RoboSim Reference Manual

Ana Cavalcanti     Augusto Sampaio
Pedro Ribeiro     Alvaro Miyazawa
Madiel Conserva Filho     Andre Didier
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.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>A3</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Assumptions</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>33</td>
</tr>
<tr>
<td>A</td>
<td>RoboSim Metamodel</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Index of Semantic Rules</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Index of Calls to Semantic Rules</td>
<td>41</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Transition events</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>Semantics of a transition’s trigger</td>
<td>27</td>
</tr>
<tr>
<td>29</td>
<td>Rename transition trigger events</td>
<td>28</td>
</tr>
<tr>
<td>30</td>
<td>Semantics of SimCall</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td>Semantics of OutputCommunication</td>
<td>28</td>
</tr>
<tr>
<td>32</td>
<td>Semantics of Assignment</td>
<td>28</td>
</tr>
<tr>
<td>33</td>
<td>ExecStatement</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>A1</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>A1Event</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>TA1</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>TA2</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>TA3</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Assumptions</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Constrained</td>
<td>31</td>
</tr>
</tbody>
</table>
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, Robotic-Platforms, Controllers, and StateMachines are ConnectionNodes. The restrictions on cardinality are well-formedness conditions. The ConnectionNodes are linked by Connections via their events from and to. 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.
A module and each of its controllers may have different specifications for the value of \texttt{cycle}; similarly, a controller and each of its machines may also have different specifications. Ultimately, it is the module that defines the \texttt{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 	exttt{StateMachine}. Another form of \texttt{StateMachine}, is a reference to a state machine.

Declarations in a simulation machine are partitioned into three \texttt{Context}s: local declarations, inputs and outputs. Input and output events have a semantics different from that in \texttt{RoboChart}. There, they are events raised by the platform or the software. In \texttt{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 \texttt{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 \textit{RoboSim}. In the metamodel (see Figure 2.1), we have an extra form of \texttt{Expression} that allows the use of an input \texttt{Event} as a variable (\texttt{InputCommunication}). If the input communicates a value, that is recorded as part of that communication. An \texttt{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 \texttt{Expression}, namely, \texttt{RequiredVariable}, allows references to a variable preceded by a $.

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

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

As opposed to \textit{RoboChart} clocks, \textit{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 $\text{Cycl}_e$ is a boolean expression. It is a predicate that specifies a restriction on the values (positive natural numbers) that the variable $\text{Cycl}_e$ may take.

SM2 The conjunction of the $\text{Cycl}_e$ 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 $\text{Cycl}_e$ for use in a simulation of the module.
Chapter 3. Well-formedness Conditions

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.
The semantics of \textit{RoboSim} is based on the timed semantics of \textit{RoboChart}, which is fully described in the \textit{RoboChart} reference manual \cite{9}. In what follows, we present the rules that are specific to \textit{RoboSim}. 
Chapter 4. Semantics

4.1 Module

Rule 1. Semantics of modules

\[ \text{cycleModule}^1 (m, \text{ctrls}, \text{inputs}, \text{outputs}, \text{rvars}, \text{evars}) \]
\[ = \text{composeControllers}^1 (m, \text{ctrls}, \text{inputs}, \text{outputs}, \text{rvars}, \text{evars}) \]
\[ \Theta_{(\text{end})} \text{SKIP} \]

where
- \( \text{rp} = \text{roboticPlatformDefinition}(m) \)
- \( \text{ctrls} = \text{controllersDefinition}(m) \)
- \( \text{cons} = m.\text{connections} \)
- \( \text{inputs} = \text{Inputs}(\text{rp}, \text{cons}) \)
- \( \text{outputs} = \text{Outputs}(\text{rp}, \text{ctrls}) \)
- \( \text{rvars} = \text{Rvars}(\text{rp}, \text{ctrls}) \)
- \( \text{evars} = \text{Internals}(\text{cons}) \)
- \( \text{setConstants} = \{ c : \text{allConstants}(\text{rp}) \times \text{set\_name}(c) \} \)
- \( \text{endExecC} = \{ c : \text{ctrls} \times \text{endexec}.\text{id}(c) \} \)

Rule 2. Function Inputs for Module

\[ \text{Inputs}(\text{rp} : \text{RoboticPlatform}, \text{cons} : \text{Set}(\text{Connection})) : \text{Set}(\text{InputsType}) = \]
\[ \{ c : \text{cons} | \text{c.from} = \text{rp} \land \text{c.to} \in \text{Controller} \times (\text{id}(\text{c.efrom}), \text{id}(\text{c.eto})) \} \]
4.1 Module

**Rule 3. Cycle for modules**

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

let CycleModule =
  (((| in : dom.inputs • registerRead.take(in) → registerReadC.give(inputs(in)) → SKIP);
    | evar : dom.evars • registerReadC.take(evar) → SKIP);
    | evar : dom.evars • registerWriteE.data(evar) → SKIP);
  (CollectOutputs(dom(allOutputs)) ||| endexec.id(m)) Monitor) △ StopY);
((end → SKIP) → (wait(\[m.cycleDef[\$exp - 1]\]));
CycleModule

CollectOutputs(sout) =
  (registerWriteC?outp : {out : dom(allOutputs \ (evars ∪ rvars)); outc : allOutputs{out :};
    outp : |{data|auto|} | out ∈ sout • outp} →
    registerWrite!lift(outp) → CollectOutputs(sout \ {project(outp)}))

□ endexec.id(m) → SKIP
□ (registerWriteC?v : {rvars : {out : dom(allOutputs \ (evars ∪ rvars)); outc : allOutputs{out :};
    outp : |{data|auto|} | out ∈ sout • outp} →
    CollectOutputs(sout \ {project(outp)})})

Monitor =
  (((| i : 1. • ctrls • endexec.id(ctrls(i)) → SKIP); endexec.id(m) → SKIP
within CycleModule
where
allOutputs = outputs ∪ rvars ∪ evars

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

**Rule 4. Module or controller memory**

memoryComp(rvars : Set(OutputsType), evars : Set(EventsType), node : ConnectionNode) : TimedCSPProcess =

let Memory(rvars) =
  (((| v : dom.evars • registerWriteC.take(v) → Memory(rvars{name(v) := true})
    □ registerWriteE.take(v) → Memory(rvars{name(v) := false})
  )
  □
  (| v : dom.rvars • registerWriteC.take(v) → Memory(rvars))
  □
  (| v : dom.evars • registerReadC.give(evars(v)) → Memory(rvars))
  □
  (| rv : ran.rvars • registerReadC.give(rv) → Memory(rvars))
within constInit(node); Memory(initial(rvars))
where
rvars = {v : dom.rvars ∪ dom.evars} • name(v)

The function **composeControllers**^{6}(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

\[
\text{composeControllers}(m : \text{SimModule}, \text{ctrls} : \text{Seq}(<\text{SimControllerDef}>, \text{inputs} : \text{Set}(<\text{InputsType}>, \text{outputs} : \text{Set}(<\text{OutputsType}>, \text{rvars} : \text{Set}(<\text{OutputsType}>, \text{evars} : \text{Set}(<\text{EventsType}>) : \text{TimedCSPProcess} =
\]

\[
\begin{cases}
\text{if } \#\text{ctrls} = 1 \\
\quad \text{then} \\
\quad \quad \text{renamingController}^1(m, \text{head} \text{ctrls}, \text{inputs}, \text{outputs}, \text{rvars}, \text{evars}) \\
\qquad \text{else} \\
\qquad \quad \text{renamingController}^2(m, \text{head} \text{ctrls}, \text{inputs}, \text{outputs}, \text{rvars}, \text{evars}) \\
\qquad \quad \{\{\text{end}\} \cup \text{setConstants}\} \\
\qquad \quad \text{composeControllers}^3(m, \text{tail} \text{ctrls}, \text{inputs}, \text{outputs}, \text{rvars}, \text{evars})
\end{cases}
\]

where

\[
\text{setConstants} = \{c : \text{allConstants}(rp) \times \text{set}_\text{vid}(c, m)\}
\]

Rule 6. Renaming controller

\[
\text{renamingController}(m : \text{SimModule}, c : \text{SimControllerDef}, \text{inputs} : \text{Set}(<\text{InputsType}>, \text{outputs} : \text{Set}(<\text{OutputsType}>, \text{rvars} : \text{Set}(<\text{OutputsType}>, \text{evars} : \text{Set}(<\text{EventsType}>) : \text{TimedCSPProcess} =
\]

\[
\begin{cases}
\text{c'} = \lfloor c \rfloor \\
\quad \{\text{ct'} : \text{allConstants}(c) \times \text{set}_\text{vid}(\text{ct'}) \leftarrow \text{set}_\text{vid}(\text{ct', m})\} \\
\quad \cup \{\text{registerRead} \leftarrow \text{registerReadC}\} \\
\quad \cup \{\text{registerWrite} \leftarrow \text{registerWriteC}\}
\end{cases}
\]

where

\[
\text{setNotConnectedEvs} = \{|\text{NotConnectedEvs}(\text{inputs}, \text{outputs}, \text{rvars}, \text{evars})|\}
\]
4.2 Controllers

Rule 7. Semantics of controllers
\[ [c : SimControllerDef]_C : TimedCSPProcess = \]
\[
\text{let } \quad \text{for each } op : c.lOperations \cdot nproc(op)(\{x : op.parameters \cdot x.name\}) = \text{opdef(op)}^{nops} \text{STM} \\
\quad \text{within } \quad \left( \begin{array}{l}
\text{cycleController}(c, inputs, outputs, dvars, evars) \\
\| \{\text{registerWriteC.data(out) | out \leftarrow \text{ran dvars}\} U [\text{registerWriteE}] \}
\| \{\text{registerReadC.data(v) | v : dom evars}\} \}
\text{memoryComp}(dvars, evars, c) \\
\| \text{setConstants} U [\text{registerReadC, registerWriteC, terminate}] U \text{endExecM} \}
\| \text{composeMachines}(c, machines, inputs, outputs, dvars, evars)^{nops} \}
\| \{\text{registerReadC, registerWriteC, registerWriteE}\} U \text{endExecM} U \text{lconsts} \}
\end{array} \right) \]
\]
\[ \Theta_{(end)} \text{SKIP} \]

where
- machines = (x : c.machines)
- cons = c.connections
- inputs = InputsC(c, cons)
- outputs = OutputsC(c, cons)
- dvars = DvarsC(c)
- evars = InternalsC(c, cons)
- setConstants = \{ct : allConstants(c) \cdot set_name(ct)\}
- endExecM = \{stm : machines \cdot endexec.id(stm)\}
- opdef(op) = findOperationDefinition(op, c.lOperations)
- nops = \{op : c.lOperations \cdot id(op) \rightarrow nproc(op)\} U \{op : requiredOperations(c) \cdot id(op) \rightarrow nproc(op)\}
- lvars = \{v : allLocalVariables(c) \cdot set_vid(v)\}
- rvars = \{v : requiredVariables(c) \cdot set_Ex_vid(v)\}
- lconsts = \{v : allLocalConstants(c) \cdot set_vid(v)\}
- rconsts = \{v : requiredConstants(c) \cdot set_vid(v)\}
Chapter 4. Semantics

Rule 8. Cycle for controllers

\[
\text{cycleController} \equiv \text{Renaming machine} \circ \text{Composition of machines}
\]

\[
\text{let } \text{CycleController} = \\
\quad (\text{CollectOutputs} \in \text{Set} \cdot \text{Set} \cdot \text{Set}) \cdot \text{Set} \cdot \text{Set} \cdot \text{Set}
\]

\[
\text{CollectOutputs(sout)} = \\
\quad (\text{CollectOutputs} \in \text{Set} \cdot \text{Set} \cdot \text{Set}) \cdot \text{Set} \cdot \text{Set}
\]

\[
\text{Monitor} = \\
\quad (\text{Collect Outputs} \in \text{Set} \cdot \text{Set} \cdot \text{Set}) \cdot \text{Set} \cdot \text{Set}
\]

Rule 9. Composition of machines

\[
\text{composeMachines} \equiv \text{Renaming machine} \circ \text{Composition of machines}
\]

\[
\text{composeMachines} \equiv \text{Renaming machine} \circ \text{Composition of machines}
\]

Rule 10. Renaming machine

\[
\text{renamingMachine} \equiv \text{Renaming machine}
\]
4.3 State Machines

**Rule 11. Semantics of state machine**

\[
\text{stm} : \text{StateMachineBody} \xrightarrow{\text{STM}} \text{TimedCSPProcess} =
\]

The rule is split according to the type of \text{stm}.

**Rule 12. Semantics of state machine**

\[
\text{stm} : \text{SimMachineDef} \xrightarrow{\text{STM}} \text{TimedCSPProcess} =
\]

\[
\begin{align*}
\text{behaviour}^{1}(\text{stm}) & \quad \text{getsetChannelsStm}^{1}(\text{stm}) \cup \{\text{registerWrite}\} \\setminus \text{getsetChannelsStm}^{2}(\text{stm}) \\
\text{stmMemory}^{1}(\text{stm}) & \quad \{\text{startexec}, \text{endexec}, \text{id}(\text{stm}), \text{end}, \text{registerRead}, \text{registerWrite}\} \\setminus \{\text{startexec}\} \\
\text{cycle}^{1}(\text{stm}, \text{inputs}, \text{outputs}) & \quad \Theta_{\text{(end) SKIP}} \\
\text{where} \\
\text{inputs} & = \text{inputs}(\text{stm}) \\
\text{outputs} & = \text{outputs}(\text{stm}) \\
\text{reqVars} & = \text{reqVars}(\text{stm}) \\
\text{consts} & = \text{allConstants}(\text{stm})
\end{align*}
\]

**Rule 13. Get and set channels**

\[
\text{getsetChannels}(\text{stm} : \text{SimMachineDef}) : \text{ChannelSet} =
\]

\[
\{v : \text{Evars}(\text{stm}) \bullet \text{get}_v(\text{id}(v)) \cup \{v : \text{Rvars}(\text{stm}) \bullet \text{get}_v(\text{id}(v))\} \cup \{v : \text{allConstants}(\text{stm}) \bullet \text{get}_v(\text{id}(v))\}
\]

**Rule 14. Cycle for machines**

\[
\text{cycle}(\text{stm} : \text{SimMachineDef}, \text{inputs} : \text{Set}(\text{InputsType}), \text{outputs} : \text{Set}(\text{OutputsType})) : \text{TimedCSPProcess} =
\]

\[
\text{let } \text{Cycle} = ((\| \text{in} : \text{inputs} \bullet \text{registerRead}.\text{take}(\text{in}) \rightarrow \text{SKIP}); \\
\text{startexec} \rightarrow \text{CollectOutputs}(\text{outputs}) \uplus \text{Stop}); \\
((\text{end} \rightarrow \text{SKIP}) \uplus (\text{wait}(\text{stm.cycleDef}_{\text{expr}} - 1))); \\
\text{Cycle} \\
\text{CollectOutputs}(\text{sout}) = \text{registerWrite}?\text{give}_T(\text{out}) \rightarrow \text{CollectOutputs}(\text{sout} \setminus \{\text{out}\}) \\
\text{endexec}.\text{id}(\text{stm}) \rightarrow \text{SKIP} \\
\text{within} \text{Cycle}
\]
**Rule 15. Constants Initialisation for State Machines**

\[
\text{constInitSTM}(cs : \text{Seq}\{\text{Variable}\}, stm : \text{StateMachineDef}, P : \text{CSPProcess}) : \text{TimedCSPProcess} =
\]

\[
\text{buildScope}(\text{undefc}, \text{stm}, \left( \begin{array}{c}
\text{let } \text{c : def} \star \text{name}(\text{c)} = [\text{c}.\text{initial}]_{\text{Expr}} \\
\text{within } P
\end{array} \right) )
\]

where
\[
\text{defc} = (x : \text{consts} | x.\text{initial} \neq \text{null}) \\
\text{undefc} = (x \star \text{consts} | x.\text{initial} = \text{null})
\]

**Rule 16. Behaviour of state machine**

\[
\text{behaviour}(stm : \text{SimMachineDef}) : \text{TimedCSPProcess} =
\]

\[
\text{let}
\]

\[
\text{Ending} = \text{endexec} \rightarrow (\text{startexec} \rightarrow \text{Ending} \bigcirc \text{end} \rightarrow \text{SKIP})
\]

\[
\text{within}
\]

\[
\left( \begin{array}{c}
\text{startexec} \rightarrow \text{stateful}(\text{stm}) \setminus \{\text{end}\} \\
\text{destruct}(\text{share} \cup \{\text{end}\}) \\
\text{SKIP}
\end{array} \right)
\]

\[
\text{Ending}
\]

**Rule 17. Semantics of state machine**

\[
\text{stateful}(stm : \text{StateMachineBody})^{nops} : \text{TimedCSPProcess} =
\]

\[
\text{let}
\]

\[
\text{Stateful} = 
\]

\[
\left( \begin{array}{c}
\text{MachineBody} \\
\text{getsetLocalChannels}(\text{stm}) \cup \text{clockSync} \cup \{\text{end}\} \\
\text{varMemory}(\text{stm}) \setminus \{\text{end}\} \cup \text{constMemory}(\text{stm}) \\
\text{clockstimed}(\text{stm}.\text{clocks}) \\
\text{getsetLocalChannels}(\text{stm}) \cup \text{clockSync}
\end{array} \right)
\]

\[
\text{within}
\]

\[
\text{Stateful}
\]

where
\[
\text{clockSync} = \{c : \text{stm}.\text{clocks} \star \text{get}\_\text{id}(c), \text{clockReset}\_\text{id}(c)\} \\
\text{enteredSS} = \{x : \text{stm}.\text{nodes} \star \text{entered}\_\text{id}(x)\}
\]
### Rule 18. State-machine memory

\[
\text{let } \text{Memory}(nvars) = \\
\begin{array}{l}
\quad (\forall v : \text{rvars} \cdot \text{get}_\text{data}(v) \cdot \text{name}(v) \rightarrow \text{Memory}(nvars)) \\
\quad (\forall r : \text{evars} \cdot \text{get}_\text{data}(v) \cdot \text{name}(v) \rightarrow \text{Memory}(nvars)) \\
\quad \text{within } \text{constInitSTM}(\text{stm}); \text{Memory}(\text{varvalues})
\end{array}
\]

where

\[
\begin{aligned}
\text{rvars} &= \text{requiredVariables}(\text{stm}) \\
\text{evars} &= \text{inputEvents}(\text{stm}) \\
\text{lvars} &= \text{allLocalVariables}(\text{stm}) \\
\text{consts} &= (v : \text{allConstants}(\text{stm}) \cdot v) \\
\text{nvars} &= (v : \text{ivars} \cup \text{lvars} \cup \text{cvars} \cdot \text{name}(v)) \cap (v : \text{consts} \cdot \text{name}(v)) \\
\text{varvalues} &= (v : \text{ivars} \cup \text{lvars} \cup \text{cvars} \cdot \text{initial}(v)) \cap (v : \text{consts} \cdot \text{name}(v))
\end{aligned}
\]

### Rule 19. Get and Set channels

\[
\text{getsetChannels}(s : \text{StateMachineDef}) : \text{ChannelSet} = \\
\begin{array}{l}
\quad \{ v : \text{allVariables}(s) \cdot \text{get}_\text{vid}(v) \} \cup \{ v : \text{allVariables}(s) \cdot \text{set}_\text{vid}(v) \} \\
\quad \{ v : \text{allConstants}(s) \cdot \text{get}_\text{vid}(v) \} \cup \{ v : \text{allConstants}(s) \cdot \text{set}_\text{vid}(v) \}
\end{array}
\]

### Rule 20. Semantics of defined operations

\[
\left[\text{stm} : \text{SimOperationDef}\right]_{\text{STM}}^{\text{nops}} : \text{TimedCSPProcess} = \\
\text{stateful}(\text{stm}) \setminus \{ \text{end} \}
\]

### 4.4 NodeContainers

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

### Rule 21. Node container semantics

\[
\left[\text{nc} : \text{NodeContainer}\right]^{\text{nops}}_{\text{NC}} : \text{TimedCSPProcess} = \\
\text{ncBehaviour}(\text{nc})^{\text{nops}}
\]
Chapter 4. Semantics

Rule 22. Node container behaviour
\[
\text{ncBehaviour}(\text{nc} : \text{NodeContainer})^{\text{tops}} : \text{TimedCSPProcess} = \\
(\text{ncCoreBehaviour}^1(\text{nc})^{\text{tops}} \setminus \{\text{enter}, \text{exit}, \text{exited}, \text{internal}\}) \uplus \text{renameTriggerEvents}^1(\text{nc})
\]

Rule 23. Semantics of node container
\[
[\text{nc} : \text{NodeContainer}]^{\text{tops}} : \text{TimedCSPProcess} = \\
\left(\begin{array}{c}
\text{ncBehaviour}^2(\text{nc})^{\text{tops}} \\
\text{clockstimesd(stateClocks(nc))}
\end{array}\right) \setminus \text{stateClockSync(nc)} \setminus \{\text{entered}\}
\]

where
\[
\begin{align*}
\text{stateClocks(nc)} &= \{s : \text{nc.nodes} \mid s \in \text{State} \land s \neq \text{Final}\} \\
\text{stateClockSync(nc)} &= \{n : \text{stateClocks(nc)} \bullet \text{entered}, \text{id}(n), \text{get}_{-\text{id}}(n)\}
\end{align*}
\]

Rule 24. Node container core behaviour
\[
\text{ncCoreBehaviour}(\text{nc} : \text{NodeContainer})^{\text{tops}} : \text{TimedCSPProcess} = \\
\left(\begin{array}{c}
(\text{composeTimedNodes}(\text{nc})^{\text{tops}} \uplus \text{tidsR(nc)} \uplus) \\
\text{vars(nc.nodes)} \uplus \text{flowevts} \uplus \text{transSync(nc)} \uplus \{\text{end}\} \uplus \{\text{startexec, endexec}\} \uplus \text{vguards(nc)} : \emptyset
\end{array}\right)
\]

where
\[
\begin{align*}
\text{tidsR(nc)} &= \left\{ t : \text{nc.transitions} \bullet \text{interrupt} \cdot \text{id}(t) \rightarrow \text{tevent}^1(t) \right\} \\
\text{flowevts} &= \{\text{enter}, \text{exit}, \text{exited}, \text{interrupt}\} \\
\text{transSync(nc)} &= \{t : \text{nc.transitions} \bullet \text{tevent}^2(t)\} \\
\text{vguards(nc)} &= \{t : \text{nc.transitions} \mid \exists v : v \in \text{usedVariables(t.condition)} \bullet v\}
\end{align*}
\]

4.5 Transitions

Rule 25. Semantics of Transitions
\[
\text{transitions}(\text{nc} : \text{NodeContainer})^{\text{tops}} : \text{TimedCSPProcess} = \\
\text{let} \\
\text{Trans} \equiv \text{readStateA}
\]

\[
\begin{align*}
\text{Trans} &= \text{readStateA} \\
\text{let} \\
\text{vguards,} \\
\text{Trans} &= \text{readStateA} \\
\text{with} \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{vguards} &= \{t : \text{nc.transitions}, v : \text{NamedElement} \mid v \in \text{usedVariables(t.condition)} \bullet v\}
\end{align*}
\]
Rule 26. Semantics of a Transition

\[ [t : \text{Transition}]^{\text{nops}} : \text{TimedCSPProcess} = \]

\[
\begin{cases}
\text{if } \text{src} \in \text{State} \land t.\text{condition} \neq \text{null} & \left( (t.\text{condition})^{\text{Expr}} \land t.\text{trigger} @ t; \text{clockResetStimed}(t); (\text{SSStop} \triangle \text{exit} \rightarrow \text{Skip}) \right) \\
\text{else if } \text{src} \in \text{State} \land t.\text{condition} = \text{null} & \left( \text{trigger} @ t; \text{clockResetStimed}(t); (\text{SSStop} \triangle \text{exit} \rightarrow \text{Skip}) \right) \\
\text{else if } \text{src} \notin \text{State} \land t.\text{condition} \neq \text{null} & \left( (t.\text{condition})^{\text{Expr}} \land \text{internal.id}(\text{src}) \rightarrow \left[ (t.\text{action})^{\text{nops}} \text{StatementInContext} \left[ \text{endexec} \leftarrow \text{endexec.action}, \text{startexec} \leftarrow \text{startexec.action} \right] ; \text{endexec} \leftarrow \text{endexec.action}, \text{startexec} \leftarrow \text{startexec.action} \right] \right) \\
\text{else if } \text{src} \notin \text{State} \land t.\text{condition} = \text{null} & \left( \text{internal.id}(\text{src}) \rightarrow \left[ (t.\text{action})^{\text{nops}} \text{StatementInContext} \left[ \text{endexec} \leftarrow \text{endexec.action}, \text{startexec} \leftarrow \text{startexec.action} \right] \right) \right) \\
\end{cases}
\]

where

\[ \text{src} = t.\text{source} \]
\[ \text{tgt} = t.\text{target} \]

Rule 27. Transition events

\[ \text{tevent}(t : \text{Transition}) : \text{CSPEvent} = \]

\[
\begin{cases}
\text{if } t.\text{trigger} = \text{null} & \text{internal.id}(t.\text{source}) \\
\text{else} & \text{endexec.__id}(t.\text{source})
\end{cases}
\]

Rule 28. Semantics of a transition’s trigger

\[ \text{trigger}(t : \text{Transition}) : \text{TimedCSPProcess} = \]

\[
\begin{cases}
\text{if } t.\text{trigger} = \text{null} & \text{internal.id}(t.\text{source}) \rightarrow \text{Skip} \\
\text{else} & \text{endexec.__id}(t.\text{source}) \rightarrow \text{Skip}
\end{cases}
\]
Chapter 4. Semantics

Rule 29. Rename transition trigger events
renameTriggerEvents(nc : NodeContainer) : RenamingSet =
\[
\{ t : ts • endexec.__id(t.source) ← endexec \}
\cup \{ endexec_action ← endexec \}
\cup \{ startexec_action ← startexec \}
where
\[
ts = \{ t : nc.transitions | t.trigger \neq null \}
\]

4.6 Statements

Rule 30. Semantics of SimCall
\[
[[ stmt : SimCall ]]^{\text{nops}}_{\text{Statement}} : TimedCSPProcess =
\]
if stmt.args > 0
\[
\text{registerWrite}.stmt.operation.name!\{ x : stmt.args • x.name \} → \text{Skip}
\]
else
\[
\text{registerWrite}.stmt.operation.name → \text{Skip}
\]

Rule 31. Semantics of OutputCommunication
\[
[[ stmt : OutputCommunication ]]^{\text{nops}}_{\text{Statement}} : TimedCSPProcess =
\]
if stmt.value \neq null
\[
\text{registerWrite}.stmt.event.name![[ stmt.value ]]_{\text{Expr}} → \text{Skip}
\]
else
\[
\text{registerWrite}.stmt.event.name → \text{Skip}
\]

Rule 32. Semantics of Assignment
\[
[[ stmt : Assignment ]]^{\text{nops}}_{\text{Statement}} : TimedCSPProcess =
\]
if stmt.left \in \text{SimVarRef}
\[
\text{registerWrite}!stmt.left.name.name![[ stmt.right ]]_{\text{Expr}} → \text{Skip}
\]
else
\[
\text{SSStop} \triangle set\_vid(stmt.left)![[ stmt.right ]]_{\text{Expr}} → \text{Skip}
\]

Rule 33. ExecStatement
\[
[[ stmt : ExecStatement ]]^{\text{nops}}_{\text{Statement}} : TimedCSPProcess =
\]
endexec → startexec → \text{SKIP}
In this chapter we describe a conformance relation that can define 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**

\[
A1(\text{inputs} : \text{Set(Events)}) : \text{TimedCSPProcess} = \\
\\\text{inputs} \cdot A1Event(i)
\]

**Rule 35. A1Event**

\[
A1Event(\text{in} : \text{Input}) : \text{TimedCSPProcess} = \\
\begin{cases}
\text{if } \text{in} \in \text{Event} \text{ then } \\
& \text{registerRead} \text{.in .name} \? \text{b?} \text{takeval(in)} \\
& \begin{cases}
& \text{if } b \text{ then } \text{in .name} \? \text{b?} \text{takeval(in)} \rightarrow A1Event(\text{in}) \text{○ A1Event(\text{in})} \\
& \text{else } A1Event(\text{in})
\end{cases}
\end{cases}
\]

\[
\begin{cases}
\text{elseif } \text{in} \in \text{Variable} \text{ then } \\
& \text{registerRead} \text{.in .name} \? \text{b?} \text{x} \\
& \begin{cases}
& \text{if } b \text{ then set.EXT} \_\text{.in .name} !\text{x} \rightarrow A1Event(\text{in}) \text{○ A1Event(\text{in})} \\
& \text{else } A1Event(\text{in})
\end{cases}
\end{cases}
\]

The function \text{takeval(x)} returns the CSP parameters for the input \(x\).
Rule 36. TA1
TA1(inputs : Set(Inputs)) : TimedCSPProcess =
A1(inputs) △ end → Skip

5.2 A2

Rule 37. TA2
TA2(outputs : Set(Outputs), cycleDef : Expression) : TimedCSPProcess =
let
A2(sout) =
\[
\begin{cases}
\text{if } \#\text{outputs} > 0 \text{ then } & (\square s : sout • read(s); A2(sout \setminus \{s\})) \\
\text{else} & STOP \\
\end{cases}
\]
\[\triangledown \text{wait}([cycleDef]\text{Expr} - 1); A2(outputs)\]
within
A2(outputs) △ end → Skip

Rule 38. read
read(o : Output) : TimedCSPProcess =
if o ∈ Event then
if o.type ≠ null then
  eventId(o).out?x → (registerWrite.o.name!x ▶ 0)
else
  eventId(o).out → (registerWrite.o.name ▶ 0)
elseif o ∈ Operation then
if #o.args > 0 then
  o.nameCall?x → (registerWrite.o.name!x ▶ 0)
else
  o.nameCall → (registerWrite.o.name ▶ 0)
elseif o ∈ Variable then
  set_vid(o)?x → (registerWrite.o.name!x ▶ 0)

5.3 A3

Rule 39. TA3
TA3(inputs : Set(Inputs), outputs : Set(Outputs), cycleDef : Expression) : TimedCSPProcess =
let
A3 =
\[
\begin{cases}
\|i : inputs • registerRead.take(i) → Skip \| & ▶ 0; \\
RUN(extEvents(inputs ∪ outputs)) △ end → Skip & \triangledown 1 (wait([cycleDef]\text{Expr} - 1); A3) \\
\end{cases}
\]
within
A3

The process \(RUN(X)\) recursively offers the events in the set \(X\). Formally, it is defined as follows:
\[ \text{RUN}(X) = \square ev : X \bullet ev \rightarrow \text{RUN}(X). \] The function \( \text{extEvents}(X) \) returns the set of events for the elements of the set \( X \).

### 5.4 Assumptions

**Rule 40. Assumptions**

Assumptions(inputs : Set(\(\text{Inputs}\)), outputs : Set(\(\text{Outputs}\)), cycleDef : Expression) : TimedCSPProcess =

\[
\begin{align*}
\text{TA1}(\text{inputs}) &= (\text{extEvents}(\text{inputs}) \cup \{\text{registerRead, end}\} \cup \text{extEvents}(\text{outputs}) \cup \{\text{registerWrite, end}\}) \\
\text{TA2}(\text{outputs, cycleDef}) &= \text{extEvents}(\text{inputs} \cup \text{outputs}) \cup \{\text{registerRead, registerWrite, end}\} \\
\text{TA3}(\text{inputs, outputs, cycleDef}) &= \text{extEvents}(\text{inputs} \cup \text{outputs}) \cup \{\text{registerRead, end}\}
\end{align*}
\]

**Rule 41. Constrained**

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

\[
\begin{align*}
\text{TA1}(\text{module}) &= (\text{end} \rightarrow \text{Skip}) \\
\text{TA2}(\text{inputsMod} \cup \text{outputsMod}) &= \text{extEvents}(\text{inputsMod} \cup \text{outputsMod}) \cup \{\text{end}\} \\
\text{TA3}(\text{inputsMod}, \text{outputsMod}, \text{cycleDef}) &= \text{extEvents}(\text{inputsMod} \cup \text{outputsMod}) \cup \{\text{end}\}
\end{align*}
\]

where

\[
\begin{align*}
\text{inputsMod} &= \text{Inputs}(\text{module}) \\
\text{outputsMod} &= \text{Outputs}(\text{module})
\end{align*}
\]

The conformance relation is then defined by refinement: Constrained(RC, cycleDef) ⊑ 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.
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.

```ecore
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::RModule.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 };
}


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 (SStateMachine), 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

C (Controller), 20

composeControllers, 18, 19, 20

composeMachines, 21, 22

cycle, 23

cycleController, 21

cycleModule, 18

getsetChannelsStm, 23, 23

memoryComp, 18, 21

NC, 24

ncBehaviour, 25, 26

ncCoreBehaviour, 26

renameTriggerEvents, 26

renamingController, 20, 20

renamingMachine, 22, 22

stateful, 24, 25

STM (State machine), 21, 22

stmMemory, 23

T (Transition), 26

tevent, 26, 26

transitions, 26

trigger, 27, 27