsf{evars}, \mathsf{rp})\end{array}\right) \\ \setminus \{\!| registerWriteE |\!\}\right) \\ [\![ \mathsf{setConstants} \cup \\ \{\!| registerReadC, registerWriteC, endExecC, end |\!\} ]\!] \\ \mathsf{composeControllers}^1(\mathsf{m}, \mathsf{ctrls}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{rvars}, \mathsf{evars})\right) \\ \setminus \mathsf{setConstants} \cup \\ \left\{\!\left| registerReadC, registerWriteC, endexec.\mathsf{id}(\mathsf{m}) \right|\!\right\} \cup \{\!| \mathsf{endExecC} |\!\}\right) \\ \Theta_{\{end\}}\ SKIP\right) \\ \setminus \{end\}$$

**where**

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

$\mathsf{ctrls} = \mathsf{controllersDefinition}(\mathsf{m})$

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

$\mathsf{inputs} = \mathsf{Inputs}(\mathsf{rp}, \mathsf{cons})$

$\mathsf{outputs} = \mathsf{Outputs}(\mathsf{rp}, \mathsf{ctrls})$

$\mathsf{rvars} = \mathsf{Rvars}(\mathsf{rp}, \mathsf{ctrls})$

$\mathsf{evars} = \mathsf{Internals}(\mathsf{cons})$

$\mathsf{setConstants} = \left\{\!\left| \mathsf{c : allConstants(rp)} \bullet set_\mathsf{name}(\mathsf{c}) \right|\!\right\}$

$\mathsf{endExecC} = \{\mathsf{c : ctrls} \bullet \mathsf{endexec.id(c)}\}$

**Rule 2. Function Inputs for Module**

$\mathsf{Inputs}(\mathsf{rp : RoboticPlatform}, \mathsf{cons : Set(Connection)}) : \mathsf{Set(InputsType)} =$

$\{\mathsf{c : cons} \mid \mathsf{c.from} = \mathsf{rp} \land \mathsf{c.to} \in \mathsf{Controller} \bullet (\mathsf{id(c.efrom)}, \mathsf{id(c.eto)})\}$

**Rule 3. Cycle for modules**

$\mathsf{cycleModule}(\mathsf{m} : \mathsf{SimModule}, \mathsf{ctrls} : \mathsf{Seq}(\mathsf{SimControllerDef}), \mathsf{inputs} : \mathsf{Set}(\mathsf{InputsType}), \mathsf{outputs} : \mathsf{Set}(\mathsf{OutputsType}), \mathsf{rvars} : \mathsf{Set}(\mathsf{OutputsType}), \mathsf{evars} : \mathsf{Set}(\mathsf{EventsType})) : \mathsf{TimedCSPProcess} =$

$$
\begin{array}{l}
\mathbf{let}\ CycleModule = \\
\quad ((\,|||\ in : \mathrm{dom}\,\mathsf{inputs} \bullet registerRead.\mathsf{take(in)} \rightarrow \\
\qquad\qquad registerReadC.\mathsf{give(inputs(in))} \rightarrow SKIP); \\
\quad (\,|||\ evar : \mathrm{dom}\,\mathsf{evars} \bullet registerReadC.\mathsf{take(evar)} \rightarrow SKIP); \\
\quad (\,|||\ evar : \mathrm{dom}\,\mathsf{evars} \bullet registerWriteE.\mathsf{data(evar)} \rightarrow SKIP); \\
\quad (CollectOutputs(\mathrm{dom}(\mathsf{allOutputs}))\ [|\ endexec.\mathsf{id(m)}\ |]\ Monitor))\ \triangle\ Stop_U); \\
\quad ((end \rightarrow SKIP) \rhd_1 (wait([\![\mathsf{m.cycleDef}]\!]_{\mathcal{E}xpr} - 1))); \\
\quad CycleModule \\
\quad CollectOutputs(sout) = \\
\qquad (registerWriteC?\mathsf{outp} : \{\mathsf{out} : \mathrm{dom}(\mathsf{allOutputs} \setminus (\mathsf{evars} \cup \mathsf{rvars}));\ \mathsf{outc} : \mathsf{allOutputs}(\!|\ \mathsf{out}\ |\!); \\
\qquad\quad \mathsf{outp} : \{\!|\mathsf{data(outc)}|\!\} \mid \mathsf{out} \in \mathsf{sout} \bullet \mathsf{outp}\} \rightarrow \\
\qquad\qquad registerWrite!\mathsf{lift(outp)} \rightarrow CollectOutputs(sout \setminus \{\mathsf{project(outp)}\})) \\
\qquad \Box\ endexec.\mathsf{id(m)} \rightarrow SKIP \\
\qquad \Box\ (registerWriteC?\mathsf{rvars} : \{\mathsf{out} : \mathrm{dom}(\mathsf{allOutputs} \cap (\mathsf{evars} \cup \mathsf{rvars}));\ \mathsf{outc} : \mathsf{allOutputs}(\!|\ \mathsf{out}\ |\!); \\
\qquad\quad \mathsf{outp} : \{\!|\mathsf{data(outc)}|\!\} \mid \mathsf{out} \in \mathsf{sout} \bullet \mathsf{outp}\} \rightarrow \\
\qquad\qquad CollectOutputs(sout \setminus \{\mathsf{project(outp)}\})) \\
\quad Monitor = \\
\qquad (\,|||\ i : 1\,..\,\#\mathsf{ctrls} \bullet endexec.\mathsf{id(ctrls(i))} \rightarrow SKIP);\ endexec.\mathsf{id(m)} \rightarrow SKIP \\
\mathbf{within}\ CycleModule \\
\mathbf{where} \\
\mathsf{allOutputs} = \mathsf{outputs} \cup \mathsf{rvar} \cup \mathsf{evars}
\end{array}
$$

The function take maps an input or output to CSP input parameters of a communication.

**Rule 4. Module or controller memory**

$\mathsf{memoryComp}(\mathsf{rvars} : \mathsf{Set}(\mathsf{OutputsType}), \mathsf{evars} : \mathsf{Set}(\mathsf{EventsType}), \mathsf{node} : \mathsf{ConnectionNode}) : \mathsf{TimedCSPProcess} =$

$$
\begin{array}{l}
\mathbf{let}\ Memory(\mathsf{nvars}) = \\
\quad (\Box\ v : \mathrm{dom}\,\mathsf{evars} \bullet registerWriteC.\mathsf{take(v)} \rightarrow Memory(\mathsf{nvars[name(v) := true]}) \\
\qquad\qquad \Box\ registerWriteE.\mathsf{take(v)} \rightarrow Memory(\mathsf{nvars[name(v) := false]})) \\
\quad \Box \\
\quad (\Box\ v : \mathrm{dom}\,\mathsf{rvars} \bullet registerWriteC.\mathsf{take(v)} \rightarrow Memory(\mathsf{nvars})) \\
\quad \Box \\
\quad (\Box\ v : \mathrm{dom}\,\mathsf{evars} \bullet registerReadC.\mathsf{give(evars(v))} \rightarrow Memory(\mathsf{nvars})) \\
\quad \Box \\
\quad (\Box\ rv : \mathrm{ran}\,\mathsf{rvars} \bullet registerReadC.\mathsf{give(rv)} \rightarrow Memory(\mathsf{nvars})) \\
\mathbf{within}\ \mathsf{constInit(node)};\ Memory(\mathsf{initial(nvars)}) \\
\mathbf{where} \\
\quad \mathsf{nvars} = \langle \mathsf{v} : \mathrm{dom}\,\mathsf{rvars} \cup \mathrm{dom}\,\mathsf{evars}) \bullet \mathsf{name(v)} \rangle
\end{array}
$$

The function composeControllers[2] (m, ctrls, inputs, outputs, rvars, evars) basically constructs a parallelism of the processes for each of the controllers in the sequence ctrls.

**Rule 5. Composition of controllers**

composeControllers(m : SimModule, ctrls : Seq(SimControllerDef), inputs : Set(InputsType), outputs : Set(OutputsType), rvars : Set(OutputsType), evars : Set(EventsType) : TimedCSPProcess =

**if** #ctrls = 1
**then**
renamingController[1] (m, head ctrls, inputs, outputs, rvars, evars)
**else**
renamingController[2] (m, head ctrls, inputs, outputs, rvars, evars)
$[| \{| end |\} \cup$ setConstants $|]$
composeControllers[3] (m, tail ctrls, inputs, outputs, rvars, evars)
**where**
setConstants = {c : allConstants(rp) • $set_$vid(c, m)}

**Rule 6. Renaming controller**

renamingController(m : SimModule, c : SimControllerDef, inputs : Set(InputsType), outputs : Set(OutputsType), rvars : Set(OutputsType), evars : Set(EventsType) : TimedCSPProcess =

$$[\![c]\!]_{c'} \left[\left[ \begin{array}{l} \{ct : allConstants(c) \bullet set_vid(ct) \leftarrow set_vid(ct, m)\} \\ \cup \{registerRead \leftarrow registerReadC\} \\ \cup \{registerWrite \leftarrow registerWriteC\} \end{array} \right]\right] \setminus setNotConnectedEvs$$

**where**
setNotConnectedEvs = $\{|$ NotConnectedEvs(inputs, outputs, rvars, evars) $|\}$

## 4.2 Controllers

**Rule 7. Semantics of controllers**

$[\![ \mathsf{c} : \mathsf{SimControllerDef} ]\!]_{\mathcal{C}} : \mathsf{TimedCSPProcess} =$

$$
\begin{array}{l}
\textbf{let} \\
\quad \mathsf{for\ each\ op} : \mathsf{c.lOperations} \bullet \\
\quad\quad \mathsf{nproc(op)}(\{\mathsf{x} : \mathsf{op.parameters} \bullet \mathsf{x.name}\}) = [\![ \mathsf{opdef(op)} ]\!]^{\mathsf{nops}}_{\mathcal{STM}'} \\
\textbf{within} \\
\quad \left( \left( \begin{array}{l}
\left( \begin{array}{l}
\mathsf{cycleController}^1(\mathsf{c}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{dvars}, \mathsf{evars}) \\
[\![ \{\!| \mathit{registerWriteC}.\mathsf{data(out)} \mid \mathsf{out} \leftarrow \mathrm{ran}\, \mathsf{dvars} |\!\} \cup \{\!| \mathit{registerWriteE} |\!\} \\
\cup \{\!| \mathit{registerReadC}.\mathsf{data(v)} \mid \mathsf{v} : \mathrm{dom}\, \mathsf{evars} |\!\} ]\!] \\
\mathsf{memoryComp}^2(\mathsf{dvars}, \mathsf{evars}, \mathsf{c})
\end{array} \right) \\
[\![ \mathsf{setConstants} \cup \{\!| \mathit{registerReadC}, \mathit{registerWriteC}, \mathit{terminate} |\!\} \cup \\
\quad \{\!| \mathsf{endExecM} |\!\} ]\!] \\
\mathsf{composeMachines}^1(\mathsf{c}, \mathsf{machines}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{dvars}, \mathsf{evars})^{\mathsf{nops}}
\end{array} \right) \right. \\
\quad \left. \setminus \{\!| \mathit{registerReadC}, \mathit{registerWriteC}, \mathit{registerWriteE} |\!\} \cup \{\!| \mathsf{endExecM} |\!\} \cup \mathsf{lconsts} \right) \\
\quad \Theta_{\{end\}}\ \mathit{SKIP}
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{machines} = \langle \mathsf{x} : \mathsf{c.machines} \rangle \\
\mathsf{cons} = \mathsf{c.connections} \\
\mathsf{inputs} = \mathsf{InputsC(c, cons)} \\
\mathsf{outputs} = \mathsf{OutputsC(c, cons)} \\
\mathsf{dvars} = \mathsf{DvarsC(c)} \\
\mathsf{evars} = \mathsf{InternalsC(c, cons)} \\
\mathsf{setConstants} = \{\!| \mathsf{ct} : \mathsf{allConstants(c)} \bullet \mathit{set_}\mathsf{name(ct)} |\!\} \\
\mathsf{endExecM} = \{\mathsf{stm} : \mathsf{machines} \bullet \mathit{endexec}.\mathsf{id(stm)}\} \\
\mathsf{opdef(op)} = \mathsf{findOperationDefinition(op, c.lOperations)} \\
\mathsf{nops} = \{\mathsf{op} : \mathsf{c.lOperations} \bullet \mathsf{id(op)} \mapsto \mathsf{nproc(op)}\} \cup \\
\quad \{\mathsf{op} : \mathsf{requiredOperations(c)} \bullet \mathsf{id(op)} \mapsto \mathsf{nproc(op)}\} \\
\mathsf{lvars} = \{\!| \mathsf{v} : \mathsf{allLocalVariables(c)} \bullet \mathit{set_}\mathsf{vid(v)} |\!\} \\
\mathsf{rvars} = \{\!| \mathsf{v} : \mathsf{requiredVariables(c)} \bullet \mathit{set_Ext_}\mathsf{vid(v)} |\!\} \\
\mathsf{lconsts} = \{\!| \mathsf{v} : \mathsf{allLocalConstants(c)} \bullet \mathit{set_}\mathsf{vid(v)} |\!\} \\
\mathsf{rconsts} = \{\!| \mathsf{v} : \mathsf{requiredConstants(c)} \bullet \mathit{set_}\mathsf{vid(v)} |\!\}
\end{array}
$$

**Rule 8. Cycle for controllers**

cycleController(c : SimControllerDef, inputs : Set(InputsType), outputs : Set(OutputsType), dvars : Set(OutputsType), evars : Set(EventsType)) : TimedCSPProcess =

$$
\begin{array}{l}
\textbf{let}\ CycleController = \\
\quad (((\interleave\ inC : \mathrm{dom}\ \mathsf{inputs} \bullet registerRead.\mathsf{take(inC)} \rightarrow \\
\qquad \interleave\ inM : \mathsf{inputs}\ \{\!|\mathsf{inC}|\!\} \bullet registerReadC.\mathsf{give(inM, inputs(inM))} \rightarrow SKIP); \\
\quad (\interleave\ evar : \mathrm{dom}\ \mathsf{evars} \bullet registerReadC.\mathsf{take(evar)} \rightarrow SKIP); \\
\quad (\interleave\ evar : \mathrm{dom}\ \mathsf{evars} \bullet registerWriteE.\mathsf{data(evar)} \rightarrow SKIP); \\
\quad (CollectOutputs(\mathrm{dom}(\mathsf{outputs} \cup \mathsf{dvars})\ [\![ endexec.\mathsf{id(c)} ]\!]\ Monitor))) \triangle Stop_U); \\
\quad ((end \rightarrow SKIP) \triangleright_1 (wait([\![\mathsf{c.cycleDef}]\!]_{\mathcal{E}xpr} - 1))); \\
\quad CycleController \\
\ CollectOutputs(sout) = \\
\quad (registerWriteC?\mathsf{outp} : \{\mathsf{outc} : \mathrm{dom}\ \mathsf{Outputs};\ \mathsf{outm} : \mathsf{Outputs}(\!|\ \mathsf{outc}\ |\!); \\
\qquad \mathsf{outp} : \{\!|\mathsf{data(outm)}|\!\} \mid \mathsf{outc} \in \mathsf{sout} \bullet \mathsf{outp}\} \rightarrow \\
\qquad registerWrite!\mathsf{lift(outp)}) \rightarrow CollectOutputs(sout \setminus \{\mathsf{project(outp)}\}) \\
\quad \Box\ (registerWriteC?\mathsf{outp} : \{\mathsf{varc} : \mathrm{dom}\ \mathsf{dvars};\ \mathsf{varm} : \mathsf{dvars}(\!|\ \mathsf{varc}\ |\!); \\
\qquad \mathsf{outp} : \{\!|\mathsf{data(varm)}|\!\} \mid \mathsf{varc} \in \mathsf{sout} \bullet \mathsf{outp}\} \rightarrow \\
\qquad CollectOutputs(sout \setminus \{\mathsf{project(outp)}\}) \\
\quad \Box\ endexec.\mathsf{id(c)} \rightarrow SKIP \\
\ Monitor = \\
\quad (\interleave\ m : \mathsf{c.machines} \bullet endexec.\mathsf{id(m)} \rightarrow SKIP);\ endexec.\mathsf{id(c)} \rightarrow SKIP \\
\ \textbf{within}\ CycleController
\end{array}
$$

**Rule 9. Composition of machines**

composeMachines(c : SimControllerDef, machines : Seq(SimMachineDef), inputs : Set(InputsType), outputs : Set(OutputsType), rvars : Set(OutputsType), evars : Set(EventsType) : TimedCSPProcess =

$$
\begin{array}{l}
\textbf{if}\ \#\mathsf{machines} = 1 \\
\textbf{then} \\
\quad \mathsf{renamingMachine}^1(\mathsf{c, head\,machines, inputs, outputs, rvars, evars}) \\
\textbf{else} \\
\quad \mathsf{renamingMachine}^2(\mathsf{c, head\,machines, inputs, outputs, rvars, evars}) \\
\qquad [\![ \{\!| end |\!\} \cup \mathsf{setConstants} ]\!] \\
\quad \mathsf{composeMachines}^2(\mathsf{c, tail\,machines, inputs, outputs, rvars, evars}) \\
\textbf{where} \\
\quad \mathsf{setConstants} = \{\!|\ \mathsf{stm} : \mathrm{ran}\ \mathsf{machines}, \mathsf{c} : \mathsf{allConstants(stm)} \bullet set_\mathsf{vid(c)}\ |\!\}
\end{array}
$$

**Rule 10. Renaming machine**

renamingMachine(c : SimControllerDef, stm : SimMachineDef, inputs : Set(InputsType), outputs : Set(OutputsType), rvars : Set(OutputsType), evars : Set(EventsType) : TimedCSPProcess =

$$
[\![\mathsf{stm}]\!]_{\mathcal{STM}^2}
\left[\!\!\left[
\begin{array}{l}
\{\mathsf{ct} : \mathsf{allConstants(stm)} \bullet set_\mathsf{vid(ct)} \leftarrow set_\mathsf{vid(ct, c)}\} \\
\cup \{registerRead \leftarrow registerReadC\} \\
\cup \{registerWrite \leftarrow registerWriteC\}
\end{array}
\right]\!\!\right] \setminus \mathsf{setNotConnectedEvs}
$$

**where**

$$
\mathsf{setNotConnectedEvs} = \{\!|\ \mathsf{NotConnectedEvs(inputs, outputs, rvars, evars)}\ |\!\}
$$

## 4.3 State Machines

**Rule 11. Semantics of state machine**

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

The rule is split according to the type of stm.

**Rule 12. Semantics of state machine**

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

$$\left(\left(\left(\begin{array}{l}\mathsf{behaviour}^{1}(\mathsf{stm}) \\ \quad [\![\mathsf{getsetChannelsStm}^{1}(\mathsf{stm}) \cup \{\!|registerWrite|\!\}]\!] \\ \mathsf{stmMemory}^{1}(\mathsf{stm})\end{array}\right) \setminus \mathsf{getsetChannelsStm}^{2}(\mathsf{stm}) \\ [\![\left\{\!\left| startexec, endexed.\mathsf{id}.(\mathsf{stm}), end, registerRead, registerWrite \right|\!\right\}]\!] \\ \mathsf{cycle}^{1}(\mathsf{stm}, \mathsf{inputs}, \mathsf{outputs})\right) \\ \setminus \{\!|startexec|\!\}\right)$$

$\Theta_{\{end\}}\ SKIP$

**where**

$\mathsf{inputs} = \mathsf{inputs}(\mathsf{stm})$

$\mathsf{outputs} = \mathsf{outputs}(\mathsf{stm})$

$\mathsf{reqVars} = \mathsf{reqVars}(\mathsf{stm})$

$\mathsf{consts} = \mathsf{allConstants}(\mathsf{stm})$

**Rule 13. Get and set channels**

$\mathsf{getsetChannels}(\mathsf{stm} : \mathsf{SimMachineDef}) : \mathsf{ChannelSet} =$

$$\{\!|\mathsf{v} : \mathsf{Evars}(\mathsf{stm}) \bullet get_\mathsf{vid}(\mathsf{v})|\!\} \cup \{\!|\mathsf{v} : \mathsf{Rvars}(\mathsf{stm}) \bullet get_\mathsf{vid}(\mathsf{v})|\!\} \cup$$
$$\{\!|\mathsf{v} : \mathsf{allConstants}(\mathsf{stm}) \bullet get_\mathsf{vid}(\mathsf{v})|\!\} \cup \{\!|\mathsf{v} : \mathsf{allLocalConstants}(\mathsf{stm}) \bullet set_\mathsf{vid}(\mathsf{v})|\!\}$$

**Rule 14. Cycle for machines**

$\mathsf{cycle}(\mathsf{stm} : \mathsf{SimMachineDef}, \mathsf{inputs} : \mathsf{Set}(\mathsf{InputsType}), \mathsf{outputs} : \mathsf{Set}(\mathsf{OutputsType})) :$
$\mathsf{TimedCSPProcess} =$

$$\begin{array}{l}\textbf{let}\ \ Cycle = (((|||\ in : \mathsf{inputs} \bullet registerRead.\mathsf{take}(\mathsf{in}) \to SKIP); \\ \qquad\qquad startexec \to CollectOutputs(\mathsf{outputs})) \triangle Stop_U); \\ \qquad\qquad ((end \to SKIP) \rhd_1 (wait([\![\mathsf{stm.cycleDef}]\!]_{\mathcal{E}xpr} - 1))); \\ \qquad\qquad Cycle \\ \quad CollectOutputs(sout) = registerWrite?\mathsf{giveT}(out) \to CollectOutputs(sout \setminus \{out\}) \\ \qquad\qquad\qquad\qquad\qquad \Box\ endexec.\mathsf{id}(\mathsf{stm}) \to SKIP \\ \textbf{within}\ Cycle\end{array}$$

**Rule 15. Constants Initialisation for State Machines**

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

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

**where**

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

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

**Rule 16. Behaviour of state machine**

$\mathsf{behaviour}(stm : \mathsf{SimMachineDef}) : \mathsf{TimedCSPProcess} =$

$$\begin{array}{l}\textbf{let} \\ \quad Ending = endexec \rightarrow (startexec \rightarrow Ending \,\square\, end \rightarrow SKIP) \\ \textbf{within} \\ \quad \left(\begin{array}{l}(startexec \rightarrow \mathsf{stateful}^{1}(stm) \setminus \{end\}) \\ \quad [\![ \{\!| share_ |\!\} ]\!] \\ SKIP\end{array}\right); \\ \quad Ending\end{array}$$

**Rule 17. Semantics of state machine**

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

$$\begin{array}{l}\textbf{let} \\ \quad Stateful = \left(\left(\begin{array}{l}MachineBody \\ [\![ \mathsf{getsetLocalChannels}(stm) \cup clockSync \cup \{end\} ]\!] \\ \left(\begin{array}{l}(\mathsf{varMemory}(stm)\ [\![\{end\}]\!]\ \mathsf{constMemory}(stm)) \\ [\![\{end\}]\!] \\ \mathsf{clockstimed}(stm.clocks)\end{array}\right)\end{array}\right) \setminus \mathsf{getsetLocalChannels}(stm) \cup clockSync\right) \\ \quad MachineBody = \left( [\![stm]\!]^{nops}_{\mathcal{NC}'}\ [\![ \{\!| interrupt |\!\} ]\!]\ Skip \right) \setminus enteredSS \\ \textbf{within} \\ \quad Stateful\end{array}$$

**where**

$clockSync = \{\!| c : stm.clocks \bullet get_id(c), clockReset.id(c) |\!\}$

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

**Rule 18. State-machine memory**

stmMemory(stm : SimMachineDef) : TimedCSPProcess =

$$
\begin{array}{l}
\textbf{let}\, Memory(\mathsf{nvars}) = \\
\quad (\Box\, v : \mathsf{rvars} \bullet \quad get_\mathsf{data}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{nvars}) \\
\qquad\qquad\qquad\quad \Box\, registerRead?\mathsf{data}(\mathsf{v})?x \to Memory(\mathsf{nvars}[\mathsf{name}(\mathsf{v}) := x]) \\
\qquad\qquad\qquad\quad \Box\, registerWrite!_\mathsf{data}(\mathsf{v})?x \to Memory(\mathsf{nvars}[\mathsf{name}(\mathsf{v}) := x])) \\
\quad \Box \\
\quad (\Box\, r : \mathsf{r : evars} \bullet \quad get_\mathsf{data}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{nvars}) \\
\qquad\qquad\qquad\quad \Box\, registerRead?\mathsf{data}(\mathsf{v})?x \to Memory(\mathsf{nvars}[\mathsf{name}(\mathsf{v}) := x])) \\
\quad \Box \\
\quad \Box\, v : \mathsf{allConstants(stm)} \bullet get_\mathsf{vid}(\mathsf{v})!\mathsf{name}(\mathsf{v}) \to Memory(\mathsf{nvars}) \\
\textbf{within}\, \mathsf{constInitSTM(stm)};\ Memory(\mathsf{varvalues})
\end{array}
$$

**where**

$$
\begin{array}{l}
\mathsf{rvars} = \mathsf{requiredVariables(stm)} \\
\mathsf{evars} = \mathsf{inputEvents(stm)} \\
\mathsf{lvars} = \mathsf{allLocalVariables(stm)} \\
\mathsf{consts} = \langle \mathsf{v : allConstants(stm)} \bullet \mathsf{v} \rangle \\
\mathsf{nvars} = \langle \mathsf{v : ivars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{name(v)} \rangle \frown \langle \mathsf{v : consts} \bullet \mathsf{name(v)} \rangle \\
\mathsf{varvalues} = \langle \mathsf{v : ivars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{initial(v)} \rangle \frown \langle \mathsf{v : consts} \bullet \mathsf{name(v)} \rangle
\end{array}
$$

**Rule 19. Get and Set channels**

getsetChannels(s : StateMachineDef) : ChannelSet =

$$
\{| \mathsf{v : allVariables(s)} \bullet get_\mathsf{vid(v)} |\} \cup \{| \mathsf{v : allVariables(s)} \bullet set_\mathsf{vid(v)} |\} \cup
$$
$$
\{| \mathsf{v : allConstants(s)} \bullet get_\mathsf{vid(v)} |\} \cup \{| \mathsf{v : allConstants(s)} \bullet set_\mathsf{vid(v)} |\}
$$

**Rule 20. Semantics of defined operations**

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

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

## 4.4 NodeContainers

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

**Rule 21. Node container semantics**

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

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

**Rule 22. Node container behaviour**

$\mathsf{ncBehaviour(nc : NodeContainer)}^{nops} : \mathsf{TimedCSPProcess} =$

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

**Rule 23. Semantics of node container**

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

$$\left(\begin{array}{l}\mathsf{ncBehaviour}^2(\mathsf{nc})^{nops}\\ [\![\mathsf{stateClockSync(nc)} \cup \{end\}]\!]\\ \mathsf{clockstimed(stateClocks(nc))}\end{array}\right) \setminus (\mathsf{stateClockSync(nc)} \setminus \{|entered|\})$$

**where**

$\mathsf{stateClocks(nc)} = \{\mathsf{s : nc.nodes} \mid \mathsf{s} \in \mathsf{State} \wedge \mathsf{s} \notin \mathsf{Final}\}$

$\mathsf{stateClockSync(nc)} = \{|\mathsf{n : stateClocks(nc)} \bullet entered.\mathrm{id}(\mathsf{n}), get_\mathrm{id}(\mathsf{n})|\}$

**Rule 24. Node container core behaviour**

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

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

**where**

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

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

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

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

## 4.5 Transitions

**Rule 25. Semantics of Transitions**

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

**let**

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

**within**

$Trans$

**where**

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

**Rule 26. Semantics of a Transition**

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

$$
\begin{array}{l}
\textbf{if } \mathsf{src} \in \mathsf{State} \wedge \mathsf{t.condition} \neq \mathsf{null} \\
\quad \left( \begin{array}{l}
\left([\![\mathsf{t.condition}]\!]_{\mathcal{E}xpr}\right) \,\&\, \mathsf{trigger}^{1}(\mathsf{t}); \; \mathsf{clockResetstimed}(\mathsf{t}); \; (SStop \triangle exit \rightarrow Skip); \\
((SStop \mid\mid\mid (\mu X \bullet endexec \rightarrow startexec \rightarrow X)) \triangle \\
(exited \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatement\mathcal{I}n\mathcal{C}ontext}[[endexec \leftarrow endexec_action, startexec \leftarrow startexec_action]]; \\
enter.\mathsf{id}(\mathsf{tgt}) \rightarrow Skip))
\end{array} \right) \\
\textbf{else if } \mathsf{src} \in \mathsf{State} \wedge \mathsf{t.condition} = \mathsf{null} \\
\quad \left( \begin{array}{l}
\mathsf{trigger}^{2}(\mathsf{t}); \; \mathsf{clockResetstimed}(\mathsf{t}); \; (SStop \triangle exit \rightarrow Skip); \\
((SStop \mid\mid\mid (\mu X \bullet endexec \rightarrow startexec \rightarrow X)) \triangle \\
(exited \rightarrow [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatement\mathcal{I}n\mathcal{C}ontext}[[endexec \leftarrow endexec_action, startexec \leftarrow startexec_action]]; \\
enter.\mathsf{id}(\mathsf{tgt}) \rightarrow Skip))
\end{array} \right) \\
\textbf{else if } \mathsf{src} \notin \mathsf{State} \wedge \mathsf{t.condition} \neq \mathsf{null} \\
\quad \left([\![\mathsf{t.condition}]\!]_{\mathcal{E}xpr}\right) \,\&\, internal.\mathsf{id}(\mathsf{src}) \rightarrow \\
\qquad\qquad [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatement\mathcal{I}n\mathcal{C}ontext}[[endexec \leftarrow endexec_action, startexec \leftarrow startexec_action]]; \\
\qquad\qquad\qquad enter.\mathsf{id}(\mathsf{tgt}) \rightarrow Skip \\
\textbf{else if } \mathsf{src} \notin \mathsf{State} \wedge \mathsf{t.condition} = \mathsf{null} \\
\qquad internal.\mathsf{id}(\mathsf{src}) \rightarrow \\
\qquad [\![\mathsf{t.action}]\!]^{\mathsf{nops}}_{\mathcal{S}tatement\mathcal{I}n\mathcal{C}ontext}[[endexec \leftarrow endexec_action, startexec \leftarrow startexec_action]]; \\
\qquad enter.\mathsf{id}(\mathsf{tgt}) \rightarrow Skip \\
\textbf{where} \\
\quad \mathsf{src} = \mathsf{t.source} \\
\quad \mathsf{tgt} = \mathsf{t.target}
\end{array}
$$

**Rule 27. Transition events**

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

$$
\begin{array}{l}
\textbf{if } \mathsf{t.trigger} = \mathsf{null} \\
\qquad internal.\mathsf{id}(\mathsf{t.source}) \\
\textbf{else} \\
\qquad endexec__.\mathsf{id}(\mathsf{t.source})
\end{array}
$$

**Rule 28. Semantics of a transition's trigger**

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

$$
\begin{array}{l}
\textbf{if } \mathsf{t.trigger} = \mathsf{null} \\
\qquad internal.\mathsf{id}(\mathsf{t.source}) \rightarrow Skip \\
\textbf{else} \\
\qquad endexec__.\mathsf{id}(\mathsf{t.source}) \rightarrow Skip
\end{array}
$$

**Rule 29. Rename transition trigger events**

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

$\{\!|\, \text{t} : \text{ts} \bullet endexec__.\text{id}(\text{t.source}) \leftarrow endexec \,|\!\}$
$\cup \{\!|\, endexec_action \leftarrow endexec \,|\!\}$
$\cup \{\!|\, startexec_action \leftarrow startexec \,|\!\}$

**where**

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

## 4.6 Statements

**Rule 30. Semantics of SimCall**

$[\![\text{stmt} : \text{SimCall}]\!]^{\text{nops}}_{Statement} : \text{TimedCSPProcess} =$

if $\text{stmt.args} > 0$

$registerWrite.\text{stmt.operation.name}!\{\text{x} : \text{stmt.args} \bullet \text{x.name}\} \rightarrow Skip$

else

$registerWrite.\text{stmt.operation.name} \rightarrow Skip$

**Rule 31. Semantics of OutputCommunication**

$[\![\text{stmt} : \text{OutputCommunication}]\!]^{\text{nops}}_{Statement} : \text{TimedCSPProcess} =$

if $\text{stmt.value} \neq \text{null}$

$registerWrite.\text{stmt.event.name}![\![\text{stmt.value}]\!]_{\mathcal{E}xpr} \rightarrow Skip$

else

$registerWrite.\text{stmt.event.name} \rightarrow Skip$

**Rule 32. Semantics of Assignment**

$[\![\text{stmt} : \text{Assignment}]\!]^{\text{nops}}_{Statement} : \text{TimedCSPProcess} =$

if $\text{stmt.left} \in \text{SimVarRef}$

$registerWrite!\text{stmt.left.name.name}![\![\text{stmt.right}]\!]_{\mathcal{E}xpr} \rightarrow Skip$

else

$SStop \bigtriangleup set_\text{vid}(\text{stmt.left})![\![\text{stmt.right}]\!]_{\mathcal{E}xpr} \rightarrow Skip$

**Rule 33. ExecStatement**

$[\![\text{stmt} : \text{ExecStatement}]\!]^{\text{nops}}_{Statement} : \text{TimedCSPProcess} =$

$endexec \rightarrow startexec \rightarrow SKIP$

# 5. Conformance

In this chapter we describe a conformance relation that can defines all the valid RoboSim simulations of a RoboChart model. It can be used to check whether a RoboSim model is a valid simulation of a RoboChart model.

## 5.1 A1

**Rule 34. A1**

$\mathsf{A1}(\mathsf{inputs} : \mathsf{Set}(\mathsf{Inputs})) : \mathsf{TimedCSPProcess} =$

$||| \; \mathsf{i} : \mathsf{inputs} \bullet \mathsf{A1Event}(\mathsf{i})$

**Rule 35. A1Event**

$\mathsf{A1Event}(\mathsf{in} : \mathsf{Input}) : \mathsf{TimedCSPProcess} =$

$$\begin{array}{l}
\mathbf{if}\ \mathsf{in} \in \mathsf{Event}\ \mathbf{then} \\
\quad \mathit{registerRead}.\mathsf{in.name}?b\mathsf{takeval(in)} \rightarrow \\
\quad \left(\begin{array}{l} \mathbf{if}\ b\ \mathbf{then}\ \mathsf{in.name}.in\mathsf{takeval(in)} \rightarrow \mathsf{A1Event(in)} \,\square\, \mathsf{A1Event(in)} \\ \mathbf{else}\ \mathsf{A1Event(in)} \end{array}\right) \\
\mathbf{else\,if}\ \mathsf{in} \in \mathsf{Variable}\ \mathbf{then} \\
\quad \mathit{registerRead}.\mathsf{in.name}?b?x \rightarrow \\
\quad \left(\begin{array}{l} \mathbf{if}\ b\ \mathbf{then}\ \mathit{set_EXT_}\mathsf{in.name}!x \rightarrow \mathsf{A1Event(in)} \,\square\, \mathsf{A1Event(in)} \\ \mathbf{else}\ \mathsf{A1Event(in)} \end{array}\right)
\end{array}$$

The function $\mathsf{takeval(x)}$ returns the CSP parameters for the input $\mathsf{x}$.

**Rule 36. TA1**
TA1(inputs : Set(Inputs)) : TimedCSPProcess =

$A1(\text{inputs}) \triangle end \rightarrow Skip$

## 5.2 A2

**Rule 37. TA2**
TA2(outputs : Set(Outputs), cycleDef : Expression) : TimedCSPProcess =

**let**

$$A2(sout) = \left( \left( \begin{array}{l} \textbf{if } \#\text{outputs} > 0 \textbf{ then} \\ \quad \left( \Box s : sout \bullet \text{read}(s); A2(sout \setminus \{s\}) \right) \\ \textbf{else} \\ \quad STOP \end{array} \right) \\ \triangleright_1 wait([\![\text{cycleDef}]\!]_{\mathcal{E}xpr} - 1); A2(\text{outputs}) \right)$$

**within**

$A2(\text{outputs}) \triangle end \rightarrow Skip$

**Rule 38. read**
read(o : Output) : TimedCSPProcess =

**if** o ∈ Event **then**
  **if** o.type ≠ null **then**
    $\text{eventId(o)}.out?x \rightarrow (registerWrite.\text{o.name}!x \blacktriangleright 0)$
  **else**
    $\text{eventId(o)}.out \rightarrow (registerWrite.\text{o.name} \blacktriangleright 0)$
**else if** o ∈ Operation **then**
  **if** #o.args > 0 **then**
    $\text{o.name}Call?x \rightarrow (registerWrite.\text{o.name}!x \blacktriangleright 0)$
  **else**
    $\text{o.name}Call \rightarrow (registerWrite.\text{o.name} \blacktriangleright 0)$
**else if** o ∈ Variable **then**
  $set_\text{vid(o)}?x \rightarrow (registerWrite.\text{o.name}!x \blacktriangleright 0)$

## 5.3 A3

**Rule 39. TA3**
TA3(inputs : Set(Inputs), outputs : Set(Outputs), cycleDef : Expression) : TimedCSPProcess =

**let**

$$A3 = \left( \begin{array}{l} \left( |||\, i : \text{inputs} \bullet registerRead.\text{take(i)} \rightarrow Skip \right) \blacktriangleright 0; \\ \left( RUN(\text{extEvents(inputs} \cup \text{outputs)}) \triangle end \rightarrow Skip \right) \triangle_1 \left( wait\left([\![\text{cycleDef}]\!]_{\mathcal{E}xpr} - 1\right); A3\right) \end{array} \right)$$

**within**

$A3$

The process $RUN(X)$ recursively offers the events in the set $X$. Formally, it is defined as follows:

$RUN(X) = \Box\, ev : X \bullet ev \rightarrow RUN(X)$. The function extEvents(X) returns the set of events for the elements of the set X.

## 5.4 Assumptions

**Rule 40. Assumptions**

Assumptions(inputs : Set(Inputs), outputs : Set(Outputs), cycleDef : Expression) : TimedCSPProcess =

$$\left( \left( \begin{array}{l} \mathsf{TA1(inputs)} \\ [\![ \mathsf{extEvents(inputs)} \cup \{\!|registerRead, end|\!\} \mid \mathsf{extEvents(outputs)} \cup \{\!|registerWrite, end|\!\} ]\!] \\ \mathsf{TA2(outputs, cycleDef)} \end{array} \right) \begin{array}{l} \\ [\![ \mathsf{extEvents(inputs \cup outputs)} \cup \{\!|registerRead, registerWrite, end|\!\} \mid \\ \quad \mathsf{extEvents(inputs \cup outputs)} \cup \{\!|registerRead, end|\!\} ]\!] \\ \mathsf{TA3(inputs, outputs, cycleDef)} \end{array} \right)$$

**Rule 41. Constrained**

Constrained(module : RCModule, cycleDef : Expression) : TimedCSPProcess =

$$\left( \begin{array}{l} ([\![\mathsf{module}]\!]_{\mathcal{M}} \,;\, end \rightarrow Skip) \\ [\![ \mathsf{extEvents(inputsMod)} \cup \mathsf{extEvents(outputsMod)} \cup \{\!|end|\!\} ]\!] \\ \left(\mathsf{Assumptions(inputsMod, outputsMod, cycleDef)} \,;\, Stop\right) \end{array} \right) \setminus \mathsf{extEvents(inputsMod)} \cup \mathsf{extEvents(outputsMod)} \cup \{\!|end|\!\}$$

**where**

inputsMod = Inputs(module)

outputsMod = Outputs(module)

The conformance relation is then defined by refinement: $\mathsf{Constrained(RC, cycleDef)} \sqsubseteq RS$.

# 6. Conclusions

We have presented here, *RoboSim*, a new notation for simulation of robots. We have described a semantics for its core constructs. It uses CSP, but we envisage its extension to use Circus [1], a process algebra that combines Z [7] and CSP, and includes time constructs [2]. Use of Circus and its UTP foundation will enable use of theorem proving as well as model checking. Work on probability is available in the UTP [6], but we will pursue an encoding of Markov decision processes in the UTP.

*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 [14] to extend the notation, and from the UTP model of continuous variables [10] to define the semantics.

# A. *RoboSim* Metamodel

The metamodel of *RoboSim* is based on the metamodel of *RoboChart*, which is fully described in the *RoboChart* reference manual [9]. In what follows, we present the metamodel that is specific to *RoboSim*. It is specified in Ecore and formatted by the tool `OCLinEcore`. The syntax of the representation used in this appendix is available here.

```
import robochart : 'platform:/resource/circus.robocalc.robochart/model/
    robochart.ecore#/';

package robosim : robosim = 'http://www.robocalc.circus/RoboSim'
{
  class InputContext extends robochart::Context;
  class OutputContext extends robochart::Context;
  class SimModule extends robochart::RCModule,Cyclic;
  class SimControllerDef extends robochart::ControllerDef,Cyclic;
  class SimMachineDef extends robochart::StateMachineDef,SimContext,Cyclic;
  class SimOperationDef extends robochart::OperationDef,SimContext;
  class ExecTrigger extends robochart::Communication;
  class SimRefExp extends robochart::Expression
  {
    property element : robochart::NamedElement[?];
    property exp : robochart::Expression[?] { composes };
    property variable : robochart::Variable[?];
    property predicate : robochart::Expression[?] { composes };
  }
  class SimCall extends robochart::Call;
  class SimVarRef extends robochart::VarRef;
  class CycleExp extends robochart::Expression;
  class ExecStatement extends robochart::Statement;
```

```
  abstract class SimContext extends robochart::Context
  {
    property inputContext : robochart::Context[1] { composes };
    property outputContext : robochart::Context[1] { composes };
  }
  abstract class Cyclic
  {
    property cycleDef : robochart::Expression[1] { composes };
  }
  class OutputCommunication extends robochart::Statement
  {
    property event : robochart::Event[1];
    property value : robochart::Expression[?] { composes };
  }
}
```

# Bibliography


[1] 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 33).

[2] 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 page 33).

[3] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011 (cited on page 8).

[4] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998 (cited on page 8).

[5] C. Pinciroli et al. "ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems". In: *Swarm Intelligence* 6.4 (2012), pages 271–295 (cited on page 7).

[6] 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 pages 8, 33).

[7] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996 (cited on page 33).

[8] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007 (cited on page 7).

[9] *RoboChart Reference Manual*. University of York. URL: https://bit.ly/2Ooe7RS (cited on pages 7, 13, 17, 35).

[10] 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 33).

[11] S. Luke et al. "MASON: A Multiagent Simulation Environment". In: *Simulation* 81.7 (2005), pages 517–527 (cited on page 7).

[12] *Simulink*. `www.mathworks.com/products/simulink`. The MathWorks,Inc. (cited on page 7).

[13] *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.com/products`. The MathWorks,Inc. (cited on page 7).

[14] 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 33).

[15] 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 8).

[16] U. Wilensky and W. Rand. *An Introduction to Agent-Based Modeling – Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press, 2015 (cited on page 7).

# Index of Semantic Rules

In this index you'll find the list of semantic functions in alphabetic order, and page where they are defined.


stateful (StateMachineBody), 24

A1, 29
A1Event, 29
Assumptions, 31

behaviour, 24

C (Controller), 21
composeControllers, 20
composeMachines, 22
constInitSTM, 24
Constrained, 31
cycle, 23
cycleController, 22
cycleModule, 19

getsetChannels, 25
getsetChannelsStm, 23

inputs, 18

M (SimModule), 18
memoryComp, 19

NC, 25
NC (NodeContainer)(Timed), 26
ncBehaviour, 26, 26

read, 30
renameTriggerEvents, 28
renamingController, 20
renamingMachine, 22

SimMachineDef, 23
Statement, 28, 28, 28
statement, 28
STM (SimOperationDef), 25
STM (SState Machine), 23
stmMemory, 25

T (Transition), 27
TA1, 30
TA2, 30
TA3, 30
tevent(Transition), 27
Transitions, 26
triggerForMemory, 27

# 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.


behaviour , **23**[1]

C (Controller) , **20**[1]
composeControllers , **18**[1], **19**[2], **20**[3]
composeMachines , **21**[1], **22**[2]
cycle , **23**[1]
cycleController , **21**[1]
cycleModule , **18**[1]

getsetChannelsStm , **23**[1], **23**[2]

memoryComp , **18**[1], **21**[2]

NC , **24**[1]
ncBehaviour , **25**[1], **26**[2]
ncCoreBehaviour , **26**[1]

renameTriggerEvents , **26**[1]
renamingController , **20**[1], **20**[2]
renamingMachine , **22**[1], **22**[2]

stateful , **24**[1], **25**[2]
STM (State machine) , **21**[1], **22**[2]
stmMemory , **23**[1]

T (Transition) , **26**[1]
tevent , **26**[1], **26**[2]
transitions , **26**[1]
trigger , **27**[1], **27**[2]