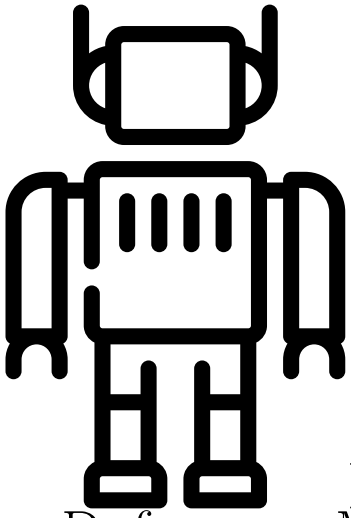

RoboSim Reference Manual

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

April 2019

Abstract

In this report, we describe a UML-like notation called RoboSim, designed specifically for simulation of autonomous and mobile robots, and including timed primitives. We provide a reference manual for RoboSim. We describe the syntax, the well-formedness conditions and semantics of RoboSim. We also present three case studies and show how we can use RoboSim models to check if a simulation is consistent with a functional design written in a UML-like notation.[1]

[1]The interested reader can reproduce the verification of all examples presented in this Reference Manual. Further information is available at https://www.cs.york.ac.uk/circus/RoboCalc/case_studies.

Contents

List of Figures

List of rules

Chapter 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 [21], MASON [10], JASON [1], Argos [15], Enky (home.gna.org/enki/), Microsoft Robotics Developer Studio (www.microsoft.com/robotics), Webots, and Simulink + Stateflow [11, 12]. The graphical notation of RoboSim is inspired on RoboChart [19], 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.
- For simplicity, no hierarchical states nor during actions are allowed in the graphical notation.

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 [17]. 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 [7]. On the other hand, CSP has an underlying model based Hoare and He's Unifying Theories of Programming [9] that makes our core semantics adequate for theorem proving and for extension to deal, for instance, with probability [23].

The remainder of this manual is structured as follows. In Chapter 2 we briefly present RoboChart the language we use to describe designs of robotic applications, and to show how we can relate RoboSim and design models. Using RoboChart we also give in Chapter 2 examples that illustrate the need for separate design and simulation notations. Chapter 3 presents the RoboSim metamodel, and Chapter 4 defines the well-formedness conditions. The semantics is presented in Chapter 5. Chapter 6 presents some case studies. Finally, Chapter 7 concludes with a summary of the results and future work.

Chapter 2

RoboChart and the need for RoboSim

We have previously presented RoboChart [16], a state-machine based notation that can be defined as a UML profile. RoboChart is distinctive in three aspects. First, it provides architectural constructs to identify the requirements for a robotic platform, and a (parallel) design of controllers. Second, it supports definition of time budgets for operations, and deadlines based on events and states. Finally, it has a CSP semantics that can be automatically calculated.

We use the toy model in Figure 2.1 to illustrate RoboChart, and later RoboSim. A robotic system is defined in RoboChart by a module. In our example, it is called CFootBot, and specifies a robot that can move around, detect obstacles, and stop. A module contains a platform and one or more controllers that run on this platform.

The robotic platform FootBot defines the interface of the system with its environment, via variables, operations, and events. In our example, the operation move(lv,av) can be used to set the robot into motion with linear speed lv and angular speed av. The operation stop can be used to instruct the robot to stop (before changing direction, for example). The event obstacle occurs when the robot gets close to any object in its environment; it is an abstraction of a sensor that detects obstacles. The variables, operations, and events of a platform characterise also the facilities that an actual platform must provide to support the implementation of the system.

There may be one or more controllers, interacting with the platform via asynchronous events, and between them via synchronous or asynchronous events. In our simple example, we have just a single controller Movement. The behaviour of a controller is defined by one or more state machines, specifying threads of execution. Again, in our simple example, the behaviour of Movement is defined just by the machine SMovement.

Interfaces can group variables, operations, and events. In Figure 2.1, the interface MovementI has only the operations move(lv,av) and stop(), provided by the robotic platform,

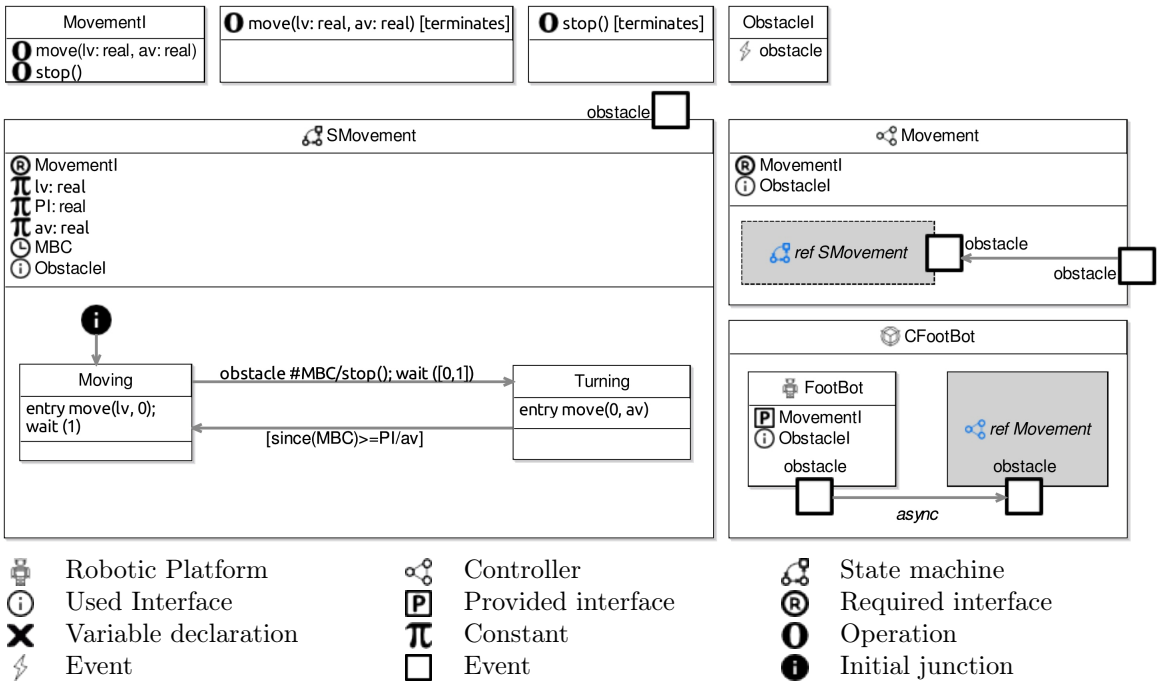

Figure 2.1: RoboChart: obstacle detection

and required by the controller. ObstacleI has just the event obstacle, which is used in the platform, the controller, and the state machine. Libraries containing definitions of platforms and operations may be provided for reuse in the modelling of robotic systems.

Connections in the module define the data flow among the platform and the controllers. In general, different events may be connected, as long as they have the same type, or no type. Types are used when an event communicates an input or output value. Connections in a controller define the flow to, from, and among machines.

State machines are similar to those in UML, except that they have a well defined action language, and time primitives. In our example, upon entry in the state Moving, after calling the operation move(lv,0), the robot waits for one time unit. Operation calls, like move(lv,0), take no time; move(lv,0) can be, for example, implemented as a simple assignment to the register of an actuator. The machine, however, is blocked by wait(1) for one time unit (which is a budget for the platform to react to this operation) before it completes entry to Moving.

In general, assignments and other data operations take no time. Equally, operation calls, in principle, take no time. An operation, however, can itself be defined by a state

machine that uses the time primitives to define budgets. In addition, if an operation is left completely undefined, that is, just a declaration is given, it can take an arbitrary amount of time. In our example, move is defined to terminate (assertion [terminates]), and since it is not given any other explicit definition using time primitives, it takes no time. Budgets need to be explicitly defined via the wait primitive.

SMovement declares a clock MBC. In Moving, when an obstacle is detected, MBC is reset (#MBC), the operation stop() is called, and then the machine may wait for one time unit. After the wait, if any, the machine moves to the state Turning. A wait statement wait([0,1]) is nondeterministic: it can block for 0 or 1 time units. The nondeterminism here may capture, for instance, that the robot may stop virtually instantaneously, or take some (small) amount of time.

In the state Turning, a call move(0,av) turns the robot. A transition back to the state Moving is guarded by since(MBC) >= PI/av. As soon as the guard is satisfied, the transition is taken. The guard requires that the value of MBC is greater than or equal to that of PI/av, to ensure that the robot waits enough time to turn PI degrees, before going back to Moving and proceeding in a straight line again. PI, lv, and av are constants loosely defined.

Although not illustrated in Figure 2.1, we can define deadlines. For instance, the occurrence of an event requires interaction with the environment, and so there is no guarantee of how much time passes before an event occurs. For example, in Moving, an arbitrary amount of time may pass before an obstacle is detected. If appropriate, we can define a deadline, as an imposition on the environment as to the maximum amount of time that may pass.

A more complete description of RoboChart is available in [13, 16], and in the reference manual [19]. Its core concepts (state machines, time budgets, and so on) are common to cyber-physical systems in general. It is the terminology and support provided with the language (in the form of libraries of definitions of platforms and operations, examples, guidelines for use, and so on) that makes it distinctively for the design of robotic systems. RoboChart models are akin to those used in the robotics literature [4, 14, 2, 20]. They are, however, not simulation models, and, if used as a guide to develop a simulation, they leave some important questions unanswered.

First of all, irrespective of the simulator used, a simulation is a cyclic mechanism whose control flow is as described in Figure 2.2. It normally proceeds indefinitely, with termi-

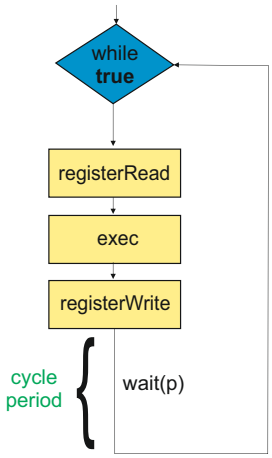

Figure 2.2: Cycle of a simulation

nation determined only by a time limit. In each cycle, the simulation reads some inputs, which correspond to registers of the sensors, executes computations to process that input and calculate outputs, and then writes those outputs to the registers of the actuators. A simulation takes an idealised view of time: input, computation, and output take place infinitely fast at the sample time, and then a period of quiescence follows. The amount of time of quiescence and the sample times are determined by the period.

In contrast, a RoboChart model, like those often written by roboticists to describe their designs, is not cyclic. In our small example, the machine SMovement evolves in a cycle, from Moving to Turning and back, but that is not the cycle of the simulation. First of all, there is no fixed period for that cycle, since, unless we know the exact configuration of the environment and starting point of the robot, we do not know when an obstacle is going to be detected. There may even be no obstacle at all. Second, the simulation cycle determines when the register of the obstacle sensor is read, and so it needs to be much faster than the cycle in the RoboChart machine.

Therefore, when using a RoboChart model to guide the development of a simulation, there is no indication of the simulation cycle. More importantly, there is, therefore, no indication of the actions and transitions that are executed or considered for execution in each cycle. In summary, there is no support for the complex task of scheduling.

Moreover, there are questions that cannot even be asked in a meaningful way. For example, in the context of a simulation, if an event occurs, it is recorded in a register, and it may be relevant to define when it should be cleared. For example, if an obstacle is

detected while SMovement is in the state Turning, that event is ignored. In a simulation, however, the question arises of whether the register should be cleared or not, so that the event can be handled when SMovement is back in the state Moving. Depending on the time the robot takes to turn and on the cycle of the simulation, different answers may be appropriate. If the turning is fast, we may need to retain the event for immediate action after turning. If it is slow, an obstacle seen during the turning may no longer be an issue when the robot finishes turning, and the right decision is to clear the register that records the event unless an obstacle is still sensed.

Finally, a simulation is typically a deterministic program. For that, it needs to prioritise events via specification of behaviour also in their absence. For example, in a state with transitions triggered by two different events e1 and e2, if both events occur, either transition can take place, and the choice is nondeterministic. A simulation typically prioritises one of them. This is achieved by handling the event with lower priority only when that of higher priority does not occur. So, it needs to specify behaviour in the absence of an event. Yet, if the particular priority to be adopted is not important, the RoboChart design model does not enforce any particular prioritisation. Whereas it is not possible to express this priority in a standard state-machine model, as we cannot specify behaviour in the absence of an event, in a simulation the values of the registers, say associated with the events e1 and e2, allows such a control.

RoboSim addresses the issues above: support for scheduling, recording and clearing of event occurrences, and specification of behaviour in the absence of an event. This is achieved by defining a cyclic model, still potentially nondeterministic, where event occurrences are captured by boolean variables, associated with additional variables when they communicate values. The definition of the cycle period and the scheduling for each cycle are explicit.

As an example, we consider the simulation of our simple robot in Figure 2.3. Its overall structure (module and controller) is similar to that of the RoboChart model in Figure 2.1, but, in general, a simulation may have an entirely different structure from that of a corresponding RoboChart model, for optimisation purposes, for instance.

In Figure 2.3, the module is SimCFootBot; it is composed by the FootBot platform and the SimMovement controller that has a reference to a single simulation machine SimSMovement. Essentially, the module differs from that in the RoboChart model just in that it specifies the cycle period: it includes a (simple) predicate stating cycle == 1. Simi-

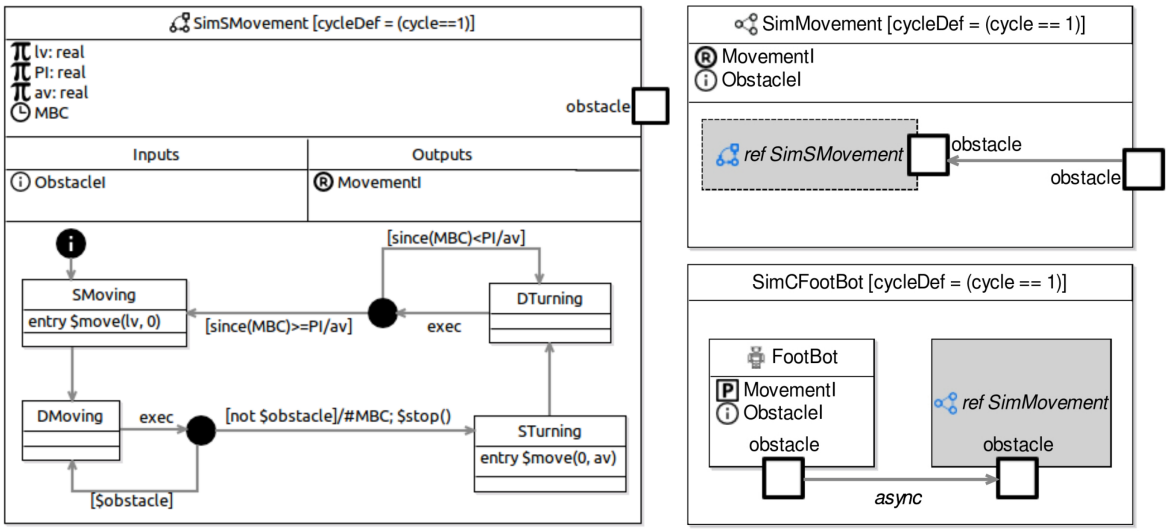

Figure 2.3: RoboSim obstacle detection

larly, the controller SimMovement differs from Movement just in the inclusion of the cycle definition.

The machine SimSMovement has the same local constants PI, lv, and av, and clock MBC defined in the RoboChart machine Movement. The event obstacle declared in the interface ObstacleI is an input, and the operations move and stop declared in the interface MovementI are outputs. Inputs and outputs are explicitly distinguished in RoboSim.

As already said, a RoboSim model specifies a cyclic mechanism; a special marker event exec defines points where behaviour evolution must stop until the next cycle. In each cycle, inputs are read from registers, processed, outputs are written to registers, and then time elapses in a period of quiescence until the next cycle (see Figure 2.2). During processing, the simulation machine takes control of execution until progress requires the (next) occurrence of exec.

The visible behaviour is the reading and writing of registers, which is ultimately characterised by the inputs and outputs. Their values capture interactions corresponding to RoboChart platform events, access to platform variables, and calls to platform operations. For instance, the event obstacle in Figure 2.1 is captured in our example as a

register with a boolean value indicating whether an obstacle has been detected or not. The boolean variable $obstacle corresponding to this input is used in guards, not triggers, of transitions. So, the only trigger used in RoboSim is exec.

The overall behaviour of SimSMovement is as follows. The first cycle starts with the transition from the initial junction to the SMoving state; its entry action records that move must be called, as indicated by $move(lv,0), with the $ as a reminder that the operation is not called immediately. Afterwards, it changes to the DMoving state, where it waits for the next cycle, because there are no transitions from DMoving not triggered by exec (the only possible trigger in RoboSim). This corresponds to the action wait(1) in the RoboChart design model, considering that the cycle period of the simulation is also 1. If we had a different wait time in the design, we would need either to change the size of the cycle, or to change the RoboSim diagram to wait the right number of cycles.

In the next cycle, SimSMovement checks whether an obstacle has been perceived. If not, it remains in DMoving. Otherwise, it moves to STurning, when it resets the MBC clock, records that stop and then move must be called and moves to DTurning, all in one cycle. This defines a simulation that implements the nondeterministic wait([0,1]) by choosing wait(0), which has no effect: it is just a skip statement that terminates immediately.

In the subsequent cycle, it remains in DTurning if the amount of time since MBC has been reset is less than PI/av, otherwise, it returns to SMoving. Here, the check of the clock can be delayed until the next cycle because PI is always greater than 0, so the state change cannot happen in the same cycle.

An alternative RoboSim model that is also correct with respect to the RoboChart model in Figure 2.1 is presented in Figure 2.4. In this case, the nondeterministic wait([0,1]) is implemented by choosing wait(1). So, the transition with guard $obstacle leads to a state Waiting, from which progress can be made only in the next cycle. Yet another option is a RoboSim model that preserves the nondeterminism. For example, we could have two transitions with guard $obstacle (and the same action): one leading to Waiting, like in Figure 2.4, and another leading directly to STurning.

Finally, we observe that, without the wait(1) in the entry action of the state Moving in Figure 2.1, it is not possible to have a simulation for the model where the nondeterministic wait([0,1]) is resolved to wait(0). In this case, if the robot is started at a position that already has an obstacle, the model would require a call to move(lv,0) and, in the same cycle, progress to Turning, and a call move(0,av). Calling the same operation in the same

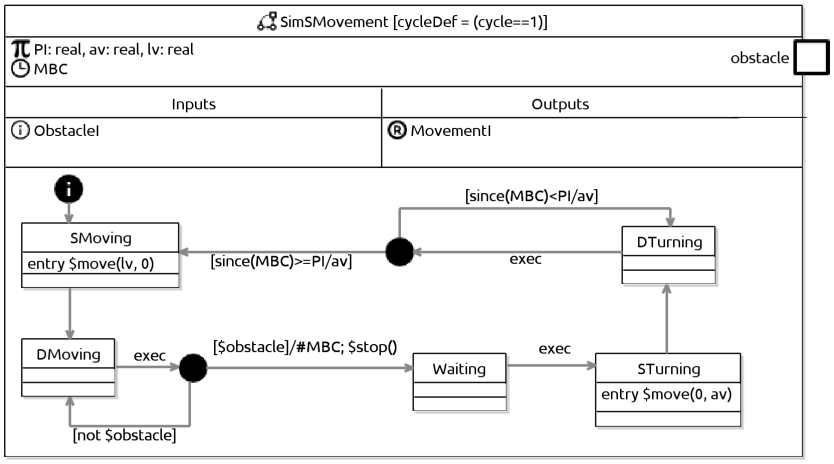

Figure 2.4: RoboSim obstacle detection - alternative

cycle is not possible. So, a RoboChart model without any of the wait primitives does not have a valid cyclic scheduling required in a simulation.

In summary, given a RoboChart model, it may have none or several correct simulations that can be described in RoboSimẆe envisage an ideal development approach where, given a RoboChart model, and extra information like the cycle period, we first of all determine whether the RoboChart model is schedulable in a simulation. If it is, a model transformation can provide a (sketch) for a RoboSim simulation. This model can replicate the structure of module and controllers of the RoboChart model, and build RoboSim state machines using, for example, an eager approach to scheduling where as many transitions as possible are taken in every simulation cycle.

As illustrated above, however, there may be several ways of scheduling the transitions and operations of a RoboChart model in a simulation. So, specialisation and optimisation of the RoboSim model may be needed. For example, a simulation may have a different parallel design. In addition, RoboSim is an independent language and, as such, it can be used to write simulations from scratch, without an explicit reference RoboChart model.

Chapter 3

RoboSim metamodel

Like RoboChart, RoboSim uses state machines to specify behaviour, but as already said, a RoboSim model specifies a cyclic mechanism. The cyclic behaviour is not implicitly defined by the states, but explicitly via the exec event. Budgets and deadlines are defined using the notion of cycle. The control flow defines what can be performed in each cycle, and wait statements and deadlines cannot be used; instead, they are defined counting cycles. The cycle period is defined as a parameter.

RoboSim models are structured using the elements sketched in Figure 3.1.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. In our example, (cycle == 1) is the simple boolean Expression that defines the value for cycle.Modules, controllers, and state machines define the valid sample times for their simulation by specifying restrictions on the value of cycle.

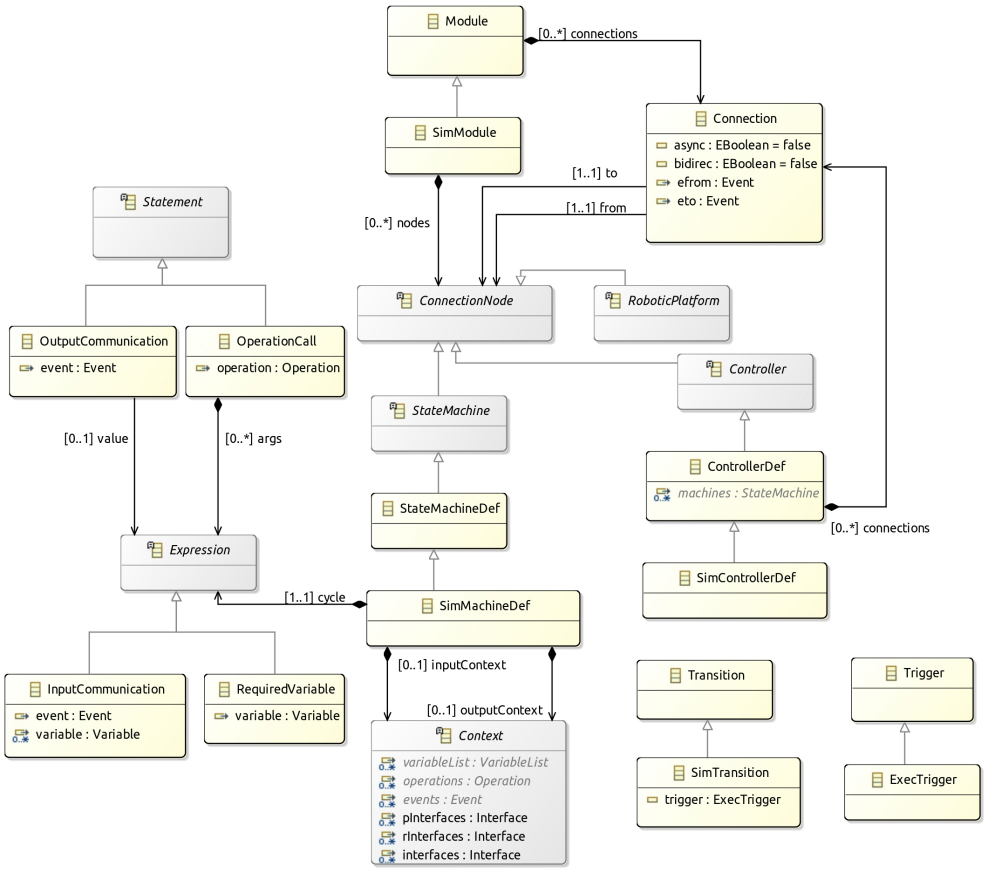

Figure 3.1: RoboSim metamodel

The cycle specifications may admit several valid periods. In our simple example (see Figures 2.3 and 2.4) the cycle is defined everywhere to last one time unit. 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 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. As illustrated in Figure 2.3, we can use references (to state machines, controllers, or platforms) to structure the diagrams, so that components can be easily used in several contexts. Similarly, simulation-controller (SimControllerDef) and simulation-module (SimModuleDef) definitions are forms of controllers and modules, as defined in RoboChart$\tilde{}$(ControllerDef and Module), and a ControllerDef can be a reference.

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. For example, the occurrence of the event obstacle in RoboChart (see Figure 2.1) may represent an indication of the presence of an obstacle implemented using some sensor. The event is used as a trigger in a transition. In contrast, in a simulation, in every cycle a register is read whose value gives an indication as to whether we have an obstacle in range. Accordingly, 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. In our example, the calls to move are written $move(lv,0) and $move(0,av).

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 3.1), 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.

Chapter 4

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 [19]. In what follows, we present the well-formedness conditions that are specific to RoboSim or related to constructs specific to RoboSim;

4.1 Robotic Platforms

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

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

- All interfaces of a platform must be provided or defined. A platform cannot require services from another component, since it is viewed as a resource on which controller software is executed, possibly sharing variables of the platform, using calls to its operations, and communicating with the platform via its events. Variables here include constants, which are distinguished in the metamodel via a modifier to a variable.
- 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.
- Defined interfaces must declare just events. There is no possibility of a platform defining variables or operations (for local use). Behaviour of a platform is not

necessarily specified, except if needed for reasoning. Its variables and operations are, therefore, provided for use in the controllers, where behaviour is specified. Communications with the platform, as well as calls to its operations and accesses to its variables constitute the visible behaviours of a RoboSim model.

- Variables and operations declared directly in the platform, outside an interface, are provided for the reasons already explained above.
- Events declared directly in the platform are defined.
- The names of variables, operations, and events are unique to the platform.

4.2 Interfaces

- The only well-formedness conditions for interfaces requires uniqueness of names of variables, events and operations.

4.3 Modules

- A module has exactly one robotic platform and at least one controller.
- 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.
- 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.
- 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.
- No state machines are included, except indirectly as part of controllers.

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

4.3.1 Module Connection

- A module connection must be asynchronous. The informal view is that the platform makes available the inputs from sensors and writes to actuators as soon as possible. It does not, for example, refuse to provide a sensor input while writing to an actuator.
- Live every other connection, only events of the same type may be connected.

4.4 Controllers

- The specification of cycle is a boolean expression.
- A controller has at least one state machine.
- 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.
- There are no provided interfaces in a controller. We expect controllers to be separate units of processing, which do not share variables or operations. Events are defined, as explained above. Controllers never provide anything to the outside components.
- Required interfaces in a controller contain only variables and operations. They must be provided by the platform in a module that uses this controller, as explained above.
- Defined interfaces contain only variables and events. They are for (shared) use of the state machines that define the behaviour of the controller.
- An operation cannot be declared in a controller outside of an interface. A declaration of an operation outside an interface may be confused with a definition for

local use and is, therefore, not allowed in controllers. Here, we make a distinction between an operation declaration and an operation definition in a controller.

- If a controller defines an operation, it is for (internal) use of its state machines. In this case, it must be fully defined in the controller.
- Variables and events declared directly in the controller, outside an interface, are defined.
- The names of variables, operations, and events are unique to the controller.
- The outputs of the state machines are disjoint, to avoid conflicts as in the case of module.
- 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.

4.4.1 Controller Connection

- Like every other connection, only events of the same type may be connected.

4.5 State Machine

- The specification of cycle is a boolean expression.
- Input declarations can be only events and variables.
- Only required variables can be inputs or outputs (but not both). Variables defined in a machine are there only for local use.
- Conversely, required variables must be an input or output (but not both). In addition, required operations may be outputs.
- All plataform operations must be outputs. Calls to operations provided by the controller, however, do not become outputs of the system.

- Inputs, outputs, and required variables are referenced using a `$`. This emphasises the fact that they are handled in a cyclic pattern. At each cycle, the input events and required variables are read, the output events and required variables are written, and the operations are called.
- There are no provided interfaces in a state machine.
- Required interfaces in a machine contain only variables and operations. A controller that uses the machine must either define or require them as well.
- Defined interfaces contain only variables and events. The events that are declared may be connected to those of other components (other machines in the same controller, or the controller itself). Events that are not connected are not visible.
- An operation cannot be declared in a state machine outside of an interface. This is for compatibility with the definition of a controller.
- The names of variables, operations, and events are unique to the machine.
- There is a single initial junction, with a single transition coming out of it with guard true and no trigger.
- Local event declarations are not allowed.
- The input context has only defined interfaces and events.
- The output context has only events and interfaces that can be defined or required.

4.6  States

- The machine of a composed state has the same well-formedness conditions of a state machine. We note, however, that a state cannot declare variables, operations, and events; a composed state is `NodeConctainer`, without a `Context` of its own.

4.7 Initial nodes

- An initial node does not have incoming transitions. Additionally, it contains at least one outgoing transition and the guards of the outgoing transitions form a cover, that is, their disjunction is true.

4.8 Junction nodes

- A junction node must contain at least one outgoing transition, and the guards of the outgoing transitions form a cover.

4.9 Final states

- A final node does not have outgoing transitions.

4.10 Transitions

- Only transitions starting in a state can have triggers.
- The ends of a transition must belong to the same `NodeContainer`. This forbids interlevel transitions.
- A transition cannot have end deadline and probability junctions.
- The only possible trigger is exec.

4.11 Operations

- There are no provided interfaces in an operation definition.

- Required interfaces in an operation definition contain only variables and operations. A machine that calls the operation must either define or require those variables and operations as well.
- Defined interfaces contain only variables and events. The variables have a local scope. The events that are declared are not connected explicitly. They become events of the state that uses the operation.
- An operation cannot be declared in an operation outside of an interface. This is for compatibility with the definition of a controller. We recall that operations cannot be defined as part of another operation's definition.
- Variables and events declared directly in the operation, outside an interface, are conducted as part of the defined interface.
- If the operation is defined by a state machine, all well-formedness restrictions on a machine's body apply.

4.12 (x) VariableList

- The only well-formedness condition for variable lists requires that the names of variables in a list are unique. This well-formedness applies equally to variables and constants.

4.13 Statements

- Timed and wait statements are not allowed.

4.14 Events

- The broadcast attribute is always false.

Chapter 5

RoboSim Semantics

We now present the formal semantics of RoboSim. In Section 5.1, we describe the notation that we use. We give an overview of the semantics in Section 5.2, and present the rules that formalise it in Section 5.3.

5.1 CSP and tock-CSP

RoboChart has a formal semantics defined using a dialect of CSP called tock-CSP. Systems and their components are defined in CSP via processes, which interact with each other and their environment via atomic and instantaneous events. In tock-CSP, an event tock marks the discrete passage of time. In FDR, there is support to write CSP processes using operators that take this interpretation of tock into account. Some properties, like maximal progress of internal actions (events), however, need to be explicitly stated as we explain in the sequel.

To illustrate the notation, we present below a very simple CSP process CFootBot that specifies the behaviour of our example in Figure 2.1. The formal semantics of RoboChart is implemented by the tool, RoboTool, that automatically calculates a process that is equivalent to CFootBot below. In this example, we do not use the FDR facilities to deal with the tock event; instead, we write processes that deal with this event explicitly.

$$\text{CFootBot} = \text{EntryMoving}; \text{Obstacle}; \text{EntryTurning}; \text{wait(PI/av)}; \text{CFootBot}$$

CFootBot composes sequentially processes EntryMoving, Obstacle, EntryTurning, and wait(PI/av) followed by a recursive call. EntryMoving is below; it engages in the event moveCall.lv.0, which represents the operation call move(lv,0) in the entry action of the state Moving. In sequence (prefixing operator $\rightarrow$), EntryMoving engages in the moveRet

event that marks the return of that operation, and then behaves like the process wait(1).

$$\text{EntryMoving} = \text{moveCall.lv.0} \rightarrow \text{moveRet} \rightarrow \text{wait}(1)$$
$$\text{wait}(n) = \text{if } n == 0 \text{ then SKIP else tock} \rightarrow \text{wait}(n-1)$$

We observe that the states of a machine that controls a robot are not visible in its behaviour. What is visible is the robot's interactions with the environment via its sensors and actuators, represented by variables, operations, and events in RoboChart, and by register reads and writes in RoboSim. This means that the structure of states and transitions of a machine used to convey a design can be completely different from that used in a simulation or in a deployment. Accordingly, variables, operations, and events are captured in the RoboChat semantics by CSP events like moveCall.lv.0 and moveRet, but states are modelled by processes. Similarly, as defined in the next section, register reads and writes are captured by events in the RoboSim semantics, but states are also modelled by processes.

The definition of the parameterised process wait(n) is recursive; it engages in n occurrences of tock to mark the passage of n time units, and after that terminates: SKIP. So, wait(1) corresponds directly to the wait(1) primitive of RoboChart. In FDR, support for the use of tock-CSP includes an in-built definition for this process.

The process Obstacle defined below allows time to pass until an event obstacle occurs, when it then terminates. So, the events obstacle and tock are offered in an external choice ($\Box$).

$$\text{Obstacle} = \text{obstacle} \rightarrow \text{SKIP} \,\Box\, \text{tock} \rightarrow \text{Obstacle}$$

In FDR, with the support provided to use tock-CSP, it is possible to define Obstacle just as obstacle $\rightarrow$ SKIP, with the possibility of the passage of time marked by occurrences of the event tock left implicit.

Finally, the process EntryTurning models the entry action of Turning.

$$\text{EntryTurning} = \text{moveCall.0.av} \rightarrow \text{moveRet} \rightarrow \text{SKIP}$$

Additional operators and examples of CSP processes are presented as needed.

An extra operator of CSP that is implemented in its model checker FDR and that we use both in the semantics of RoboSim and in our verification technique is prioritise. The

general form of this operator is prioritise(P, R), where R is a sequence $\langle X_1, \ldots, X_n \rangle$ of disjoint subsets X_i of the whole set of events Σ in scope for P, with decreasing priority through the list. The process prioritise(P, R) behaves like P, but it prevents any event in X_i, for $i > 1$, from taking place when τ (an internal event), ✓ (termination) or an event in some X_j, with $j < i$, is possible. The events in X_1 have the same priority as that of τ and ✓. Events not in R are incomparable to all other members of R.

We use priorities here to address a technicality of the support for tock-CSP in FDR: we have to use it to ensure maximal progress of internal actions by giving internal events priority over tock. We also use prioritise to encode the assumptions for a simulation; we explain why this is necessary in Section 5.4.

CSP has consistent denotational, algebraic, and operational accounts of its semantics. FDR automatically calculates labelled transition systems for processes, using the operational semantics. Here, the notion of state captures history of interactions and records information like refusal sets to support reasoning about deadlocks.

Distinctively, CSP is a language for refinement. In general, a process P is said to be refined by another process Q when every behaviour of Q is also a possible behaviour of P. In this work, we use the failures-divergences model, where behaviour captures order of events, deadlocks, and livelocks. In addition, in the case of tock-CSP, it captures the time in which events happen. It is possible, however, to use the RoboChart and RoboSim models for less expensive checks that require reasoning just about the traces of events of the models, or just about traces and deadlocks. Although we define a different notion of conformance between RoboChart and RoboSim models, it is, at its core, a refinement, where additional assumptions are added to the specification characterised by a RoboChart model.

As already said, we use the CSP refinement model checker FDR for validation of our semantics and of our verification approach. Using FDR, we can automatically check deadlocks and refinement assertions, for example. FDR, in addition, supports tock-CSP. It recognises the special nature of the tock event; it is possible, for example, to define a timed section where the tock events are introduced automatically.

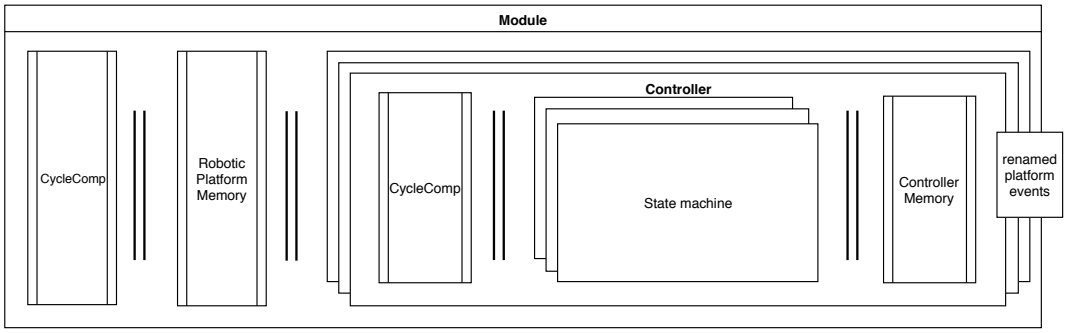

Figure 5.1: Structure of RoboSim tock-CSP semantics

5.2 Overview

The semantics of a RoboSim model is a process that captures the behaviour of the module. Its structure is depicted in Figure 5.1. Controllers, machines, and states are also modelled as CSP processes. We give more details regarding each of the components in what follows.

The visible events are communications over channels registerRead and registerWrite that represent reads and writes to registers of the sensors and actuators. The types of these channels are defined by the types of the inputs and outputs of the model. In all cases, we have a cartesian product whose first component represents an input or output, and the second, and perhaps last, component is a boolean. Its value indicates whether the input or output is available: the event occurred, or the operation was called, for instance. Additional components may be needed when the input or output communicates values, or when the operations have parameters, for example.

In Figure 2.3, for the input event obstacle, we have CSP events registerRead.obstacleU.true and registerRead.obstacleU.false for when the event has and for when the event has not happened. The value obstacleU is a constructor of a data type that represents the inputs. For the operation call $move(lv,av), we have an event registerWrite.moveU.true.lv.av. For cycles in which the operation is not called, registerWrite.moveU.false.lv.av events have arbitrary values for lv and av. Here, moveU is a constructor of a data type that represents outputs.

The semantics formalised in the next section defines a single CSP process for a module, but, for the sake of readability, in examples here, we use process declarations. For example, the semantic function $[\![\_]\!]_{\mathcal{M}}$ specifies the definition below of the process SimCFootBot

for the module in our example in Figure 2.3. That function, however, does not define the declaration that names the process. Besides improving readability, declarations have a positive impact in optimising both the generation and analysis of the models.

Module and controllers The process for a module composes in parallel: a CycleComp(p) process, which orchestrates the overall cyclic execution flow of a component with a cycle p, a memory process, and processes for the controllers of the module. Renaming links the events of the platform and of the controllers as defined by the module connections.

SimCFootBot_CycleComp(p), defined later on, uses the CSP events registerRead and registerWrite to represent interactions with the sensors and actuators of the robot. To pass on register values to and from the controllers, it uses additional channels registerReadC and registerWriteC. For a channel c : T, the notation $\{|c|\}$ is the set of events c.t, for $t \in T$. So, in the parallelism below, synchronisation is required on all events that use registerReadC and registerWriteC, besides tock. Synchronisation on tock ensures that there is a single clock for all parallel components.

$$
\begin{aligned}
\text{SimCFootBot} = \; & ((\text{SimCFootBot\_CycleComp}(1) \\
& \quad [\![\, \{|\text{registerReadC}, \text{registerWriteC}, \text{tock}|\} \,]\!] \\
& \quad \text{SimMovement}[[\text{registerRead} \leftarrow \text{registerReadC}, \\
& \qquad\qquad \text{registerWrite} \leftarrow \text{registerWriteC}]]) \\
& \setminus \{|\text{registerReadC}, \text{registerWriteC}|\}) \; \Theta_{\{\text{end}\}} \; \text{SKIP}) \setminus \{\text{end}\}
\end{aligned}
$$

Termination is handled by an exception ($\Theta_{\{\text{end}\}}$) on the event end of the semantics. If, using end, the controller process SimMovement signals termination, SimCFootBot terminates.

A controller process also uses channels registerRead and registerWrite, but it is defined compositionally, and so the types of registerRead and registerWrite for the controller are defined solely in terms of the inputs and outputs of that controller. Any other inputs and outputs handled by other controllers, which are considered in the module, are not taken into account. So, when a controller process is composed in a module process, we carry out a renaming: the channels registerRead and registerWrite used in the controller are renamed to registerReadC and registerWriteC of the right type for it to communicate with the CycleComp process. The channels registerRead and registerWrite are left for the module readings and writes. The channels registerReadC and registerWriteC are

internal, and so hidden (\). The special channel end used to control termination as explained above is also hidden.

To ensure that, as usual in CSP, we have maximal progress, we need to give internal (hidden) events and the ✓ event, which signals termination, priority over tock. In this way, internal events and termination happen as soon as possible, and cannot be delayed indefinitely. So, the overall semantics of the module is given by the process below defined using the prioritise operator. Since the events in the first element of the sequence provided to prioritise have the same priority as that of τ, which represents hidden events, and ✓, we place tock in the second set of the sequence. With the empty set as the first element, we give priority to hidden events and ✓, and to no others, over tock.

$$\text{PSimCFootBot} = \text{prioritise}(\text{SimCFootBot}, \langle \{\}, \text{tock} \rangle)$$

A prioritised version needs to be defined also for the controller, machine, and state processes. The use of prioritise here is a technicality that can be avoided when using a tool that provides full support for tock-CSP.

Platform variables are outputs of the robotic system, and so are represented by registerWrite events. These variables can, however, be read by other controllers. So, their values need to be recorded in the memory. Variables for internal connections between controllers are neither inputs nor outputs of the module; they are basically shared variables between controllers. So, they are held in the platform memory as well. In our example, there are no platform variables or internal connections. So, the platform-memory process is omitted.

The semantics of a controller is similar to that of a module, except that it composes processes that model the machines of the controller, instead of controller processes (see Figure 5.1). So, we focus here first on the definition of the CycleComp(p) process, which captures the flow in Figure 2.2. The definition for our example is as follows.

$$\begin{aligned} \text{SimCFootBot\_CycleComp(p)} = \; & \text{SimCFootBot\_TakeInputsComp;} \\ & \text{SimCFootBot\_ProvideOutputsComp;} \\ & \text{wait(p);} \\ & \text{SimCFootBot\_CycleComp(p)} \end{aligned}$$

The TakeInputsComp process takes the readings of the registers in any order using registerRead and passes them to the controllers (or machines) that deal with those values

using a different channel readRegisterC. As mentioned above, in general, registerRead and registerReadC have different types, with registerRead accepting readings from any of the sensors of the module, and registerReadC accepting the readings of interest to a particular controller.

In our simple example, with a single controller with a single input, SimCFootBot_TakeInputsComp is very simple.

$$\text{SimCFootBot\_TakeInputsComp} = \text{registerRead?in} \rightarrow \text{registerReadC.in} \rightarrow \text{SKIP}$$

ProvideOutputsComp processes take the outputs from the controllers (or machines) and write to the actuator registers. For our example, with a single output, we have the following. In the case of registerWriteC events, we do not have to identify a controller, since a registerWriteC gives rise to a single registerWrite event.

$$\begin{aligned}\text{SimCFootBot\_ProvideOutputsComp} = \;& \text{registerWriteC?out} \rightarrow \\ & \text{registerWrite.out} \rightarrow \text{SKIP}\end{aligned}$$

In general, for multiple inputs, TakeInputsComp reads and produces them in interleaving. For multiple outputs, their order is defined by the machine that produces them, and is potentially relevant because there can be data dependency between outputs. This is particularly critical when they are calls to operations that access and modify required (and so, shared) variables. The order between the controllers and machines, however, is not fixed, and so ProvideOutptusComp is also an interleaving when there are multiple controllers or machines. In CycleComp(p), after these processes finish, there is a waiting period of p time units, before a recursion to handle the next cycle of the simulation.

Strictly speaking, more elaborate renamings are necessary to map the events of the controller to those of the platform as required in the connections. For instance, the event obstacle used in the machine SimMovement is different from the event obstacle in the platform FootBot, although they have the same name. It is not their names that connect them, but the explicit connection in the module SimCFootBot (see Figure 2.3). So, in the semantics we need different event variables, like SimMovement_obstacle and Foot_Bot_obstacle, and in SimSMovement the event registerRead.SimMovement_obstacle must be matched to the event registerReadC.Foot_Bot_obstacle in the process SimCFoo−tBot_CycleComp(1). We omit these renamings for simplicity here.

The controller process for our example is as follows; its structure is similar to that of a module process.

$$
\begin{aligned}
\mathrm{SimMovement} = \; & ((\mathrm{SimMovement\_CycleComp}(1) \\
& \quad [\![\{\!| \mathrm{registerReadC}, \mathrm{registerWriteC}, \mathrm{tock} |\!\}]\!] \\
& \quad \mathrm{SimSMovement}[[\mathrm{registerRead} \leftarrow \mathrm{registerReadC}, \\
& \qquad\qquad \mathrm{registerWrite} \leftarrow \mathrm{registerWriteC}]]) \\
& \setminus \{\!| \mathrm{registerReadC}, \mathrm{registerWriteC} |\!\}) \; \Theta_{\{\mathrm{end}\}} \; \mathrm{SKIP}
\end{aligned}
$$

Here the termination event end is not hidden so that the controller can signal termination to the module.

State machines For a state machine, the parallelism includes two other components: a Clocks process to represent the clocks used in the machine, and a Buffer process to collect the outputs. The overall structure is described in Figure 5.2. The process for the machine memory records local variables as well as variables for required input variables and for the input events. The Behaviours process captures the core behaviour of the machine itself with a parallelism of processes that model the initialisation and its states. For our example in Figure 2.3, we have the following process.

$$
\begin{aligned}
& \mathrm{SimSMovement} = \\
& \quad (((((\mathrm{SimSMovement\_Cycle}(1) \\
& \qquad [\![\{\!| \mathrm{registerRead}, \mathrm{endexec}, \mathrm{tock} |\!\}]\!] \\
& \quad (\mathrm{SimSMovement\_Memory} \\
& \quad [\![\{\!| \mathrm{setWC\_C0}, \mathrm{setWC\_C1}, \mathrm{internal.tid5}, \mathrm{internal.tid6} |\!\}]\!] \\
& \quad \mathrm{Clocks}) \setminus \{\!| \mathrm{setWC\_C0}, \mathrm{setWC\_C1} |\!\}) \\
& \qquad [\![\{\!| \mathrm{cyclein} |\!\}]\!] \\
& \quad \mathrm{SimSMovement\_Buffer}) \setminus \{\!| \mathrm{cyclein} |\!\}) \\
& \qquad [\![\{\!| \; \mathrm{get\_obstacle}, \mathrm{internal.tid5}, \mathrm{internal.tid6}, \mathrm{clockreset}, \mathrm{stmout}, \\
& \qquad\quad \mathrm{startexec}, \mathrm{endexec}, \mathrm{end} \; |\!\}]\!] \\
& \quad \mathrm{SimSMovement\_Behaviours}) \\
& \qquad \setminus \{\!| \mathrm{get\_obstacle}, \mathrm{stmout}, \mathrm{startexec}, \mathrm{endexec}, \mathrm{clockreset}, \mathrm{internal} |\!\}) \Theta_{\{\mathrm{end}\}} \mathrm{SKIP}
\end{aligned}
$$

The Clocks process is just like in RoboChart, despite the fact that, in a RoboSim model, time can only be observed at the sample times defined by the cycle period. The set_WC_ events are for variables that represent the clock conditions in the machine, and clockreset

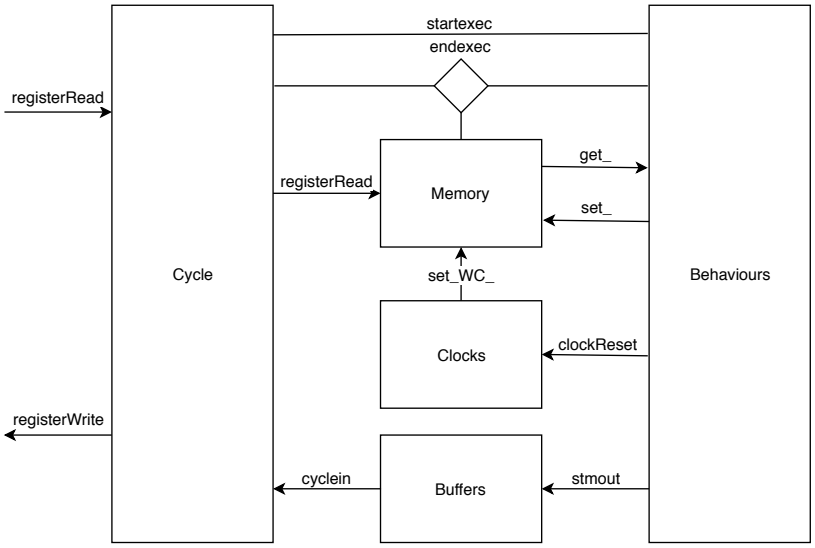

Figure 5.2: Semantics of a RoboSim state machine

events are used to reset clocks. In the Clocks process, we also deal with the transitions that have time restrictions as part of their guards. Events of a special channel internal are used as semantic triggers for transitions that have no trigger (RRoboCharteventt) in the model. In our example, these are the transitions from a junction to SMoving and DTurning (see Figure 2.3), which have identifiers tid5 and tid6. The synchronisation with the Clocks process is, therefore, on events internal.tid5 and internal.tid6 representing the triggers for these transitions. In general, guards are handled by the memory process, so for transitions with time restrictions in their guards, we require cooperation between the memory and clock processes, as well as the Behaviours process.

Cycle(p) The definition of a Cycle(p) process is similar to that of a CycleComp(p) process. As depicted in Figure 5.2, Cycle(p) differs in that it records the sensor readings in the memory, and receives via the channel cyclein output events from the Buffer to be written to the actuators. Moreover, after TakeInputs, Cycle(p) signals to Behaviours that it can start the execution phase of the cycle using an event startexec. For our

example, we have the following definition.

$$\begin{aligned}\text{SimSMovement\_Cycle(p)} = \;& \text{SimSMovement\_TakeInputs};\\ & \text{startexec} \rightarrow \text{endexec?tid} \rightarrow\\ & \text{SimSMovement\_ProvideOutputs};\\ & (\text{wait(p)} \;\Box\; \text{end} \rightarrow \text{SKIP});\\ & \text{SimSMovement\_Cycle(p)}\end{aligned}$$

The RoboSim event exec is realised in the semantics using events startexec and endexec. The event startexec signals to the Behaviours process to start the cycle execution, and endexec is used by Behaviours to signal that it has finished. The event startexec is, for instance, implicitly associated with the transition from the initial node of the machine. After startexec, the Cycle(p) process waits for an endexec, and finally provides the outputs before waiting and recursing. At the point where it is ready to wait, it is also ready to accept a signal from the Behaviours process to terminate via the event end. Termination cannot occur in the middle of the cycle processing or quiescence phases.

In the Behaviours process, the points where endexec is raised are defined by the transitions with trigger exec in the machine. Such transitions may have guards, which control whether the exec event is really available or not. As mentioned above, guards are handled by the memory process, which records the values of the variables that may be referenced in these guards. For this reason, the event endexec takes as parameter the identifier tid of a transition with an exec event. The memory process enables or disables particular events endexec.tid depending on whether the guard for the transition with identifier tid holds or not. In Cycle(p), the particular value of tid is irrelevant, so tid is taken as input: in other words, any exec transition enabled signals the end of the cycle.

The TakeInputs process is similar to those used in CycleComp(p), except that the inputs are recorded directly by the memory process, which uses the registerRead channel to update the values of the input variables (see Figure 5.2). When the inputs communicate no values, like the event obstacle in our example, for instance, the definition of the TakeInputs process is as sketched below. Here, we consider that the inputs are identified in a set Inputs. They are read in interleaving ($|||$); for each input, a boolean b indicates whether the event has occurred or not.

$$\text{TakeInputs} = |||\ \text{in} : \text{Inputs} \bullet \text{registerRead.in?b} \rightarrow \text{SKIP}$$

Because the inputs are stored directly in the memory, there is no need to relay them. On the other hand, Cycle(p) needs to know when all inputs have been read, so that it can determine when execution of the machine can start. For this reason, TakeInputs still needs to be involved in the registerRead events.

ProvideOutputs relays outputs from Behaviours collected in the buffer, and writes to all actuators to signal whether the output has happened or not. Its definition for when no values are communicated as part of the outputs is sketched below. When the output has occurred, it communicates true via registerWrite; otherwise it communicates false for that output. For every output out, ProvideOutputs uses registerWrite.out.b to communicate the boolean b for out.

$$\text{ProvideOutputs} = \text{CollectOutputs}(\text{Outputs})$$

$$\begin{aligned}\text{CollectOutputs}(\text{sout}) = \text{cyclein?out} \rightarrow\ & \text{if}\ (\text{out} == \text{noOut}) \\ & \text{then}\ \text{NoOther}(\text{sout}) \\ & \text{else}\ (\text{registerWrite.out.true} \rightarrow \\ & \qquad \text{CollectOutputs}(\text{sout} \setminus \{\text{out}\}))\end{aligned}$$

$$\text{NoOther}(\text{sout}) = |||\ \text{out} : \text{sout} \bullet \text{registerWrite.out.false} \rightarrow \text{SKIP}$$

In ProvideOutputs, the CollectOutputs process is instantiated with the set of all Outputs. When CollectOutputs receives from the buffer the indication of an output (cyclein?out), it checks whether it is a real output, that is, whether it is different from noOut, communicates registerWrite.out.true, and recurses, removing out from its argument sout. So, the order in which the outputs are buffered is preserved. When noOut is communicated, that is, the buffer is empty, for each remaining output in sout, registerWrite.out.false is communicated by NoOther; as these communications represent absence of the outputs, they are interleaved: their order is irrelevant.

Buffer The definition of the Buffer process is reasonably standard. A buffer is not needed for a controller or module because the register reads and writes are synchronous and the behaviours of the controllers and modules are ultimately defined by the behaviours of the state machines. So, the only role of a controller or module is to relay data to and from the machines synchronously as required. In the case of a machine, on the other hand, the reads and writes are still carried out synchronously, but the values

are kept locally for use during processing (see Figure 2.2), as needed, according to the flow of execution of the machine, which is captured by the Behaviours process.

Buffer takes inputs from the Behaviours process via a channel stmout and produces outputs to the Cycle(p) process via a channel cyclein (see Figure 5.2). As mentioned, an empty buffer, however, provides a noOut output to indicate to Cycle(p) that no more outputs are available. The buffer is finite, with the size of the sequence of outputs sout that records its content limited by the number of outputs of the machine: numOutputs_CFootBot in our example. It is an (implicit) assumption in a simulation that an output cannot occur twice in the same cycle: each cycle of the simulation invokes a sequence of different outputs, and if the same output needs to be performed several times, this is done over several cycles. So, the number of outputs is an adequate limit for the size of the buffer.

The process for our example is as follows. SimSMovement_Buffer represents an empty buffer, and the parametrised process BufferF(sout) represents a buffer holding outputs in the sequence sout.

$$
\begin{array}{l}
\text{SimSMovement\_Buffer} = \text{let} \\
\quad \text{BufferF}(\text{sout}) = \\
\qquad \text{length}(\text{sout}) < \text{numOutputs\_CFootBot} \;\&\; \text{stmout?out} \rightarrow \\
\qquad\quad \text{BufferF}(\text{sout} \frown \langle \text{out} \rangle) \\
\qquad \Box \\
\qquad \text{Flush}(\text{sout}) \\
\quad \text{Flush}(\langle\rangle) = \text{SimSMovement\_Buffer} \\
\quad \text{Flush}(\langle \text{out} \rangle \frown \text{sout}) = \text{cyclein!out} \rightarrow \text{Flush}(\text{sout}) \\
\text{within}\, \text{stmout?out} \rightarrow \text{BufferF}(\langle \text{out} \rangle) \;\Box\; \text{cyclein!noOut} \rightarrow \text{SimSMovement\_Buffer}
\end{array}
$$

We use $\text{sout} \frown \langle \text{out} \rangle$ to describe the concatenation ($\frown$) of sout with the singleton sequence $\langle \text{out} \rangle$. If the buffer has at least one element, when there is a request via cyclein to output, all outputs in the buffer are flushed by the process Flush, until the buffer is again empty (that is, holds the empty sequence $\langle\rangle$), and we have a recursion to Buffer.

Memory The process Memory is similar to that of RoboChart and is reasonably standard in allowing set and get events for the variables of the machine. It, however, records also the inputs of the simulation, and for those it uses registerRead and get_ channels. Cycle (particularly, TakeInputs) can set these inputs, and Behaviours can read them.

In addition, Memory controls when a transition can be taken. Transitions may have, in addition to the trigger exec, guards that depend on the value of the variables in the scope of the machine. It is, therefore, convenient for guards to be evaluated by Memory; the result of that evaluation is used to decide when the transitions can be enabled, pending only on the trigger, if any. We sketch the definition of SMovement_Memory in our example below. It is defined in terms of a local parametrised process called Memory.

SMovement_Memory = let
  Memory(obstacle, ...) = get_obstacle!obstacle →
    Memory_SMovement(obstacle, ...)
  □
  registerRead.obstacleU?x →
    Memory_SMovement(x, ...)
  □
  endexec.tid2 → Memory_SMovement(obstacle, ...)
  □
  ...
within Memory(false, ...)

The parameters of Memory record values for the variables in the memory. In the example here, besides the variable that corresponds to the event obstacle, we have other variables to handle the clocks, omitted above. In the semantics, a transition with trigger exec is treated as a transition with trigger endexec, which is the event that signals to the Cycle(p) process that the execution of the machine for the current cycle has finished. It is raised by Behaviours in accordance with the definition of the machine at the right points, but if there are guards, it can be further constrained by the memory and clock processes. In our example, tid2 is the identifier of the transition in Figure 2.3 from DMoving with trigger exec. Since that transition has no guards, the memory allows the occurrence of that event at any point.

Behaviours The definition of Behaviours process is similar to that for a state machine in RoboChart that has a single event exec. We sketch it below for our example, and focus

on what is different from the RoboChart semantics.

$$\text{SMovement\_Behaviours} = \text{startexec} \rightarrow \left(\begin{array}{l} \text{Init} \\ \quad [\![\{\!| \text{enter.STM.SM}, \text{entered.STM.SM}, \ldots |\!\}]\!] \\ (\text{SMoving\_R} \; [\![\ldots]\!] \; (\text{DMoving\_R} \; [\![\ldots]\!] \; (\text{STurning\_R} \; [\![\ldots]\!] \; \text{DTurning\_R}))) \end{array} \right)$$

$$\text{Init} = \text{enter.STM.SM} \rightarrow \text{entered.STM.SM} \rightarrow \text{SKIP}$$

Just like in the RoboChart semantics, we use special flow events enter, entered, exit, and exited to allow a machine or state to request that a state is entered or exited. These events take two parameters: the identifier of the component that made the request and that of the component to which the request is being made. In the example above, we use identifiers STM and SM for the state machine and for the state SMoving.

A Behaviours process starts with the event startexec to ensure that the machine starts only when the Cycle(p) process has already read the sensors. It is then defined by a parallelism between an initialisation process Init and another parallelism of processes that model each of the states. The Init process proceeds with a request from the machine for, in our example, the state SMoving to be entered, and then waits for the acknowledgement that any entry actions and the initialisation of any substate is completed. In our example, there are none.

The state processes are like those of RoboChart; they establish a control flow using the flow events. There are, however, some key differences. The semantics of actions changes for the case of an operation call. For example, the semantics of $move(lv,0) in the entry action of SMoving is the process $\text{stmout!output}(\text{moveU}, \text{lv}, 0) \rightarrow \text{SKIP}$, which communicates to the buffer the fact that the operation move has been called, and gives its parameters.

The semantics of a trigger also changes in RoboSim. As expected, in the RoboChart semantics a trigger gives rise to a process that waits for the occurrence of the event. In RoboSim, the only trigger is exec, and it gives rise to a process that raises the semantic events endexec and startexec. For instance, for the transition in Figure 2.3 from DMoving with trigger exec, if its identifier is tid2, the trigger is captured by the following process.

$$\text{endexec.tid2} \rightarrow \text{startexec} \rightarrow \text{SKIP}$$

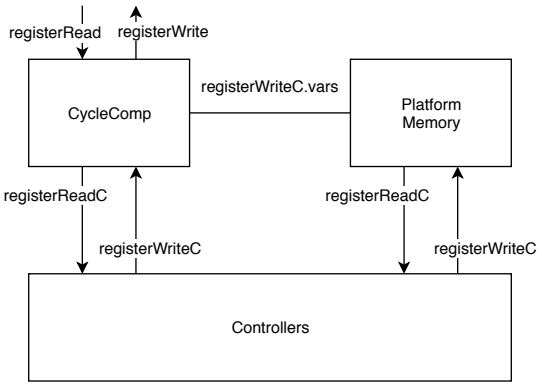

Figure 5.3: Semantics of a RoboSim module

The endexec event takes the transition identifier as a parameter, because its occurrence in different transitions may be associated with different guards, and the memory process needs to identify which guard is relevant in each case, as indicated above. In the case of the implicit event startexec, there is no such need.

5.3 Formalisation

The mapping from RoboSim to tock-CSP is defined by a collection of compositional rules that translate an element of the RoboSim metamodel to a CSP term. The main Rule 1 defined below maps a module to a CSP process. The structure of the process that it defines is depicted in Figure 5.3.

The header of the rule identifies a semantic function and its parameters. In Rule 1, the function $[\![\_]\!]_{\mathcal{M}}$ takes a module m as argument and defines a CSPProcess. Meta-notation used to characterise the process is underlined, and CSP terms are written in italics as usual. The meta-notation is straightforward and explained as needed.

In Rule 1, we name rp, ctrls, and cons components of the metamodel of m extracted using syntactic functions that identify its roboticPlatform, a sequence of its controllers, and a set of its connections. The semantics is defined in terms of further syntactic and semantic functions described below. Rule 1 justifies the definition of SimCFootBot presented in the previous section after simplifications already explained.

The functions Inputs(rp, cons) and Outputs(rp, cons) used in Rule 1 define the inputs and outputs of the module. The inputs are the events associated with the connections from the robotic platform (including those associated with the bidirectional connections to or from the platform). Outputs(rp, cons) contains some of the outputs of the module, namely, the operations of the platform, and the events associated with the connections to the platform (including those associated with the bidirectional connections to or from the platform). Finally, Vars(rp) identifies the rest of the outputs: those corresponding to variables of the platform.

Rule 1. Semantics of modules
$[\![\text{m} : \text{Module}]\!]_{\mathcal{M}} : \text{CSPProcess} =$

$$
\begin{array}{l}
\text{let Module(p : Period)} = \\
\quad ((((\text{cycleComp}^1(\text{p}, \text{inputs}, \text{outputs} \cup \text{vars}) \\
\quad [\![\{ | \text{registerWriteC.data(v)} \mid \text{v} : \text{vars} | \}]\!] \\
\quad \text{memoryComp}^1(\text{vars}, \text{evars}, \text{rp})) \\
\qquad [\![\text{setConstants} \cup \{| \text{registerReadC}, \text{registerWriteC} |\}]\!] \\
\quad \text{composeControllers}^1(\text{m}, \text{ctrls})) \\
\quad \setminus \text{setConstants} \cup \{| \text{registerReadC}, \text{registerWriteC} |\}) \ \Theta_{\{\text{end}\}} \ \text{SKIP}) \setminus \{\text{end}\} \\
\text{within Module(cycle)} \\
\text{where} \\
\quad \text{rp} = \text{m.roboticPlatform} \\
\quad \text{ctrls} = \text{m.controllers} \\
\quad \text{cons} = \text{m.connections} \\
\quad \text{inputs} = \text{Inputs(rp, cons)} \\
\quad \text{outputs} = \text{Outputs(rp, cons)} \\
\quad \text{vars} = \text{Vars(rp)} \\
\quad \text{evars} = \text{Internals(rp, cons)} \\
\quad \text{setConstants} = \{ | \text{c} : \text{allConstants}^1(\text{rp}) \bullet \text{set\_name(c)} | \}
\end{array}
$$

The function Internals(rp, cons) identifies the events that are neither inputs nor outputs of the module, but are used to connect controllers. This function identifies a set of pairs of events: a function evars from events to events. In the domain, we have the events used

as output in a controller. Associated to each such event oute in evars, we have the event inpc used as input in another controller and connected to oute in the module.

Together, the cycleComp^2 and platform-memory processes (memoryComp^2) depicted in Figure 5.3 manage all the data used in the controllers. The CycleComp process reads and writes all registers. The values read from and written to registers for all events and operations are passed on to or from the controllers directly. There is always only one controller that uses the input or produces the output. The values of the platform variables are recorded in the memory, since they are potentially shared between several controllers, although just one of them may update the variable. The memory process also sets the values of the constants using set channels. In Rule 1, the names of these channels are collected in the set setConstants defined using the name name(c) of the platform constants c identified by allConstants^2(rp).

The function memoryComp^3(vars, evars, node) defines a memory process for a module or controller. It holds values for its variables vars and constants, and for the variables in evars representing its internal events. The constants are those whose name and, possibly, value are defined in node. For the constants, there are just set channels to define their values; get channels are available only in machine-memory processes (see Rule 12). For the variables, there are set and get channels; the set channel is registerWriteC and the get channel is registerReadC, which are those used by the controller or machine processes to read and write all inputs and outputs (see Figure 5.3).

In Rule 2, the parameters nvars of the local process Memory represent the values of the variables in vars and in the range of evars. The names of the parameters are the names of these variables. In the memory process for a module or controller, the variables in vars are set through the channel registerWriteC; the get channel is registerReadC.

A variable in the domain of evars corresponds to an output event. When its value is updated via a registerWriteC event, the value of the corresponding variable in the range of evars is updated.

The process defined by constInit^1(node) is an interleaving of set events for the constants of the platform or controller identified by node. For each constant, if its value c.initial is defined in the model (it is not NULL), then that value is set. If it is not, then an arbitrary value name(c) is accepted. When the set channel is hidden, the arbitrary value is chosen in a nondeterministic choice. The controllers and machines that require the constants

synchronise on the set channels to get the same value determined either uniquely or nondeterministically.

Rule 2. Module or controller memory

memoryComp(vars : Set(OutputType), evars : Seq(EventType × EventType,)
node : ConnectionNode) : CSPProcess =

$$
\begin{array}{l}
\text{let Memory}(\text{nvars}) = \\
\quad \left(\Box\, \text{v} : \text{vars} \bullet \left(\begin{array}{l} \text{registerReadC!name(v)} \rightarrow \text{Memory}(\text{nvars}) \\ \Box \\ \text{registerWriteC?take(v)} \rightarrow \text{Memory}(\text{nvars}[\text{name(v)} := \text{give(v)}]) \end{array} \right) \right) \\
\quad \Box \\
\quad (\Box\, \text{v} : \text{ran evars} \bullet \text{registerReadC!name(v)} \rightarrow \text{Memory}(\text{nvars})) \\
\quad \Box \\
\quad (\Box\, \text{v} : \text{dom evars} \bullet \text{registerWriteC?take(v)} \rightarrow \text{Memory}(\text{nvars}[\text{name(evars v)} := \text{give(v)}])) \\
\text{within constInit}^2(\text{node});\ \text{Memory}(\text{initial(nvars)}) \\
\text{where} \\
\quad \text{nvars} = \langle \text{v} : \text{vars} \cup \text{ran evars} \bullet \text{name(v)} \rangle
\end{array}
$$

The function composeControllers^2(m, ctrls) basically constructs a parallelism of the processes for each of the controllers in the sequence ctrls.

Rule 3. Composition of controllers

composeControllers(m : Module, ctrls : Seq(Controller)) : CSPProcess =

$$[\![\{\text{tock}, \text{end}\}]\!]\ \text{c} : 1 \,..\, \#\text{ctrls} \bullet \text{renamingController}(\text{m}, [\![\text{ctrls(c)}]\!]_{\mathcal{C}})$$

It is defined in terms of the function $[\![\text{ctrl}]\!]_{\mathcal{C}}$, which specifies the process for one controller (see Rule 4). The controller processes synchronise only on tock and end because, even if they are connected, that connection is captured in the semantics by variables of the platform memory, not by CSP events of the semantics. Time passage and termination, however, require the agreement of all controllers. Renaming establishes the

connections with the platform of m using registerReadC and registerWriteC for interaction with CycleComp.

Rule 4. Semantics of controllers
$[\![\text{c} : \text{Controller}]\!]_{\mathcal{C}} : \text{CSPProcess} =$

$$
\begin{array}{l}
\text{let Controller(p : Period)} = \\
\quad (((\text{cycleComp}^3(\text{p}, \text{inputs}, \text{outputs}) \;|||\; \text{memoryComp}^4(\text{vars}, \text{evars}, \text{c}) \\
\qquad [\![\text{setConstants} \cup \{|\text{registerReadC}, \text{registerWriteC}, \text{tock}|\}]\!] \\
\quad \text{composeMachines}^1(\text{c}, \text{machines})) \\
\quad \setminus \text{setConstants} \cup \{|\text{registerReadC}, \text{registerWriteC}|\}) \; \Theta_{\{\text{end}\}} \; \text{SKIP} \\
\text{within Controller(cycle)} \\
\text{where} \\
\quad \text{machines} = \text{c.machines} \\
\quad \text{cons} = \text{c.connections} \\
\quad \text{inputs} = \text{InputsC(c, cons)} \\
\quad \text{outputs} = \text{OutputsC(c, cons)} \\
\quad \text{vars} = \text{Vars(c)} \\
\quad \text{evars} = \text{InternalsC(c, cons)} \\
\quad \text{setConstants} = \{ | \text{ct} : \text{allConstants(c)} \bullet \text{set\_name(ct)} | \}
\end{array}
$$

In the function $\text{cycleComp}^4(\text{p}, \text{inputs}, \text{outputs})$, the types InputsType and OutputsType provide data representations for inputs and outputs. For instance, variables of the platform are outputs of the robotic system: any update to such a variable is visible. So, if there is, for example, a variable speed of type Speed in the platform, then OutputType includes a constructor, speedU, for instance, that takes a boolean and a CSP representation of Speed to represent that variable. Since these types are specific for each component, strictly speaking they are parameters of any rules that use them. We omit these parameters for simplicity.

Rule 5. Cycle for modules and controllers

cycleComp(p : Period, inputs : Set(InputsType), outputs : Set(OutputsType)) : CSPProcess =

```
let CycleComp(period) =
    (||| in : inputs • registerRead.take(in) → registerReadC.give(in) → SKIP);
    ((||| out : outputs • registerWriteC.take(out) → registerWrite.give(out) → SKIP)
    wait(period);
    CycleComp(period)
within CycleComp(p)
```

In cycleComp[5], each input in is read via registerRead and passed on using registerReadC to the controller that expects that input. The function take maps an input or output to CSP input parameters of a communication. For example, for an input obstacle, it gives the parameter obstacleU?b, which takes a boolean b as input to indicate whether that event has occurred. For an output moveU.lv.av, we get moveU?b?lv?av, which takes the boolean and the arguments of the operation. Correspondingly, the function give defines output parameters obstacleU.b and moveU.b.lv.av.

The cycle of the module gives rise to a CSP constant cycle whose value needs to be instantiated based on its specification in the module and controllers. It is used as argument for the local process Module defined in Rule 1. Strictly speaking, in the CSP model, we need to qualify the name of this constant, and all names used in global declarations, to avoid clashes. This is because the hierarchical scope structure defined by modules, controllers, and machines is not replicated in CSP. In our formalisation, however, we do not address this issue. It can be solved, for instance, using the CSP notion of modules. In our tool we use fully qualified names based on the names for the modules, controllers, and machines to improve traceability for counterexamples.

Rule 6. Composition of machines

composeMachines(machines : Seq(StateMachineDef)) : CSPProcess =

$$[|\{\text{tock}, \text{end}\}|]\ \text{stm} : 1 \,..\, \#\text{composeMachines} \bullet \text{renamingMachine}(\llbracket \text{machines}(\text{stm}) \rrbracket_{\mathcal{STM}})$$

In the case of the outputs corresponding to a platform variable v, the communication registerWriteC.take(v) is shared with the memory (see Figure 5.3). In this way, the update is both kept in the memory and written to a register.

We next consider Rule 7 for state machines, which formalises the process network in Figure 5.2.

Rule 7. State machine

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

$$
\begin{array}{l}
(((((\text{cycle}^1(\text{cycle}, \text{inputs}, \text{outputs}) \\
\quad [\![\{\!| \text{registerRead}, \text{endexec}, \text{tock} |\!\}]\!] \\
\text{constInitSTM}(\text{consts}, \text{stm}, (\text{stmMemory}^1(\text{stm}, \text{wcs}) \; [\![\text{clockMemSync}]\!] \; \text{stmClocks}^1(\text{wcs}))) \\
\quad \setminus (\text{clockMemSync} \setminus \text{trigEvents}^1(\text{stm}))) \\
\quad [\![\{\!| \text{cyclein} |\!\}]\!] \\
\text{buffer}^1(\text{outputs})) \\
\setminus \{\!| \text{cyclein} |\!\}) \\
\quad [\![\text{getsetChannels}^1(\text{stm}) \cup \text{trigEvents}^2(\text{stm}) \cup \text{clockResets}^1(\text{wcs}) \cup \\
\qquad \{\!| \text{stmout}, \text{startexec}, \text{endexec}, \text{end} |\!\}]\!] \\
\text{behaviours}^1(\text{stm})) \\
\quad \setminus \text{getsetChannels}^2(\text{stm}) \cup \text{clockResets}^2(\text{wcs}) \cup \\
\qquad \{\!| \text{internal}, \text{entered}, \text{stmout}, \text{startexec}, \text{endexec} |\!\}) \\
\Theta_{\{\text{end}\}} \; \text{SKIP}
\end{array}
$$

where

$$
\begin{array}{l}
\text{inputs} = \text{stm.inputs} \\
\text{outputs} = \text{stm.outputs} \\
\text{wcs} = \{\text{t} : \text{allTransitions}^1(\text{stm}) \mid \text{t.condition} \neq \text{null} \bullet \text{t} \mapsto \text{wc}(\text{t.condition})\} \\
\text{clockMemSync} = \left\{ \left| \text{t} : \text{Transition} \mid \text{t} \in \operatorname{dom} \text{wcs} \bullet \text{triggerEvent}^1(\text{t}) \right| \right\} \cup \\
\qquad \left\{ \left| \text{v} : \text{allClockVariables}(\text{wcs}) \bullet \text{setWC\_vid}(\text{v}) \right| \right\}
\end{array}
$$

Rule 8 defines the cycle process for a machine. Its interest on the input registers is just in establishing when all inputs have been read, so that it can signal via startexec the start of the execution of a cycle. The writes to the registers preserve the order in which the

outputs are produced, using CollectOutputs. The defined process is very similar to that presented in the previous section, but considers inputs and outputs that communicate values. The syntactic functions giveT and giveF define the communication parameters to be used when an output has occurred, and when it has not. So, giveT(moveU.lv.av) is moveU.true.lv.av and giveF(moveU.lv.av) is moveU.false.lv.av.

Rule 8. Cycle for machines

cycle(p : Period, inputs : Set(InputsType), outputs : Set(OutputsType)) : CSPProcess =

$$
\begin{array}{lll}
\textbf{let}\ \ \mathrm{Cycle}(\mathrm{period}) = & (\|\!\|\ \mathrm{in} : \mathrm{inputs} \bullet \mathrm{registerRead.take(in)} \rightarrow \mathrm{SKIP}); & \\
& \mathrm{startexec} \rightarrow \mathrm{endexec?tid} \rightarrow \mathrm{SKIP}; & \\
& \mathrm{CollectOutputs(outputs)}; & \\
& (\mathrm{wait(period)} \ \Box\ \mathrm{end} \rightarrow \mathrm{SKIP}); & \\
& \mathrm{Cycle(period)} & \\
\mathrm{CollectOutputs(sout)} = \mathrm{cyclein?out} \rightarrow & \textbf{if}\ (\mathrm{out} == \mathrm{noOut}) & \\
& \textbf{then}\ \mathrm{NoOther(sout)} & \\
& \textbf{else}\ (\mathrm{registerWrite.giveT(out)} \rightarrow & \\
& \mathrm{CollectOutputs(sout} \setminus \{\mathrm{out}\})) & \\
\mathrm{NoOther(sout)} = \|\!\|\ \mathrm{out} : \mathrm{sout} \bullet \mathrm{registerWrite.giveF(out)} \rightarrow \mathrm{SKIP} & & \\
\textbf{within}\ \mathrm{Cycle(p)} & &
\end{array}
$$

The clock process is defined by stmClocks^2(wcs) just like in the RoboChart semantics. Its definition uses a function wcs, which associates every transition t of the machine (in allTransitions^2(stm)) that has a guard (t.condition $\neq$ null) with a representation of that guard using variables. This representation is defined by the function wc, omitted here, and maps clock conditions to variables. These variables v are collected in the set allClockVariables(wcs) and their identifiers vid(v) are used to defined setWC channels in the set clockMemSync.

Rule 9. Clocks for machines

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

$$|| \ (\mathrm{t}, \mathrm{e}, \mathrm{v}) : \{\mathrm{t} : \mathsf{Transition}, \mathrm{e} : \mathsf{Expression}, \mathrm{v} : \mathsf{Variable} \mid \mathrm{t} \in \mathrm{dom}\, \mathsf{wcs} \land (\mathrm{e} \mapsto \mathrm{v}) \in \pi_2(\mathsf{wcs}(\mathrm{t}))\}$$

$$\bullet \ [\![\alpha\mathsf{WC}(\mathrm{t}, \mathrm{e}, \mathrm{v})]\!] \ \mathsf{compileWC}^1(\mathrm{t}, \mathrm{e}, \mathrm{v})$$

where

$$\alpha\mathsf{WC}(\mathrm{t}, \mathrm{e}, \mathrm{v}) = \left\{ \left| \mathsf{triggerEvent}^2(\mathrm{t}), \mathrm{setWC\_vid}(\mathrm{v}) \right| \right\} \cup \mathsf{alphaClockReset}^1(\mathrm{e})$$

Together, the memory and clock processes manage all the data and guards of the machine. They synchronise on the setWC channels for the clock variables, so that guards involving both time primitives and machine variables can be handled. The synchronisation set clockMemSync also includes the $\mathsf{triggerEvent}^1$ for each transition $\underline{\mathrm{t}}$ in the domain of wcs. These include events that use the channel internal. The setWC channels, but not the triggers are local to the memory and clocks processes, and so hidden in the parallelism. The function constInitSTM defines a process that sets the values of the constants, which are used by both the memory and clock processes, putting them in scope for the parallelism of the memory and clock processes. The values of required constants are defined by synchronisation with the controller and module memory processes. So, their values are agreed by all relevant components.

The buffer for a set of outputs is just like that presented in the previous section (Rule 10). The only point that needs to be adjusted for each machine is the limit on the size of the buffer, which must be the size of outputs.

Rule 10. Buffer for machines

buffer(outputs : Set(OutputsType)) : CSPProcess =

$$
\begin{array}{l}
\textbf{let}\ \ \mathrm{Buffer} = \mathrm{stmout?out} \rightarrow \mathrm{BufferOF}(\langle \mathrm{out} \rangle) \,\Box\, \mathrm{cyclein!noOut} \rightarrow \mathrm{Buffer} \\
\quad \mathrm{BufferF}(\mathrm{sout}) = \ \ \mathrm{length}(\mathrm{sout}) < \#\mathrm{outputs} \,\&\, \mathrm{stmout?out} \rightarrow \mathrm{BufferF}(\mathrm{sout} \frown \langle \mathrm{out} \rangle) \\
\qquad\qquad\qquad\qquad\ \ \Box \\
\qquad\qquad\qquad\qquad\ \ \mathrm{Flush}(\mathrm{sout}) \\
\quad \mathrm{Flush}(\langle\rangle) = \mathrm{Buffer} \\
\quad \mathrm{Flush}(\langle \mathrm{out} \rangle \frown \mathrm{sout}) = \mathrm{cyclein!out} \rightarrow \mathrm{Flush}(\mathrm{sout}) \\
\textbf{within}\ \mathrm{Buffer}
\end{array}
$$

The process that defines the control flow of the machine, defined by behaviours^2(stm), is oblivious to the registers, and is defined in terms of the RoboSim variables, including those recording operation calls and events, which are held in the memory and buffer processes (seeRule 11). The inputs read from the registers are put directly in the memory as formalised below. The outputs are stored in the buffer for later use by the cycle process. This rule uses some semantic functions. They give a compositional semantics also at the level of states. What is inherently characteristic of RoboSim is the semantics of the trigger exec and the outputs. Rule 6 specifies the trigger.

Rule 11. Behaviour of state machine

behaviours(stm : StateMachineDef) : CSPProcess =

let

Ending = endexec.terminate → (startexec → Ending □ end → SKIP)

within

startexec →

((((initialisation(stm) |[flowevts]| composeStates[1](⟨s : stm.nodes | s ∈ State⟩, stm))

\ {|enter, exit, exited|})Θ$_{\{end\}}$SKIP) \ {end});

Ending

where

flowevts = ⋃{ x : SIDS \ states(stm); y : states(stm)

• {|enter.x.y, entered.x.y, exit.x.y, exited.x.y|} }

The memory for a state machine stm is defined in Rule 12. It considers the input variables ivars, the local variables lvars, the clock variables cvars, and a list consts of the constants (local and required) in an arbitrary order. The sequence nvars of the names name(v) of all these variables and constants v determines the parameters for Memory. Their initial values initial(v) are collected in varvalues in the order determined by nvars; the list varvalues defines the parameters passed to Memory to define the memory process. For the constants, as explained above, the process defined by constInitSTM puts their names in scope. So, there is no need for a parameter for a constant.

Rule 12. State-machine memory

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

$$
\begin{array}{l}
\text{let } \mathrm{Memory}(\mathsf{nvars}) = \\
\quad (\Box\, \mathrm{v} : \mathsf{ivars} \bullet \mathrm{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \to \mathrm{Memory}(\mathsf{nvars}) \;\Box\; \mathrm{registerRead?x} \to \\
\quad\quad \mathrm{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad (\Box\, \mathrm{v} : \mathsf{lvars} \bullet \mathrm{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \to \mathrm{Memory}(\mathsf{nvars}) \;\Box\; \mathrm{set\_}\mathsf{vid(v)}?\mathrm{x} \to \\
\quad\quad \mathrm{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad (\Box\, \mathrm{v} : \mathsf{cvars} \bullet \mathrm{setWC\_}\mathsf{vid(v)}?\mathrm{x} \to \\
\quad\quad \mathrm{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad \Box\, \mathrm{v} : \mathsf{allConstants}^3(\mathrm{stm}) \bullet \mathrm{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \to \\
\quad\quad \mathrm{Memory}(\mathsf{nvars}) \\
\quad \Box \\
\quad (\Box\, \mathrm{t} : \mathsf{allTransitions}^3(\mathrm{stm}) \bullet \mathsf{memoryTransition}^1(\mathrm{t}, \mathrm{wcs});\ \mathrm{Memory}(\mathsf{nvars})) \\
\quad \Box \\
\quad \mathrm{endexec.terminate} \to \mathrm{Memory}(\mathsf{nvars}) \\
\text{within } \mathsf{Memory(varvalues)}
\end{array}
$$

where

$\mathsf{ivars} = \mathsf{inputVariables(stm)}$

$\mathsf{lvars} = \mathsf{allLocalVariables}^1(\mathsf{stm})$

$\mathsf{cvars} = \mathsf{clockVariables}^1(\mathsf{wcs})$

$\mathsf{consts} = \langle \mathrm{v} : \mathsf{allConstants}^4(\mathsf{stm}) \bullet \mathrm{v} \rangle$

$\mathsf{nvars} = \langle \mathrm{v} : \mathsf{ivars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{name(v)} \rangle \frown \langle \mathrm{v} : \mathsf{consts} \bullet \mathsf{name(v)} \rangle$

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

With $\mathsf{nvars[name(v) := x]}$ we denote the list of variable names (used as argument to Memory), with the name $\mathsf{name(v)}$ replaced with the value x. For the local variables, we have get and set events; the inputs are taken from registerRead directly. The clock variables can be set, but not retrieved, so that there is no get channel for them. They are used only in the Memory process itself to guard transitions. For the constants, we have

only a get channel, since their values cannot be changed. For each transition t in the set allTransitions^4(stm) of transitions of stm, a process memoryTransition^2(t, wcs) models its guard.

Rule 13. memoryTransition function

memoryTransition(t : Transition, wcs : Transition ⇸ (Expression, WC)) : CSPProcess

---

if (t.condition ≠ null) then

$$(\underbrace{[\![\pi_1(\text{wcs}(t))]\!]}_{\mathcal{E}xpr'}) \;\&\; \text{triggerForMemory}^1(t.\text{trigger}, \text{id}(t))$$

else

triggerForMemory^2(t.trigger, id(t))

If the transition has a trigger, which is necessarily exec, then memoryTransition^3(t, wcs) is a prefixing on endexec as formalised in Rule 14. For transitions without a trigger, we have a prefixing on the special event of the semantics called internal as said before.

Rule 14. ExecTrigger

$[\![\text{exec}]\!]^{\text{tid}}_{\mathcal{T}rigger}$: CSPProcess =

---

endexec.tid → startexec → SKIP

As already mentioned, exec is the only possible trigger. The parameter of the semantic function is the identifier tid of the transition where it occurs. The CSP events raised by exec are endexec.tid to indicate the end of the cycle execution, followed by startexec to wait for the next cycle. The transition identifier associates the trigger with the guard of the transition handled in the memory process (see Rule 12).

Finally, a prefixing on endexec.terminate allows this event to happen unconstrained. This event does not represent a transition of the machine. Instead, if the machine terminates, this event is used to keep the cycles going, even in the absence of any useful processing, until all the machines in all controllers terminate. Termination of the simulation is

synchronous, and so the reading of registers need to proceed until no further processing is needed by any component.

Rule 15. Semantics of final states

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

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

In Rule 7, the function $\mathsf{getsetChannels}^3(\mathrm{stm})$ defines the get and set channels for the variables that are internal to the machine.

Rule 16. Get and Set channels

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

$$\left\{ \left| \mathsf{v : allVariables}^1\mathsf{(s)} \bullet \mathrm{get\_}\mathsf{vid(v)} \right| \right\} \cup \left\{ \left| \mathsf{v : allVariables}^2\mathsf{(s)} \bullet \mathrm{set\_}\mathsf{vid(v)} \right| \right\} \cup$$
$$\left\{ \left| \mathsf{v : allConstants}^5\mathsf{(s)} \bullet \mathrm{get\_}\mathsf{vid(v)} \right| \right\} \cup \left\{ \left| \mathsf{v : allConstants}^6\mathsf{(s)} \bullet \mathrm{set\_}\mathsf{vid(v)} \right| \right\}$$

The set $\mathsf{trigEvents}^3(\mathrm{stm})$ in Rule 7 collects the events that represent a trigger in a transition in the machine. Channels used to reset the clocks are included in the set $\mathsf{clockResets}^3(\mathrm{wcs})$; there are channels for the clocks of the machine and for the states for which we have a sinceEntry primitive. This information is available in wcs.

Rule 17. Trigger events

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

$$\left\{ \left| \mathsf{t : allTransitions}^5\mathsf{(s)} \bullet \mathsf{triggerEvent}^3\mathsf{(t.trigger, id(t))} \right| \right\}$$

The rules for outputs define actions that buffer them. We give the action for an operation call in Rule 18. There, eval(e) defines the parameters for the communication corresponding to the list of arguments e for the operation.

Rule 18. Operation call
$[\![\text{op(e)}]\!]_{\mathcal{A}ction}$: CSPProcess =

---

stmout!opU.true.eval(e) → SKIP

Here, eval(e) defines the parameters for the communication corresponding to the list of arguments e for the operation.

5.3.1 Memory

Rule 19. Semantics of triggers for memory
triggerForMemory(t : Trigger, tid : TIDS) : CSPProcess =

---

if t.event.type ≠ null
  eventId(t.event).tid.in?x__ → Skip
else
  eventId(t.event).tid.in → Skip

5.3.2 Clocks

Rule 20. clockVariables function
clockVariables(wcs : Transition ⇸ (Expression, WC)) : ℙ Variable

---

clockVariables(wcs) =
  {t : Transition, e : Expression, v : Variable | t ∈ dom wcs ∧ (e ↦ v) ∈ π_2(wcs(t)) • v}

Rule 21. clockResets function

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

$$\text{clockResets}(\text{stm}) = \bigcup \left\{ \begin{array}{l} \text{t} : \text{Transition}, \text{e} : \text{Expression}, \text{v} : \text{Variable} \mid \\ \text{t} \in \text{dom}\, \text{wcs} \land (\text{e} \mapsto \text{v}) \in \pi_2(\text{wcs}(\text{t})) \\ \bullet\ \text{alphaClockReset}^2(\text{e}) \end{array} \right\}$$

Rule 22. stmClocks function

stmClocks(wcs : Transition ⇸ (Expression, WC)) : TimedCSPProcess =

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

$$\bullet\ [\![\alpha\text{WC}(\text{t}, \text{e}, \text{v})]\!]\ \text{compileWC}^2(\text{t}, \text{e}, \text{v})$$

where

$$\alpha\text{WC}(\text{t}, \text{e}, \text{v}) = \left\{ \left| \text{triggerEvent}^4(\text{t}), \text{setWC\_vid}(\text{v}) \right| \right\} \cup \text{alphaClockReset}^3(\text{e})$$

Rule 23. alphaClockReset function

alphaClockReset(e : Expression) : ChannelSet =

This rule is defined by multiple rules according to the subtype of the expression: (24, 25, 26, 28, 29, 30, 31, 32, 33, 34, 35, 36).

Rule 24. alphaClockReset function

alphaClockReset(e : ParExp) : ChannelSet =

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

Rule 25. alphaClockReset function

alphaClockReset(e : Not) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{5}(e.exp)

Rule 26. alphaClockReset function

alphaClockReset(e : CallExp) : ChannelSet =

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

Rule 27. alphaClockResetCallArgs function

alphaClockResetCallArgs(s : seq Expression) : ChannelSet =

if #(s) > 0 then
  alphaClockReset^{6}(head(s)) ∪ alphaClockResetCallArgs^{2}(tail(s))
else
  ∅
endif

Rule 28. alphaClockReset function

alphaClockReset(e : And) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{7}(e.left) ∪ alphaClockReset^{8}(e.right))

Rule 29. alphaClockReset function

alphaClockReset(e : Or) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{9}(e.left) ∪ alphaClockReset^{10}(e.right))

Rule 30. alphaClockReset function

alphaClockReset(e : Implies) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{11}(e.left) ∪ alphaClockReset^{12}(e.right))

Rule 31. alphaClockReset function

alphaClockReset(e : Iff) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{13}(e.left) ∪ alphaClockReset^{14}(e.right))

Rule 32. alphaClockReset function

alphaClockReset(e : GreaterThan) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{15}(e.left) ∪ alphaClockReset^{16}(e.right))

Rule 33. alphaClockReset function

alphaClockReset(e : GreaterOrEqual) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{17}(e.left) ∪ alphaClockReset^{18}(e.right))

Rule 34. alphaClockReset function

alphaClockReset(e : LessThan) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{19}(e.left) ∪ alphaClockReset^{20}(e.right))

Rule 35. alphaClockReset function

alphaClockReset(e : LessOrEqual) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{21}(e.left) ∪ alphaClockReset^{22}(e.right))

Rule 36. alphaClockReset function

alphaClockReset(e : Equals) : ChannelSet =

alphaClockReset(e) = alphaClockReset^{23}(e.left) ∪ alphaClockReset^{24}(e.right))

Rule 37. alphaClockReset function

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

$\mathsf{alphaClockReset(e)} = \{|\, \mathrm{clockReset.id(e.clock)} \,|\}$

5.3.3 Expressions

Rule 38. Semantics of expressions

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

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

Rule 39. Semantics of and expression

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

$[\![\mathsf{s.left}]\!]_{\mathcal{E}xpr^2} \wedge [\![\mathsf{s.right}]\!]_{\mathcal{E}xpr^3}$

Rule 40. Semantics of array expression

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

$[\![\mathsf{s.value}]\!]_{\mathcal{E}xpr^4}(\{\mathsf{p : s.parameters} \bullet [\![\mathsf{p}]\!]_{\mathcal{E}xpr^5}\})$

Rule 41. Semantics of boolean expression

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

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

Rule 42. Semantics of call expression

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

if (name = ‘size‘ ∧ (head s.args has type SetType)) then
  card($[\![\text{head s.args}]\!]_{\mathcal{E}xpr^6}$)
else if (name = ‘size‘ ∧ (head s.args has type SeqType)) then
  length($[\![\text{head s.args}]\!]_{\mathcal{E}xpr^7}$)
else
  name({a : s.args • $[\![\text{a}]\!]_{\mathcal{E}xpr^8}$})

where

name = s.function.name

Rule 43. Semantics of concatenation expression

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

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

Rule 44. Semantics of not equal expression

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{11}} \neq [\![\text{s.right}]\!]_{\mathcal{E}xpr^{12}}$$

Rule 45. Semantics of division

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{13}} / [\![\text{s.right}]\!]_{\mathcal{E}xpr^{14}}$$

Rule 46. Semantics of equality

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{15}} = [\![\text{s.right}]\!]_{\mathcal{E}xpr^{16}}$$

Rule 47. Semantics of greater or equal expression

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{17}} \geq [\![\text{s.right}]\!]_{\mathcal{E}xpr^{18}}$$

Rule 48. Semantics of greater than

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{19}} > [\![\text{s.right}]\!]_{\mathcal{E}xpr^{20}}$$

Rule 49. Semantics of if and only if expression

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{21}} \Leftrightarrow [\![\text{s.right}]\!]_{\mathcal{E}xpr^{22}}$$

Rule 50. Semantics of implication

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{23}} \Rightarrow [\![\text{s.right}]\!]_{\mathcal{E}xpr^{24}}$$

Rule 51. Semantics of integer expression

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

$$\text{s.value}$$

Rule 52. Semantics of less or equal expression

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{25}} \leq [\![\text{s.right}]\!]_{\mathcal{E}xpr^{26}}$$

Rule 53. Semantics of less than

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{27}} < [\![\text{s.right}]\!]_{\mathcal{E}xpr^{28}}$$

Rule 54. Semantics of minus

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{29}} - [\![\text{s.right}]\!]_{\mathcal{E}xpr^{30}}$$

Rule 55. Semantics of modulus

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

$$[\![\text{s.left}]\!]_{\mathcal{E}xpr^{31}} \text{ mod } [\![\text{s.right}]\!]_{\mathcal{E}xpr^{32}}$$

Rule 56. Semantics of multiplication

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

$$[\![s.\mathsf{left}]\!]_{\mathcal{E}xpr^{33}} \times [\![s.\mathsf{right}]\!]_{\mathcal{E}xpr^{34}}$$

Rule 57. Semantics of arithmetic negation

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

$$-[\![s.\mathsf{exp}]\!]_{\mathcal{E}xpr^{35}}$$

Rule 58. Semantics of logical negation

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

$$\neg\, [\![s.\mathsf{exp}]\!]_{\mathcal{E}xpr^{36}}$$

Rule 59. Semantics of or expression

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

$$[\![s.\mathsf{left}]\!]_{\mathcal{E}xpr^{37}} \vee [\![s.\mathsf{right}]\!]_{\mathcal{E}xpr^{38}}$$

Rule 60. Semantics of parenthesised expression

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

$$([\![s.\mathsf{exp}]\!]_{\mathcal{E}xpr^{39}})$$

where

Rule 61. Semantics of plus

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

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

Rule 62. Semantics of range expression

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

$$\{ x : \mathbb{N} \mid [\![s.\mathsf{lrange}]\!]_{\mathcal{E}xpr^{42}} \; \mathsf{rel1}\, x \wedge x \, \mathsf{rel2} \, [\![s.\mathsf{rrange}]\!]_{\mathcal{E}xpr^{43}} \}$$

where

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

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

Rule 63. Semantics of sequence expression

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

$$\langle \{ x : s.\mathsf{values} \bullet [\![x]\!]_{\mathcal{E}xpr^{44}} \} \rangle$$

Rule 64. Semantics of set expression

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

$\{\{x : s.values \bullet [\![x]\!]_{\mathcal{E}xpr^{45}}\}\}$

Rule 65. Semantics of tuple expression

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

$(\{x : s.values \bullet [\![x]\!]_{\mathcal{E}xpr^{46}}\})$

5.3.4 Auxiliary Functions

Rule 66. Function allVariables

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

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

Rule 67. Function allLocalVariables

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

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

Rule 68. Function allConstants

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

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

Rule 69. Function requiredVariables

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

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

Rule 70. Function allTransitions

allTransitions(s : NodeContainer) : Set(Transition) =

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

Rule 71. triggerEvent function

triggerEvent(t : Transition) : CSPEvent =

if t.trigger $\neq$ null
  endexec.id(t)
else
  internal.id(t)

Rule 72. Composition of states

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

if #ss = 1
then
  restrictedState^1(p, head ss)
else
  (restrictedState^2(p, head ss)
    [|shflowevts|]
    composeStates^2(tail ss, p))

where

shflowevts = flowEvents^1(head ss, p) ∩ ⋃{x : tail ss • flowEvents^2(x, p)}

Rule 73. Restricted semantics of states

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

⟦s⟧$_§$ [| all_other_transitions_S \ all_transitions_PS |] Skip

where

tidsfromwithin = {t : transitionsFrom^1(s) ∪ allTransitions^6(s) • id(t)}
all_other_transitions_S = {e : Event; tid : TIDS \ tidsfromwithin • eventId(e).tid}
all_transitions_PS = {e : Event; tid : TIDS • eventId(e).tid}
  \{t : allTransitions^7(p) • eventId(t.trigger.event).id(t)}

Rule 74. Flow events

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

$$\bigcup \{x : \text{states}^{1}(p);\ y : \{\text{id}(s)\} \bullet \left\{ \begin{array}{l} \text{enter}.y.x, \text{entered}.y.x, \text{exit}.y.x, \text{exited}.y.x, \\ \text{enter}.x.y, \text{entered}.x.y, \text{exit}.x.y, \text{exited}.x.y, \end{array} \right\} \}$$

Rule 75. Function transitionsFrom

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

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

Rule 76. Function states

states(n : NodeContainer) : Set(State) =

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

Rule 77. Constants Initialisation for Controllers and Modules

constInit(node : ConnectionNode) : CSPProcess =

$$||| c : \text{consts} \bullet \left(\begin{array}{l} \text{if } c.\text{initial} \neq \text{NULL then} \\ \quad \text{set\_vid}(c)![\![c.\text{initial}]\!] \rightarrow \text{Skip} \\ \text{else} \\ \quad \text{set\_vid}(c)?\text{name}(c) \rightarrow \text{Skip} \end{array} \right)$$

where

$$\text{consts} = \text{allConstants}^{7}(\text{node})$$

5.3.5 Waiting Condition as CSP processes

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

Rule 78. compileWC function

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

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

let

Reset = clockReset.id(C) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\|\text{triggerEvent}^{5}(t)\|\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{47}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\|\text{triggerEvent}^{6}(t)\|\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

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

let

Reset = clockReset.id(C) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\|\text{triggerEvent}^{7}(t)\|\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{48}}} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\|\text{triggerEvent}^{8}(t)\|\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\|\text{triggerEvent}^{9}(t)\|\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{49}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\|\text{triggerEvent}^{10}(t)\|\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\|\text{triggerEvent}^{11}(t)\|\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{50}}} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\|\text{triggerEvent}^{12}(t)\|\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

Rule 78. compileWC function

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{13}(t) \mid\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{51}}} +1\ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{14}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{15}(t) \mid\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{52}}} +1\ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{16}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{17}(t) \mid\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{53}}} +1\ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{18}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{19}(t) \mid\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{54}}} +1\ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{20}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

Rule 78. compileWC function

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

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

let

Reset = clockReset.id(C) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\!|\text{triggerEvent}^{21}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{55}}} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\!|\text{triggerEvent}^{22}(t)|\!\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

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

let

Reset = clockReset.id(C) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\!|\text{triggerEvent}^{23}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{56}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\!|\text{triggerEvent}^{24}(t)|\!\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\!|\text{triggerEvent}^{25}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{57}}} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\!|\text{triggerEvent}^{26}(t)|\!\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\!|\text{triggerEvent}^{27}(t)|\!\}) \\ \quad \triangle_{[\![e]\!]_{Expr^{58}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\!|\text{triggerEvent}^{28}(t)|\!\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

Rule 78. compileWC function

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

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

let

Reset = clockReset.id(C) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{29}(t) \mid\}) \\ \quad \triangle_{(\llbracket e \rrbracket \atop \mathcal{E}xpr^{59}} +1)\ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{30}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

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

let

Reset = clockReset.id(C) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{31}(t) \mid\}) \\ \quad \triangle_{(\llbracket e \rrbracket \atop \mathcal{E}xpr^{60}} +1)\ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{32}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!true → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{33}(t) \mid\}) \\ \quad \triangle_{(\llbracket e \rrbracket \atop \mathcal{E}xpr^{61}} +1)\ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{34}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!true → Monitor

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

let

Reset = entered?x.id(S) → setWC_vid(v)!false → Monitor

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{35}(t) \mid\}) \\ \quad \triangle_{(\llbracket e \rrbracket \atop \mathcal{E}xpr^{62}} +1)\ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{36}(t) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

Rule 78. compileWC function

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{37}(\text{t}) \mid\}) \\ \quad \triangle_{[\![\text{e}]\!]_{Expr^{63}}} \ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{38}(\text{t}) \mid\}) \\ \quad\quad \triangle_{([\![\text{e}]\!]_{Expr^{64}}+1)} \ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{39}(\text{t}) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{40}(\text{t}) \mid\}) \\ \quad \triangle_{[\![\text{e}]\!]_{Expr^{65}}} \ \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{41}(\text{t}) \mid\}) \\ \quad\quad \triangle_{([\![\text{e}]\!]_{Expr^{66}}+1)} \ \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{42}(\text{t}) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

setWC_vid(v)!false → Monitor

Rule 78. compileWC function

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{43}(\text{t}) \mid\}) \\ \quad \triangle_{[\![\text{e}]\!]_{\mathcal{E}xpr^{67}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{44}(\text{t}) \mid\}) \\ \quad \triangle_{([\![\text{e}]\!]_{\mathcal{E}xpr^{68}}+1)} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{45}(\text{t}) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

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

let

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

$$\text{Monitor} = \left(\begin{array}{l} \text{RUN}(\{\mid \text{triggerEvent}^{46}(\text{t}) \mid\}) \\ \quad \triangle_{[\![\text{e}]\!]_{\mathcal{E}xpr^{69}}} \text{setWC\_vid(v)!true} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{47}(\text{t}) \mid\}) \\ \quad \triangle_{([\![\text{e}]\!]_{\mathcal{E}xpr^{70}}+1)} \text{setWC\_vid(v)!false} \rightarrow \text{RUN}(\{\mid \text{triggerEvent}^{48}(\text{t}) \mid\}) \end{array} \right) \triangle \text{Reset}$$

within

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

In the next section, we explain how we use the semantics of RoboSim to verify consistency of models.

5.4 Automated verification

Although a RoboSim model can be developed from scratch, verifications can be performed if there is a reference design model. As previously explained, the comparison between the models is not straightforward for a few reasons. Basically, a RoboChart model describes order, availability, and time of updates to platform variables, of platform events, and of calls to platform operations. These updates, events, and calls can happen at any time. RoboSim models, on the other hand, describe order, availability, and time of registerRead and registerWrite events, which occur at the sample times of the simulation defined by the cycle period. Any straightforward comparison fails.

Roughly speaking, to compare the models, we need to relate occurrences of registerRead and registerWrite events to updates to platform variables, platform events, and calls to platform operations. We also need to justify how a model that deals with occurrences of events at any point in time corresponds to a model where events are only perceived at sample times. For that, we formalise assumptions that describe the input events as registerRead events, and describe updates to platform variables, output events, and calls to platform operations as registerWrite events. We also formalise the assumptions embedded in a simulation that events do not occur between cycles, each cycle has at most one occurrence of each output and, in each cycle, each operation is called at most once. It is these assumptions routinely made by simulation developers that allow us to relate the models in the two abstraction levels.

In summary, formally, what specifies a simulation is not a RoboChart model, but a restriction of that model to consider a mapping of inputs and outputs into register reads and writes, and a cycle for the simulation. We note, however, that the specification so obtained (a) may not be feasible (schedulable), (b) it is described in CSP, but our approach does not require practitioners to know CSP, and (c) it may be implemented by any of a collection of RoboSim models, which may vary in component structure and scheduling. So, the availability of RoboSim allows design of simulations using a language that adopts the right paradigm, but still insulates practitioners from formalisms. In addition, it is, of course, possible to design a simulation in RoboSim without reference to a design in RoboChart.

Our formalisation of the assumptions takes the form of processes TA1, TA2, and TA3 (Section 5.4.1). Using these assumptions, we can, first of all, verify that the RoboChart model can be scheduled in a cyclic design as imposed by a simulation. We illustrate that such

scheduling or even its existence is not always obvious, and our technique supports practitioners with early verification of schedulability via a deadlock check. If the RoboChart model is schedulable, we can then compare it, in the context of the assumptions, to a RoboSim model (Section 5.4.2). Although we consider here properties of our running example, the assumptions and assertions define general templates that can be automatically generated from the RoboChart and the RoboSim models, along with their CSP semantics.

5.4.1 Relating design and simulation models

The first assumption relates RoboChartand RoboSim inputs, expressed as events and register reads, respectively. As the order of inputs is not relevant, that is, the registers can be read in any order, the association between the two input representations is modelled using interleaving. This is captured in the process A1 below, used to define TA1.

$$\mathrm{A1} = (|||\, \mathrm{in} : \mathrm{Inputs} \bullet \mathrm{A1Event(in)})$$

For an input in, the process A1Event(in) engages in the event registerRead.in?b?x, which corresponds to a register read for input in. If in does not communicate any values, like obstacle in our example, just a boolean b is input; otherwise, it is a tuple with a boolean and the communicated values x. If b is true, the robotic platform has raised the (RoboChart) event: it may happen, as modelled by useinv(in), or may be ignored by the software, and so does not happen, as captured in the RoboChart design model. If b is false, the event cannot happen.

$$\begin{aligned} \mathrm{A1Event(in)} \;=\; & \mathrm{registerRead.in?b?x} \rightarrow \\ & \quad \mathrm{if\ b} \\ & \quad \mathrm{then\ (useinv(in);\ A1Event(in)} \;\Box\; \mathrm{A1Event(in))} \\ & \quad \mathrm{else\ A1Event(in)} \end{aligned}$$

The process useinv(in) maps an event in to a process that raises the event itself and then terminates. In our example, we relate registerRead.obstacleU.true to obstacle. So, we define the input event obstacleU as a data representation of the RoboChart event obstacle that is communicated in registerRead, and the process useinv(obstacleU) = obstacle → SKIP. The assumption captured by A1 not only maps the platform events to register reads, but also ensures that, for each register reading, the input can occur at most once.

We constrain the semantics of the RoboChart model by composing it in parallel with the assumptions. Therefore, a process that models an assumption (like A1) must terminate when the specification does. The event end is used to interrupt ($\triangle$) A1 and enforce its termination: $\mathrm{TA1} = \mathrm{A1} \triangle (\mathrm{end} \rightarrow \mathrm{SKIP})$.

Our second assumption relates the RoboChart and RoboSim outputs, taking into account that the order of the outputs to be performed in a cycle may be relevant. In the definition of the process A2 below, the process Order has two arguments: the sequence of outputs that have occurred in the cycle, and the set of those that have not. The initial arguments to Order are the empty sequence $\langle\rangle$ and the set with all the possible outputs.

$$\mathrm{A2} = \mathrm{Order}(\langle\rangle, \mathrm{OutputEvents});\ \mathrm{A2}$$

For our example, OutputEvents is $\{\mathrm{moveCall}, \mathrm{stopCall}\}$; we record the operation calls, but not their return.

Order, defined below, records each output out that occurs in the cycle in the sequence outOcc, and removes them from the set outNOcc. This is guarded by a condition that requires the number of output events that have occurred to be smaller than the total number numOuts of such events. Here we take advantage of the assumption that the same output cannot occur more than once in a cycle just to ensure that Order is finite; it is enforced by A3 defined blow.

$$\begin{array}{l}
\mathrm{Order}(\mathrm{outOcc}, \mathrm{outNOcc}) = \\
\quad \mathrm{length}(\mathrm{outOcc}) < \mathrm{numOuts}\ \&\ (\Box\, \mathrm{out} : \mathrm{OutputEvents} \bullet \mathrm{out} \rightarrow \\
\qquad \mathrm{Order}(\mathrm{outOcc} \frown \langle \mathrm{out} \rangle, \mathrm{outNOcc} \setminus \{\mathrm{out}\})) \\
\quad \Box \\
\quad \mathrm{outOcc} \neq \langle\rangle\ \&\ \mathrm{registerWrite.use}(\mathrm{head}(\mathrm{outOcc})).\mathrm{true} \rightarrow \\
\qquad \mathrm{Order}(\mathrm{tail}(\mathrm{outOcc}), \mathrm{outNOcc}) \\
\quad \Box \\
\quad \mathrm{outOcc} == \langle\rangle\ \&\ (|||\, \mathrm{out} : \mathrm{outNOcc} \bullet \mathrm{registerWrite.use}(\mathrm{out}).\mathrm{false} \rightarrow \mathrm{SKIP})
\end{array}$$

Once the cycle finishes, the outputs are written out via the channel registerWrite. The outputs that have actually occurred in the cycle (those in outOcc) are written in the same order as they are recorded. The remaining ones (those in outNOcc) have not occurred in the current cycle, and so their absence is communicated as an interleaving, since the order is irrelevant in this case. We note, however, that Order allows outputs to be written during the actual processing of the cycle. What ensures that they are written only at

the end of the cycle is the use of the prioritise operator of CSP (as further detailed in the sequel) to allow registerWrite to occur only when no more external events can.

The function use specifies the inverse mapping of the previously defined process useinv(in); it maps an event into its data representation that is communicated via the channel registerWrite. In the case of an operation op, only the event opCall is considered. For example, use(moveCall.lv.0) = moveU.lv.0, where moveU, as previously indicated, is a data value used to signal the occurrence of the move operation and its arguments.

As before, TA2 is the version of A2 that handles termination using interrupt as shown above for TA1.

The third and final assumption captures the pattern of a simulation cycle, which, as already discussed, involves iterations of sensor readings (via the channel registerRead); performing the control behaviour related to the current cycle; writing to the actuators (via the channel registerWrite); and then waiting for p time units (the cycle period). Unlike the other two assumptions, termination is handled as part of the assumption specification, rather than separately using interruption. The reason is that, for this third assumption, termination is allowed only at the end of the cycle. The process RUN(X) continuously offering the events in the set X. Formally, RUN(X) = $\Box$ x : X $\bullet$ x $\rightarrow$ RUN(X).

$$
\begin{aligned}
\text{TA3} = \; & (|||\ \text{in} : \text{Inputs} \bullet \text{registerRead.in?b?x} \rightarrow \text{SKIP}); \\
& (\text{RUN}(\text{ExternalEvents}) \triangle (|||\ \text{out} : \text{OutputEventsOp} \bullet \text{registerWrite.out?b?x} \rightarrow \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{SKIP})); \\
& (\text{end} \rightarrow \text{SKIP} \;\Box\; \text{wait(p);}\ \text{TA3})
\end{aligned}
$$

In TA3, after the registerRead events take place in interleaving for all inputs, the external events are allowed indefinitely in any order. This proceeds until a registerWrite event takes place. The outputs are allowed also in interleaving. When all outputs take place then either the simulation terminates, as signalled by the end event, or a period of p time units takes place, and then a new cycle is started via a recursion.

The composition of the three assumptions is given by the following process.

$$
\begin{array}{lll}
\mathrm{TA1A2A3} & = & (\mathrm{TA1} \,[\![\, \mathrm{InputEvents} \cup \{\!|\mathrm{registerRead}, \mathrm{end}|\!\} \\
& & \quad | \\
& & \quad \mathrm{OutputEvents} \cup \{|\, \mathrm{registerWrite}, \mathrm{end} \,|\} \,]\!]\, \mathrm{TA2}) \\
& & [\![\mathrm{InputEvents} \cup \{\!|\mathrm{registerRead}, \mathrm{end}|\!\} \cup \mathrm{OutputEvents}]\!] \\
& & \cup \{|\, \mathrm{registerWrite}, \mathrm{end} \,|\} \\
& & | \\
& & \mathrm{ExternalEvents} \cup \{|\, \mathrm{registerRead}, \mathrm{registerWrite}, \mathrm{tock}, \mathrm{end} \,|\} \,]\!]\, \mathrm{TA3}
\end{array}
$$

This definition uses the alphabetised parallel operator of CSP. The sets InputEvents and ExternalEvents contain the input and external events of the process $[\![\mathrm{RCM}]\!]_{\mathcal{M}}$ that defines the semantics for a RoboChart module RCM used below. In our example, these sets of events are $\{\mathrm{obstacle}\}$ and $\{\mathrm{obstacle}, \mathrm{moveCall}, \mathrm{moveRet}, \mathrm{stopCall}, \mathrm{stopRet}\}$, respectively. The alphabets of each process in the definition of TA1A2A3 is simply the set of events used by that process.

TA1A2A3 is composed in parallel with $[\![\mathrm{RCM}]\!]_{\mathcal{M}}$ to define a specification Constrained for a simulation.

$$
\begin{array}{l}
\mathrm{Constrained} = ([\![\mathrm{RCM}]\!]_{\mathcal{M}};\ \mathrm{end} \rightarrow \mathrm{SKIP} \,[\![\, \mathrm{ExternalEvents} \cup \{\mathrm{tock}, \mathrm{end}\} \,]\!]\, \mathrm{TA1A2A3}) \\
\qquad\qquad \setminus \{\mathrm{end}\}
\end{array}
$$

The sequential composition of $[\![\mathrm{RCM}]\!]_{\mathcal{M}}$ with the process $\mathrm{end} \rightarrow \mathrm{SKIP}$ is a technicality. The purpose of the event end, as already explained, is to ensure distributed termination. It is used but hidden in $[\![\mathrm{RCM}]\!]_{\mathcal{M}}$; so if $[\![\mathrm{RCM}]\!]_{\mathcal{M}}$ terminates, the process $\mathrm{end} \rightarrow \mathrm{SKIP}$ signals termination to TA1A2A3. The event end is hidden in the outermost scope. The parallel processes in Constrained synchronise on their common events: tock, end and the external events.

Although the purpose of TA3 is to capture the pattern of a simulation cycle, it does not enforce that registerWrite events can happen only after all the external events of the current cycle, since RUN(ExternalEvents) can be interrupted at any point. To impose the cycle policy, we use the prioritise operator. The resulting process is PConstrained below.

$$\mathrm{PConstrained} = \mathrm{prioritise}(\mathrm{Constrained}, \langle \mathrm{ExternalEvents}, \{|\, \mathrm{registerWrite}, \mathrm{tock} \,|\} \rangle)$$

We give priority to the RoboChart events over the RoboSim output events, that is, the registerWrite events, and so we use the first communication on registerWrite to mark the end of a cycle of the RoboChart machine. We also give RoboChart events priority over tock to ensure the events available are urgent, as required.

PConstrained imposes all the constraints of the simulation assumptions on the RoboChart model so that it can now be directly compared to a RoboSim simulation model. This allows us to perform some interesting verifications.

5.4.2 Verification

With a tock-CSP semantics for the RoboChart and the RoboSim models, we can verify their properties in isolation, as well as check the soundness of the RoboSim model with respect to the constrained RoboChart model. We now explain how to perform application-independent checks for schedulability and soundness.

Schedulability As already explained, it is useful to check that the RoboChart model can be captured in a simulation, given the necessary assumptions defined above. Verification of deadlock freedom of the constrained RoboChart model captures the analysis of this meaningful property for the simulation: schedulability in cycles. The assumptions impose additional restrictions on the RoboChart model. If those restrictions are unsatisfiable, then we have a deadlock. Absence of deadlock of the constrained RoboChart model, as formalised by PConstrained, on the other hand, indicates that it is schedulable. This can be verified in FDR using the following assertion.

```
assert PConstrained :[deadlock free]
```

Of course, if the RoboChart model itself deadlocks, this check is not useful; PConstrained, in this case, is certain to deadlock. If, however, we suppose that it is not useful to simulate a model that has a deadlock, and so the RoboChart model is itself deadlock free, a deadlock here is caused by the assumption TA3. The deadlock indicates that, when the RoboChart model is further restricted by the order and timing imposed by TA3, there is a conflict. So, the scheduling in cycles imposed by TA3 cannot be satisfied by the RoboChart model.

This assertion holds for our running example, considering p = 1 as the cycle period, but it is instructive to explore scenarios where it does not hold. In our first RoboChart

model, the wait(1) in the entry action of Moving (see Figure 2.1) was not included, but without it the model is not schedulable. In this case, the shortest trace yielded by FDR as a counterexample of the deadlock verification is as shown below.

$$\langle\ \text{registerRead.obstacleU.true},\ \text{moveCall.lv.0}, \text{moveRet}, \text{obstacle}, \text{stopCall}, \text{stopRet},\ \text{registerWrite.moveU.true.lv.0}, \text{registerWrite.stopU.true}\rangle$$

After the operation call stop(), represented in the RoboChart model by the events stopCall and stopRet, the RoboChart model needs to call the move operation in the entry action of Turning; at the semantic level, this means engaging in the events moveCall and moveRet. However, the assumptions forbid more than one call of the same operation in the same cycle. So, the composition of the RoboChart model with the assumptions deadlocks.

This schedulability analysis also detects inconsistencies between time requirements in the design with respect to values of the cycle period. In our example, our constrained design reveals a problem if we consider p = 2 as the period, for example. In this case, no obstacle would be detected and no move or stop operation would be called except every 2 time units. First, this needs to be justified by an (informal) argument that the sensor is powerful enough to detect any obstacle that could be in the distance travelled in 2 time units. Moreover, with PI = 45 and av = 15, the transition from Turning with guard since(PI/av) is equivalent to a wait(3). After 3 time units, the transition must be taken and move(lv,0) called. With 2 as the cycle period, however, that might be during a cycle of the simulation, when nothing happens. We, therefore, have an inconsistency in the simulation due to lack of response.

Formally, with 2 as the period, TA3 engages in two consecutive tock events in each cycle. So, when wait(3) finishes, it may be the case that the simulation is in the middle of a cycle. The RoboChart model requires an immediate call move(lv,0), but this is modelled by an external event and cannot take place mid-cycle because of A3. So their parallel execution deadlocks. This indicates that the RoboChart model cannot be scheduled with a cycle period 2.

Besides 1, the only other valid cycle period for our running example is 3 when PI = 45 and av = 15. In large examples, schedulability analysis can be extremely subtle. So, a way of checking it via a classical property (deadlock freedom) is extremely convenient. We note, however, that verification of the consistency of a RoboSim model with respect

to a RoboChart model does not require proof of schedulability. The check of consistency can also flag the schedulability problem. The advantage of the deadlock check is that it is cheaper, and, for scalability, it can be easily automated with proof techniques or handled using techniques such as those in [?]. Moreover, if it fails, the source of the problem is clearly an issue with the RoboChart design, rather than any issue in the RoboSim model.

Soundness We verify soundness of a RoboSim model by checking whether it refines the constrained RoboChart model in the failures-divergences semantics of CSP. This ensures that the traces, deadlocks, livelocks, and timings of the simulation are all possible for the RoboChart model as well. RoboChart external events are hidden, as the relevant events for comparison are those that represent register reading and writing: registerRead and registerWrite. So, our specification is not just the RoboChart model as it is; on the other hand, the correctness verification has at its heart the standard notion of refinement in the canonical semantics of CSP.

For our example, as mentioned, both simulations in Figures 2.3 and 2.4 are sound with respect to the design model in RoboChart. In both cases, soundness can be mechanically checked by the following FDR assertion, where `[FD=` is failures-divergences refinement, and `SimCFootBot` is the process $[\![\mathsf{SimCFootBot}]\!]_{\mathcal{M}}$ for the relevant module.

```
assert PConstrained \ ExternalEvents  [FD= SimCFootBot
```

It is worthwhile exploring examples of mistakes in simulations we can uncover. We focus on scenarios where a RoboSim simulation fails to refine the RoboChart model constrained by the assumptions, that is, the assertion above fails. For that, we consider variations of the model in Figure 2.3.

As a first example, we miss the entry action of the state SMoving (see Figure 5.4). The assertion fails and a counterexample shows that the simulation is unable to call move as required by the RoboChart model.

As another example, if the exec event is not included as trigger in the two transitions from DMoving, this allows the simulation to call move twice in the same cycle. A counterexample shows the two calls.

Another category of mistakes involves the erroneous use of guards. Considering the same transitions, if the designer misses the guard in the transition that leads to STurning (as

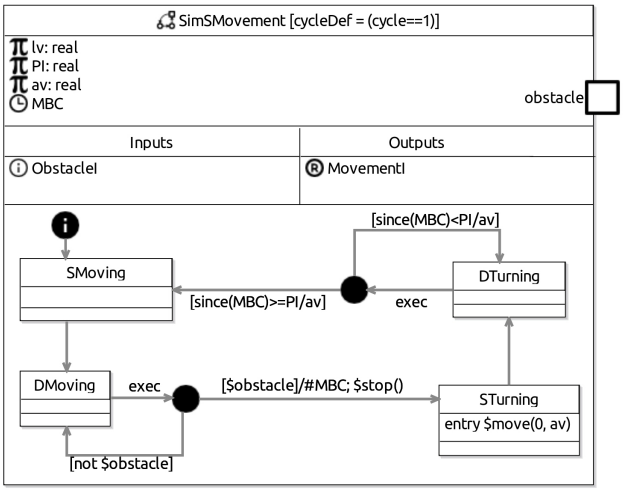

Figure 5.4: Simulation machine: missing entry action for state SMoving

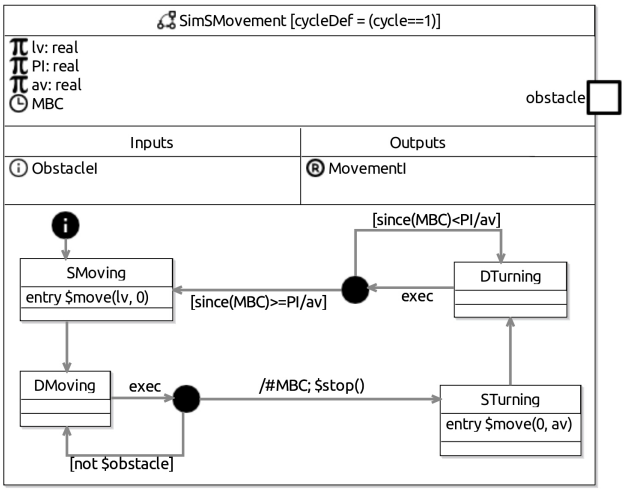

Figure 5.5: Simulation machine: missing guard to state STurning

shown in Figure 5.5), or swaps the two guards (see Figure 5.6), these are also detected by the consistency checks.

The more subtle situation where a simulation might ignore the occurrence of a RoboChart event in a state where the RoboChart model necessarily handles this event is also caught. The inclusion of an extra transition back to the state DMoving triggered by the exec event and with no guards (which is the same as a [true] guard) is an example; see Figure 5.7. In this case, when a new cycle starts and an obstacle is detected, if the machine is in the state DMoving, there is a nondeterminism between this new transition and the one that

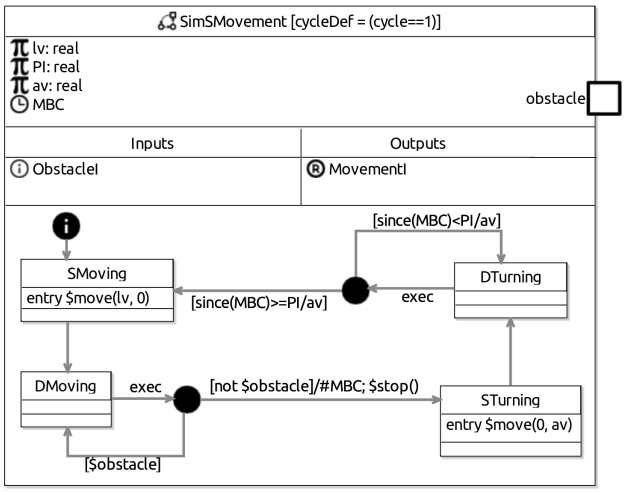

Figure 5.6: Simulation machine: swapping guards in transitions from DMoving

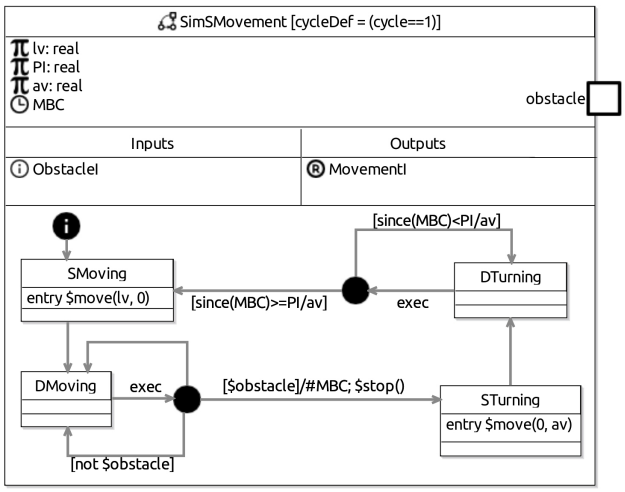

Figure 5.7: Simulation machine: possibly ignoring an obstacle occurrence

leads to STurning. So the occurrence of an obstacle can be ignored, which is disallowed by the RoboChart machine when it is in the Moving state (see Figure 2.1).

Finally, we consider another subtle mistake where the wait(1) construct in the RoboChart model (see Figure 2.1) is not properly simulated. In the simulation in Figure 5.8, instead of a junction with a single exec event from the state DMoving, as in the correct simulation (Figure 2.3), two transitions are used with exec guarded by the respective disjoint conditions on the occurrence (or the absence) of an obstacle. However, the guards are

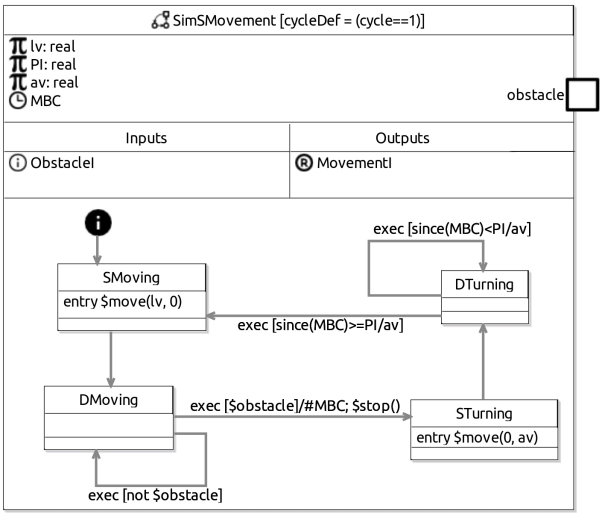

Figure 5.8: Simulation machine: wrong cycle allocation

checked in the current cycle, whereas the wait(1) in the RoboChart model imposes that these should be checked in the next cycle.

In summary, like the analysis of schedulability, soundness check of simulations with respect to design models can be very elaborate. Our reasoning framework allows (automatic) verification that can reveal subtle problems.

Chapter 6

Examples

We present below a few case studies. The first is a simple robot that performs a square trajectory, which shows how composite states, applications that terminate, and deadlines may be handled. A Transporter used in a swarm application, specified in [?], illustrates simulations with during actions. The Alpha Algorithm from [5] has multiple machines that can be independently simulated. Each of these examples is briefly discussed below, where we explore distinctive aspects of the simulation and conformance of the simulation model with respect to the RoboChart model.

6.1 Square

This case study involves a robot that performs a square trajectory, trying to avoid possible obstacles.

6.1.1 RoboChart model

Here, we consider the RoboChart model from [16], reproduced here in Figure 6.1.

The initial transition leads to the state MovingForward; in this transition, the clock C is reset and the variable segment is initialised with 0; this variable is used to control the number of sides of the square that have been currently traversed. The execution flow terminates (reaches a final state) when the robot has covered all the sides of the square (segment == 4), in which case it communicates the event stop with deadline 0.

Since MovingForward is a composite state, entering in this state means executing its entry action (moveForward with the constant parameter linear, for a linear motion) as well as the entry action of the state Observing, namely, a call to enableCollisionDetection, as there is an initial transition to this substate.

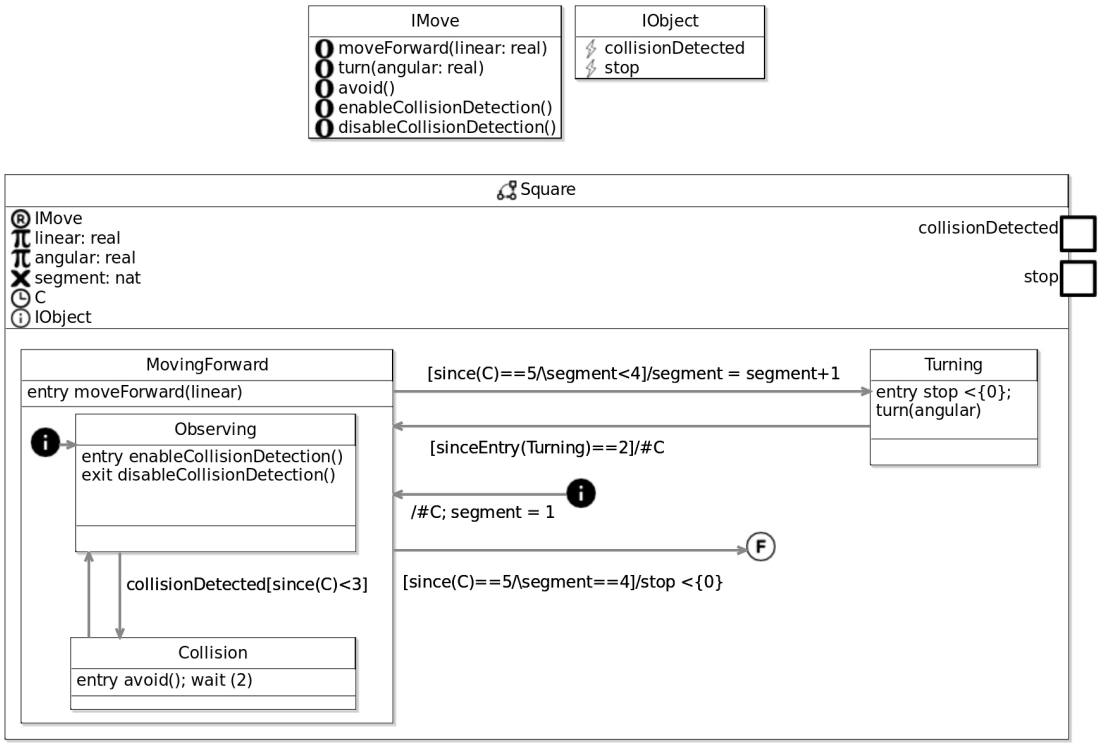

Figure 6.1: RoboChart: square trajectory state machine. The module contains a single controller, which contains this machine.

The internal transition inside the state MovingForward happens when a collision is detected and the time guard since(C)<3 is satisfied, in which case the control moves to state Collision. This transition causes the exit action of Observing to be executed, and then the entry action of Collision. After the collision has been handled, an immediate transition leads the control back to the substate Observing.

A transition can also be taken from the composite state MovingForward to Turning, when 5 time units have passed; in this case the value of segment is incremented and the robot starts an angular movement. This is interrupted after 2 time units, and the control goes back to MovingForward, when the clock is reset.

6.1.2 RoboSim model

A simulation model with a cycle value 1 is presented in Figure 6.2. The structure of the simulation could have mimicked that of the RoboChart model, since RoboSim also

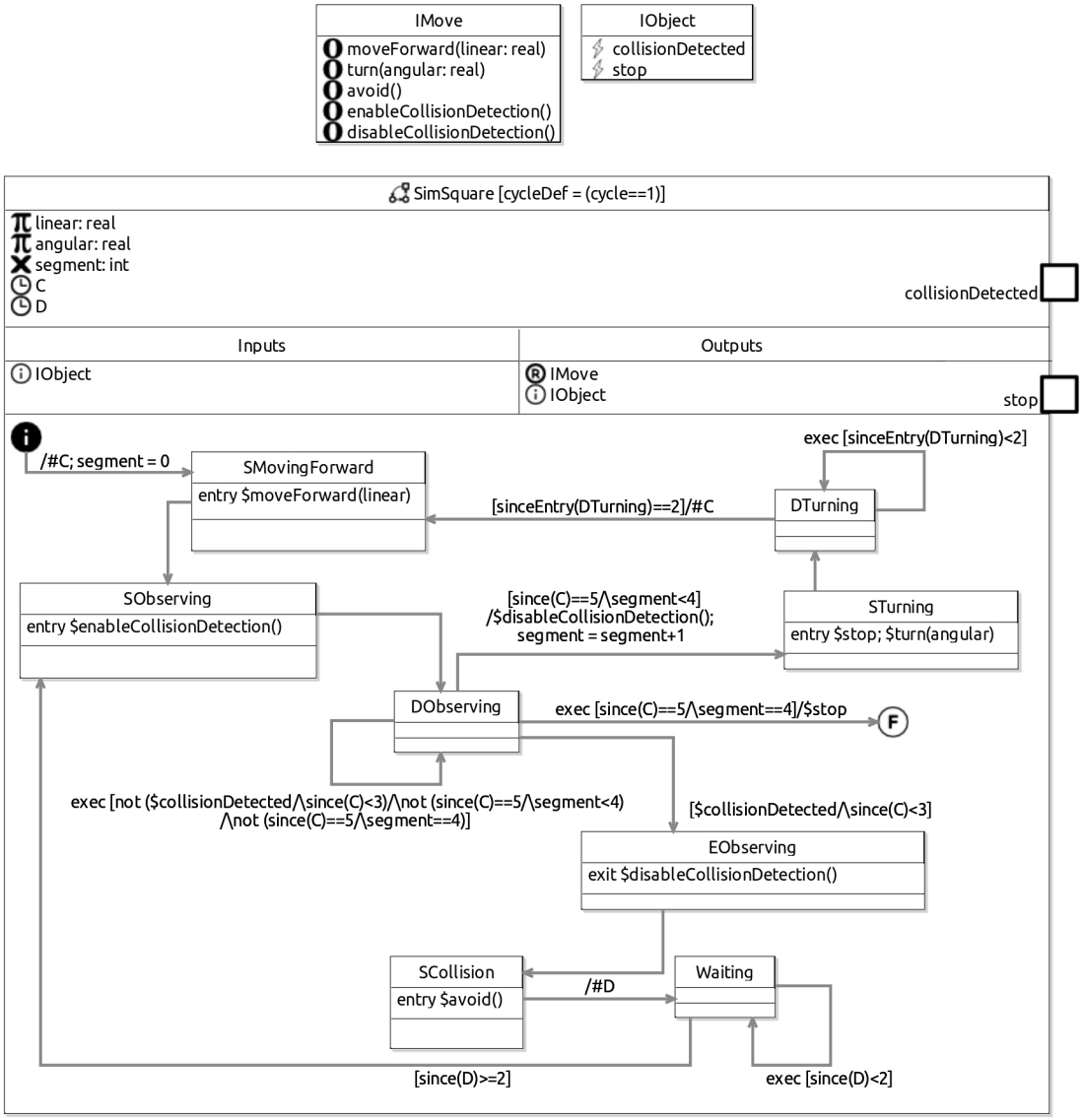

Figure 6.2: RoboSim: square trajectory state machine

allows composite states. Instead, this simulation model linearises the RoboChart machine, including only simple states. This might serve as a more concrete reference for an implementation, for instance. In the following section, we illustrate the use of composite states in the simulation.

The actions from the initial state up to the state SObserving are similar to those in the RoboChart model. We recall, nevertheless, that RoboSim does not allow deadlines.

Deadline constraints do not impact the RoboSim model itself, but rather they may require the definition of explicit assumptions on the environment to ensure that such constraints are satisfied. After explaining the RoboSim model we further discuss the impact of deadlines in verification.

From the state DObserving, a number of transitions can be taken. If a collision is detected within the allowed amount of time, the control flow transitions to EObserving and, then, immediately, to SCollision and to Waiting. This latter state allows two new cycles without any effect, except for time to pass, representing the wait(2) action in the RoboChart model, and then the control is transferred back to SObserving. Another transition from DObserving is the one to state STurning, which immediately transitions to DTurning, from which one transition allows to return to SMovingForward and another self-transition allows the change of cycle. Finally, a self-transition from DObserving has disjoint conditions to the other two and allows the change of cycle.

There is a deadline 0 on the event stop in the transition from MovingForward to the final state. The simulation ensures that the relevant stop action is executed in the current cycle by disallowing any transition triggered by exec.

This model is a provably correct simulation with respect to the RoboChart model in Figure 6.1. In this case, no special treatment of the deadline is necessary for the verification. In general, however, particularly when deadlines are imposed on input events, it is necessary to define an environment assumption that ensures that the deadline constraints are satisfied, namely, that the environment provides the input to satisfy the required deadline.

The asymmetry between inputs and outputs is because a deadline is an assumption on the environment handled differently in RoboChart and RoboSim. In RoboChart, both inputs and outputs require agreement of the environment to provide the input or accept the output. In RoboSim, the environment is always ready to accept an output via registerWrite, as reflected in the model by buffering the outputs. On the other hand, the environment must provide the inputs: we can always read an input via registerRead, but it may not be ready as indicated by the boolean false.

In this case study, we have explored some variations of the RoboChart model. For example, we have considered different time constraints in the since and sinceEntry guards in the RoboChart model, replacing 3 and 5 with 8 and 12 time units. We have also replaced wait(2) in state Collision with wait(4) and the guard SinceEntry(2) with SinceEntry(4). In

this case, we obtain a model that can be scheduled with a cycle at most 4, which is the greatest common divisor of 4, 8 and 12. We have checked that cycle periods of size 1 and 2 are also valid for that scenario.

As another variation, we have introduced nondeterminism in the RoboChart model, replacing the wait(2) action in the entry action of Collision with the nondeterministic action wait([2..4]), which defines a more flexible budget, with 2, 3 or 4 time units. The nondeterministic choice here determines the length of the cycles that are schedulable. For instance, for wait(3), a cycle of size 2 is not schedulable, as we need a common divisor of the time restrictions.

For these variations, we have also developed conforming simulation models.

6.2 Transporter

The transporter system proposes use of a swarm of small robots for moving an object to a specified goal. The robots move the object by pushing it.

The swarm is a fully distributed system: it involves no interaction among the robots. Our models capture the behaviour of a single robot; the swarm is formed of a number of robots with the behaviour specified here.

6.2.1 RoboChart model

Here we describe the RoboChart state machine of the Transporter system (see Figure 6.3).

In state `Searching` the robot searches for an object by calling `searchObject` and, once it sees one, it transitions to state `MovingToObject` as triggered by the event `objectSeen`, which receives and assigns the estimated distance to variable `distance`. If the object is nearby, so the condition `distance<close` holds, and the object has been seen for less than `Ta` time units, then the state-machine transitions to state `ClosingInOnObject`. While in states `MovingToObject` and `ClosingInOnObject` if the object is lost for a certain amount of time `Ta`, then the robot initiates another search for the object by transitioning to state `Searching`.

In states `MovingToObject` and `ClosingInOnObject` we have the possibility to keep treating the event `objectSeen` and updating the current `distance`, as long as, for state

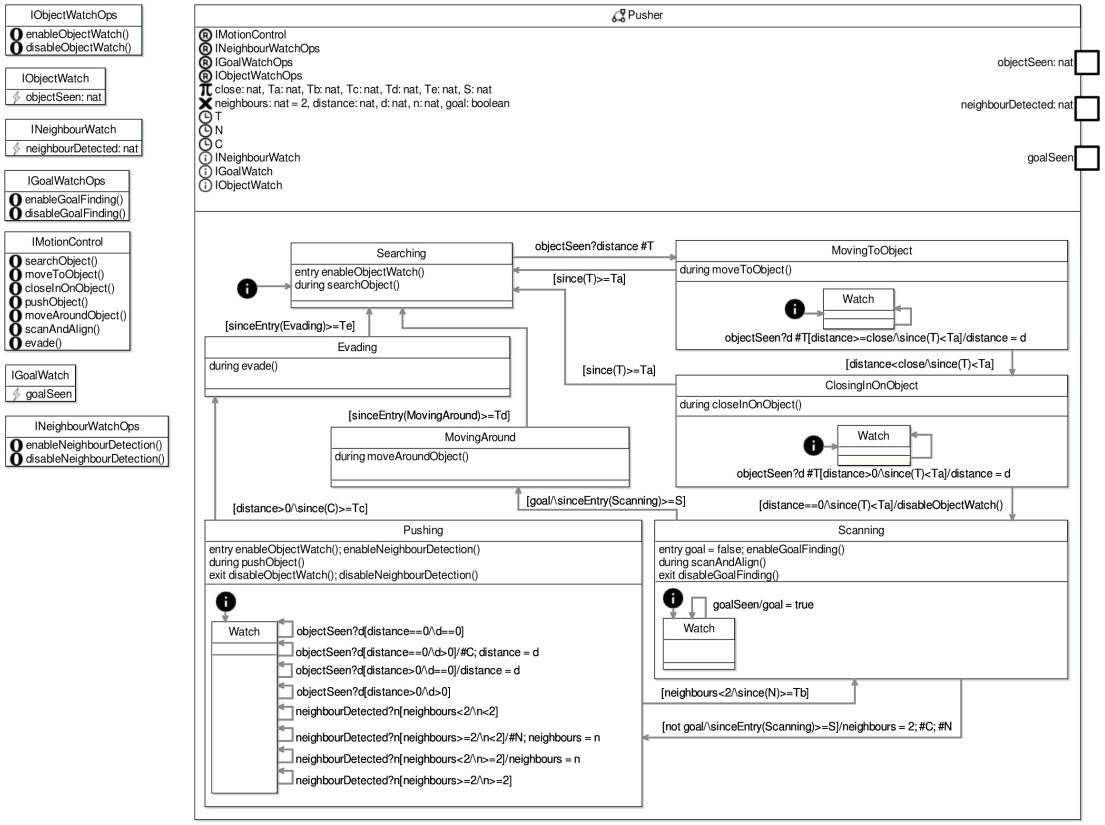

Figure 6.3: RoboChart state machine of the Transporter

`MovingToObject` the object has been last seen for less than `Ta` time units, and the distance is greater than or equal to `close`, and similarly for state `ClosingInObObject` the distance is greater than `0`.

When the robot is close enough to the object, satisfying condition `distance == 0`, then it transitions from state `ClosingInOnObject` to `Scanning` and disables the object tracking facility by calling `disableObjectWatch`. In state `Scanning` the robot initialized the variable `goal` to `false`, assuming that the goal is not visible until the first time event `goalSeen` takes place, and enables the capability to find the goal by calling `enableGoalFinding`.

Once the state `Scanning` has been entered, `scanAndAlign` is called, so that the robot can perform an alignment procedure, and a decision is taken `S` time units after entering the state `Scanning`, to either transition to `MovingAround` if the goal is seen, or to transition to `Pushing` if the goal is not seen. When taking this transition we initialize the variable

neighbours, which keeps track of the number of nearby neighbours, to 2. This prevents the transition from Pushing to Scanning from being triggered unless Tb time units have passed since the most recent time, after having entered Pushing, where the number of neighbours was observed to drop below 2, which can only be appropriately checked once at least one reading of neighbourDetected has occurred. In state Pushing the robot first calls enabledObjectWatch to enable the facility to track the object, followed by the call to enableNeighbourDetection to enable the facility to track neighbours. Once state Pushing is entered it finally calls the operation pushObject and can receive events objectSeen and neighbourDetected.

The transition from Pushing to Evading takes place whenever the object has been lost for at least Tc time units. We specify this behaviour by taking into account the distance to the object as given by objectSeen together with clock C. Whenever the distance changes to being greater than 0, we reset clock C. The four possible cases are treated by separate transitions whose guards are summarized below:

- objectSeen?d[distance==0/\d==0]: the object is seen at distance 0, so there is no need to change the existing value of distance.
- objectSeen?d[distance>0/\d==0]/distance=d: if the current distance is greater than 0, and the new distance is 0, then we just record the new value d in distance.
- objectSeen?d[distance>0/\d>0]: in every other case, we do not update the value of distance.

The transition from Pushing to Scanning takes place whenever less than 2 neighbours have been observed for at least Tb time units. We specify this behaviour using the variable neighbours together with clock N, so that whenever the value drops below 2 we count the time using clock N. We specify this using four transitions, whose guards are summarized below:

- neighbourDetected?n[neighbours<2/\n<2]: the current count of neighbours is less than 2 and the new value being communicated is also less than 2, so there is no need to record a change.
- neighbourDetected?n[neighbours<2/\n>=2]/neighbours=n: the current count of neighbours is less than 2 and the new value n is greater than or equal to 2, so

there is a change to a situation in which at least 2 neighbours are around, so we updated the value of `neighbours` to `n`.

- `neighbourDetected?n[neighbours>=2/\n>=2]`: in every other case in which the count of current neighbours or the new value `n` is above 2 we do not update the variable `neighbours`.

Before exiting state `Pushing` the capability to track the object and the neighbours is disabled by calling `disableObjectWatch` and `disableNeighbourDetection`, respectively.

In general, the value of a `since(C)` condition for a clock `C` that has not been reset is arbitrary. We observe that in the case of the guard `distance>0/\since(C)>=Tc`, `since(C)>=Tc` has an appropriate value when `distance>0` is true because clock `C` is reset precisely when the value of distance becomes greater than 0. Similarly in the case of the guard `neighbours<2/\since(N)>=Tb`, `since(N)>=Tb` has an appropriate value when `neighbours<2` is true because clock `N` is reset precisely when the number of `neighbours` goes below 2 as explained above.

In state `Evading` the robot evades the vicinity by calling the operation `evade`. After exactly `Te` time units it transitions to state `Searching`. Similarly, in state `MovingAround` the robot calls `moveAroundObject` which takes the robot around the object. After exactly `Td` time units it transitions to state `Searching`.

We observe that the state-machine, and indeed the robotic controller, never terminates. This follows closely the description of the algorithm in the original paper. However in the reported trials, robots could be stopped by a central controller that can ascertain whether the object has been made contact with the goal, or after a certain amount of time. We do not model that facility in this model.

6.2.2 RoboSim model

Figure 6.4 shows the RoboSim model of the algorithm, adding details about the operational cycle, which refines our RoboChart model[1].

[1]The interested reader can reproduce the verification of the simulation model. All CSP scripts are available at goo.gl/1so4uK.

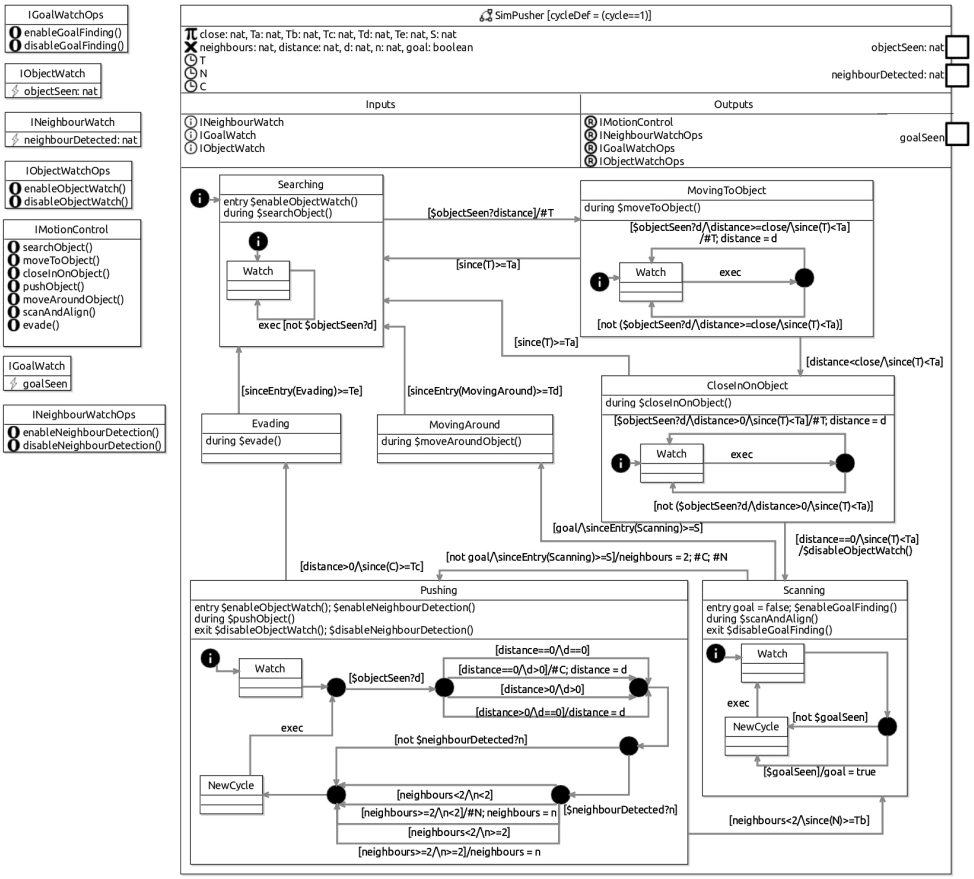

Figure 6.4: RoboSim model of the Transporter

Rather than discussing the model of the simulation in detail, we focus here on some distinctive aspects. We note, for instance, the use of composite states as a way to model behaviour in the presence of during actions in the design. For example, the model of the state MovingToObject is very similar in the RoboChart and RoboSim machines. In RoboSim, however, an additional transition with exec as trigger allows the change of cycle when no object is detected.

As another example, in the states Scanning and Pushing we observe the use of transitions in the RoboSim model, to and from junctions, when contrasted with the several self-transitions in the respective states in the RoboChart model. This is a consequence of the need to control that the sensor readings for detecting the presence of an event must not be checked more than once in the same cycle. For instance, if $goalseen (in state Searching) is checked to set the variable goal accordingly, which itself is used to enable the transitions

from this state, then it does not make sense to allow $goalseen to be checked more than once in a same cycle. Another point we would like to emphasise is that the RoboChart model of the transporter is nondeterministic, and the simulation is deterministic. this case we have a refinement, not an equality between the RoboChart and RoboSim models.

6.3 Alpha Algorithm

Our third case study is the model of a single robot in a swarm acting under the Alpha Algorithm [5]. This involves two machines: one specifying the robot movement, and the other, the communication with the other robots in the swarm.

6.3.1 RoboChart model

Figure 6.5 presents the RoboChart model of the more elaborate Movement machine, which is our focus here. A point of note is that, due to the compositional nature of our semantics, we can verify each machine separately.

The initial transition leads to the state MovementAndAvoidance. Here, a clock MBC is reset. Since the state MovementAndAvoidance is composite, entering it means executing the entry action of the state Move, namely, move(lv,0); this operation is similar to the homonymous ones of our running example and the Square example in Section 6.1, where the first parameter indicates the linear motion of the robot, and the second one the angular motion.

The transition from Move to Avoid happens when an obstacle is detected (represented by the occurrence of the event obstacle) and no more than a certain amount of time, given by (MB-360/av), has passed. Once in Avoid, its entry action is executed, and a period of time must pass (as stated in the wait action) before returning to Move.

A transition can also be taken from MovementAndAvoidance to Turning if some amount of time equal or greater than MB time units has passed since the last clock reset. In this case, the clock MBC is reset. This means that the robot is required to turn after every period of MB time units. Afterwards, the control goes back to MovementAndAvoidance.

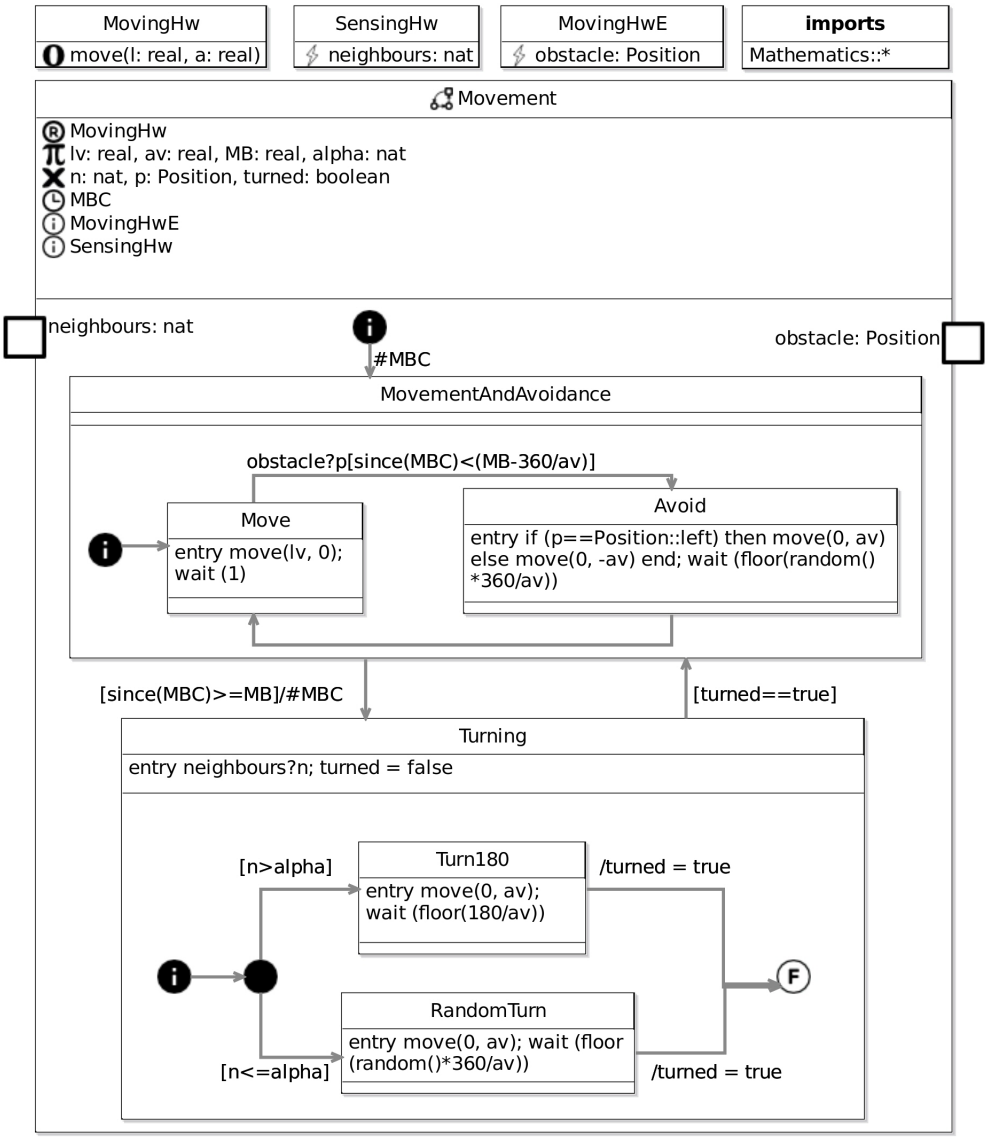

Figure 6.5: RoboChart: Alpha Algorithm

6.3.2 RoboSim model

The simulation model for this example is presented in Figure 6.6. The simulation model for this example is presented in Figure 6.6. Its structure is similar to that of the RoboChart model, but it is also possible to notice some differences. As usual, we have included an extra state (here, DMove) in the composite state MovementAndAvoidance and an initial state in Turning to control the cycle scheduling. In between the states

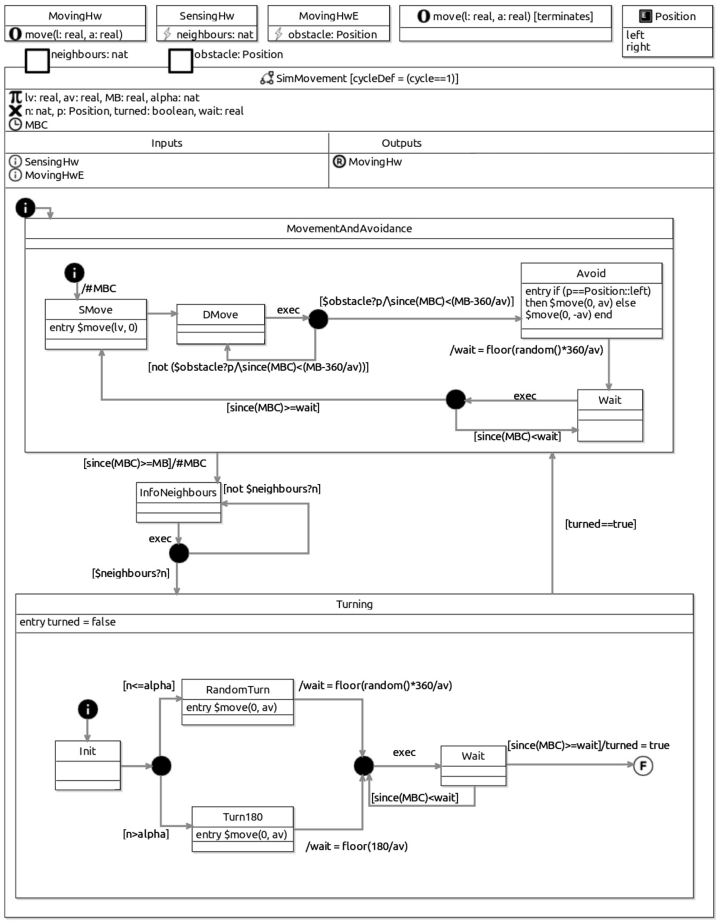

Figure 6.6: Simulation machine of the Alpha Algorithm in RoboSim

MovementAndAvoidance and Turning we have included the new state InfoNeighbours. This is necessary because, in RoboSim, input communications, such as $neighbours?n, are part of guards, and cannot happen in actions, since they are boolean expressions (possibly associated with an assignment to variables). We note that in the RoboChart model in Figure 6.5, the communication neighbours?n is a statement in the entry action of the state Turning. Finally, as already explained, the wait action is not available in RoboSim, so the occurrences of wait in the states Avoid, RandomTurn and Turn180 are captured by counting cycles in the two Wait states.

We have proved that the simulation model is a correct implementation of the RoboChart model of the Movement machine[2]. During this verification, we have detected a schedu-

[2]The interested reader can reproduce the verification of the simulation model. All CSP scripts are available at goo.gl/fp8j2G.

lability issue in the RoboChart model. To avoid that two conflicting move operations happen in the same cycle, the wait(1) action in Move (see Figure 6.5) had to be included. It can be difficult to get these right without proper verification support, as we are proposing here.

Chapter 7

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 [3], a process algebra that combines Z [22] and CSP, and includes time constructs [18]. 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 [23], 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 [8] to extend the notation, and from the UTP model of continuous variables [6] to define the semantics.

Appendix A

Complete Metamodel

This appendix contains the metamodel for RoboSim as an extension of that for RoboChart. It is specified in Ecore and formatted by the tool `OCLinEcore`. The syntax of the representation used in this appendix is available here.

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

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;
import robochart : 'http://www.robocalc.circus/text/RoboChart' ;

/*
* Controller
*/
class ControllerDef extends robochart::ControllerDef
{
  property cycleDef : CycleDec[1]
}
class CycleDec extends robochart::NamedExpression
{
  property cycle : robochart::Variable[1] { composes };
  property cycleExp : robochart::Expression[1] { composes };
}
/*
* State Machine
*/
class StateMachineDef extends robochart::StateMachineDef
{
  property cycleDef : CycleDec[1]
  property inputContext : Context[?]
  property outputContext : Context[?]
}
/*
* Statement
*/
class PlatformCall extends robochart::Statement
{
```

```
  property operation : robochart::Operation[1];
  property args : robochart::Expression[*|1] { ordered composes };
}
/*
* Expression
*/
class InputCommunication extends robochart::Expression
{
  property event : robochart::Event[1];
  property variable : robochart::Variable[?];
  property predicate : robochart::Expression[?] { composes };
  property value : robochart::Expression[?] { composes };
}
class PlatformVariable extends robochart::Expression
{
  property variable : robochart::Variable[1];
}
/*
* Module
*/
class Module extends robochart::Module
{
  property cycleDef : CycleDec[1]
}
```

Bibliography

[1] R. H. Bordini, J. F. Hübner, and M. Wooldridge. Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons, 2007.

[2] S. G. Brunner, F. Steinmetz, R. Belder, and A. Domel. Rafcon: A graphical tool for engineering complex, robotic tasks. In IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3283–3290, 2016.

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

[4] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. Simulation, Modeling, and Programming for Autonomous Robots, chapter RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pages 149–160. Springer, 2012.

[5] C. Dixon, A. F. T. Winfield, M. Fisher, and C. Zeng. Towards temporal verification of swarm robotic systems. Robotics and Autonomous Systems, 60(11):1429–1441, 2012.

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

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

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

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

[10] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multiagent simulation environment. Simulation, 81(7):517–527, 2005.

[11] The MathWorks,Inc. Simulink. www.mathworks.com/products/simulink.

[12] The MathWorks,Inc. Stateflow and Stateflow Coder 7 User's Guide. www.mathworks.com/products.

[13] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3869–3876, 2017.

[14] I. Pembeci, H. Nilsson, and G. Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pages 168–179. ACM, 2002.

[15] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. Swarm Intelligence, 6(4):271–295, 2012.

[16] P. Ribeiro, A. Miyazawa, W. Li, A. L. C. Cavalcanti, and J. Timmis. Modelling and verification of timed robotic controllers. In N. Polikarpova and S. Schneider, editors, Integrated Formal Methods, pages 18–33. Springer, 2017.

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

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

[19] University of York. RoboChart Reference Manual. www.cs.york.ac.uk/circus/RoboCalc/robotool/.

[20] M. Wachter, S. Ottenhaus, M. Krohnert, , N. Vahrenkamp, and T. Asfour. The armarx statechart concept: Graphical programing of robot behavior. Frontiers in Robotics and AI, 3:33, 2016.

[21] U. Wilensky and W. Rand. An Introduction to Agent-Based Modeling – Modeling Natural, Social, and Engineered Complex Systems with NetLogo. MIT Press, 2015.

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

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

Index of Semantic Rules

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

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.