Verified Simulation for Robotics

Ana Cavalcanti[a,c,*], Augusto Sampaio[b,d,*], Alvaro Miyazawa[c], Pedro Ribeiro[c],
Madiel Conserva Filho[d], André Didier[d], Wei Li[e], Jon Timmis[e]

[a]*Ana.Cavalcanti@york.ac.uk*
[b]*acas@cin.ufpe.br*
[c]*Department of Computer Science, University of York, UK*
[d]*Centro de Informática, Universidade Federal de Pernambuco, Brazil*
[e]*Department of Electronic Engineering, University of York, UK*

Abstract

Simulation is a favoured technique for analysis of robotic systems. Currently, however, simulations are programmed in an ad hoc way, for specific simulators, using either proprietary languages or general languages like C or C++. Even when a higher-level language is used, no clear relation between the simulation and a design model is established. We describe a tool-independent notation called RoboSim, designed specifically for modelling of (verified) simulations. We describe the syntax, well-formedness conditions, and semantics of RoboSim. We also show how we can use RoboSim models to check if a simulation is consistent with a functional design written in a UML-like notation akin to those often used by practitioners on an informal basis. We show how to check whether the design enables a feasible scheduling of behaviours in cycles as needed for a simulation, and formalise implicit assumptions routinely made when programming simulations. We develop a running example and three additional case studies to illustrate RoboSim and the proposed verification techniques. Tool support is also briefly discussed. Our results enable the description of simulations using tool-independent diagrammatic models amenable to verification and automatic generation of code.

Keywords: state machines, process algebra, CSP, semantics, refinement

1. Introduction

Current practice in software development for robotic systems is often based on standard state machines [11, 36, 6, 49], without formal semantics, to specify the robot controller only. A state machine guides the development of a simulation, but no rigorous connection between the machine and the simulation code is established.

There are several well established simulators for robotics [34, 40, 37, 28], each with its own (often proprietary) language, and specific API to support simulation of robotic platforms and the environment, as well of the software controller. There is no portability, and no traceability to design models given by state machines; support focusses on the design of the API, execution of simulations, and generation of code for deployment in several platforms.

Our goal is to enrich the practice of software engineering in robotics via the design and implementation of a diagrammatic domain-specific notation for modelling and verification of simulations. With this notation, we can establish mathematically a relation between a design and a simulation. We can also define simulation models that are tool independent, but are detailed enough to allow automatic generation of simulation code.[C1,C18]

We present our notation, namely, RoboSim: a language to model simulations of robotic systems by state machines combined to define concurrent and distributed designs that use specified services of a platform. As a diagrammatic notation, RoboSim is more appealing than programming languages and yet akin to notations in current use by practitioners. Distinctively, a RoboSim model can be verified against a UML-like design of a controller. Our vision is the future use of RoboSim as an intermediate, tool and implementation independent, notation to describe verified simulations that can be automatically translated into code for use with standard robotic simulators.

*Corresponding author.

Feature	RoboChart	RoboSim
Events	Synchronous interrupt communications	Boolean variables
Control flow	Determined by events	Determined by cycle periods
Operation executions	At the point of call	At the sample times
State machine	Reactive behaviour	Data model
Interaction with environment	Via events	Via data in registers
Time budgets and deadlines	Associated with events, states, and operations	Defined by cycle period

C2,C4,C18

Table 1: RoboChart versus RoboSim state machines

With RoboSim, we bridge the gap between the state-machine modelling and simulation paradigms. For modelling, machines typically use events as interrupts in triggers, time annotations are nonexistent, or based on budgets and deadlines for triggers and operations, and there is no explicit notion of cycle. In a state machine for simulation, events are asynchronous, and behaviour evolution is characterised by sample times that define a cyclic pattern. RoboSim follows the simulation paradigm, but has a semantics unified with that of RoboChart [31], a UML-like design language. So, RoboSim can be used to verify consistency between designs and simulations.

Table 1 summarises the key differences between the modelling paradigms adopted by RoboChart and by RoboSim. RoboChart supports the definition of more abstract design models where interaction is captured by events that define the flow of executions. Operations are executed at the point of call, and can depend on the occurrence of events and raise events themselves. Time properties are defined by budgets and deadlines on events, states, and operations.

In contrast, RoboSim supports the definition of cyclic models for simulation, whose evolution in time is determined by a period that defines the length of the cycle. Sample times occur at the start of each cycle. Interaction with the environment is via reading and writing to registers representing sensors and actuators. Events are variables that record the values of the registers. Operations are called at the sample times, and can update registers. As a modelling language, however, RoboSim still allows the definition of nondeterministic models. So, a RoboSim model may define several correct simulations that eliminate some or all of that nondeterminism.[C2,C5]

In this paper, we describe the RoboSim metamodel and well-formedness conditions, and give a formal account of its semantics in CSP [41]. This is a process algebra for refinement with well established tools [19, 45]. Here, we use CSP as a front-end to a predicative relational semantics in the UTP (Unifying Theories of Programming) [20]. Since the UTP includes a CSP theory, it is possible to encode our CSP semantics in the UTP, and in this way obtain support for theorem proving using the powerful prover Isabelle/HOL [15]. For early validation of our semantics and verification approach, however, we have used the model checker FDR [19]. To deal with time primitives and properties, we use tock-CSP, an approach to modelling time in CSP [43, 41] used to give semantics to RoboChart.

The refinement semantics in CSP enables the verification of RoboSim models against a design. Verification, however, is not straightforward. Because of the gap between the design and simulations paradigms, a simulation can be considered correct with respect to a design only if we take into account important assumptions. For example, a cyclic simulation assumes that events do not occur between the sample times. In this paper, we first explain how we can use CSP to check that there is a feasible scheduling of the design into cycles, so that we can write a simulation for it. We then show how to check that a simulation preserves the properties of the design. In doing so, we formalise the implicit informal assumptions made in the development of any simulation.

In more detail, the specification for a simulation is not simply the semantics of the RoboChart model. We need to cater for some of the key differences summarised in Table 1, especially the treatment of events and the definition of the control flow. In addition, we need to record assumptions that need to be made in the development of a simulation. For that, we take as specification the semantics of the RoboChart model in conjunction with a formalisation of a mapping between the RoboChart and the RoboSim events, operations, and variables, and of the cyclic execution.[C3]

There are other domain-specific languages (DSL) for robotics targeted at simulation [22, 11, 24], and general-purpose graphical languages for simulation, notably Simulink/Stateflow [30], a *de facto* standard in the transport industry. Our focus with RoboSim is a customised language with notation and semantics that is vendor and platform independent, with a clear and precise semantics, and that can be used to relate design and simulation models.

The main contribution in this paper is more than just the development of a new language, but a new approach to engineering and verification of sound robotic simulations and, ultimately, applications. The issues that we identify

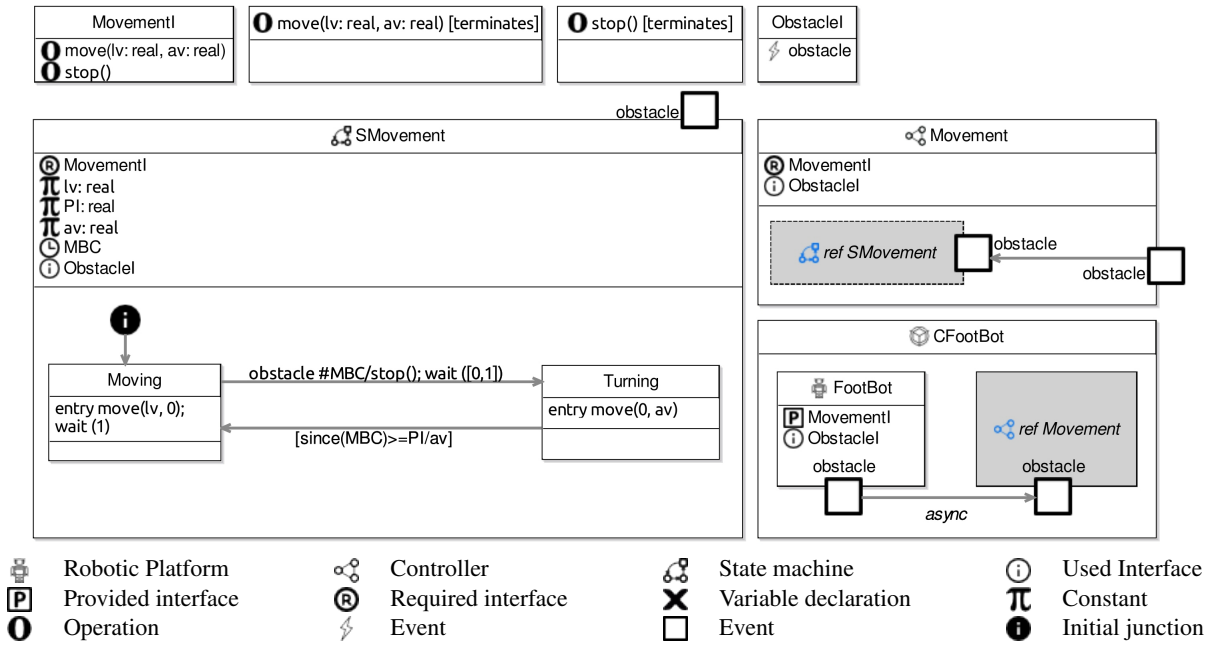

Figure 1: RoboChart: obstacle detection

and address in our work are relevant for comparing any state-machine design with cyclic simulation models. With such a consistency check, we can be sure that problems revealed by the simulation are problems of the design, and not a consequence of programming errors introduced when developing the simulation.

The remainder of this paper is structured as follows. In Section 2 we 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 Section 2 examples that illustrate the need for separate design and simulation notations. In Section 3, we describe other notations for robotics, and discuss further the distinctive features of RoboSim and its verification technique. Section 4 presents the RoboSim metamodel and well-formedness conditions. The semantics is presented in Section 5; at that point, we also describe tock-CSP. Our verification technique is the subject of Section 6. Tool support and examples are discussed in Section 7. In Section 8, we conclude and discuss future work.

2. RoboChart and the need for RoboSim

We have previously presented RoboChart [31], 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 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 1, the interface MovementI has only the operations move(lv,av) and stop(), provided by the robotic platform, 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 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 [31, 39], and in the reference manual [47]. 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.[C9]

RoboChart models are akin to those used in the robotics literature [11, 36, 6, 49]. 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. It normally proceeds indefinitely, with termination 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.

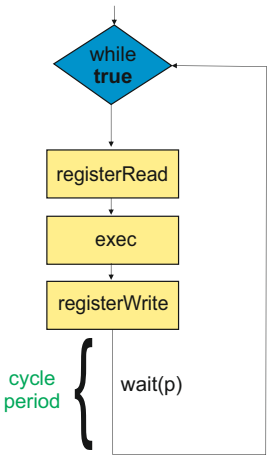

Figure 2: Cycle of a simulation

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.[C11] 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 3. Its overall structure (module and controller) is similar to that of the RoboChart model in Figure 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 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. Similarly, 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). During

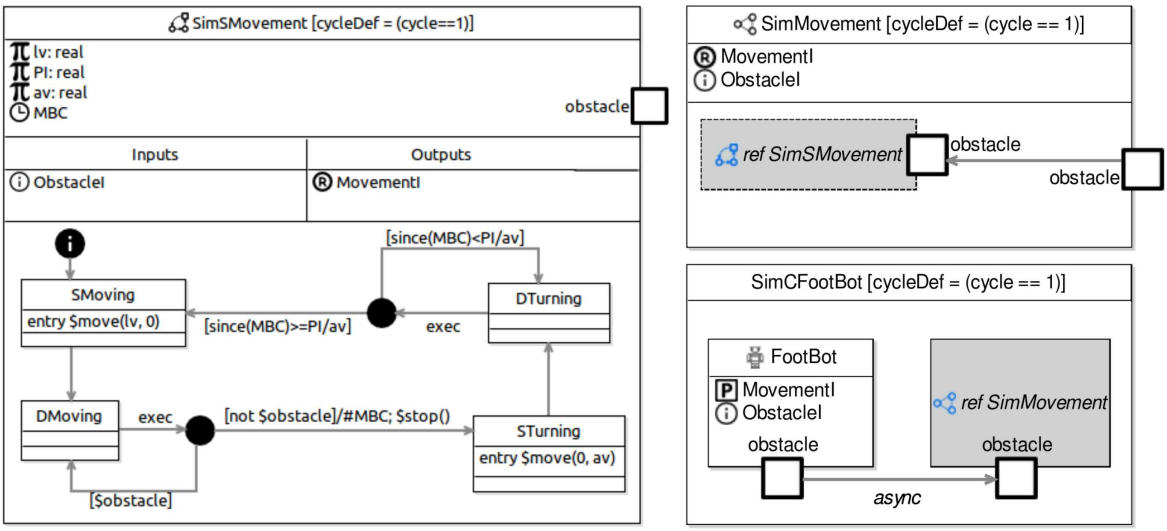

Figure 3: RoboSim: obstacle detection

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 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 1 is presented in Figure 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 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 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

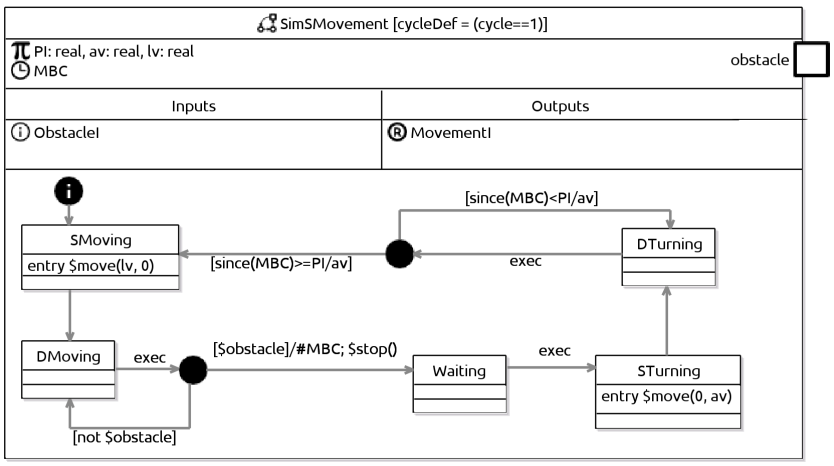

Figure 4: RoboSim: obstacle detection - alternative

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 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 RoboSim. We 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.

What we present here is an approach to check schedulability, and a conformance relation that can be used to justify model transformation or to check if the optimisations or simulations written independently are sound.[C6,C7,C8,C19,C20,C21]

Before presenting RoboSim in detail, we discuss related work.

3. Related work

There are several general-purpose simulation frameworks in widespread use: Simulink and Stateflow [29, 30], 20-sim[1], and Modelica [17] are examples in widespread use. They can, no doubt, be used in robotics. The literature, however, suggests that roboticists often describe state machines using an informal notation [35, 38, 46] before moving to developing a simulation, often by writing optimised code (in C or C++, for instance) for a simulator tailored for robotics. When there are complex control laws involved, the general simulators are useful. This is, however, not often the case in many applications, and the flexibility of code-based simulations makes it possible to develop fast simulations that can be used to run an extensive number of experiments in an effective way. [C22,C23,C24]

There is support for verification for general languages such as C Simulink/Stateflow [30]. Early efforts on verification for robotics apply existing mathematical techniques as they are [22]. A recent comprehensive survey on specification and verification in robotics [13, 27] indicates that model checking is the most popular approach at the moment.[C25] What we present here is a customised approach. Customisation allows us to use a simple language akin to

[1] www.20sim.com/

what is already used by practitioners, and to explore optimisation in the automation of verification. This has potential impact on verification by model checking and theorem proving.

Simulators for robotics vary widely in the language they adopt. Webots [34] and V-REP [40] provide different graphical interfaces. In Webots, models can be programmed using a human-readable customised notation. In V-REP, various general languages can be used. ARGoS [37] and Enki (`home.gna.org/enki/`) are programmed using different C++ libraries. The Microsoft Robotics Developer Studio (`www.microsoft.com/robotics`) provides environments and platforms programmed using VPL or C#. Player/Stage [18] provides a device server, and clients can be programmed in a variety of popular languages. MASON [28] and BREVE [23] adopt the agent paradigm; for programming, BREVE adopts a custom language or Python, and MASON, Java.

None of these simulators adopt a diagrammatic notation to specify simulation code. Moreover, a simulation written for one of them cannot be easily ported to another one. Our vision is that a simulation model written in RoboSim is used for automatic generation of code for such simulators. Work on automatic generation of code for ARGoS is under way. Writing translations from RoboSim for other platforms is future work.

There are many DSL for robotics that are diagrammatic. Several of them might be used to describe models that can be verified using the technique we present here for RoboSim. RoboFlow [5] is a language for programming, not simulation, with an operational semantics. While the existence of a formal semantics is very positive and a clear way to define sound tools, there is no support for reasoning about RoboFlow models in relation to designs at any level.

RobotML [11], rFSM [24], and SmartSoft [44] are DSL for simulation and deployment, but do not have a formal semantics. They provide no support for analysis of models, either in isolation or in relation to designs.

ArmarX and Rafcon are programming languages for robotics based on state machines without formal semantics [49, 6]. Some of their core restrictions, like absence of interlevel transitions, are similar to those of RoboChart and RoboSim, and ultimately can facilitate the provisioning of reasoning facilities like those proposed here.

A pioneering effort in robotics defines Orccad [22], for modelling, simulation, and programming. Translation to two different notations allows verification of timed properties. Instead of using state machines, Orccad models are formed of tasks defined by control laws, combined by procedures defined by reactive programs. RoboChart and RoboSim instead use the more widely accepted approach of state machines. Combined use of RoboChart with control laws is addressed in [8], and that approach, based on modern co-simulation standards [14], can be used for RoboSim.

Verification by model checking is available for the executable deployment language for robotics GenoM [16, 4]. Models are translated for use with the Petri Net model checker TINA. Verification focusses on schedulability and deadlock freedom, while verification of functional properties is possible, but not yet pursued. As opposed to RoboSim, GenoM is an executable language for deployment rather than simulation; models, for example, include C code.

In summary, most of the DSL for robotics currently described in the literature are not associated with a proof technique. A recent survey [32] indicates a rise in interest in model-based engineering, and, therefore, DSL for robotics. The focus, however, has not been on proof of behavioural properties, or linking design and simulation models, in a unified semantic and verification framework, as we do here for RoboChart and RoboSim.[C25]

Compared to Stateflow, for instance, RoboChart and RoboSim are much more restrictive languages. On the other hand, the cycle of a RoboSim model can be more flexibly defined. Particularly, the occurrence of events or the structure of the machine does not implicitly define the behaviour in each cycle. Moreover, in Stateflow, in each cycle, for each event that has occurred, the machine is executed once. RoboSim adopts the approach of reactive simulators, where the machine is executed once, when all events are normally considered. We expect, however, that it is possible to define a pattern for Stateflow models to allow their verification following the RoboSim approach here.

There are also works that do not follow the model-based approach. Notably, there has been interest in runtime verification to avoid the need to construct complex models. The framework in [10] uses a programming language P that can be used to describe state machines. Systematic extensive testing, called model checking in that context, but not guaranteeing full coverage, is used for verification. In addition, temporal logic specifications are checked at runtime to monitor violation of assumptions used for verification. The RoboChart and RoboSim approach can be combined with such frameworks; we note, for example, the use of an informal state machine to describe the example there. Systematic test generation from RoboChart and RoboSim models is part of our agenda for future work.

Runtime verification is also adopted in [21]. Here, the target is ROS (Robotic Operating System) software. A property language deals with safety and security properties, and monitors are automatically generated. Translation from RoboChart and RoboSim models to ROS simulations are currently under development.

A hybrid approach is presented in [42], where Java PathFinder is used to deal with control software implemented in Java using a cyclic executive real-time pattern. To handle the environment, a simulation of a model that uses differential equations to capture the expected behaviour of sensors and actuators is implemented. As usual in robotics, there is no rigorous link between the simulation and the model. RoboChart and RoboSim as yet do not provide support to deal with environment models and assumptions, an aspect we will consider in the future.

In summary, what is distinctive about our work is the fact that we have a language that is small and controlled. The architectural design pattern that it embeds can guide roboticists when developing models. It can be much more difficult if practitioners have open languages like UML or Simulink. In addition, restrictions on the language simplify the semantics and both facilitate and enable verification. Moreover, beyond support for verifying desirable properties of the models, our approach provides a conformance notion for a simulation with respect to a more abstract design model. More than a new notation, we present a modelling and verification approach for simulation of robotic applications that can be useful in the context of all notations based on state machines above.[C25]

4. Metamodel and well-formedness conditions

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. The RoboSim metamodel and the associated well-formedness conditions are presented in Sections 4.1 and 4.2. A formal semantics in tock-CSP is the subject of Section 5.

4.1. Metamodel

RoboSim models use the elements sketched in Figure 5. 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. In Figure 5, we show this feature just for the simulation machines. The complete metamodel, including components and attributes omitted here, is available in [48].

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.[C12] Modules, controllers, and state machines define the valid sample times for their simulation by specifying restrictions on the value of cycle.

The cycle specifications may admit several valid periods. In our simple example (see Figures 3 and 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, omitted in Figure 5, is a reference to a state machine. As illustrated in Figure 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 (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

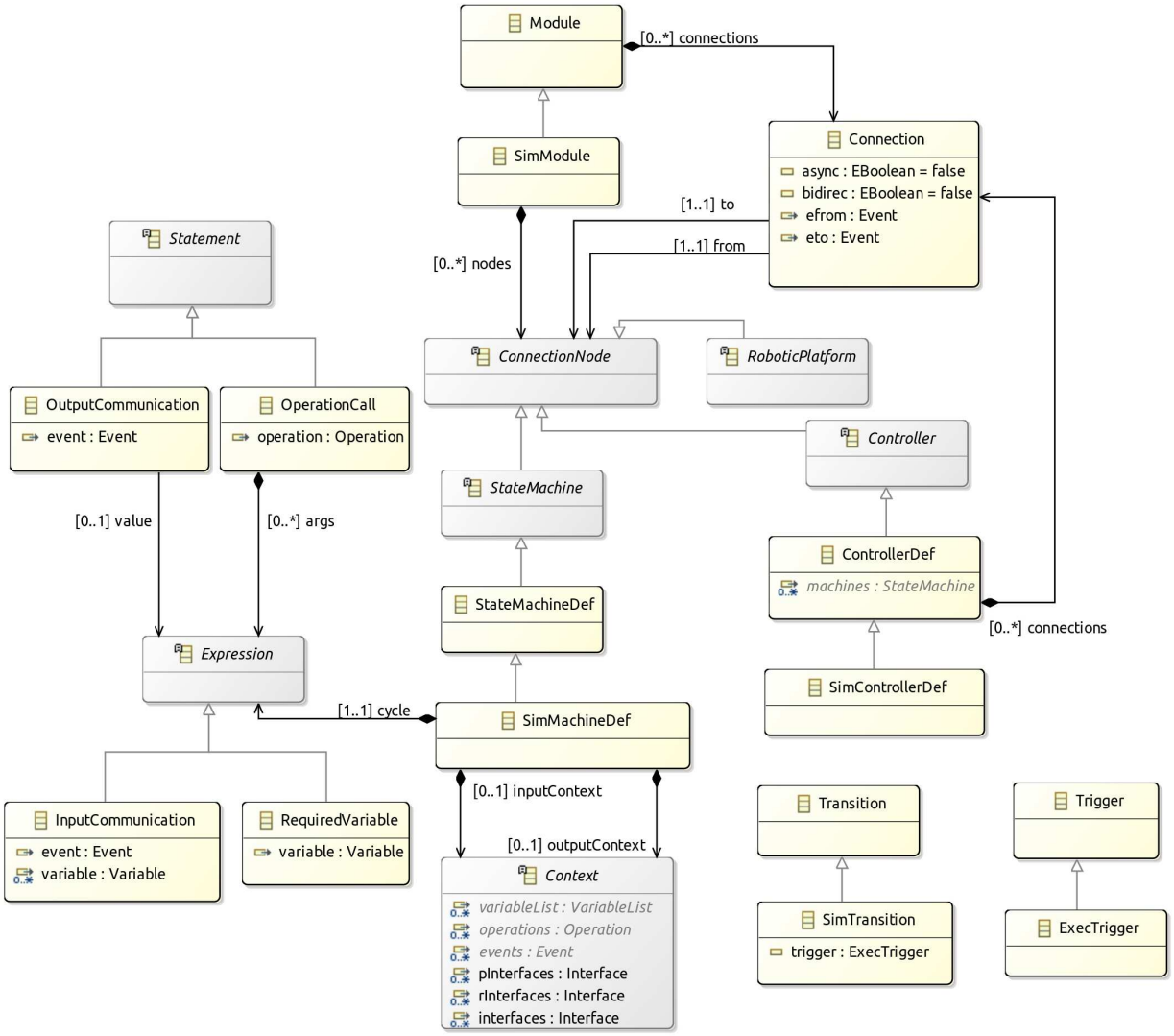

Figure 5: RoboSim metamodel

platform or the software. For example, the occurrence of the event obstacle in RoboChart (see Figure 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 5), 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. RoboSim allows other types of Statement, omitted in Figure 5, 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.

In the next section, we explain the well-formedness restrictions that a RoboSim model (written using the metamodel just described) must satisfy, so that its semantics presented later in Section 5 is meaningful.

4.2. 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 section. Many conditions are those expected of a standard state-machine notation, and are fully described in the definition of RoboChart [47]. Here, we focus on well-formedness conditions that are specific to RoboSim or related to constructs specific to RoboSim.

4.2.1. Module

M1 *The specification of* cycle *is a boolean expression.* As already explained, it is a predicate that specifies a restriction on the values (positive natural numbers) that the variable cycle may take.

M2 *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.

M3 *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 by at most one. An operation should be required just by one controller.

4.2.2. Controller

C1 *The specification of* cycle *is a boolean expression.*

C2 *The conjunction of the* cycle *specifications of all the simulation machines, of all the machines that define operations, and of the controller itself is not false.* Besides machines, a controller may define operations used by its machines. These operations may be defined by simulation machines themselves, and their cycle specifications also need to be feasible in the context of the controller. This condition is similar to M2, but refers to the specifications of cycle in the machines of a controller and in the controller itself, while M2 refers to the specifications of the controllers of a module and of the module itself. M2 is not concerned with restrictions in machines.[C13]

C3 *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.

C4 *The outputs of the state machines are disjoint*, to avoid conflicts as in the case of module.

4.2.3. State machine

SM1 *The specification of* cycle *is a boolean expression.*

SM2 *Input declarations can be only events and variables.*

SM3 *Only required variables can be inputs or outputs (but not both).* Variables defined in a machine are there only for local use.

SM4 *Conversely, required variables must be an input or output (but not both). In addition, required operations are outputs.*

$SKIP$	Terminating process
$STOP$	Deadlock process
$a \longrightarrow P$	Prefixed operator: initially offers to engage in the event a, and then behaves as a process P
$g \mathbin{\&} P$	Guarded process: behaves as P if the predicate g is true. Otherwise, it behaves like $STOP$
if g **then** P **else** Q	Conditional: behaves like P or Q depending on whether g is true or false
$P \Box Q$	External choice of P or Q made by the environment
$P; Q$	Sequence: behaves as P until it terminates successfully, and, then it behaves as Q
$P \setminus X$	Hiding: behaves like P but with all communications in the set X hidden
$P \mid\mid\mid Q$	Interleaving: P and Q run in parallel and do not interact with each other
$P \llbracket X \rrbracket Q$	Generalised parallel: P and Q must synchronise on events that belong to the set X
$P \llbracket X \mid Y \rrbracket Q$	Alphabetised parallel: P can communicate events in X and Q in Y; they synchronise on $X \cap Y$
$P[[c \leftarrow d]]$	Renaming: replaces uses of channel c with channel d in P
$P \Delta Q$	Interrupt: behaves like P until an event offered by Q occurs
$P \Theta_{cs} Q$	Exception: behaves as P until it raises an event in cs, and, then it behaves like Q
$\Big\|\Big\|\Big\| i : I \bullet P(i)$	Iterated Interleaving
$prioritise(P, R)$	Prioritise: behaves as P with priorities defined by R

Table 2: CSP and tock-CSP operators: P and Q are metavariables that stand for processes, a, c, and d for events, g for a condition, X, Y, and cs for sets of events, I for a set, and R for a list of sets of events.[C27]

SM5 *All operations must be outputs.* Even if they are provided by the controller, from the point of view of a machine, they are outputs. Calls to operations provided by the controller, however, do not become outputs of the system.

SM6 *Inputs, outputs, and required variables are referenced using a $.* This emphasises the fact that they are handled in a cyclic pattern, instead of at the points defined in the machine.

The complete set of well-formedness conditions is in [48]. In the next section, we define a formal CSP (and UTP) semantics of RoboSim models that are well formed according to these restrictions.

5. 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.[C14,C15]

The CSP (and tock-CSP) language contains a large number of operators that can be used to model complex concurrent systems. Table 2 presents the operators that we use in this work. Further information can be found in [41].

To illustrate the notation, we present below a very simple CSP process *CFootBot* that specifies the behaviour of our example in Figure 1. The formal semantics of RoboChart is implemented by the tool presented in Section 7 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.

$$CFootBot = EntryMoving; Obstacle; EntryTurning; wait(PI/av); CFootBot$$

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 $\longrightarrow$), *EntryMoving* engages in the *moveRet* event that marks the return of that operation, and then behaves like the process *wait*(1).

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

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. [C26]

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 ($\Box$).

$$Obstacle = obstacle \longrightarrow SKIP \,\Box\, tock \longrightarrow Obstacle$$

In FDR, with the support provided to use tock-CSP, it is possible to define *Obstacle* just as $obstacle \longrightarrow SKIP$, 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.

$$EntryTurning = moveCall.0.av \longrightarrow moveRet \longrightarrow SKIP$$

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), $\checkmark$ (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 $\checkmark$. 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 6.

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.[C26]

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.[C10]

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.

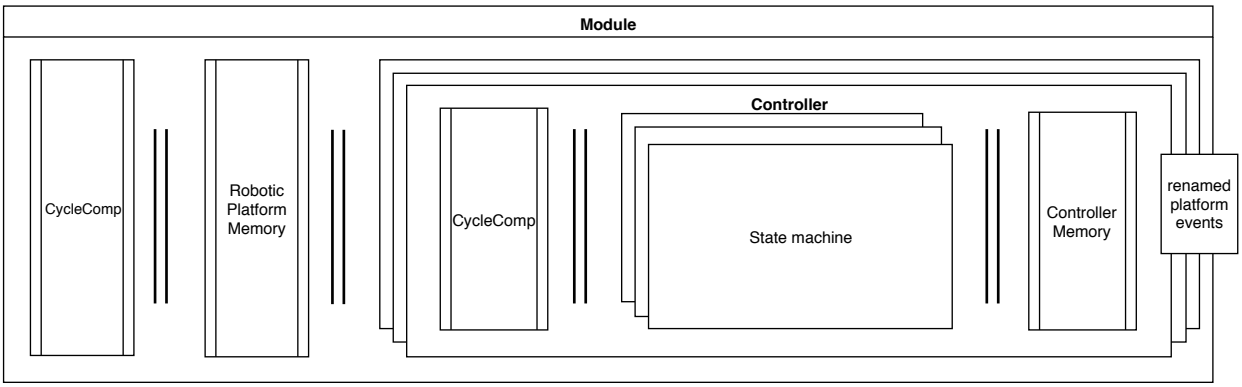

Figure 6: Structure of RoboSim tock-CSP semantics

5.2. Overview

The semantics of a RoboSim model is a CSP process that captures the behaviour of the module. Its structure is depicted in Figure 6. Controllers, machines, and states are also modelled as CSP processes. We give details below.

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.

For our example in Figure 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. Similarly, 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 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. So, they are implemented in our tool (see Section 7).

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.

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

Termination is handled by an exception ($\Theta_{\{end\}}$) 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 ($\setminus$). 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*.[C28]

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

A prioritised version needs to be defined also for the controller, machine, and state processes. This is taken into account in the automatic generation of the semantics of a RoboSim model using the tool discussed in Section 7, but we omit further discussion in the formalisation presented in the next section. 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 6). So, we focus here first on the definition of the *CycleComp*(p) process, which captures the flow in Figure 2. The definition for our example is as follows.

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

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.

$$SimCFootBot\_TakeInputsComp = registerRead?in \longrightarrow registerReadC.in \longrightarrow SKIP$$

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.

$$SimCFootBot\_ProvideOutputsComp = registerWriteC?out \longrightarrow registerWrite.out \longrightarrow SKIP$$

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 (see Figure 2).

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 3). So, in the semantics we need different event variables, like *SimMovement_obstacle* and *Foot_Bot_obstacle*, and in the process *SimSMovement* the event *registerRead.SimMovement_obstacle* must be matched to the event *registerReadC.Foot_Bot_obstacle* in the process *SimCFootBot_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.

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

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 7. 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 its initialisation and its states. For our example in Figure 3, we have the following process.

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

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* 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 (RoboChart event) in the model. In our example, these are the transitions from a junction to SMoving and DTurning (see Figure 3), which have identifiers *tid*5 and *tid*6. The synchronisation with the *Clocks* process is, therefore, on events *internal.tid*5 and *internal.tid*6 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 7, *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.

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

The RoboSim event exec is realised in the semantics using events *startexec* and *endexec*. The event *startexec* signals

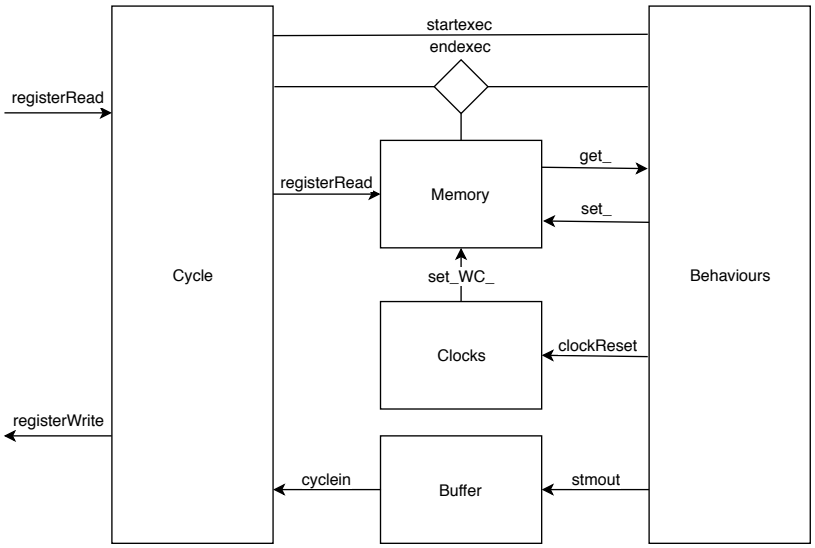

Figure 7: Semantics of a RoboSim state machine

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 7). 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.

$$TakeInputs = \;|||\; in : Inputs \bullet registerRead.in?b \longrightarrow SKIP$$

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*.

$$ProvideOutputs = CollectOutputs(Outputs)$$

$$CollectOutputs(sout) = cyclein?out \longrightarrow \mathbf{if}(out == noOut)\ \mathbf{then}\ NoOther(sout)\ \mathbf{else}\ (registerWrite.out.true \longrightarrow CollectOutputs(sout \setminus \{out\}))$$

$$NoOther(sout) = \Big|\Big|\Big|\ out : sout \bullet registerWrite.out.false \longrightarrow SKIP$$

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), 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 7). 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*.

$$\begin{aligned}
&SimSMovement_Buffer = \mathbf{let}\\
&\quad BufferF(sout) =\\
&\qquad length(sout) < numOutputs_CFootBot \mathbin{\&} stmout?out \longrightarrow BufferF(sout \frown \langle out \rangle)\\
&\qquad \Box\\
&\qquad Flush(sout)\\
&\quad Flush(\langle\rangle) = SimSMovement_Buffer\\
&\quad Flush(\langle out \rangle \frown sout) = cyclein!out \longrightarrow Flush(sout)\\
&\mathbf{within}\ stmout?out \longrightarrow BufferF(\langle out \rangle) \Box cyclein!noOut \longrightarrow SimSMovement_Buffer
\end{aligned}$$

We use $sout \frown \langle out \rangle$ to describe the concatenation ($\frown$) of *sout* with the singleton sequence $\langle out \rangle$. 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 *Flush* process, until the buffer is again empty (that is, holds the empty sequence $\langle\rangle$), and we have a recursion.

Memory. The memory process 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, the memory process controls when a transition can be taken by handling guards. 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 the memory process; 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*.

$$
\begin{array}{l}
SMovement\_Memory = \textbf{let} \\
\quad Memory(obstacle, \ldots) = \; get\_obstacle!obstacle \longrightarrow Memory\_SMovement(obstacle, \ldots) \\
\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\quad registerRead.obstacleU?x \longrightarrow Memory\_SMovement(x, \ldots) \\
\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\quad endexec.tid2 \longrightarrow Memory\_SMovement(obstacle, \ldots) \\
\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\quad \ldots \\
\textbf{within}\ Memory(false, \ldots)
\end{array}
$$

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, *tid*2 is the identifier of the transition in Figure 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 the *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.

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

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 *stmout.moveU.lv*.0 $\longrightarrow$ *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 3 from DMoving with trigger exec, if its identifier is *tid*2, the trigger is captured by the following process.

$$endexec.tid2 \longrightarrow startexec \longrightarrow SKIP$$

The *endexec* event takes the transition identifier as a parameter, because its occurrence in different transitions may be

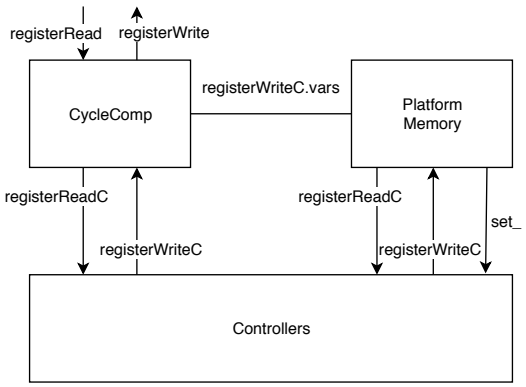

Figure 8: Semantics of a RoboSim module

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

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

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.

The complete semantics of the example in Figure 3 is in [48]. It is written in the machine version of CSP, and generated with the help of our tool (see Section 7). In the next section, we describe the rules that formalise the semantics of RoboSim, and are used to automate the generation of formal models of diagrams.

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, which is Rule 1 defined below, maps a module to a CSP process. The structure of the process that it defines is depicted in Figure 8.

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 fully defined below, in Appendix A, or in [47]. 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

Rule 2. 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)
```

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.

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* and platform-memory processes depicted in Figure 8 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(rp).

The function memoryComp(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 (see Rule 8 in Appendix A). 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 5). 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 8).

The function composeControllers(m, ctrls) (see Rule 9) basically constructs a parallelism of the processes for each of the controllers in the sequence ctrls. It is defined in terms of the function $[\![\text{ctrl}]\!]_C$, which specifies the process for one controller (see Rule 10). 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*.

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 (see Section 7) we use fully qualified names based on the names for the modules, controllers, and machines to improve traceability for counterexamples.

The function cycleComp(p, inputs, outputs) is defined by Rule 2. 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

Rule 3. State machine $[\![\text{stm} : \text{StateMachineDef}]\!]_{STM} : \text{CSPProcess} =$

$$
\begin{array}{l}
((((\text{cycle}(cycle, \text{inputs}, \text{outputs}) \\
\quad [\![\{ registerRead, endexec, tock \}]\!] \\
\text{constInitSTM}(\text{consts}, \text{stm}, (\text{stmMemory}(\text{stm}, \text{wcs}) \ [\![\text{clockMemSync}]\!] \ \text{stmClocks}(\text{wcs}))) \setminus (\text{clockMemSync} \setminus \text{trigEvents}(\text{stm}))) \\
\quad [\![\{ cyclein \}]\!] \\
\text{buffer}(\text{outpus})) \setminus \{ cyclein \}) \\
\quad [\![\text{getsetChannels}(\text{stm}) \cup \text{trigEvents}(\text{stm}) \cup \text{clockResets}(\text{wcs}) \cup \{ stmout, startexec, endexec, end \}]\!] \\
\text{behaviours}(\text{stm})) \\
\quad \setminus \text{getsetChannels}(\text{stm}) \cup \text{clockResets}(\text{wcs}) \cup \{ internal, entered, stmout, startexec, endexec \}) \\
\Theta_{\{end\}} \ SKIP \\
\quad \textbf{where} \\
\qquad \text{inputs} = \text{stm.inputs} \\
\qquad \text{outputs} = \text{stm.outputs} \\
\qquad \text{wcs} = \{ t : \text{allTransitions}(\text{stm}) \mid t.\text{condition} \neq \text{null} \bullet t \mapsto \text{wc}(t.\text{condition}) \} \\
\qquad \text{clockMemSync} = \{ t : \text{Transition} \mid t \in \text{dom wcs} \bullet \text{triggerEvent}(t) \} \cup \{ v : \text{allClockVariables}(\text{wcs}) \bullet setWC\_\text{vid}(v) \}
\end{array}
$$

parameters of any rules that use them. We omit these parameters for simplicity.

In *CycleComp*, 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*.

In the case of the outputs corresponding to a platform variable v, the communication *registerWriteC*.take(v) is shared with the memory (see Figure 8). In this way, the update is both kept in the memory and written to a register.

The controller processes are defined in much the same way as a module process, and we include the relevant rule in Appendix A. We next consider Rule 3 for state machines, which formalises the process network in Figure 7. The cycle process is defined in Rule 4, and the machine-memory process in Rule 5, both explained in the sequel.

The clock process is defined by stmClocks(wcs) just like in the RoboChart semantics. Its definition uses a function wcs, which associates every transition t of the machine (in allTransitions(stm)) that has a guard (t.condition ≠ 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.

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 triggerEvent for each transition 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 12). 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.

The function getsetChannels(stm) defines the get and set channels for the variables that are internal to the machine. The set trigEvents(stm) collects the events that represent a trigger in a transition in the machine. Channels used to reset the clocks are included in the set clockResets(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.

The process that defines the control flow of the machine, defined by behaviours(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 (see Rule 13). 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.

Rule 4. Cycle for machines

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

```
let Cycle(period) = (||| in : inputs • registerRead.take(in) ⟶ SKIP);
                    startexec ⟶ endexec?tid ⟶ SKIP;
                    CollectOutputs(outputs);
                    (wait(period) □ end ⟶ SKIP);
                    Cycle(period)
    CollectOutputs(sout) = cyclein?out ⟶ if(out == noOut)
                                         then NoOther(sout)
                                         else (registerWrite.giveT(out) ⟶ CollectOutputs(sout \ {out}))
    NoOther(sout) = ||| out : sout • registerWrite.giveF(out) ⟶ SKIP
within Cycle(p)
```

Rule 4 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*.

The memory for a state machine stm is defined in Rule 5. 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.

With nvars[name(v) := x] we denote the list of variable names (used as argument to *Memory*), with the name 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(stm) of transitions of stm, a process memoryTransition(t, wcs) models its guard. If the transition has a trigger, which is necessarily exec, then memoryTransition(t, wcs) is a prefixing on *endexec* as formalised below. For transitions without a trigger, we have a prefixing on the special event of the semantics called *internal* as said before.

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.

The rule that specifies the function behaviours(stm) that formalises the core behaviour of the state machine uses semantic functions defined in [47]. 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.
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 5).

The rules for outputs define actions that buffer them. We give the action for an operation call in Rule 7. There, eval(e) defines the parameters for the communication corresponding to the list of arguments e for the operation.

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

Rule 5. State-machine memory stmMemory(stm : StateMachineDef, wcs : Transition ⇸ (Expression, WC)) : CSPProcess =

$$
\begin{array}{l}
\textbf{let}\ \mathit{Memory}(\mathsf{nvars}) = \\
\quad (\Box\ v : \mathsf{ivars} \bullet \mathit{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \longrightarrow \mathit{Memory}(\mathsf{nvars}) \ \Box\ \mathit{registerRead}?x \longrightarrow \mathit{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad (\Box\ v : \mathsf{lvars} \bullet \mathit{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \longrightarrow \mathit{Memory}(\mathsf{nvars}) \ \Box\ \mathit{set\_}\mathsf{vid(v)}?x \longrightarrow \mathit{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad (\Box\ v : \mathsf{cvars} \bullet \mathit{setWC\_}\mathsf{vid(v)}?x \longrightarrow \mathit{Memory}(\mathsf{nvars[name(v) := x]})) \\
\quad \Box \\
\quad \Box\ v : \mathsf{allConstants(stm)} \bullet \mathit{get\_}\mathsf{vid(v)}!\mathsf{name(v)} \longrightarrow \mathit{Memory}(\mathsf{nvars}) \\
\quad \Box \\
\quad (\Box\ t : \mathsf{allTransitions(stm)} \bullet \mathsf{memoryTransition(t, wcs)};\ \mathit{Memory}(\mathsf{nvars})) \\
\quad \Box \\
\quad \mathit{endexec.terminate} \longrightarrow \mathit{Memory}(\mathsf{nvars}) \\
\textbf{within}\ \mathsf{Memory(varvalues)} \\
\textbf{where} \\
\quad \mathsf{ivars} = \mathsf{inputVariables(stm)} \\
\quad \mathsf{lvars} = \mathsf{allLocalVariables(stm)} \\
\quad \mathsf{cvars} = \mathsf{clockVariables(stm)} \\
\quad \mathsf{consts} = \langle \mathsf{v} : \mathsf{allConstants(stm)} \bullet \mathsf{v} \rangle \\
\quad \mathsf{nvars} = \langle \mathsf{v} : \mathsf{ivars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{name(v)} \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name(v)} \rangle \\
\quad \mathsf{varvalues} = \langle \mathsf{v} : \mathsf{ivars} \cup \mathsf{lvars} \cup \mathsf{cvars} \bullet \mathsf{initial(v)} \rangle \frown \langle \mathsf{v} : \mathsf{consts} \bullet \mathsf{name(v)} \rangle
\end{array}
$$

Rule 6. Trigger $[\![\textbf{exec} : \mathsf{ExecTrigger}]\!]^{\mathsf{tid}}_{\mathit{Trigger}} : \mathsf{CSPProcess} =$

$$\mathit{endexec}.\mathsf{tid} \longrightarrow \mathit{startexec} \longrightarrow \mathit{SKIP}$$

Rule 7. Operation call $[\![\mathsf{op(e)}]\!]_{\mathit{Statement}} : \mathsf{CSPProcess} =$

$$\mathit{stmout}!\mathit{opU}.\mathit{true}.\mathsf{eval(e)} \longrightarrow \mathit{SKIP}$$

6. 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.[C7]

Our formalisation of the assumptions takes the form of processes $TA1$, $TA2$, and $TA3$ (Section 6.1). 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 6.2). 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 (see Section 7).

6.1. Relating design and simulation models

The first assumption relates RoboChart and 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 $TA1$.

$$A1 = (|||\ in : Inputs \bullet A1Event(in))$$

For an input in, the process $A1Event(in)$ defined below 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.

$$\begin{array}{lcl} A1Event(in) & = & registerRead.in?b?x \longrightarrow \\ & & \quad \textbf{if}\ b \\ & & \quad \textbf{then}\ (useinv(in);\ A1Event(in) \,\Box\, A1Event(in)) \\ & & \quad \textbf{else}\ A1Event(in) \end{array}$$

The process $useinv(in)$ maps an event in to a process that raises the event itself (as declared in CSP) 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 $useinv(obstacleU) = obstacle \longrightarrow SKIP$. 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 (Δ) $A1$ and enforce its termination: $TA1 = A1\ \Delta\ (end \longrightarrow SKIP)$.

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.

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

For our example, $OutputEvents$ is $\{moveCall, stopCall\}$; 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 *A*3 defined blow.

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

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 explained 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, *TA*2 is the version of *A*2 that handles termination using interrupt as shown above for *TA*1.

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, $RUN(X) = \Box\, x : X \bullet x \longrightarrow RUN(X)$.

$$
\begin{array}{lll}
TA3 = & (|||\, in : Inputs \bullet registerRead.in?b?x \longrightarrow SKIP); & \\
& (RUN(ExternalEvents) \;\Delta\; (|||\, out : OutputEventsOp \bullet registerWrite.out?b?x \longrightarrow SKIP)); & \\
& (end \longrightarrow SKIP \;\Box\; wait(p);\; TA3) &
\end{array}
$$

In *TA*3, 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.

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

This definition uses the alphabetised parallel operator of CSP. The sets *InputEvents* and *ExternalEvents* contain the input and external events of the process $[\![RCM]\!]_{\mathcal{M}}$ that defines the semantics for a RoboChart module RCM used below. In our example, these sets of events are {*obstacle*} and {*obstacle*, *moveCall*, *moveRet*, *stopCall*, *stopRet*}, respectively. The alphabets of each process in the definition of *TA*1*A*2*A*3 is simply the set of events used by that process.

*TA*1*A*2*A*3 is composed in parallel with $[\![RCM]\!]_{\mathcal{M}}$ to define a specification *Constrained* for a simulation.

$$
Constrained = ([\![RCM]\!]_{\mathcal{M}};\; end \longrightarrow SKIP \;[\![\; ExternalEvents \cup \{tock, end\} \;]\!]\; TA1A2A3) \setminus \{end\}
$$

The sequential composition of $[\![RCM]\!]_{\mathcal{M}}$ with the process $end \longrightarrow SKIP$ is a technicality. The purpose of the event

end, as already explained, is to ensure distributed termination. It is used but hidden in $[\![RCM]\!]_{\mathcal{M}}$; so if $[\![RCM]\!]_{\mathcal{M}}$ terminates, the process $end \longrightarrow SKIP$ signals termination to *TA*1*A*2*A*3. 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 *TA*3 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.

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

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.

6.2. 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 *TA*3. The deadlock indicates that, when the RoboChart model is further restricted by the order and timing imposed by *TA*3, there is a conflict. So, the scheduling in cycles imposed by *TA*3 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 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.

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

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

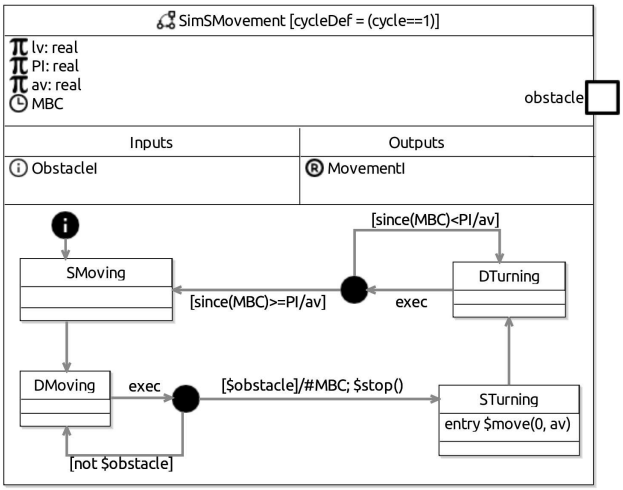

Figure 9: Simulation machine: missing entry action for state SMoving

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, *TA*3 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 *A*3. 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 [33]. 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.[C29]

For our example, as mentioned, both simulations in Figures 3 and 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 3.

As a first example, we miss the entry action of the state SMoving (see Figure 9). 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 (as in Figure 10), this allows the simulation to call move twice in the same cycle. A counterexample shows the two calls.

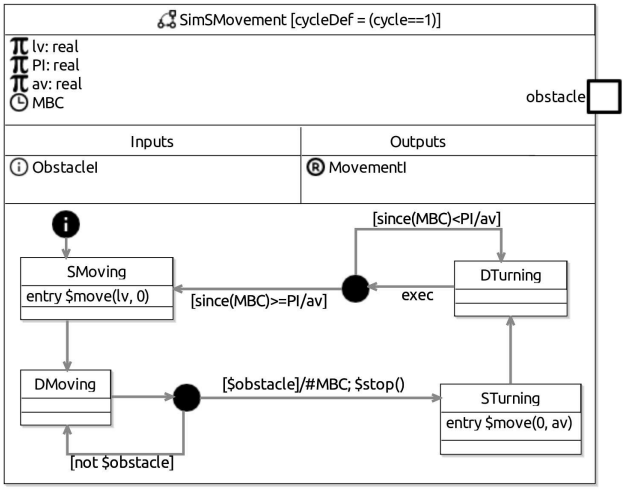

Figure 10: Simulation machine: missing exec in transitions from DMoving

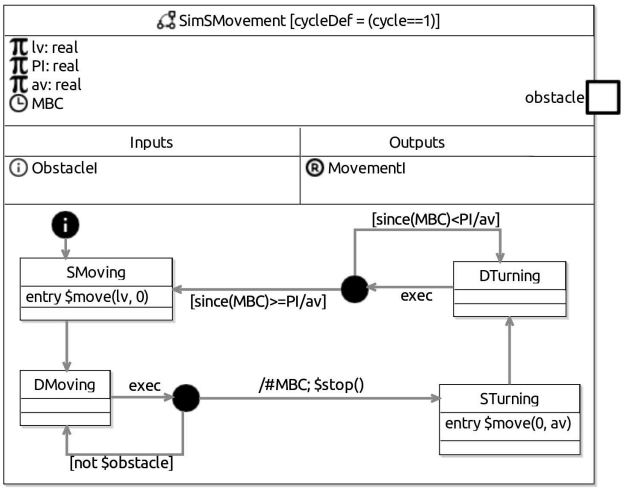

Figure 11: Simulation machine: missing guard to state STurning

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 shown in Figure 11), or swaps the two guards (see Figure 12), 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 13. 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 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 1).

Finally, we consider another subtle mistake where the wait(1) construct in the RoboChart model (see Figure 1) is not properly simulated. In the simulation in Figure 14, instead of a junction with a single exec event from the state DMoving, as in the correct simulation (Figure 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 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

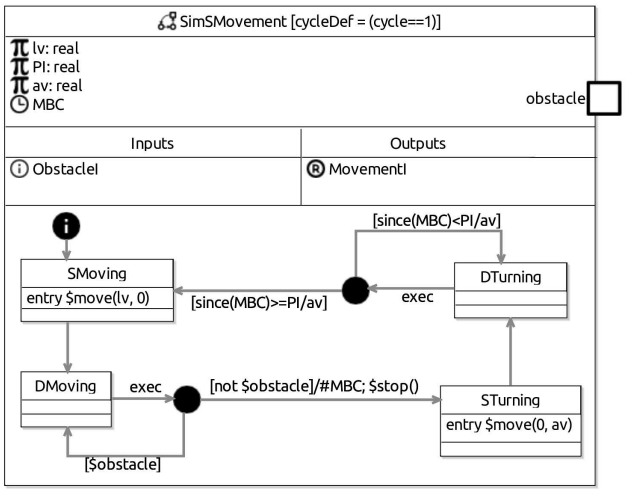

Figure 12: Simulation machine: swapping guards in transitions from DMoving

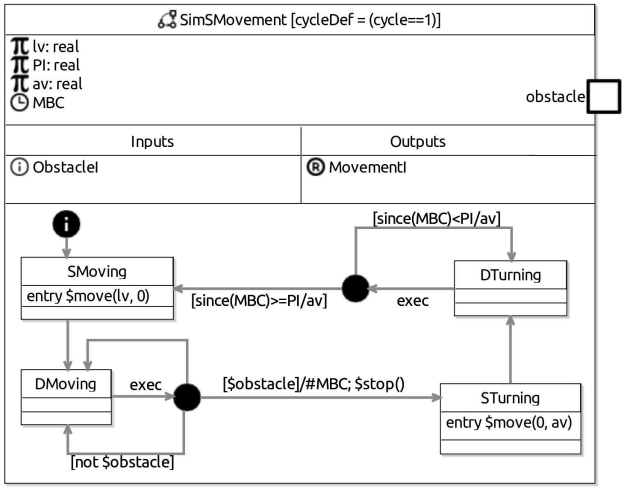

Figure 13: Simulation machine: possibly ignoring an obstacle occurrence

be very elaborate. Our reasoning framework allows (automatic) verification that can reveal subtle problems.

7. Tool support and examples

Support for modelling and reasoning in both RoboChart and RoboSim is provided by RoboTool[2]. The graphical interface uses similar visual elements in both notations to keep a unified interface for end users.

RoboTool is a set of Eclipse [1] plugins implemented using the Xtext [3] and Sirius [2] frameworks. Xtext automatically generates plugins that implement a parser and a linker, and provides mechanisms for the implementation of scope providers, validators, type checkers, code generators, and so on. Sirius allows us to define graphical representations for metamodels, and produces a plugin for construction and visualisation of instances of the metamodel.

RoboSim's grammar extends RoboChart's grammar and also extends its implementations (scope providers, validators, type checkers, and so on). For example, as already mentioned, the semantics of a RoboSim's simulation machine

[2] `www.cs.york.ac.uk/circus/RoboCalc/robotool`

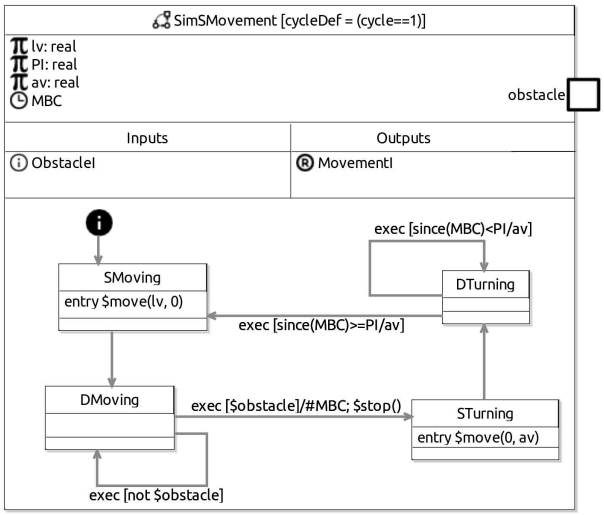

Figure 14: Simulation machine: wrong cycle allocation

is similar to RoboChart's state machine, augmented with the semantics of the cycle period, buffer, and treatment of *registerRead* and *registerWrite*. Only specific methods are overridden in RoboSim's implementation classes.

Besides generating the CSP semantics of RoboChart and RoboSim models automatically, RoboTool also generates assertions for checking classic properties, like deadlock and livelock freedom, and determinism. These are all (implicitly) refinement assertions. It is also possible to use refinement to verify properties directly, and generate reports. A fully fledged property language that is accessible by practitioners is in our plans for future work.[C10]

We present below a few examples. The first is a simple robot that performs a square trajectory, which shows how composite RoboChart states, applications that terminate, and deadlines may be handled. A transporter used in a swarm application, specified in [9], illustrates simulations with during actions. The Alpha Algorithm from [12] 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 with respect to the RoboChart model.

7.1. Square

This case study involves a robot that performs a square trajectory, trying to avoid possible obstacles. We consider the RoboChart model from [39], reproduced here in Figure 15.

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 raises 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.

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.

A simulation model with a cycle value 1 is presented in Figure 16. The structure of the simulation could have mimicked that of the RoboChart model, since RoboSim also allows composite states. Instead, this simulation model

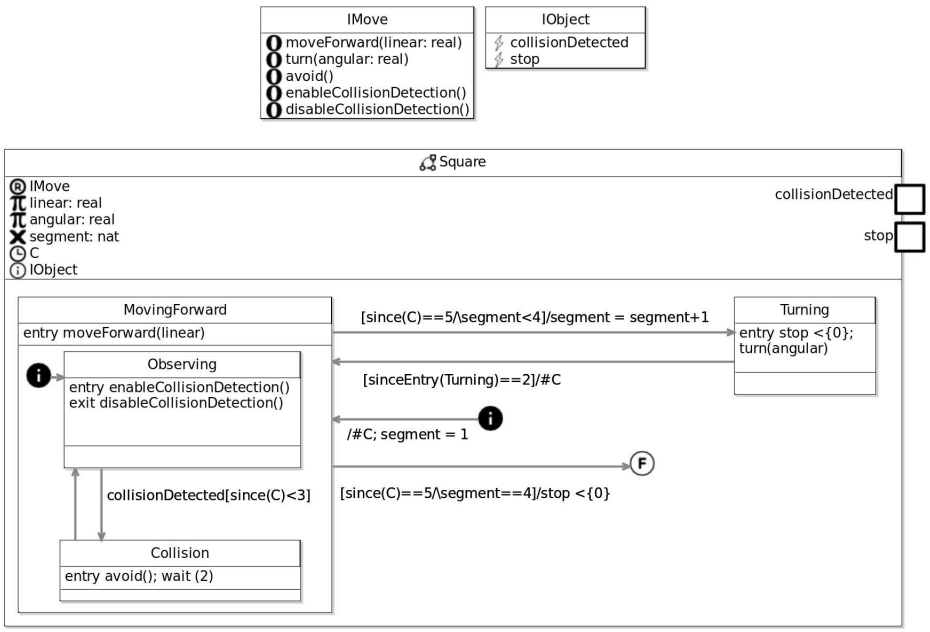

Figure 15: RoboChart: square trajectory state machine. The module contains a single controller, which contains this machine.

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 15. 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

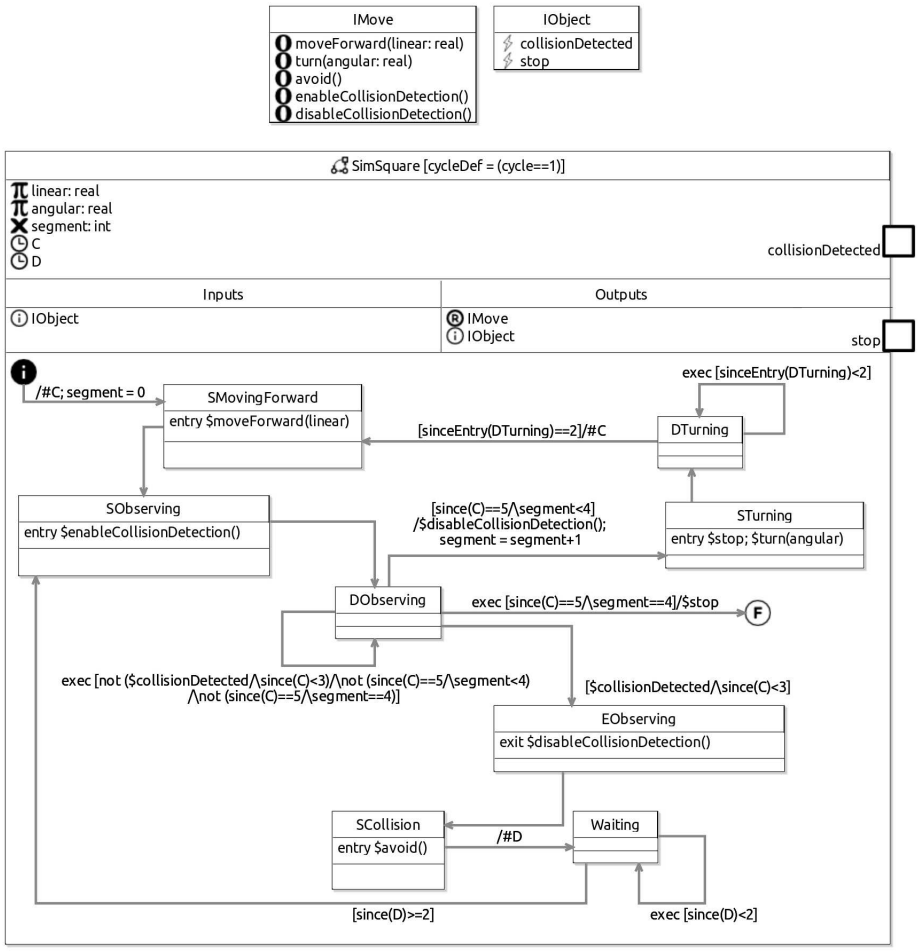

Figure 16: RoboSim: square trajectory state machine

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.

7.2. Transporter

We present now RoboChart and RoboSim models of the transporter described in [9], which proposes use of a swarm of small robots for moving a large 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. The RoboChart state machine is presented in Figure 17. A verified simulation machine in RoboSim is depicted in Figure 18.

As an overview, some relevant events used in this model are objectSeen, whose occurrence indicates the presence of the object and whose parameter records an estimated distance; goalSeen indicates the detection of the goal; and neighbourDetected, as the name suggests, captures the presence, and its parameter the number, of neighbours. Some operations that appear as entry, during, and exit actions are used to enable the platform to receive those events.

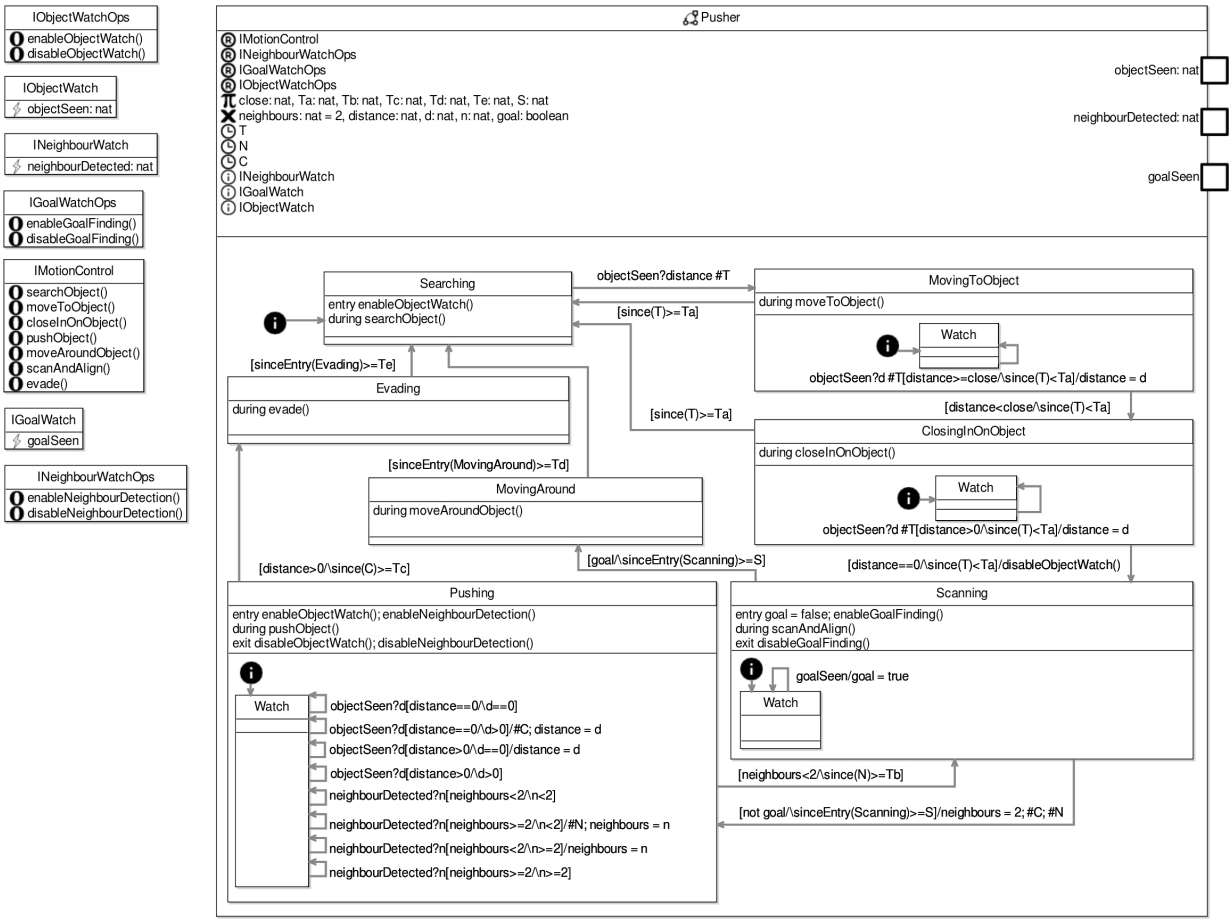

Figure 17: RoboChart: transporter state machine

Rather than discussing the model and 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.

The full details of the case study, including the proof that the RoboSim model refines the RoboChart model, constrained by the assumptions *TA*1, *TA*2 and *TA*3 (see Section 5), can be found in [48]. We have started the verification by confirming the schedulability of the RoboChart model. It was schedulable, but several modelling mistakes were uncovered. Particularly, marking the end of a cycle can be very subtle. For instance, determining that $goalseen must be first read in the current cycle, and, subsequently, in consecutive new cycles, did not seem obvious at first.

Another point we would like to emphasise is that the RoboChart model of the transporter is nondeterministic, and the simulation is deterministic; it is in line with an existing simulation presented in [9].

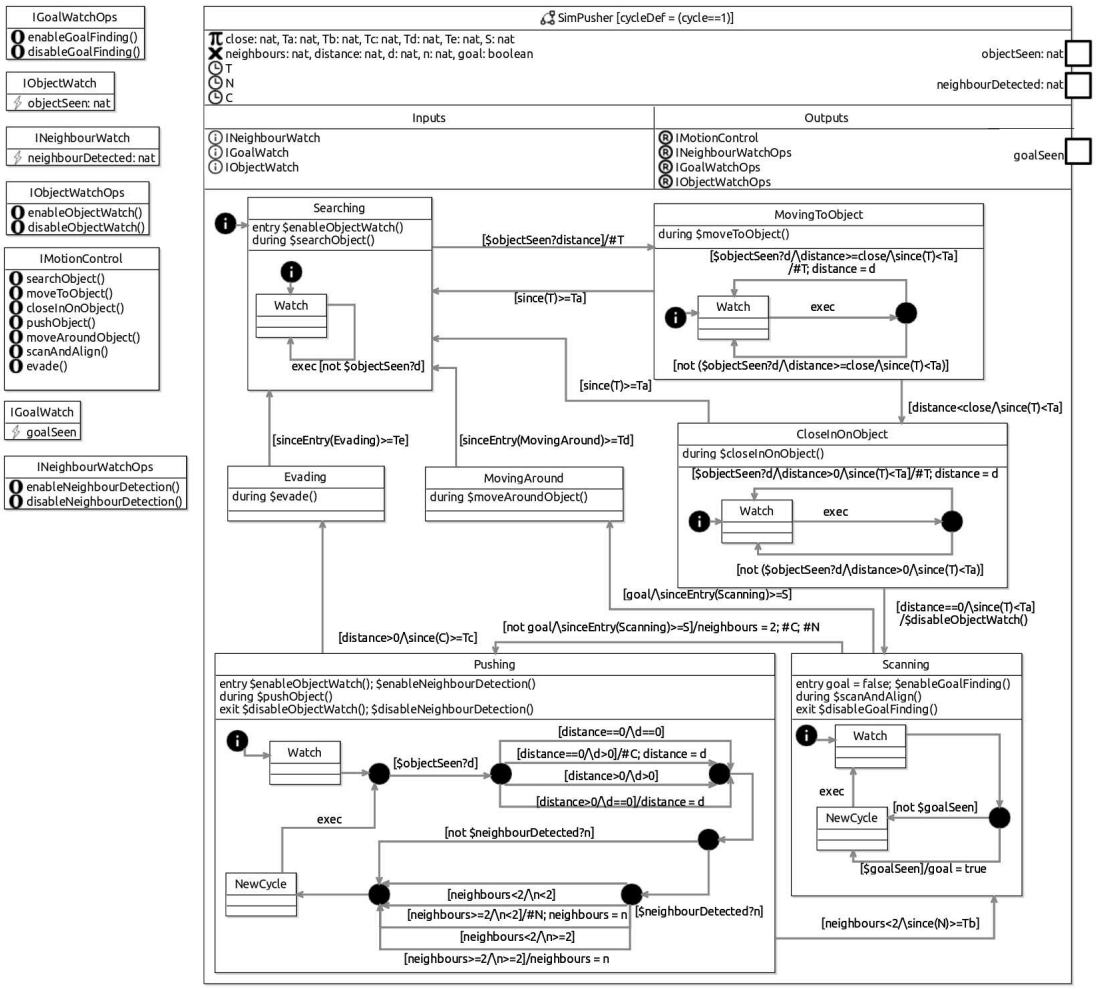

Figure 18: RoboSim: transporter state machine

7.3. Alpha Algorithm

Our third case study is the model of a single robot in a swarm acting under the Alpha Algorithm [12]. This involves two machines: one specifying the robot movement, and the other, the communication with the other robots in the swarm. Figure 19 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 7.1, 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.

The simulation model for this example is presented in Figure 20. 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)

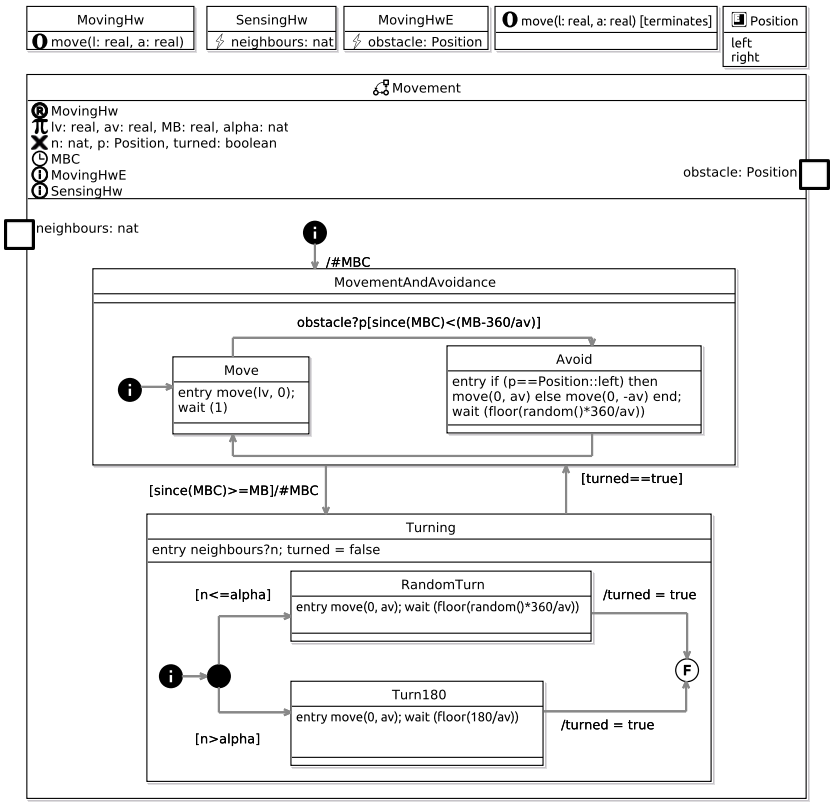

Figure 19: RoboChart: Alpha Algorithm

in the composite state MovementAndAvoidance and an initial state in Turning to control the cycle scheduling. In between the states 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 19, 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. During this verification, we have detected a schedulability 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 19) had to be included. It can be difficult to get these right without proper verification support, as we are proposing here.

8. Conclusions and future work

In this paper, a main contribution is an approach to verify consistency of simulations with respect to design models. We identify the assumptions that are routinely made in an implicit way when developing a simulation and formalise them. We show how to use refinement to use these assumptions to check that the design model has a valid cyclic schedule useful for simulation, and to compare a design and a simulation. The approach is justified using CSP, but its use does not require knowledge of CSP, since CSP models and assertions can be automatically generated.[C7]

Our work is carried out in the context of two domain-specific diagrammatic languages for robotics: RoboChart and RoboSim. The central modelling concept in both of them is state machines, but RoboChart is targeted at designs, and RoboSim is targeted at simulation models. Another contribution here is RoboSim: we define its metamodel,

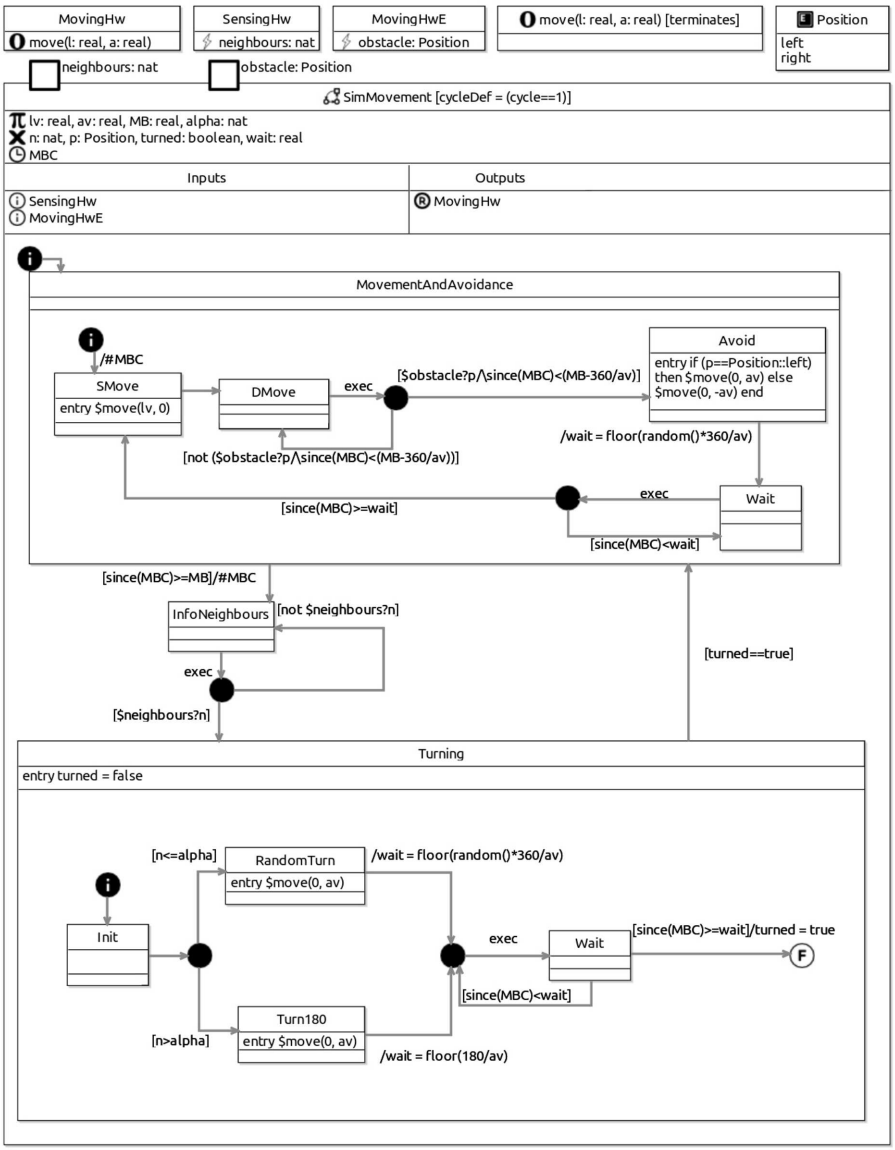

Figure 20: RoboSim: Alpha Algorithm

well-formedness conditions, and process algebraic semantics. Our long-term goal is to establish RoboSim as a tool-independent notation that is agnostic to programming languages adopted in simulation implementations and simulators. Work on automatic translation from (restricted) RoboChart models to code for the ARGoS simulator has already been carried out [25]. Replicating that work for RoboSim is easier.

Tool support is under development, but can generate the semantics of a RoboSim model automatically, as well as the verification conditions to ascertain schedulability and correctness. Automation via model checking has been feasible for our case studies, but for larger examples, we advocate the use of proof-based verification using the relational semantics of CSP encoded in the UTP. Results available for Isabelle/HOL [15] are relevant.

RoboChart and RoboSim are specialised languages that impose an architectural pattern of design on the models, and their automatically generated semantics also has a fixed structure. This opens the perspective of high levels of automation in proof via a variety of techniques, not only model checking. Optimisation of the consistency checks

Rule 8. Module or controller memory

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

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

Rule 9. Composition of controllers composeControllers(m : Module, ctrls : Seq(Controller)) : CSPProcess =

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

using FDR is a topic for future work, as is automation of theorem proving via tactics.

In the long run, however, we propose the use of model-to-model transformations to derive a candidate RoboSim model from a RoboChart design that is guaranteed to be correct according to the notion defined here. Since a RoboChart design can be simulated in many different ways, these transformations have to make some design assumptions for the simulation: a cycle must be as given, the component structure of the models can be maintained, and the scheduling can be eager, in the sense of executing the machines as fast as possible without breaking the assumptions. The definition of such a sound transformation technique is future work. What we present here are key ingredients to enable this automated approach: a modelling language for simulation and a notion of correctness.[C7]

To validate the definition of the RoboSim semantics and our notion of correctness, we have taken several steps. We have considered, first of all, a series of small examples that illustrate common errors. We have also implemented an automatic generator for the semantics and for key components of the correctness notion, namely, the assumptions that need to be associated with a RoboChart (design) model. This ensures, for example, that the definitions are well formed. Moreover, we have proved key properties of those processes (deadlock freedom, determinism, and so on) using a well established model checker, namely FDR. Finally, we have tried some larger examples. We are also reassured by the fact that the definition relies on a well-established notion of conformance, namely, behaviour subsetting, of CSP.[C29]

In terms of future work on RoboSim as a language, we plan to consider examples that make use of multi-rate simulations, involving components (state machines and controllers) with different cycle periods in the same module. This will require the use of buffers to handle the inputs and outputs also at the controller and module level.

Our work is not relevant only in the context of RoboChart and RoboSim. We have already investigated the use of RoboChart in conjunction with Simulink [8], and we plan to extend the investigation to RoboSim, possibly with the use of design patterns in Simulink inspired by RoboSim. We have given process algebraic semantics to Simulink [7] and SysML [26]. It is, therefore, possible to consider verification as we have presented here in the context of those languages. We have identified issues relevant for the verification of simulations no matter what notation is used.

Appendix A. Semantic functions

In Rule 8, 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

Rule 10. Semantics of controllers $[\![$c : Controller$]\!]_C$: CSPProcess =

```
let Controller(p : Period) =
    (((cycleComp(p, inputs, outputs) ||| mcMemory(vars, evars, c))
      [| setConstants ∪ {| registerReadC, registerWriteC, tock |} |]
    composeMachines(c, machines) )
    \ setConstants ∪ {| registerReadC, registerWriteC |} ) Θ{|end|} SKIP
within Controller(cycle)
where
    machines = c.machines
    cons = c.connections
    inputs = InputsC(c, cons)
    outputs = OutputsC(c, cons)
    vars = Vars(c)
    evars = InternalsC(c, cons)
    setConstants = {| ct : allConstants(c) • set_name(ct) |}
```

Rule 11. Composition of machines composeMachines(machines : Seq(StateMachineDef)) : CSPProcess =

```
[| {tock, end} |] stm : 1 .. #machines • renamingMachine([|machines(stm)|]STM)
```

Rule 12. Buffer for machines
buffer(outputs : Set(OutputsType)) : CSPProcess =

```
let  Buffer = stmout?out ⟶ BufferOF(⟨out⟩) □ cyclein!noOut ⟶ Buffer
     BufferF(sout) =  length(sout) < #outputs & stmout?out ⟶ BufferF(sout ⌢ ⟨out⟩)
                      □
                      Flush(sout)
     Flush(⟨⟩) = Buffer
     Flush(⟨out⟩ ⌢ sout) = cyclein!out ⟶ Flush(sout)
within Buffer
```

Rule 13. Behaviour of state machine behaviours(stm : StateMachineDef) : *CSPProcess* =

```
let
   Ending = endexec.terminate ⟶ (startexec ⟶ Ending □ end ⟶ SKIP)
within
    startexec⟶
    ((((initialisation(stm) [| flowevts |] composeStates(⟨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 |} }
```

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(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.

In the definition of the composition of controller processes in Rule 9, we use the iterated generalised parallel operator $[\![cs]\!]\ i : I \bullet P(i)$, which composes processes $P(i)$, all synchronising on the events of a set cs. In Rule 9, the set contains just *tock* and *end*, and the processes are defined by the semantics of controllers. A renaming defined by a function renamingController requires the inputs from *registerRead* to be obtained via the channel *registerReadC*, and the outputs to be provided via *registerWriteC* by renaming of *registerWrite*. In addition, the renaming captures the connections between a controller and the platform by mapping the events used in the controller to the events of the platform to which they are connected. These events are parameters of the *registerReadC* and *registerWriteC* channels.

For the semantics of controllers (defined by Rule 10), we use InputsC(c, cons), OutputsC(c, cons), VarsC(c), and InternalsC(c, cons) to define its inputs, outputs, variables, and internal events. The inputs are the events associated with connections to a state machine and required variables that are used as input in one of its state machines. The outputs are required variables used as outputs in one of the state machines, required operations, and events associated with connections from a state machine. Vars(c) includes variables and operations declared locally in the controller. Finally, InternalsC(c, cons) determines the events associated with connections between machines to define a function from output to input events. Otherwise, the semantics of a controller is very much like that of a module.

In a controller process, the controller-memory process does not need to interact with the cycle process, since it holds only variables local to the controller. (We recall that a platform-memory process may hold platform variables whose updates are visible, and so it does interact with the cycle process for the module.) In the controller-memory process, the set channels for the required constants synchronise with the matching set channels in the module memory.

In Rule 13, the *entered* events are not hidden, because they are used in the *Clocks* process to deal with the sinceEntry time primitives. They are hidden in the overall semantics of a machine (see Rule 3). The special event *end* is used in the machine to signal termination of all processes that represent its states. That event is local to the parallelism of state processes that define the machine. Once the machine finishes, however, an extra *endexec* event is raised, so that the final cycle can complete, and then another event *end* can be raised to signal termination to other machines or controllers in the same module. It is possible that the cycles go on for a while, with repeated events *startexec* and *endexec* taking place, but no further processing, until there is an agreement for termination.

Acknowledgements. This work is funded by the EPSRC grants EP/M025756/1 and EP/R025479/1, by the Royal Academy of Engineering, and by INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14. No new primary data was created as part of the study reported here. We are grateful to anonymous reviewers and James Baxter for detailed comments that helped us improve the presentation of our work.

References

[1] Eclipse. `http://eclipse.org/`. Accessed: 2018-01-06.
[2] Sirius. `https://www.eclipse.org/sirius/`. Accessed: 2016-05-25.
[3] Xtext. `https://eclipse.org/Xtext/`. Accessed: 2016-05-25.
[4] T. Abdellatif, S. Bensalem, J. Combaz, L. deSilva, and F. Ingrand. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578, 2012.
[5] S. Alexandrova, Z. Tatlock, and M. Cakmak. Roboflow: A flow-based visual programming language for mobile manipulation tasks. In *IEEE International Conference on Robotics and Automation*, pages 5537–5544, 2015.
[6] 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.
[7] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via Circus. *Formal Aspects of Computing*, 23(4):465–512, 2011.
[8] A. L. C. Cavalcanti, A. Miyazawa, R. Payne, and J. Woodcock. Sound simulation and co-simulation for robotics. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 173–194. Springer International Publishing, 2017.
[9] J. Chen, M. Gauci, and R. Gross. A strategy for transporting tall objects with a swarm of miniature mobile robots. In *2013 IEEE International Conference on Robotics and Automation*, pages 863–869. IEEE, 2013.
[10] A. Desai, T. Dreossi, and S. A. Seshia. Combining model checking and runtime verification for safe robotics. In S. Lahiri and G. Reger, editors, *Runtime Verification*, pages 172–189. Springer, 2017.
[11] 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.
[12] 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.
[13] M. Farrell, M. Luckcuck, and M. Fisher. Robotics and integrated formal methods: Necessity meets opportunity. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods*, volume 11023 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2018.

[14] FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org`, 2014.
[15] S. Foster, F. Zeyda, and J. C. P. Woodcock. Unifying heterogeneous state-spaces with lenses. In A. C. A. Sampaio and F. Wang, editors, *Theoretical Aspects of Computing*, volume 9965 of *Lecture Notes in Computer Science*, pages 295–314, 2016.
[16] M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal Methods and Software Engineering*, pages 383–399. Springer, 2016.
[17] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.
[18] B. Gerkey, R. T. Vaughan, and H. Andrew. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *11th International Conference on Advanced Robotics*, pages 317–323, 2003.
[19] 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.
[20] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
[21] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu. ROSRV: Runtime Verification for Robots. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, pages 247–254. Springer, 2014.
[22] K. Kapellos, D. Simon, M. Jourdant, and B. Espiau. Task level specification and formal verification of robotics control systems: State of the art and case study. *International Journal of Systems Science*, 30(11):1227–1245, 1999.
[23] J. Klein. BREVE: a 3D Environment for the Simulation of Decentralized Systems and Artificial Life. In *8th International Conference on Artificial Life*, pages 329–334. The MIT Press, 2003.
[24] M. Klotzbucher and H. Bruyninckx. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *Journal of Software Engineering for Robotics*, 2(13):28–56, 2012.
[25] W. Li, A. Miyazawa P. Ribeiro, A. L. C. Cavalcanti, J. C. P. Woodcock, and J. Timmis. *From Formalised State Machines to Implementations of Robotic Controllers*, pages 517–529. Springer International Publishing, 2018.
[26] L. Lima, A. Miyazawa, A. L. C. Cavalcanti, M. Cornélio, J. Iyoda, A. C. A. Sampaio, R. Hains, A. Larkham, and V. Lewis. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, pages 1 – 28, 2015.
[27] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. Formal specification and verification of autonomous robotic systems: A survey. *CoRR*, abs/1807.00048, 2018.
[28] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.
[29] The MathWorks,Inc. *Simulink*. `www.mathworks.com/products/simulink`.
[30] The MathWorks,Inc. *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.com/products`.
[31] 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.
[32] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede. A survey on domain-specific modeling and languages in robotics. *Journal of Software Engineering for Robotics*, 7(1):75–99, 2016.
[33] M. V. M. Oliveira, P. Antonino, R. Ramos, A. C. A. Sampaio, A. C. Mota, and A. W. Roscoe. Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing*, 28(6):937–1004, 2016.
[34] M. Olivier. Webots^{TM}: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.
[35] H. W. Park, A. Ramezani, and J. W. Grizzle. A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. *IEEE Transactions on Robotics*, 29(2):331–345, 2013.
[36] 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.
[37] 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.
[38] C. A. Rabbath. A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. *Journal of Intelligent & Robotic Systems*, 70(1):233–253, 2013.
[39] 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.
[40] E. Rohmer, S. P. N. Singh, and M. Freese. V-rep: A versatile and scalable robot simulation framework. In *IEEE International Conference on Intelligent Robots and Systems*, volume 1, pages 1321–1326. IEEE, 2013.
[41] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
[42] S. Scherer, F. Lerda, and E. M. Clarke. Model checking of robotic control systems. In *8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2005.
[43] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.
[44] D. Stamper, A. Lotz, M. Lutz, and C. Schlegel. The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software. *Journal of Software Engineering for Robotics*, 7(1):3–19, 2016.
[45] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.
[46] T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. L. Grixa, F. Ruess, M. Suppa, and D. Burschka. Toward a Fully Autonomous UAV: Research Platform for Indoor and Outdoor Urban Search and Rescue. *IEEE Robotics Automation Magazine*, 19(3):46–56, 2012.
[47] University of York. *RoboChart Reference Manual*. `www.cs.york.ac.uk/circus/RoboCalc/robotool/`.
[48] University of York. *RoboSim Reference Manual*. `www.cs.york.ac.uk/circus/RoboCalc/robotool/`.

[49] M. Wachter, S. Ottenhaus, M. Krohnert, , N. Vahrenkamp, and T. Asfour. The ArmarX Statechart Concept: Graphical Programing of Robot Behavior, journal = Frontiers in Robotics and AI, volume = 3, pages = 33, year = 2016,.

Index of Changes

This index lists, for each comment, the pages where the text has been modified to address it. Since the same page may contain multiple changes, the page number contains the index of the change in superscript to identify different changes. Finally, the page number contains a hyperlink that takes the reader to the corresponding change.