RoboChart: modelling and verification of the functional behaviour of robotic applications

Article *in* Software and Systems Modeling · January 2019

DOI: 10.1007/s10270-018-00710-z

6 authors, including:

Alvaro Miyazawa
The University of York
28 PUBLICATIONS **207** CITATIONS
SEE PROFILE

Pedro Ribeiro
The University of York
16 PUBLICATIONS **65** CITATIONS
SEE PROFILE

Wei Li
The University of York
19 PUBLICATIONS **234** CITATIONS
SEE PROFILE

Jon Timmis
The University of Sunderland
408 PUBLICATIONS **10,735** CITATIONS
SEE PROFILE

Software & Systems Modeling
https://doi.org/10.1007/s10270-018-00710-z

REGULAR PAPER

RoboChart: modelling and verification of the functional behaviour of robotic applications

Alvaro Miyazawa[1] · Pedro Ribeiro[1] · Wei Li[2] · Ana Cavalcanti[1] · Jon Timmis[2] · Jim Woodcock[1]

Received: 21 June 2018 / Revised: 21 December 2018 / Accepted: 21 December 2018

Abstract
Robots are becoming ubiquitous: from vacuum cleaners to driverless cars, there is a wide variety of applications, many with potential safety hazards. The work presented in this paper proposes a set of constructs suitable for both modelling robotic applications and supporting verification via model checking and theorem proving. Our goal is to support roboticists in writing models and applying modern verification techniques using a language familiar to them. To that end, we present RoboChart, a domain-specific modelling language based on UML, but with a restricted set of constructs to enable a simplified semantics and automated reasoning. We present the RoboChart metamodel, its well-formedness rules, and its process-algebraic semantics. We discuss verification based on these foundations using an implementation of RoboChart and its semantics as a set of Eclipse plug-ins called RoboTool.

Keywords State machines · Formal semantics · Process algebra · CSP · Model checking · Timed properties · Domain-specific language for robotics

1 Introduction

The current practice of programming robotic applications is often based on standard state machines, without a formal semantics or even precise syntax, to describe the controller only, with time properties discussed in natural language [76,78,95]. For analysis, simulation is often used during design to understand the behaviour of the controller for particular robots and environments. A state machine guides the development of the simulation, but only a loose connection between these artefacts is claimed. For implementation in a robotic platform, ad hoc adjustments to the deployed code are normally carried out to cater for the reality gap between the simulation, and the platform and actual environment.

Robotics can benefit from techniques used in modern software engineering, involving precise modelling and rigorous verification. Our goal is to support this agenda, and here we report on a key result: definition, formalisation, and application of a state machine based notation, called RoboChart, for the design of robotic systems. RoboChart is akin to informal notations in current use, but it is precise and specialised to enable automated reasoning , catering for proof of functional properties that can be specified as a refinement check, including, deadlock, livelock, and timelock freedom, for instance. Moreover, RoboChart enforces design patterns appropriate for robotics, where the physical robot is explicitly modelled in terms of only its variables, events, and operations. RoboChart also supports the definition of a dedicated library of components to aid the development of robotic applications. The core concepts of RoboChart are common to cyber-physical systems in general, but the terminology and support provided, such as libraries, examples, guidelines, and

Communicated by Dr. Jeff Gray.

✉ Alvaro Miyazawa
Alvaro.Miyazawa@york.ac.uk

Pedro Ribeiro
Pedro.Ribeiro@york.ac.uk

Wei Li
Wei.Li@york.ac.uk

Ana Cavalcanti
Ana.Cavalcanti@york.ac.uk

Jon Timmis
Jon.Timmis@york.ac.uk

Jim Woodcock
Jim.Woodcock@york.ac.uk

[1] Department of Computer Science, University of York, York YO10 5GH, UK

[2] Department of Electronic Engineering, University of York, York YO10 5DD, UK

Published online: 23 January 2019

simulation facilities, makes it distinctive as a domain-specific language (DSL) for robotics.

RoboChart can be regarded as a profile of UML state machines and their derivatives, enriched with facilities to define time properties. We adopt a minimalist core of the UML state machine notation to enable a simplified semantics suitable for automated reasoning, and add time primitives often demanded to model robotic applications. In the literature, there are descriptions of many general-purpose notations (CSP [85], timed automata [6], and others), but we propose specialisation to facilitate use by practitioners, optimised verification, and automatic generation of simulations.

In addition to state machines, RoboChart includes elements to organise specifications and foster reuse: constructs to model robotic platforms and their controllers. Communication between state machines is synchronous to model parallel threads of behaviour, while communication between controllers can also be asynchronous, like in actual implementations of robotic systems. Operations used in a state machine can be taken from a domain-specific API, or defined by another state machine as well as pre- and postconditions. The action language used in states and transitions is fully specified.

To specify budgets and deadlines for operations and events, directly as part of a state machine, we use time primitives. They are inspired by constructs of timed automata [2] and Timed CSP [88], but are included in a notation where, for example, states have actions and can call operations. Constraints can be specified in association with the relative-time elapsed since the occurrence of events or the entering of states. Timed automata and Timed CSP, for example, are alternatives to give semantics to RoboChart.

Here, we formalise the RoboChart semantics using CSP and its dialect, tock-CSP [85], tailored for modelling time. Via their CSP semantics, we provide support for verification of RoboChart models using the FDR [41] refinement model checker. FDR provides a high degree of automation for early validation of our semantics and has the additional advantage of supporting tock-CSP. Ultimately, however, the semantic underpinning that we envisage for RoboChart is Hoare and He's Unifying Theories of Programming [46] (UTP). This is motivated by a requirement for extensibility of RoboChart to tackle additional aspects of robotic systems, such as, probabilistic [100] and continuous behaviour [36]. Our use of CSP is primarily as a front end for a UTP theory [19], which defines a mathematical predicative relational model that supports verification based on theorem proving. Our choice of CSP as a front end is motivated by the availability of a powerful model checker for early validation. An encoding of CSP using the UTP is also available in the theorem prover Isabelle/HOL [37,71].

Use of RoboChart is supported by RoboTool[1]; it enables modelling, performs type checking and analysis of well-formedness, and automatically calculates CSP models. Here, we describe RoboTool and its design. It has been applied in several case studies from the literature, which we also discuss here.

We introduced RoboChart in [68]; there, we give an overview of its metamodel and four case studies. We briefly mentioned the semantics, but here give a full overview and formalisation. In [82], we presented the timed semantics, but not the complete semantics or a formalisation, not the complete metamodel and well-formedness rules, and not the design of RoboTool.

An early version of RoboChart was discussed informally in [51], where we approached the issue of automatic simulation generation. This important aspect of our work was partially addressed in [51] via a discussion of an initial version of our facilities for automatic code generation in C++. Our focus in this paper, however, is on the RoboChart notation, its semantics, and its tool. A revision of the early approach in [51] to sound simulation is ongoing work.

In summary, we address the problem of model-based design of robotic applications. Our contribution is a novel DSL for verification, namely RoboChart, which provides constructs to model real-time concurrent designs. It includes the notions of robotic platforms with (distributed) controllers, possibly with parallel threads of behaviour. Threads are modelled by state machines, distinctively, with powerful time primitives to capture budgets and deadlines. Most importantly, a formal semantics enables verification of RoboChart models.

Section 2 reviews related works. Section 3 describes RoboChart—its metamodel and well-formedness conditions—and presents an example. Section 4 describes the CSP semantics, abstracting away temporal aspects, and formalises it as functions from RoboChart to CSP models. Section 5 presents RoboTool and extra examples: a chemical detector [44], a transporter [20], and the alpha algorithm [10]. In Sect. 6, we focus on the temporal aspects of RoboChart, revisiting its metamodel, well-formedness conditions, and semantics. Finally, Sect. 7 concludes and describes our next steps.

2 Related work

Early efforts on verification for robotics apply existing mathematical techniques [30], and while there are many general-purpose languages for which model checking support is available (like C, for example), our goal here is a

[1] Available for download at www.cs.york.ac.uk/circus/RoboCalc/robotool/.

customisation to produce a simple language akin to what is already used by practitioners [14,25,77], with friendly support for graphical modelling, and optimisations in the semantics and verification that do not apply to arbitrary uses of general-purpose languages.

UML and its derivatives have been used for modelling in various application domains, from business to safety-critical systems. While UML has been given many formalisations, in general, only subsets of UML are covered. The works in the literature either define tailored semantic domains [12], or use existing techniques such as graph transformations [48] and CSP [22,81].

There are several general languages for architectural and behavioural modelling. Notable examples of widely used notations are SysML [73], AADL [32], and Focus [13]. For SysML, a comprehensive semantics in a CSP-like language is available [52]. For AADL, we are aware of a semantics that uses rewriting logic [75].

For Focus, the semantics is based on streams, which map time to messages; communication is asynchronous, but without delay. Refinement allows the introduction of new behaviours. This is different from the CSP models, based on atomic, instantaneous, synchronous events, with refinement captured by behaviour subsetting.

The AutoFocus [93] approach caters for the whole development process, from informal textual specification to code. This tool chain is similar to RoboTool in some ways, particularly with respect to its goals. On the other hand, where AutoFocus targets embedded software with behaviour defined by automata or functions, RoboTool focuses on robotic applications with behaviour defined by state machines. Verification in AutoFocus uses theorem proving with Isabelle/HOL. Semi-automatic model transformation encodes properties into temporal logic; the transformation generates a refinement of the original model, rather than encoding its semantics. So the properties of the generated model can be slightly different. AutoFocus also provides facilities for code generation. Work on support for proof and code generation from RoboChart is ongoing.

RoboChart is, however, distinctive as a small language, supporting a particular component model, and with a tailored semantics to enable sound generation of usable formal models . In our previous work [53,66], we have given semantics to comprehensive subsets of Stateflow [62] and UML state machines using CSP-based languages. The models, however, require further transformation to enable even simple checks. Results with RoboChart, on the other hand, point to tractable models that enable model checking directly. This is likely to facilitate theorem proving as well.

Nordmann et al. [72] suggest that domain-specific languages for robotics are becoming popular, providing extra motivation for our focus on a small DSL. While the majority of works aim at code generation for deployment or simulation, we target generation of mathematical models for verification. Some of the proposed notations and tools can be complemented by RoboChart models to support modelling and verification of the functional behaviour of components and systems.

RobotML [25] is a UML-based DSL for robotics for automatic generation of platform-independent code; reasoning about non-functional properties is envisaged but not available yet. In the same vein, Schlegel et al. [87] advocate model-based engineering for robotic systems using a UML-based framework, but without support for formal verification.

SafeRobots [79] is a general framework that supports a component-based approach, where components are defined using a data-flow architecture, and OCL is adopted for definition of properties. Specification of behaviour is via code from libraries.

The work in [47] covers architectural design and deployment. There is, however, no support for modelling behaviour, time properties, or verification.

The MontiArcAutomaton framework [83] provides extension and composition mechanisms for languages and code generators. They can accommodate use and integration of multiple modelling languages and generators, and support of heterogeneous target platforms. At its core, MontiArcAutomaton comprises an ADL based on components and connectors that allows extension with component-behaviour modelling languages. RoboChart, as a language based on components and connectors, could be integrated in this setting.

Yakindu[2] is an Eclipse-based tool for constructing UML statecharts, with support for code generation and animation. Like RoboChart, it supports the definition of events, variables, interfaces, actions, (timed) triggers, and operations. Its action language is comparable, but RoboChart supports rich data types based on Z [99]. Compared to Yakindu, in RoboChart operations can be defined abstractly, without determining specific behaviours, and capturing time budgets and deadlines, but optionally also as state machines. Similarly to RoboTool, Yakindu implements validation rules, but some of them are incomplete [3]. For example, states are deemed unreachable if there are no incoming transitions, but, if there are, path reachability is not taken into account. Yakindu could serve as a basis for implementing RoboChart, but this would require adapting Yakindu's metamodel to support specific elements of RoboChart, namely controllers and modules, and to restrict the use of features that hinder compositional semantic models. Yakindu does not currently have a public API that could support such a task.

FlexBE [86] is a behaviour engine for the Robot Operating System (ROS) that enables human operators to specify and observe a robotic system's behaviour, and if necessary inter-

[2] www.itemis.com/en/yakindu/state-machine.

vene at runtime, by pausing or modifying behaviours. These are specified by hierarchical state machines with actions implemented by Python classes. Composition is realised via code generation. Similar, but more abstract, models can be developed in RoboChart using shared variables and multiple state machines. FlexBE's tool implements validation rules, but does not support formal verification.

MissionLab [29] is specific for military applications. It provides support for end users, rather than developers, to specify behaviour as mission plans. A wizard allows the definition of mission parameters, like task, environment, possibility of presence of enemies, and so on. Lower-level behaviour is captured by simple state machines. Move operations may be specified on a map. A formal semantics and verification are not mentioned, but usability studies indicate ease of use. Such studies for RoboTool are not available yet.

There are also vendor-specific tools for developing controllers for robots like the NAO[3] and LEGO Mindstorms[4] EV3 provides a didactic visual programming environment for specifying behaviours using blocks, a precursor of the educational language Scratch [57]. Similarly, the NAO software provides blocks and parametrisation that is specific to the NAO hardware.

A rather different approach to the development of robotic controllers is the use of temporal logic formulae as a model [63]. For example SPECTRA [61] allows behaviour and assumptions about the environment to be modelled using patterns [58] of a subset of LTL for which there are efficient synthesis algorithms [11]. In practice, this also requires suitable discrete data type abstractions [60]. Time constraints, however, cannot be directly specified, and so the model needs to account for the target cyclic executive, which could have a fixed delay, or run as fast as possible. Empirical evidence [59] suggests that modelling of realistic environments can be challenging, as is establishing traceability for failures observed in deployments and in the case of infeasible designs. Simulation is not directly considered.

The approaches proposed in [30,33,39,42] are closest to ours in addressing aspects such as architectural design, concurrency, control of events, and verification. GenoM [39] supports verification of schedulability via translation to Petri Nets and model checking [9], and deadlock checking using BIP [5]. GenoM is an executable language (potentially including C code). RoboChart, on the other hand, is a self-contained modelling language supporting various levels of abstraction.

Mauve [42] supports component-based models with interfaces defined by constants, operations, and ports, but not shared variables like RoboChart. Behaviour can be defined by code or simple textual state machines. On the other hand, a contract language can be used to specify behaviour using temporal logic and observation points of the code or state machine. Code generation is supported for Orocos [92] platforms. Based on a deployment that associates tasks with components and the executable code, an optimised WCET analysis is used to ensure schedulability. Time properties are derived from this analysis, rather than specified like in RoboChart. Proof of properties uses TINA like [39].

The verification approach in [33] is based on an adaptive architecture and supports identification of optimal configurations based on various proof techniques including model checking. Verification of behavioural properties, however, is not the focus. Finally, Orccad [30] is a notation that supports modelling, simulation and programming, and verification of timed properties by translating models into formal languages. Orccad has limited support for graphical modelling, using constructs that are closest to those of RoboChart's semantics.

In summary, to our knowledge, most of the DSLs for robotics in the literature do not have a technique for proof of behavioural properties. They also provide limited or no support to deal with time.

As previously said, the RoboChart time primitives are inspired by timed automata and Timed CSP. Timed automata use synchronous continuous-time clocks, and properties expressed in temporal logic can be checked using the model checker UPPAAL. RoboChart, in contrast, provides abstractions specific for robotic applications and has a semantics for which there is a notion of refinement. UPPAAL is limited in its support for modelling state, with comparable models requiring additional states, interleaved automata, and definition of appropriate state invariants. Ongoing work, however, is exploring a RoboChart semantics using timed automata for the purpose of property verification.

A real-time extension of UML statecharts, Hierarchical Timed Automata (HTA), is proposed in [21]. In that work, statecharts are translated to timed automata for use with UPPAAL. Some of the restrictions on UML are similar to those of RoboChart; for example, neither notation includes inter-level transitions. On the other hand, HTA imposes severe restrictions on data, guards, and use of events, and has no support for operations. Roughly speaking, HTA is timed automata with hierarchy and history. On the positive side, the target timed automata models remain decidable.

UML [74] has a simple notion of time and does not provide appropriate facilities to model timed properties. Instead, a profile such as UML-MARTE [90] can be used, which features support for logical, discrete, and continuous time through the notion of clocks. Complex clock constraints may be specified using CSSL [56]. A constraint solver [24] exists to find solutions suitable for deployment. Support for specifying deadlines and the duration of behaviours is largely focused on particular instances of behaviour, as specified

[3] doc.aldebaran.com/1-14/software/choregraphe.

[4] lego.com/mindstorms/downloads/download-software.

through UML sequence and time diagrams. While it is possible to define the potential execution duration of a particular behaviour, it is not possible to define timed constraints in terms of transitions and states.

UML-RT, an extension to UML, focuses on the architectural description of systems by providing a clear separation of concerns using the notions of capsules, ports, and protocols. Capsules encapsulate finite state machines, while communication between capsules takes place through ports, whose valid communications are defined by protocols. A timing protocol can act as a timer by raising timeouts in response to the passage of a certain amount of time [89]. It is not obvious how timed constraints, such as deadlines, can be specified directly on UML-RT state machines using anything more precise than informal annotations.

In [80] UML-RT is given semantics in *Circus* without considering time. An extension to UML-RT is proposed in [1] with semantics given in CSP+T [101], an extension of CSP that supports the timing of events. Inspired by the constructs of CSP+T, in [1] annotations are added to record the occurrence time of events and constrain the occurrence time of subsequent events. Although some RoboChart time primitives are similar, we have a richer set of primitives.

RoboChart is mentioned in a comprehensive survey on formal specification and verification in robotics [31,55]. It highlights that model checking is the most prominent approach in the literature at the moment, and the importance of integrating formal methods. RoboChart supports verification by model checking, but the long-term plan is use of theorem proving to deal with larger models and collections. Future work will explore combined use of several verification approaches.

3 Language

The core of RoboChart is a subset of the UML state machine notation that excludes non-essential features that lead to an increase in the complexity of the underlying mathematical models. Parallel states are not available in RoboChart as they introduce difficulties both in the semantics and modelling practice. For instance, based on examples found in the robotics literature [76,78,95], it is not clear what form, if any, interaction between parallel states should take. On the other hand, we include architectural components, namely modules and controllers, which support the modelling of parallel behaviours in a customised and precise manner.

Controllers are collections of state machines that can communicate with each other through their events, or relay information through the events of the controllers. Similarly, modules are collections of controllers that may interact with each other. A module defines the boundaries of the robotic application modelled and specifies the interface of such application through a robotic platform, which provides an abstraction of the hardware in terms of variables, events, and operations.

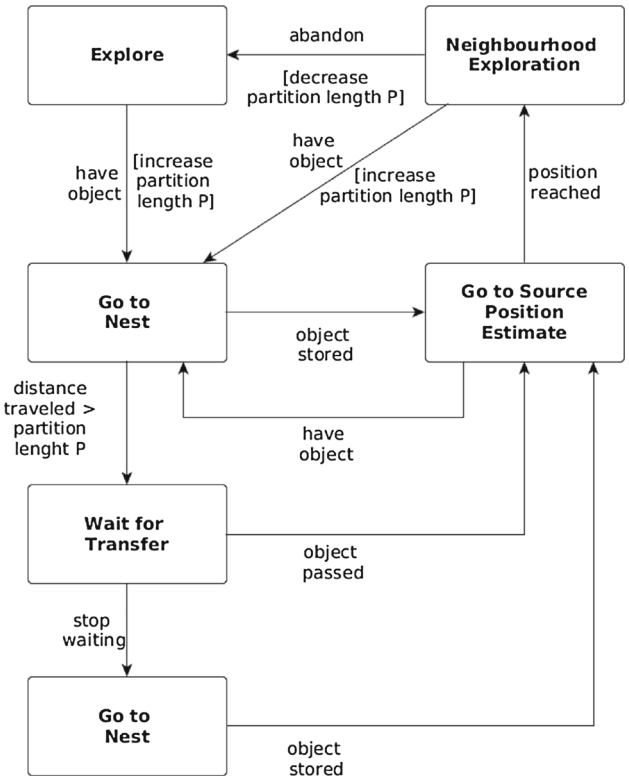

Fig. 1 State machine from [16]

In the following sections, we expand on the main constructs of RoboChart. In Sect. 3.2, we describe in more detail parts of the metamodel of RoboChart, and in Sect. 3.3, describe and motivate well-formedness conditions associated with RoboChart. First, however, in Sect. 3.1, we give a more complete, though informal, overview of RoboChart via an example.

3.1 Notation

In this section, we use the model of a foraging robot described in [16] to present RoboChart. The goal of this robot is to search an area (the source) for objects, collect one object, and deliver it to another area (the nest). It achieves this by coordinating with other similar robots to only traverse part of the arena. Essentially, a robot tries to find an object, collect it, and carry it up to a certain distance P (the partition length), where it will wait to transfer to another robot that will continue the task. Failures in locating objects reduces the partition length P, and success increases it.

In [16], the authors present the state machine in Fig. 1. Besides the fact that it cannot be analysed using a tool, because the notation is informal, this account leaves key questions open. For example, the initial state is not indicated, and it is not possible to establish when the robot gives up waiting

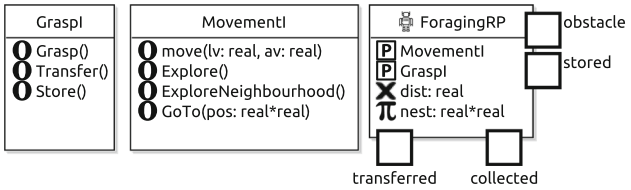

Fig. 2 Robotic platform for the foraging example

for a transfer, or when it abandons the neighbourhood exploration. By reading the English description of the robot, we have developed the precise RoboChart model in Fig. 3.

Robotic platform An important aspect of modelling a robotic application is understanding any assumptions related to the functionalities of the physical robot that are required. Such assumptions can be recorded in a RoboChart model via a robotic platform definition.

As already mentioned, a robotic platform abstracts a physical robot in terms of the functionalities it makes available for the controllers. These may be of three types: (a) variables, (b) operations, or (c) events. These elements can be either declared directly in the robotic platform, or grouped in interfaces. For our example, the robotic platform ForagingRP is shown in Fig. 2 alongside two interfaces GraspI and MovementI. The platform declares a number of events represented as boxes on the border of the platform, a variable dist of type real, a constant nest of type real*real (pair of real numbers), and provides two interfaces MovementI and GraspI. The first declares operations associated with movement and exploration, while the second contains operations associated with the mechanism for manipulating objects. The visible behaviour of the robot is precisely characterised by accesses to variables of the platform, calls to its operations, and occurrences of its events.

Different elements in a RoboChart model are often identified by icons. For instance, a variable declaration is identified by ✕, while a constant is indicated by π. Table 1 summarises the icons used in the RoboChart diagrams presented in this paper. For a comprehensive account of the RoboChart icons, we refer to [96].

Variables represent information the platform makes available to its controllers and may be used to share information among the controllers or to record information for the execution of basic operations of the hardware. For instance, it is common in wheeled robots for movement to be established through assignment of speed values to specific variables associated with the wheels. In such case, a robotic platform could abstract such a behaviour by simply providing the variables left_wheel: real and right_wheel: real , and, at this level, omitting any relationship between such variables and the hardware. In our example, the platform state contains a variable dist that records the distance travelled by the robot, and

Table 1 Summary of RoboChart icons

Symbol	Description
O	Operation declaration
P	Provided interface
R	Required interface
✕	Variable declaration
π	Constant declaration
	Module definition
	Robotic Platform definition
	Robotic Platform reference
	Controller definition
	Controller reference
	State machine definition
	State machine reference
	Clock declaration
	Initial junction
f_x	Function definition
	Precondition
	Postcondition

a constant nest that records the position of the nest in two dimensions. The variable dist is an abstraction for an odometer of the platform.

Operations represent functionalities provided by the physical robot. For instance, an alternative to the variables for the speed of the wheels is the definition of an operation move(ls:real,as:real), which, given the desired linear and angular speeds, causes the robot to move. This can be, in a loose sense, a more abstract approach to modelling; in this example, the variables left_wheel and right_wheel can indicate reliance on the fact that the robot is wheeled, while the operation move relies only on the fact that the robot is capable of moving forward and turning. In our model of the foraging robot, seven operations are provided by the platform. Grasp(), Transfer() and Store() provide functionality to manipulate objects, while the operations move(ls,as), Explore(), ExploreNeighbourhood(), and GoTo(pos: real*real) provide the means for the robot to navigate.

Finally, events model simple, atomic interactions between the robot and its environment. This feature is the most abstract of the three, and provides a means for the robot (and its controllers) to interact with the environment in a reactive manner. For example, obstacle detection is often implemented by reading the value of a sensor, analysing the results, and deciding whether an obstacle is present or not. When modelling this application, however, the only information of interest is the presence of an obstacle, since we are not concerned with properties of this algorithm. We can, therefore, abstract the reading and analysis of sensor information as a single event obstacle that only occurs when an obstacle is present. Furthermore, typed events, such as obstacle:real, can carry extra

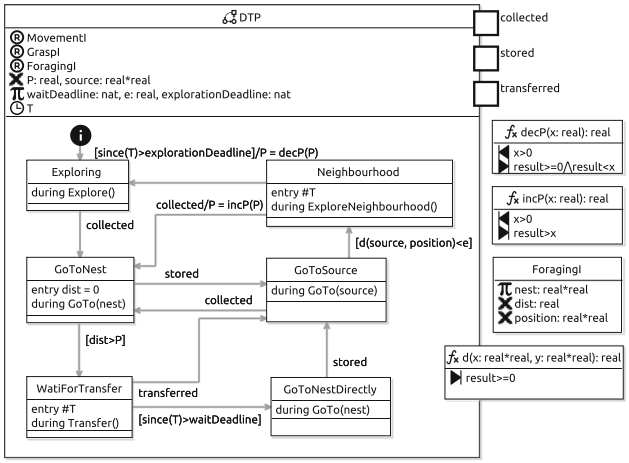

Fig. 3 Foraging state machine

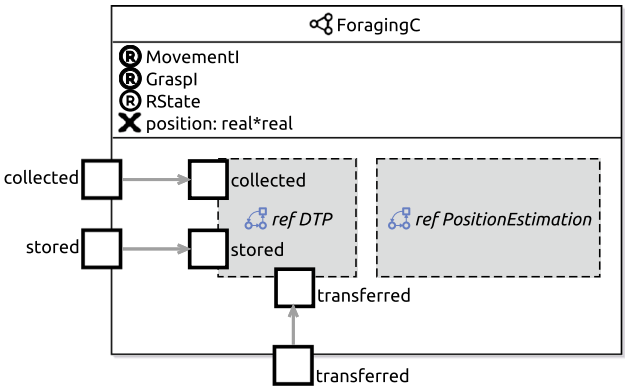

Fig. 4 Foraging controller

information such as the distance of the obstacle. In our example, the event obstacle is as discussed above, and the events collected, stored, and transferred mark the successful completion of the operations Grasp(), Transfer() and Store().

State machine A state machine is the main behavioural construct of RoboChart; Fig. 3 shows the machine DTP for our example that specifies foraging using dynamic task partitioning. A state machine contains states, junctions, and transitions.

Transitions connect states and junctions, which represent decision points. It is at states and junctions that the state machine decides how to proceed, that is, which transition, if any, to take next. The difference between states and junctions is their stability: while a machine may wait in a state for an event or condition before proceeding by taking a transition out of that state, a junction must be left immediately.

States are represented by rectangular boxes and may specify three types of actions: entry, during and exit actions, associated with the different phases of execution of the state. Additionally, a state may be composite, that is, it may itself contain junctions, states, and transitions. **Final states** are special types of state, drawn as a solid white circle with the letter F in the middle. It indicates termination of the containing state machine or state. We note that a final state is a state, not a junction, and therefore is stable.

Junctions are identified by solid black circles, and, as previously mentioned, represent a decision that must be made immediately. As a consequence, it must always be possible for a transition to be taken out of a junction. This leads to well-formedness conditions that forbid the use of a trigger in such an outgoing transition (that is, the decision cannot depend on an external interaction) and require that any conditions form a cover (that is, in all circumstances, at least one outgoing transition has a condition that is true). **Initial junctions**, represented by solid black circles with the letter i inside, indicate the starting point of the execution of a state machine or composite state.

Similarly to a robotic platform, a state machine may declare events and variables, but not operations. It can, however, require operations as illustrated in Fig. 3. The operations in the interfaces GraspI and MovementI are required by the state machine DTP. Additionally, it requires the interface ForagingI, which declares the constant nest, and variables dist and position. Both nest and dist are provided by the robotic platform, but position is not. This variable is provided by the controller that uses DTP shown in Fig. 4 described later.

DTP declares two local variables itself: P records the current length of the partition, and source, the estimated position of the source of objects. DTP also declares waitDeadline, e, and explorationDeadline, which are constants that specify how long to wait for a transfer, how close to get to the source, and how long to explore a neighbourhood. The time spent performing each task is recorded using the clock T, which can be reset using the statement #T in an action.

The state machine DTP consists of an initial junction and six states. As previously said, the initial junction determines the first state to be executed. In our example, this is the state Exploring, which executes the operation Explore() in its during action indefinitely. If an event collected occurs, indicating an object has been found and collected, Exploring is exited and the state GoToNest is entered. An entry action, executed upon entering a state, sets the variable dist to zero, and a during action calls the operation GoTo with parameter nest, that is, the target position to store the object.

From GoToNest, three possible behaviours can occur. If the nest is reached and the object is stored, indicated by the event stored, the DTP moves to the state GoToSource. If, while moving towards nest, the distance traversed (recorded in dist) becomes greater than P (the partition length), the state WaitForTransfer is entered. In the state GoToNest, we have a nondeterministic behaviour: after the entry action is executed, both transitions out of GoToNest become enabled if stored occurs and the condition dist>P is true. In this sce-

nario, the robot has traversed a distance larger than P (that is, the condition holds), but has arrived in the nest and stored the object. This nondeterminism is perhaps not obvious, but is present in the original model (see Fig. 1) and has been revealed following analysis of our RoboChart model. It is likely that it needs to be resolved in a simulation or deployment. Nevertheless, any properties of DTP remain true; however, the nondeterminism is resolved in a more concrete model.

In the state WaitForTransfer, the robot stops and waits to transfer the object to another robot. If a transfer occurs, indicated by the event transferred, the robot moves to the state GoToSource. If, however, the robot waits for longer than waitDeadline to transfer the object, it moves directly to the nest and stores the object (state GoToNestDirectly), before moving to GoToSource.

In GoToSource, the robot uses the operation GoTo to move to the position recorded in source, which estimates the location of the objects. If the distance between the current position and source is less than a constant e, the machine decides that the source has been reached and enters the state Neighbourhood. In this state, the clock is reset (#T) and then ExploreNeighbourhood() is called. If an object is found and collected (that is, collected occurs) the partition length P is increased using the function incP, and GoToNest is entered. Otherwise, if the robot has explored the neighbourhood for longer than explorationDeadline, then P is decreased using the function decP, and Exploring is entered.

This state machine uses three functions: d, incP, and decP. The first is a distance function whose only requirement is to return a value greater than or equal to 0; it can be implemented by the Euclidean distance, for example. The parameter of the other two functions must be positive, and their results must both change monotonically. These properties are specified with pre (◀) and postconditions (▶), as shown in Fig. 3. Although this is not a common practice in robotics, these simple specifications capture the properties of the two implementations experimented with in simulations reported in [16]. RoboChart does not require the definition of pre- and postconditions; we can leave the functions completely unspecified and provide an implementation just for verification, as is done with simulations.

Controller Several state machines can be encapsulated in a controller to model either different threads of execution or to capture independent functionalities. The controller, not a state machine, interacts with other controllers or directly with the robotic platform.

Similarly to a state machine, a controller may require operations and variables, and declare events and local variables. For example, the controller ForagingC shown in Fig. 4 encapsulates a reference to the DTP state machine, and a reference

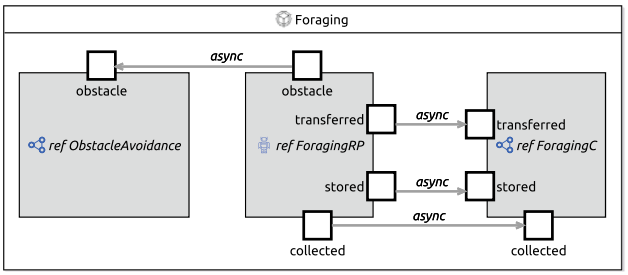

Fig. 5 Foraging module

to a state machine called PositionEstimation[5]. This second machine models functionality to estimate the position of the robot and record it in the shared variable position used by DTP.

The events of ForagingC are the same as those of DTP, and they are connected with a directional arrow, indicating that the events are received by the controller and relayed to the state machine. The names of the events, however, need not be the same. While the state machine PositionEstimation is not connected to either the controller or DTP via events, it can still interact with DTP through the shared variable position. Even though our approach encourages the use of events for interaction between parallel components, RoboChart permits the definition of shared variables due to their use in practice, particularly in the design of simulations.

Module The overall application is specified by a construct called module, which encapsulates a robotic platform and one or more controllers. In our example, the module Foraging is presented in Fig. 5. It includes references to the robotic platform ForagingRP, and the controllers ForagingC and ObstacleAvoidance (omitted here). The two controllers do not interact with each other, only with the robotic platform through the events obstacle, transferred, stored, and collected.

All communications with the robotic platform are asynchronous. This restriction reflects the fact that in a physical robot, represented by a robotic platform in RoboChart, interactions with the control software, represented by controllers in RoboChart, never block.

Modules differ from controllers in that they have a robotic platform, and characterise not only the software components (represented by controllers, and ultimately, state machines), but also requirements on the hardware and its built-in libraries (represented by a platform).

[5] Omitted here, but available at www.cs.york.ac.uk/circus/RoboCalc/case_studies/.

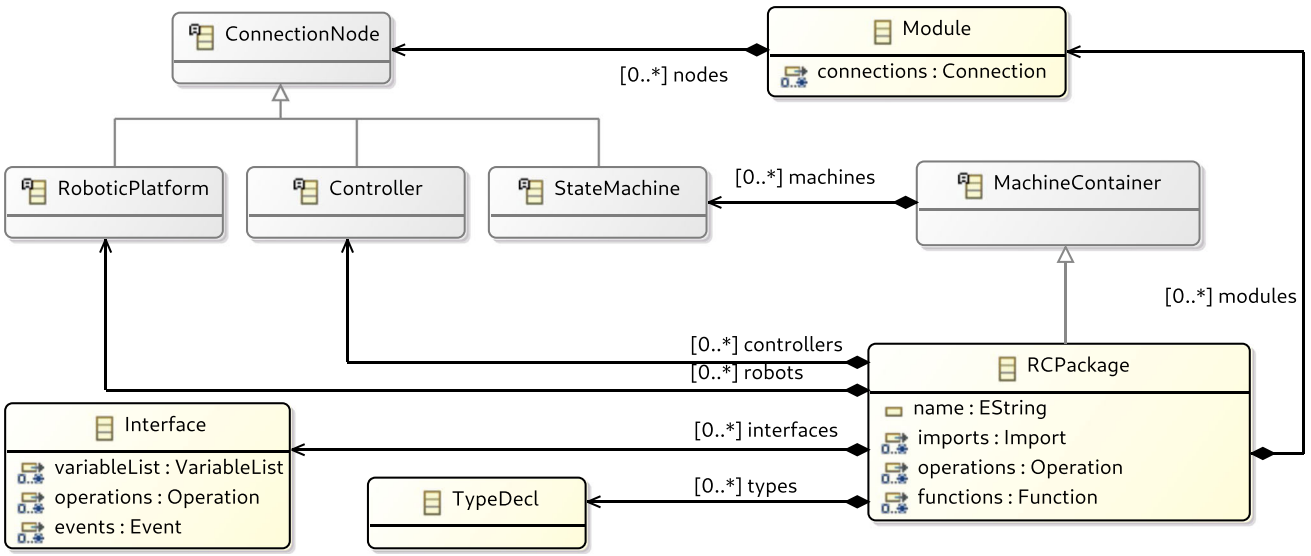

Fig. 6 Metamodel of RoboChart packages

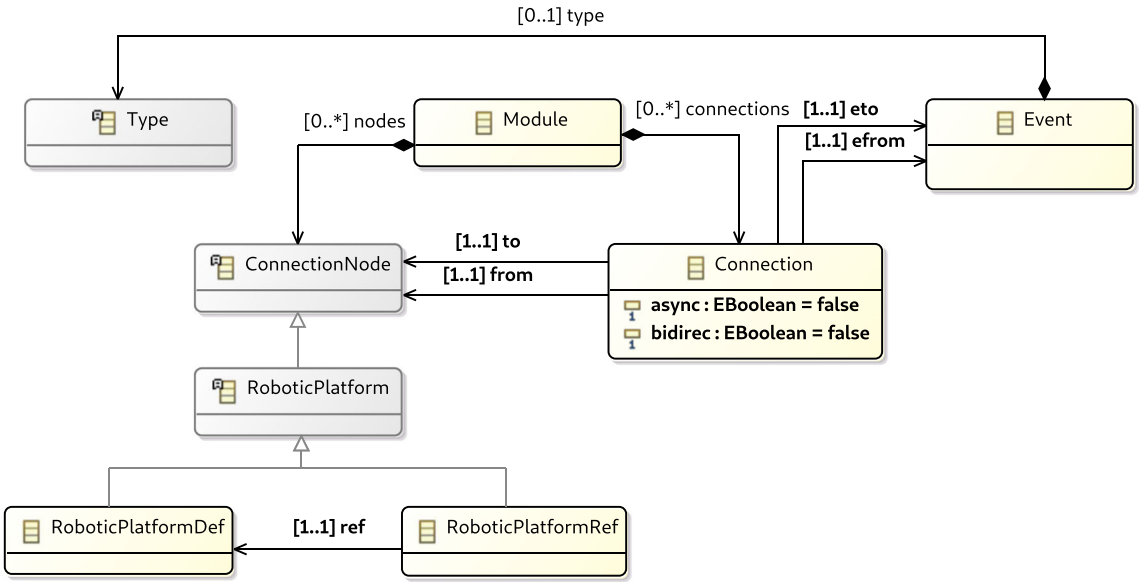

Fig. 7 Metamodel of RoboChart modules

3.2 Metamodel

The structure of RoboChart models is determined by a metamodel that specifies the types of constructs available and how they relate to each other. Here, we focus on the top-level structure of RoboChart models and on modules, controllers, and state machines. The complete metamodel with additional details is available in [96].

A RoboChart model is organised in packages, with their definitions shared using an imports mechanism similar to that of Java. Figure 6 defines a RoboChart package RCPackage. It has an optional name, and optionally imports other packages. All elements of a model are defined in a package. So, an RCPackage can include declarations of types, interfaces, modules, robotic platforms, controllers, and state machines.

RCPackages and ControllerDefs (in Fig. 8) contain a collection of state machines (machines). This feature is captured by the class MachineContainer, inherited by RCPackage and ControllerDef as shown in Figs. 6 and 8. While the two containment relations are realised in the same way in the metamodel, a machine contained in a controller is declared and used by that controller, but a machine contained directly in a package is just declared. Such machine can be used via references in various controllers. In any case, our semantics presented later characterises a state machine as a component that can be analysed in isolation.

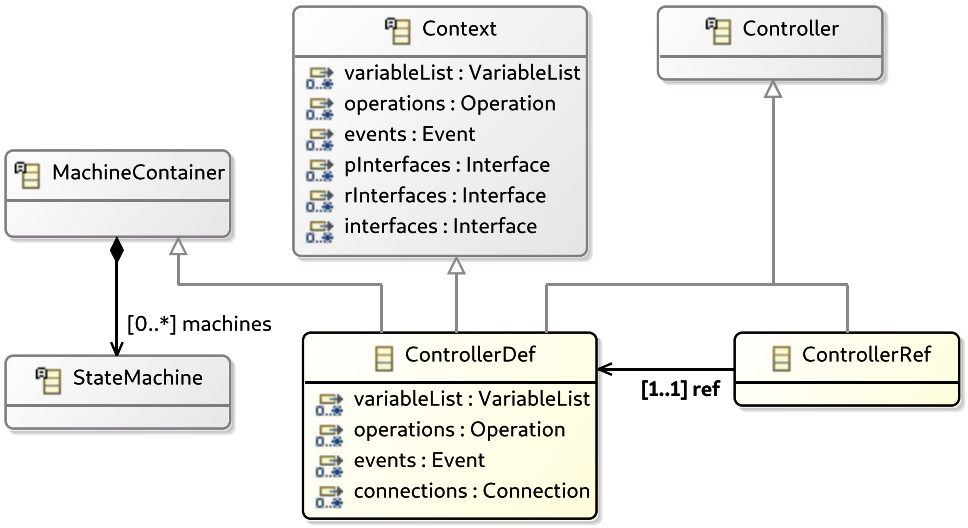

Fig. 8 Metamodel of RoboChart controllers

RoboChart is a typed notation, with types drawn from a set TypeDecls (type declarations, contained in the field types of an RCPackage). The type system is based on that of the Z notation [99]. A type declaration can be a given set [99] (that is, an abstract type without definition), an enumeration (free type), a record type (called Data Type like in UML, corresponding to Z schema types), a cartesian product, or a set. Unlike Z, RoboChart, through its library, provides a set of core given types: real, nat, int, boolean, and char. The Z mathematical toolkit, including definitions of data types for sequences, function, and relations, for example, is available in RoboChart. There is also a definition for a data type string, which is not available in Z.

The structure of a module is detailed in Fig. 7. It comprises a number of connection nodes and connections. The class ConnectionNode characterises the elements that can be connected through their events; they are: platforms, controllers, and state machines. Modules, however, cannot contain state machines directly. While the metamodel allows this, state machines are explicitly excluded via a well-formedness condition presented in the next section (see condition M1).

On the other hand, as shown later in Fig. 8, a controller can have several StateMachines connected among themselves. The RoboticPlatform is defined by a RoboticPlatformDefinition or by a RoboticPlatformReference. A RoboticPlatformReference is associated with a RoboticPlatformDefinition and allows the definition to be made independently of a particular module. In this way, it can be reused, and even provided in a library.

Connections are between a source (from) and a target (to) node, and in a module they establish the relationship between a platform and its controllers, and between the controllers themselves. Connections are established via a source (efrom) and a target (eto) event. They can be asynchronous and bidirectional, as indicated by the boolean attributes async and bidirec. An event may have a Type, which, if present, defines the values that can be communicated via the connection.

The structure of a Controller is shown in Fig. 8. It can be specified by a ControllerDefinition or a ControllerReference, which just names a controller defined elsewhere. A ControllerDefinition encapsulates any number of state machines. The class MachineContainer, which groups state machines, is inherited by both ControllerDef and RCPackage. The possibility, in the metamodel, for an empty set of state machines in a controller is useful in RoboTool to support partial models. In the case of RCPackage, it captures the fact that it is possible to have a package without state machines.

Lastly, a ControllerDefinition implements a Context, which defines the variables, including constants, operations, events, and provided, required, and defined interfaces of an element. Defined interfaces declare the variables and events that are used for the specification of behaviour; they are possibly shared if several elements are used to specify that behaviour.

Figure 9 shows the metamodel of state machines. Similarly to robotic platforms and controllers, a StateMachine is either a StateMachineReference or a StateMachineDefinition. A StateMachineReference points to a StateMachineDefinition, supports reuse, and is a key factor in allowing the specification of a library of modelling components. A StateMachineDefinition, on the other hand, fully specifies the behaviour of a state machine, which is captured by the class StateMachineBody inherited by StateMachineDefinition. A StateMachineBody is similar to a ControllerDefinition in that it is a Context, but it is also a NodeContainer, which records the nodes and transitions of the state machine.

A node can be either a State or a Junction, with FinalStates and InitialJunctions as special types of States and Junctions.

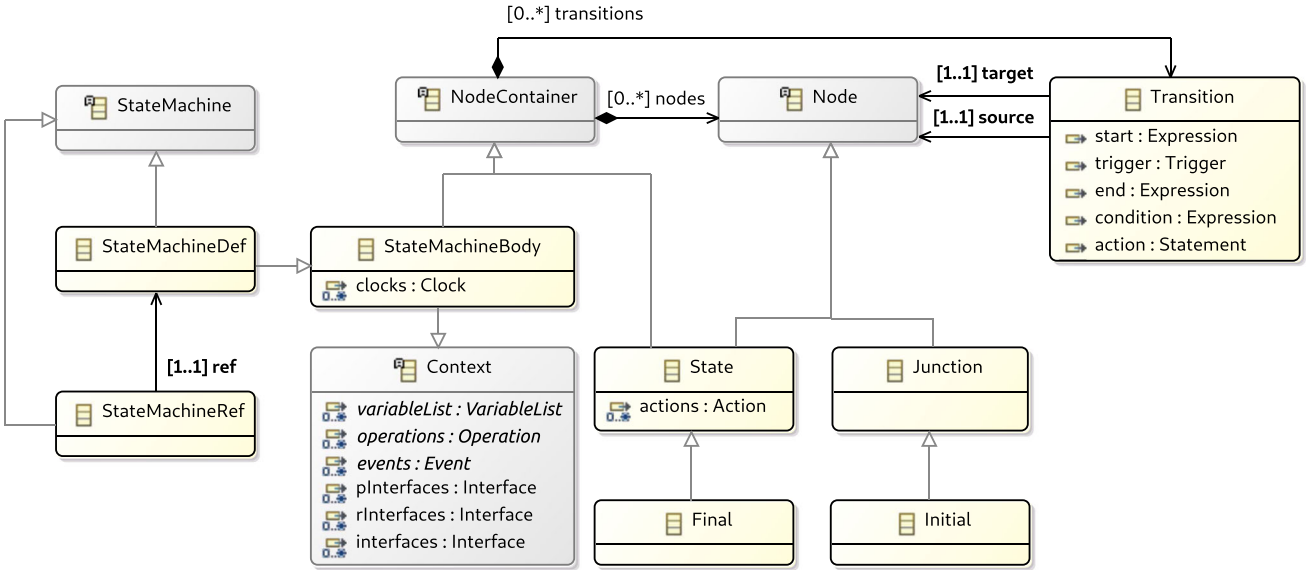

Fig. 9 Metamodel of RoboChart state machine

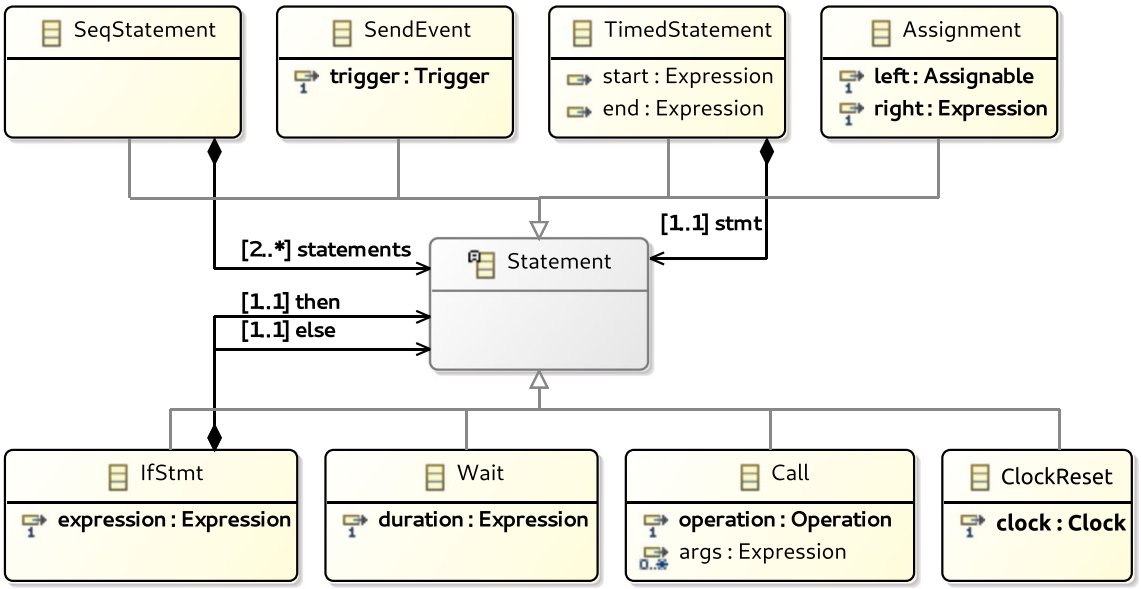

Fig. 10 Metamodel of RoboChart statements

A State is also a NodeContainer accounting for the possibility of hierarchical state machines. Finally, a Transition connects two nodes and can contain a number of features: a trigger and a condition describing when a transition can take place, an action describing the effect of the transition, and a deadline over the availability of the transition. (We envisage that an extension of RoboChart will also allow for ProbabilisticJunctions, and for a probability expression used in transitions starting in ProbabilisticJunctions.)

RoboChart specifies an action language to be used in the specification of the behaviours of state and transition actions. Figure 10 shows the possible statements of our action language. They include basic statements such as Assignment, which assigns a value to a variable, SeqStatement, which composes two or more statements in sequence, Call, which calls an operation with a sequence of expressions as arguments, and IfStmt, which conditionally executes one of two statements depending on the truth value of an expression. Additionally, the action language includes statements to interact with other parallel elements of the model using events (SendEvent), and timed primitives that allow time to pass (Wait), the specification of bounds on the start and end times of statements, and the resetting of clocks.

The expression language used in statements is inspired by Z and its mathematical toolkit. The syntax is omitted here due to its similarities to Z and for the sake of conciseness. A detailed account of the syntax and semantics of expression can be found in [96].

Not all models that respect the metamodel can be given semantics. For example, it is not possible to give a reasonable interpretation to modules with connections between events of different types. To assign meaning to RoboChart models, we must first identify the set of valid models. We do this by defining well-formedness conditions. This is the purpose of the next section.

3.3 Well-formedness

The well-formedness conditions include the typing and scope rules for expressions and actions, and unicity of names in all components. These conditions are standard and omitted here. Below, we concentrate on elements of larger granularity. A complete account of the well-formedness conditions can be found in [96].

Some conditions may have been avoided by a more strict metamodel that already enforces the restrictions. Our metamodel, however, allows us to record partial models. In addition, the emphasis on well-formedness conditions makes it possible to selectively enforce specific subsets of conditions based on the application envisaged for the models (for instance, model checking, theorem proving, or simulation). Finally, the conditions below are a more readable account of how RoboChart should be used than a concise metamodel.

Modules

M1 *A module must contain exactly one robotic platform, at least one controller, and no machines*. The platform describes the hardware on which the controllers are executed, and identifies the variables and operations it has available for use by the controllers. Additionally, through events, the platform provides points of interaction with the controllers. The behaviour of a module is defined by controllers, and not by state machines directly.

M2 *All variables and operations required by the module's controllers must be provided by the platform*. This guarantees that the module is self-contained, that is, the resources required by its controllers are provided by the platform. We note that, in particular, operations required by a controller cannot be provided by another controller, since there is no facility for remote operation calls in a robotic design.

Robotic platforms

RP1 *Robotic platforms cannot require interfaces*. Since a platform abstracts a self-contained hardware component, it cannot itself require any resources.

RP2 *Defined interfaces can only have events* since there is no point in platforms having local access to variables, because they do not specify a behaviour.

We note that variables and operations declared directly in the platform, outside an interface, are considered as if declared in a provided interface, for the reasons already explained above. Events declared directly in the platform, on the other hand, are defined.

Controllers

C1 *A controller must contain at least one state machine* so that it has a behavioural specification.

C2 *Controllers cannot provide variables or operations to other controllers*. As explained above, controllers are taken to model separate units of processing, and so may require variables and operations, or define its own for use by its own state machines, but not provide any themselves. As a consequence, a controller cannot have provided interfaces.

C3 *All variables required by the controller's state machines must be defined or required by the controller*. If a state machine assumes the existence of a variable, it must be defined by any component that contains the state machine, that is, it must be defined either by its controller, or indirectly provided by a platform associated with its module.

C4 *All operations required by the controller's state machines must be defined or required by the controller.* This is similar to the previous condition, except that definition of an operation entails not just a declaration, like in the case of a variable, but a local specification. So, an operation can be required by a controller, but cannot be just declared in the controller. If it is required by a machine in the controller, the operation must either be required (from the platform) or fully defined within the controller.

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

Interfaces Interfaces are meant to group sets of variables, operations, and events. They exist to foster reuse, but are also an important mechanism to describe dependencies. As explained previously, provided and required interfaces describe sharing, while defined interfaces simply declare its elements locally.

I1 *Provided and required interfaces contain only variables and operations* because these are the only elements that can be shared; events are locally defined and interaction is established using connections.

I2 *Defined interfaces can only have variables and events* because operations of a platform are always provided, in a controller they are either required or defined (but never just declared as explained above, and so cannot be in an interface), and in a machine, they are always required. We note that, if the defined interface is used in a platform, then, as stated by a well-formedness condition for robotic platforms presented above, it can actually only have events.

State machines The well-formedness conditions that apply to state machines encompass aspects such as usage of interfaces, declaration of variables, events, and operations, and specification of nodes.

STM1 *State machines cannot have provided interfaces.* As the smallest self-contained model components that encapsulate behaviour, machines do not contain independent elements that might be provided for separate use by other components.

STM2 *Operations in state machines can only be required, not defined*. Operations either describe an abstraction of functionality provided by the robotic platform or encapsulate behaviours for reuse. In the first case, operations can be left unspecified, but in the second case, they must be defined by a state machine. Since state machines cannot contain other state machines, there are no means for an operation to be fully defined in the context of a state machine, and, therefore, it cannot provide or declare operations.

STM3 *Every state machine must have exactly one initial junction*, which determines its starting state.

STM4 *State machines must contain at least one state* (possibly a final state), because junctions (including initial junctions) are not stable, that is, they are transitory decision steps towards entering a state. A state machine cannot, therefore, rest in the initial junction.

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

States and final states The conditions for states and final states restrict the usage of actions.

S1 *If a state has a non-empty set of nodes, then conditions STM3 and STM4 of* state machines *apply.*

S2 *A state has at most one of each type of action:* entry, during, *and* exit, because they define actions that specify behaviours that are executed in well-defined phases of the overall execution of a state.

S3 *Final states cannot be the source of transitions*, because they model termination of the behaviours of a state machine or composite state.

Junctions and initial junctions As discussed before, junctions are transitory steps towards a state. The well-formedness conditions associated with junctions aim to guarantee this transient nature.

J1 *A junction that is not initial must contain at least one outgoing transition*, since otherwise it would not be possible to reach a state from the junction.

J2 *The guards of the transitions out of a junction must form a cover*, that is, their disjunction is true to guarantee that it is always possible to take at least one transition out of a junction.

J3 *Transitions starting in junctions cannot have triggers*. Again, this condition guarantees that the state machine does not become stuck in a junction.

J4 *An initial junction must have exactly one outgoing transition*, and, because of the above conditions J2 and J3, does not have a guard or trigger.

Transitions

T1 *The source and target of a transition must belong to the same container.* This guarantees that there are no inter-level transitions. These are a common feature of state machine notations, but make the semantics non-compositional, because when an inter-level transition is taken, it does not affect the behaviour of only its source and target nodes, but also of all the substates (at any level) of their least common ancestor. A parent state of a state S is a composite state that includes S directly. The ancestor states include the parent, its parent, and so on, at all levels. A direct substate is called a child state.

The condition T1 is particularly important because it enables us to define a compositional semantics. This means that the semantics of a composite construct is a function of the semantics of its components. For example, the semantics of a controller is determined by the semantics of its state machines composed in such way as to allow communication between them.

Connections Modules and controllers contain connections. Our conditions restrict the types of the connected elements, the nature of the connections, and the types of the associated events, which must be the same.

Cn1 *Connections of a module must associate only events of the robotic platform and its controllers*. Connections in a module describe only the interactions of the immediate components of the module, that is, the platform and the controllers.

Cn2 *Connections with a robotic platform are always asynchronous*. Events in platforms are points of interaction with sensor and actuators of the hardware, which are asynchronous by nature. Our hardware abstraction does not provide means for interaction to be refused, only ignored. This is in line with existing robotics programming interfaces.

Cn3 *Connections of a controller must associate only its events and those of its machines*. As in the case of modules, connections in a controller describe only the interactions of its immediate components.

Cn4 *Connections must not associate events of the same component.* So, in particular, events cannot be connected to themselves. Events establish interaction with other components and the environment.

RoboChart has a rich expression language, with universal and existential quantifications, lambda expressions, and definite descriptions. These features have been added for use in the definitions of pre- and postconditions of functions and operations, but not in guards and statements. If needed, they can be used indirectly in these contexts via the definition of boolean-valued functions. In this way, expressions that can render the diagrams unreadable can still be effectively used.

As previously said, the expression languages are inspired by the Z notation and omitted here due to space limitations; its complete syntax is presented in [96].

The well-formedness conditions presented in this section are necessary for the untimed semantics discussed in the next section to be defined. Extra conditions are necessary for the timed semantics as discussed in Sect. 6 (and stronger restrictions are necessary to allow automatic generation of simulation code).

In the next section, we define the semantics of models that are well formed according to these rules.

4 Semantics

As previously mentioned, RoboChart models are endowed with a formal semantics. Our formalisation relies on the UTP framework, but we use CSP as a front end to the UTP to support early validation via model checking. In Sect. 4.1, we briefly introduce CSP and describe the operators used in our semantics. In Sect. 4.2, we provide an overview of our semantics, and in Sect. 4.3, we discuss its formalisation as a function from RoboChart to CSP models.

4.1 CSP

Communicating sequential process [45,85] (CSP) is one language out of a large family of specification notations for concurrent systems referred to as process algebras. This family includes notations such as CCS [64], Pi-Calculus [65], and ACP [8]. CSP is distinctive in its denotational nature, while CCS focuses on operational semantics, and ACP on algebraic semantics. The denotational models of CSP give rise to notions of refinement that are particularly useful when verifying properties and establishing correctness of implementations.

The central constructs of CSP are processes and channels. Processes can specify patterns of interactions, including aspects such as deterministic and nondeterministic choice, deadlock, and termination. Process definitions can be made via parallel composition of other processes, where interaction occurs through channels. The communications between parallel processes are instantaneous, atomic events and can carry values.

Table 2 gives the CSP operators used in our semantics. For each operator, it provides its symbol, name, and an informal description of its behaviour. Section 4.2 presents a number of examples of usage of CSP. We explain each of the examples as they appear.

4.2 Overview

The structure of our CSP semantics is sketched in Fig. 11. The semantics of modules, controllers, and state machines is given by processes. A module process is defined by the parallel composition of the processes for its controllers interacting according to the connections in the module, as well as a memory process recording the variables (and constants) of the robotic platform.

While the semantics presented in Sect. 4.3 produces a single CSP process definition, in examples we use process declarations to modularise it and improve readability. For example, the semantic function $[\![\_]\!]_{\mathcal{M}}$ presented in Sect. 4.3 specifies the definition below of the process *Foraging* for the module of our running example (Fig. 5), but not the declaration shown that names it. Here, we name this process and others to facilitate presentation. The process names that we use match applications of the semantic functions described in Sect. 4.3. The resulting declarations have a positive impact also in optimising both the generation and analysis of the models. Therefore, we implement them in RoboTool; this is further discussed in Sect. 5.1.

The fact that the semantics presented in Sect. 4.3 generates a single process is not detrimental to compositionality. Each semantic function is defined compositionally, that is, purely in terms of the semantics of the components of the RoboChart element that it defines. Furthermore, compositionality of

Table 2 Summary of CSP operators

Symbol	Name	Description
$Skip$	skip	Terminate immediately without any side effects
$Stop$	deadlock	Refuse all interactions, but does not change the state
$P \llbracket cs \rrbracket Q$	generalised parallel composition	Run P and Q in parallel synchronising on events in cs and terminate only when P and Q terminate
$P[cs_1 \parallel cs_2]Q$	alphabetised parallel composition	Run P, engaging in events in cs_1, and Q, in events in cs_2, in parallel, synchronising on the intersection of cs_1 and cs_2, and terminate only when P and Q terminate. The channel sets cs_1 and cs_2 are the alphabets of the processes P and Q
$P \mid\mid\mid Q$	interleaving	Run P and Q in parallel without synchronisation, but terminate only when both P and Q terminate
$\{\mid c \mid\}$	channel set	Set of all possible events corresponding to communications via the channel c
$c \rightarrow P$	prefix	Synchronise on channel c and then behave like P
$c?x \rightarrow P$	input	Synchronise on channel c with any possible value, store the chosen value in x, and behave like P
$c?x : S \rightarrow P$	restricted input	Synchronise on channel c with any value in S, store the chosen value in x, and behave like P
$c!e \rightarrow P$	output	Synchronise on channel c with value e and behave like P
$P \,;\, Q$	sequential composition	Behave like P, and once P terminates, behave like Q
$P \Box Q$	external choice	Allow the environment to choose between P and Q
$P \sqcap Q$	internal choice	Nondeterministically choose between P and Q
$P \triangle c \rightarrow Q$	interrupt	Behave like P with $c \rightarrow Q$ in choice, until P terminates or the choice is resolved in favour of $c \rightarrow Q$
$P \Theta_{cs} Q$	exception	Behave like P, until P raises an event in cs, at which point, behave like Q
$P \setminus cs$	hiding	Run P with events in cs hidden
(e) & P	guard	If e is true, behave like P, otherwise deadlock
$P[\![c \leftarrow d]\!]$	renaming	Rename the occurrences of event c to d in P
$\llbracket cs \rrbracket\ i : I \bullet P(i)$	replicated generalised parallelism	Run $P(i)$ in parallel for all i in I synchronising in cs
$\Big\|\Big\|\ i : I \bullet [A(i)]P(i)$	replicated alphabetised parallelism	Run $P(i)$ in parallel with alphabet $A(i)$ for all i in I, synchronising in the intersection of their alphabets
$\Big\vert\Big\vert\Big\vert\ i : I \bullet P(i)$	replicated interleave	Interleave $P(i)$ for all i in I
$⨟\ i : I \bullet P(i)$	replicated sequential composition	Run $P(i)$ in sequence for all i in I
$\Box\ i : I \bullet P(i)$	replicated external choice	Offers a choice of processes $P(i)$, where i is in I

refinement follows directly from compositionality of the CSP operators. For instance, if a state machine S of a controller C is refined by another machine S′, then S can be replaced with S′ to form a controller C′ that is a refinement of C.

Foraging defines our example module as the parallel composition of the processes for its controllers, namely ObstacleAvoidance and ForagingC, and a memory process *Memory_Foraging* for the platform ForagingRP. Synchronous connections between controllers are captured by a parallel composition ($\llbracket \ldots \rrbracket$) with the events involved in the connections hidden ($\setminus$) because these connections are not visible. Since, in our example, the controllers are not connected directly, in the composition of the controller processes the parallelism is an interleaving ($|||$) and there is no hiding.

$$
\begin{array}{l}
Foraging = \\
\left(\begin{array}{l}
\left(\begin{array}{l}
ObstacleAvoidance[\![\ldots]\!] \\
\quad ||| \\
ForagingC \\
\quad \left[\!\!\left[\begin{array}{l}
set\_FC\_dist \leftarrow set\_FRP\_dist, \\
set\_FC\_nest \leftarrow set\_FRP\_nest, \\
FC\_transferred \leftarrow FRP\_transferred, \\
FC\_stored \leftarrow FRP\_stored, \\
FC\_collected \leftarrow FRP\_collected
\end{array}\right]\!\!\right]
\end{array}\right) \\
\quad \llbracket \{\!\mid set\_FRP\_dist, set\_Ext\_FC\_dist, \ldots \mid\!\} \rrbracket \\
Memory\_Foraging(0)
\end{array}\right) \\
\setminus \{\!\mid set\_Ext\_FC\_dist, set\_FRP\_nest, \ldots \mid\!\}
\end{array}
$$

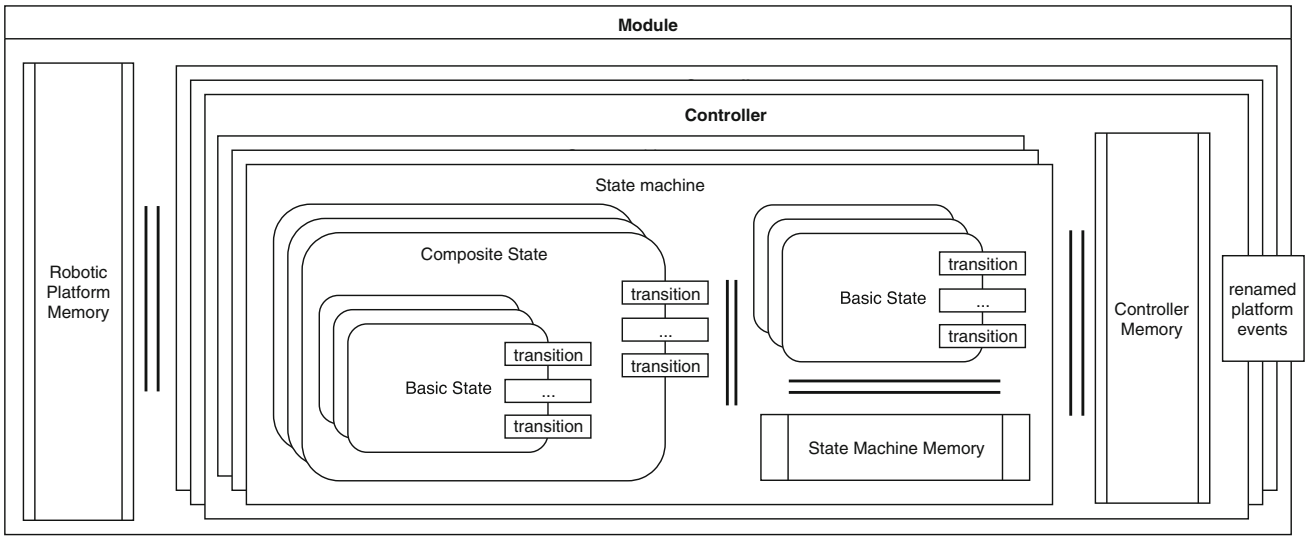

Fig. 11 Structure of the RoboChart semantics: stacked components and parallel lines indicate parallel composition; bordered boxes indicates points of interaction

Asynchronous connections, if present, are realised via a buffer modelled by another process composed in parallel with the composition of the controller processes. Interactions with the buffer are hidden.

When referring to RoboChart elements we use sans serif font, and use italics for CSP terms. For example, ObstacleAvoidance is a RoboChart state machine, and *ObstacleAvoidance* is the CSP process that specifies its semantics. Table 3 describes the names of elements (processes and channels) in our semantics.

The memory model of RoboChart is hierarchical, with a memory for the robotic platform at the top of the hierarchy, memories for the controllers at the next tier, and finally the memories for the state machines under those for their controllers. The memory for a robotic platform records its variables for sharing between controllers (and their state machines). In our example, the process *Memory_Foraging*(0) defined below models a shared memory recording the platform variable dist. The value of the constant nest is defined using the channel *set_FRP_nest*. Since a specific value is not determined in the RoboChart model, the communication accepts any possible value (and introduces a nondeterminism when *set_FRP_nest* is hidden).

The memory processes for a robotic platform or controller are slightly different from those for state machines. A process for a platform or controller not only accepts updates to the memory variables, but also propagates updates down the hierarchy to the memory of the state machines that require the updated variables. The memory of a state machine caches the variables it requires, so that the model of the machine itself is independent of the location of the variables that it uses, that is, the particular controller or robotic platform that provides the variables that it requires.

Table 3 Summary of naming conventions in RoboChart semantics

Name	Description
N_C	Qualified name of component C, where *N* is the qualified name of its immediate parent
set_C_v!e	Channel modelling assignment to variable v of component C
set_Ext_C_v	Channel modelling external assignment to shared variable v of component C
get_C_v	Channel modelling reading of value in variable v of component C
$enter.id_1.id_2$	Event marking the start of a state activation, where id_1 is the identifier of the state that requested activation and id_2 is the identifier of the state being activated
$entered.id_1.id_2$	Event marking the completion of a state activation
$exit.id_1.id_2$	Event marking the start of a state deactivation, where id_1 is the identifier of the state that requested deactivation and id_2 is the identifier of the state being deactivated
$exited.id_1.id_2$	Event marking the completion of a state deactivation
Memory_C	Process modelling the memory of a component C
S_main	Process modelling the behaviour of state S without its substates
S_R	Process modelling the behaviour of state S taking into account its substates, and restricting transitions of S that can no longer occur

The memory of a robotic platform is modelled by a recursive process that, at each step, accepts, for each variable *v*, updates to *v* through a channel *set_v*, and, for each controller C_i that requires *v*, propagates the new value through channels $set\_Ext\_C_i\_v$. There are no get channels, since, as mentioned

above, the values of the variables are cached in the memory processes for the state machines that use them.

The memory process for the platform in our example, namely, *Memory_Foraging*(*dist*), is as follows. It accepts a value x for dist through *set_FRP_dist*, and propagates it to *ForagingC* through *set_Ext_FC_dist*. ObstacleAvoidance does not require dist, so no additional propagation is needed.

$$
\begin{array}{l}
\mathit{Memory\_Foraging}(\mathit{dist}) = \textbf{let} \\
\quad \mathit{Memory}(\mathit{dist}) = \\
\qquad \mathit{set\_FRP\_dist}?x \rightarrow \\
\qquad \mathit{set\_Ext\_FC\_dist}!x \rightarrow \mathit{Memory}(x) \\
\textbf{within} \\
\quad \mathit{set\_FRP\_nest}?\mathit{nest} \rightarrow \mathit{Memory}(\mathit{dist})
\end{array}
$$

The parameter defines the initial value of dist.

In the definition of a module process, the platform-memory process is composed in parallel with the processes for the controllers synchronising on the *set* channels. In our example, *Memory_Foraging*(0) is composed in parallel with the parallel composition of the processes *ObstacleAvoidance* and *ForagingC*. They synchronise on *set_FRP* and *set_Ext_FC* events for dist. The *set* channels used to update the variables are visible, since they represent changes to attributes of the platform, but the *set* channels used to define the values of the constants and the *set_Ext* channels used just to define the internal propagation protocol are hidden.

Write access to a memory higher up in the hierarchy is accomplished by renaming ($[\![\ldots \leftarrow \ldots]\!]$) the *set* channels of a controller (or machine) process, when it is composed to define a module (or controller). In our example, the channel *set_FC_dist* used by the controller process *ForagingC* (defined below) to update the variable dist is renamed to *set_FRP_dist*, resulting in a process that interacts directly with *Memory_Foraging*.

The two channels *set_FC_dist* and *set_FRP_dist* for the same variable dist represent assignments in different contexts. The channel *set_FRP_dist* represents assignment to dist as a provided variable of the module with platform ForagingRP (abbreviated to *FRP*). In the process for the controller ForagingC (abbreviated here to *FC*), however, we do not use *set_FRP_dist* to avoid dependence between the semantics of the controller and that of the module where it is used. This allows independent definition and, therefore, analysis of the controller. On the other hand, when defining the module, the two channels are identified (via renaming and synchronisation) to guarantee that when the controller assigns a value to dist, it is captured by the module.

Renaming is also used to deal with connections between a controller and the platform. A controller event is uniquely identified in the semantics by a qualified name determined by the controller. If such an event is connected to an event of the platform, a renaming to the platform event models the connection. For example, the event transferred of the controller ForagingC, whose qualified name in our example is *FC_transferred*, is renamed to *FRP_transferred*, which is the qualified name of the event transferred of the platform. The same sort of renamings are applied to *ObstacleAvoidance*, but are omitted in the sketch above for simplicity.

The visible interactions of a module, represented by visible CSP events of the module process, correspond to updates to platform variables, via *set* channels, to events of the platform, when they are accepted by a controller, and to calls and returns to and from the platform operations. In our example, they are events that use the channel *set_FRP_dist*, the channels named after the events of ForagingRP (namely, *FRP_collected*, *FRP_stored*, and so on), and channels named after the operations in the interfaces GraspI and MovementI, provided by ForagingRP, with suffix *Call* or *Ret* to indicate an operation call or return (Fig. 2).

The semantics of a controller is the parallel composition of the processes for its state machines interacting according to their connections, and a memory process for the controller variables. This semantics is similar to that of a module, but the components are processes for state machines (Fig. 11) and the controller memory.

For instance, the process below for ForagingC is the parallel composition of a process that interleaves the behaviours of its state machines DTP and PositionEstimation, and its memory process *Memory_ForagingC*.

$$
\begin{array}{l}
\mathit{ForagingC} = \\
\left(\left(\begin{array}{l}
\mathit{DTP} \\
[\![\mathit{set\_DTP\_position} \leftarrow \mathit{set\_FC\_position}, \ldots]\!] \\
\quad |||
\end{array}\right)\right. \\
\quad \mathit{PositionEstimation}[\![\ldots]\!] \\
\left.\quad [\{\mathit{set\_FC\_position}, \ldots, \mathit{set\_Ext\_DTP\_dist}\}]\right) \\
\quad \mathit{Memory\_ForagingC}(0, (0, 0)) \\
\setminus \{\ldots\}
\end{array}
$$

Here, the machine processes are composed in interleaving because they do not interact directly via RoboChart events, only through the shared variable position. Like in the definition of a module, renamings deal with associations of local and shared variables and with connections of events. In the example, renamings are applied to the processes that model the individual state machines to associate local (machine) updates to the shared variable position into global (controller) updates, and to associate connected events of the machines. For instance, while the semantics of DTP uses the channel *set_DTP_position* to write to position, the composition of *DTP* in the model of the controller renames this channel to *set_FC_position*, thus allowing *DTP* to interact directly with the controller's memory process.

Like in a module process, the parallel composition of the parallelism of the state machine processes with the memory process synchronises on events that update and propagate changes to the variables. In addition, the channels used to update variables in the controller memory (*set_FC_position* in our example) and the channels used to propagate changes to the state machine memories are hidden.

The memory process for a controller is similar to that of a robotic platform, except that it must not only receive updates and propagate changes, but also relay propagations (of updates from the robotic platform to the state machines). In our example, the memory process for the controller ForagingC is shown below.

$$
\begin{array}{l}
Memory\_ForagingC(dist, position) = \textbf{let} \\
\quad Memory(dist, position) = \\
\qquad \left(\begin{array}{l}
set\_FC\_position?x \rightarrow \\
\quad set\_Ext\_DTP\_position!x \rightarrow \\
\quad set\_Ext\_PositionEstimation\_position!x \rightarrow \\
\quad Memory(dist, x) \\
\Box\ set\_Ext\_FC\_dist?x \rightarrow \\
\quad set\_Ext\_DTP\_dist!x \rightarrow \\
\quad set\_Ext\_PositionEstimation\_dist!x \rightarrow \\
\quad Memory(x, position)
\end{array} \right) \\
\textbf{within} \\
\quad set\_FC\_nest?nest \rightarrow Memory(dist, position)
\end{array}
$$

The required constant nest is set at the start. Since the controller declares a variable position and requires the variable dist, it behaves differently for each of these variables. The variable position is treated similarly to dist in *Memory_Foraging*: updates are accepted and propagated. The required variable dist, on the other hand, is not updated directly here. *Memory_ForagingC* simply relays values propagated by the robotic platform, received through the channel *set_Ext_FC_dist*, to any state machine that requires that value. Both machines of ForagingC require dist, so the value received is propagated through the channel *set_Ext_DTP_dist* to DTP and through *set_Ext_PositionEstimation_dist* to the state machine PositionEstimation. The order chosen in the semantics for propagation is arbitrary.

Below, we show the process for the machine DTP. It is a parallel composition of two processes. One of them models the behaviour of the state machine. It is itself defined by the parallel composition of a process *Init*, which describes the transition to the initial state, with the parallelism of processes modelling the states.

$$
\begin{array}{l}
DTP = \\
\left(\begin{array}{l}
\left(\begin{array}{l}
Init \\
\quad [\![EnterExitChannels]\!] \\
\left(\begin{array}{l} ExploringR \\ [\![Exploring\_S \cap \ldots]\!] \end{array} \right) \\
\ldots \\
\setminus\ (Exploring\_S \cap \ldots)
\end{array} \right) \setminus (\Sigma \setminus RoboEvents) \\
\quad [\{ set\_DTP\_P, \ldots, collected, \ldots \}] \\
Memory\_DTP(0, (0,0), 0, (0,0))
\end{array} \right) \\
\quad [\![collected.t0 \leftarrow collected, \ldots]\!] \\
\setminus \left\{ \begin{array}{l} set\_DTP\_P, set\_DTP\_source, get\_DTP\_P, \\ get\_DTP\_source, get\_DTP\_dist, \\ get\_DTP\_position, get\_DTP\_nest \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
Init = enter.DTP.Exploring \rightarrow \\
\qquad entered.DTP.Exploring \rightarrow SKIP
\end{array}
$$

Memory_DTP models the state machine memory; it records its local variables (and constant) and caches any required variables. *RoboEvents* is the set of all CSP events representing visible interactions of the machine, namely RoboChart events, accesses to shared variables, and operation calls and returns. All CSP events (Σ), except those in *RoboEvents*, are hidden. *Memory_DTP* synchronises with the process that defines the behaviour of the machine on the *set* and *get* channels of all variables, and the events of the machine. The events are renamed to remove transitions identifiers, and the *get* and *set* channels are hidden, except for the *set* events of the shared variables: dist and position in our example.

CSP events are used to model the control flow defined by entering and exiting states via the channels *enter*, *entered*, *exit*, and *exited*. They model the beginning and end of these phases: entering or exiting a state is only completed when the entry and exit actions are finished. This has an impact on availability of transitions; for instance, the transitions of a state are only possible once that state is entered. So, we use the channel *enter* to start the entering stage, and *entered* to signal its completion; similarly for *exit* and *exited*. Each channel takes two parameters: the component that has requested the action to start and the target of the request. For instance, in *Init* above, *DTP* itself requests that the state *Exploring* is entered using the event *enter.DTP.Exploring*. This event synchronises with the event *enter?s.Exploring* offered by the process *ExploringR*, which models the state Exploring. The synchronisation instantiates *s* as *DTP*.

A process that models a state does so compositionally, capturing only information about the state itself, irrespective of the context (composite state or state machine) where it occurs. In general, a process for a state S may have two components: processes *S_main*, modelling the behaviours of S, and *S_ch*, capturing the behaviours of the children states, if any. In this view, a state is potentially itself an indepen-

dent software component that we can consider separately in verification.

S_main has the form sketched below.

$$
\begin{array}{l}
S\_main = \\
\quad enter?s.SID \rightarrow \\
\quad \left(\begin{array}{l}
entry; \\
enter.SID.SSID \rightarrow entered.SID.SSID \rightarrow \\
entered.s.SID \rightarrow \\
\quad \left(\begin{array}{l}
during;\ STOP \ \triangle \\
\quad \left(\begin{array}{l}
transitions\_of\_S \\
\Box \\
all\_other\_transitions\_S
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
$$

This process uses the identifier *SID* of S, and, for a composite state, the identifier *SSID* of the state targeted by its initial junction. *S_main* accepts communications over *enter* that request entry to S, executes its *entry* action, requests activation of *SSID*, waits for it to be completed, that is, for that state to be *entered*, acknowledges entry to S, and executes its *during* action while offering a choice of events that trigger transitions. If a transition is taken, the *during* action is interrupted ($\triangle$). To cater for a during action that terminates, the process *during* is followed by the deadlocked process *STOP*. This ensures that the interruptions arising from the transitions are not discarded by the termination, and remain available for as long as the state is active.

The transitions offered are those of S (modelled by processes denoted by *transitions_of_S* in the sketch above) and all possible transitions, that is, transitions with all possible valid triggers from and to all possible states, whether in the diagram or not, except those from S and its substates (modelled by processes denoted by *all_other_transitions_S* above). These are all the transitions that, if taken, lead to an exit of S, like those of an ancestor of S. The transitions of the substates of S are the only ones not considered here, because they cannot lead to an exit of S, and, therefore, interruption of the during action, since there are no inter-level transitions.

The role of the processes in *all_other_transitions_S* is to capture the possibility of a transition of an ancestor state of S interrupting its execution without losing compositionality in the model. We do not consider specifically the transitions of the states where S occurs. The definition of *S_main* does not depend on the particular transitions of the ancestor states of S: it accepts any transitions, not only its own, except those of its substates. Since there are no inter-level transitions, the transitions in the substates of S are dealt with separately, in the processes for these substates themselves. As explained, these transitions are not modelled in either *transitions_of_S* or *all_other_transitions_S*.

For illustration, Fig. 12 presents part of a diagram showing a composite state S that is itself part of a composite state PS with transitions numbered. Transitions like (3), from S, are modelled in *transitions_of_S*, the transitions like (4), (5), (6), (7), and (8) are modelled in *all_other_transitions_S*, and those like (1) and (2), between substates of S, are not modelled in either *transitions_of_S* or *all_other_transitions_S*.

Of course, as part of the behaviour of S, transitions from states that are not related to S, that is, not substates nor ancestors of *S*, are never taken. If S is the current state, those transitions are not actually available. Moreover, transitions that are not in the diagram also are obviously never taken. These transitions in *S* that cannot be taken are restricted as part of defining the process for the parent state PS as explained later in this section. The definition of *S*, however, does not depend on the identification of the transitions of its ancestors or even of the machine as a whole.

The transitions modelled in *transitions_of_S* and *all_other_transitions_S* are offered in choice ($\Box$). In our example, the process *Exploring* for the state of the same name accepts the event *collected*, which is associated with its own transition, but also all other possible transitions whether in the diagram or not.

The semantics of a transition with identifier *tid* and trigger e?x, receiving a value x, with a guard g, and with an action tact, possibly defined in terms of x, from the state S to another state R, with identifier *RID*, is captured by a process *T* of the form indicated below.

$$
\begin{array}{l}
T = e.tid?x \rightarrow exit.SID.SID \rightarrow \\
\quad exit.SID?s \rightarrow exited.SID.s \rightarrow eact; \\
\quad exited.SID.SID \rightarrow tact; \\
\quad enter.SID.RID \rightarrow entered.SID.RID \rightarrow S\_main
\end{array}
$$

T is composed in choice with similar processes for each transition of S to define the process indicated in the sketch of *S_main* by *transitions_of_S*.

In *T*, if *e* occurs, then exiting of *S* is indicated by *exit*.*SID*.*SID* and *exited*.*SID*.*SID*. In between, if S is a composite state, we have the request from *S* for its active child state (with identifier *s*) to *exit* and, after the confirmation via *exited*.*SID*.*s*, the execution of the exit action *eact* of S. Afterwards, the transition action *tact* is executed, and then there is a request for R to be entered using *enter* and *entered*, before recursing back to *S_main*. This recursion makes the behaviour of S available again; it can once again be requested to enter. The guard g is not modelled in *T*, but in the memory process for the machine discussed later.

Since, in the context of a transition for a state S, the decision to exit S comes from that state itself, in *exit*.*SID*.*SID* and *exited*.*SID*.*SID* both identifiers are that of S, that is, *SID*. In general, however, these channels are used to accept any requests to, and acknowledge, exit from the state. So, we need two identifiers. For instance, when S, as a parent state, asks a child state to exit, the values of these identifiers are different as shown in the sketch *T* itself.

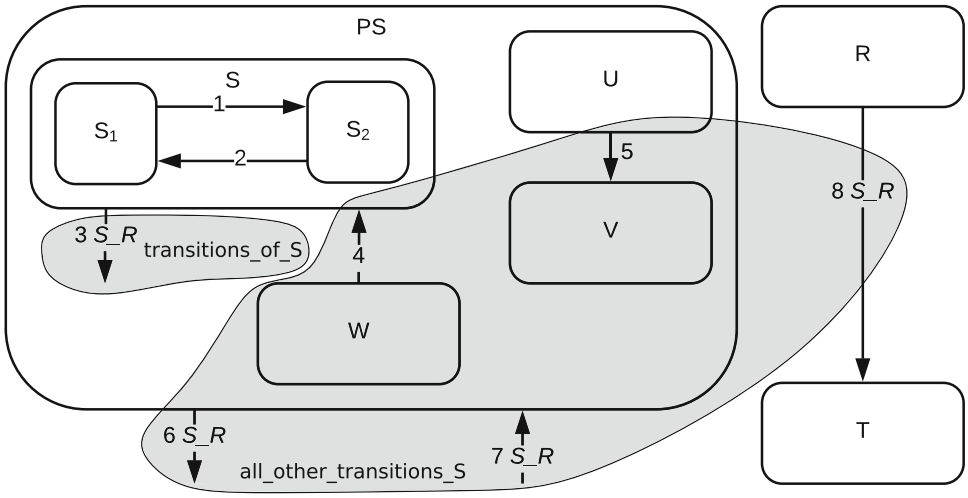

Fig. 12 Example of a partial diagram with collections of transitions identified. The shaded areas represent transitions modelled by the processes *transitions_of_S* or *all_other_transitions_S* in our sketch of *S_main*

The events *exit.SID.SID* and *exited.SID.SID* enables validation of a state machine by analysis of its internal control flow. We can use *exited* events to analyse, for example, the time spent in a state, by considering a version of the state machine process where such events are visible. This kind of validation is discussed later in Sect. 6.3.2. (We observe, however, that in the overall semantics such events are not visible, and so properties specified in terms of these events are not expected to be preserved by refinement. For example, in a refinement, states may even be removed or added as long as the externally observable behaviour is correct).

The processes T_O in *all_other_transitions_S* are similar. In this case, the request to exit comes from another state *as*. In addition, no account of the transition action is given, and no new state is entered, because the interruption here is associated with transitions other than those of S and of its substates. The control flow for these other transitions is handled in the processes for their source state, if any, as further discussed below.

$$\begin{aligned} T_O = \; & e.tid?x \rightarrow exit?as.SID \rightarrow \\ & exit.SID?s \rightarrow exited.SID.s \rightarrow eact; \\ & exited.as.SID \rightarrow S\_main \end{aligned}$$

In specifying the semantics of a parent state PS, from the state process *S*, we need to define a more restricted process *S_R* that excludes transitions from a sibling state of S. (A sibling is a state that has the same parent.) This is because it is the model for the parent state that specifies the transitions available between its children. So, the availability of transitions in *S* that are actually controlled by one of its sibling states needs to be blocked. In our example diagram sketch in Fig. 12, the process *S_R* still captures the transitions (3), (6), (7), and (8) like *S_main*, but not (4) and (5), which are for its sibling states, and still not (1) and (2). Since we define *S_R* as a component of the process for its parent state (if any), we still do have a compositional definition. In our example, for instance, the model for *DTP*, uses the restricted version of the state processes: *Exploring_R*, *GoToNest_R*, and so on.

In general, a process *S_R* is defined as follows.

$$S\_R = \left(\begin{array}{l} S \\ [\![\alpha all\_other\_transitions\_S \setminus \alpha all\_transitions\_PS]\!] \\ SKIP \end{array} \right)$$

Here, we use $\alpha all\_other\_transitions\_S$ to denote the set of events for the transitions captured by the process denoted by *all_other_transitions_S* in *S_main*. Similarly, $\alpha all\_transitions\_PS$ contains the events for the transitions of PS, including those whose semantics is captured in *transitions_of_PS* and *all_other_transitions_PS*.

The parallelism with *SKIP* blocks the transitions in the synchronisation set. To explain this definition, we observe that *PS* itself captures transitions as indicated in Fig. 12 for any state. So, in this example, the set of transitions modelled by *all_transitions_PS* includes (6), (7), and (8). These are the transitions not to or from a substate of *PS*. In *S_R*, we block the transitions captured by *all_other_transitions_S*, that is, those that are not from S or its substates, that are not captured by *PS*. These are exactly the transitions that are not from S and are from a substate of PS. In Fig. 12, *S_R* captures the transitions like (3), (6), (7), and (8). The transitions like (4) and (5), originating from a sibling state of S in PS, are blocked. They are captured by the processes for those sibling states.

This restricted process *S_R* is used to complement the model of PS, defined by the process *PS_main* of the form sketched above, with a model *PS_ch* of its children. For the example in Fig. 12, *PS_ch* is defined in terms of *S_R*, *U_R*, *V_R* and *W_R*. In general, for a state S, if S does not have any substates, like Exploring, the overall model *S* of S is just the process *S_main*. If, on the other hand, S does have

substates, its model is a parallel composition between *S_main* and another process *S_ch* that models the children (Fig. 11). The transitions captured by *S_ch* are all those captured by *S_main*, plus those among the children (but not those among further nested substates that might exist) of S.

The parallelism between *S_main* and *S_ch* captures the transitions modelled in *S_main* and the additional ones in *S_ch*. Synchronisation is required on the transitions captured by both processes, namely, those of *S_main*. For S in Fig. 12, this parallelism captures transitions like all those shown: (1) to (8). The behaviour for transitions like (1) and (2) is defined solely by *S_ch*, while that for the other transitions it requires agreement. The specific behaviour of a transition like (3) is defined in *S_main*, by a process like *T* above. In *S_ch*, that transition is enabled without further restrictions. It is considered in *S_ch* only because it is allowed by the definitions of S_1 and S_2, which are independent of the context defined by S as their parent state.

S_main and *S_ch* also synchronise on the flow events *enter*, *entered*, *exit*, and *exited* that target a child state, but are not from another child state. This ensures that entering and exiting of the children states that are not requested by another child are requested by *S_main*.

S_ch is a parallel composition of the restricted processes for S's children synchronising on the flow events. Use of the restricted processes ensures that the model of each child does not affect the transitions from its sibling states. Synchronisation on the flow events captures the sequential flow among the children states.

Memory_DTP(*P*, *source*, *dist*, *position*), the memory process for the state machine DTP, is below.

$$
\begin{array}{l}
Memory\_DTP(P, source, dist, position) = \textbf{let} \\
\quad Memory(P, source, dist, position, nest) = \\
\quad\quad \left(\begin{array}{l}
set\_DTP\_P?x \rightarrow \\
\quad\quad Memory(x, source, dist, position, nest) \\
\Box\ get\_DTP\_P!P \rightarrow \\
\quad\quad Memory(P, source, dist, position, nest) \\
\Box\ set\_DTP\_dist?x \rightarrow \ldots \\
\Box\ get\_DTP\_dist!dist \rightarrow \ldots \\
\Box\ set\_Ext\_DTP\_dist?x \rightarrow \ldots \\
\Box\ get\_DTP\_nest!nest \rightarrow \ldots \\
\Box\ (dist > P) \& internal.t2 \rightarrow \ldots \\
\Box\ \ldots
\end{array} \right) \\
\textbf{within} \\
\quad set\_DTP\_nest?nest \rightarrow \\
\quad\quad Memory(P, source, dist, position, nest)
\end{array}
$$

While a state machine can write directly to the memory of the controller, it does not read values directly. The state machine memory process keeps a copy of the required variables, and accepts updates that keep their values synchronised through the use of *set_Ext* events. Our example above is for a process that manages four variables: local variables P and source, and required variables dist and position. For P and source, we have *set* and *get* channels. For dist and position, we in addition have *set_Ext* channels used by the controller to update the variables. The value for the required constant nest is defined like in the memory processes for platforms and controllers, but, unlike in those cases, the value chosen is passed to the local recursive process *Memory*, where it is offered through a *get* event. We note that *Memory_DTP* is agnostic to the particular controller that uses this machine and may actually hold the values of the required variables.

A state machine memory process also models the evaluation of guards, this is captured by extra processes combined in the choice above. For instance, the transition from GoToNest to WaitForTransfer in Fig. 3 has a guard [dist > P]. As explained, this transition is modelled as part of the semantics of the state GoToNest by the process below, where, we do not model the guard, and since the transition is not associated with an event, we use a channel *internal* for its trigger.

This process waits for a synchronisation on the channel *internal*, which is later hidden. The parameter of *internal*, *t*2, is the identifier of the transition. Next, the process indicates that the state is being exited using the channel *exit*; the parameters are the identifier *GTN* of GoToNest and indicate that this state is exiting itself. Since there are no substates and no exit actions, the exiting terminates immediately, which is indicated by the channel *exited*, and the entering of the state WaitForTransfer with identifier *WFT* is requested using the channel *enter*. Finally, the transition waits on the channel *entered* for the state WaitForTransfer to finish entering and recurses to the process *GoToNest*, which models GoToNest when inactive.

$$
\begin{array}{l}
internal.t2 \rightarrow \\
\quad exit.GTN.GTN \rightarrow exited.GTN.GTN \rightarrow \\
\quad enter.GTN.WFT \rightarrow entered.GTN.WFT \rightarrow \\
\quad GoToNest
\end{array}
$$

As illustrated, a transition process does not evaluate any guard. This is done by the machine memory process. In our example, *Memory_DTP* includes a choice $[dist > P] \,\&\, internal.t2 \rightarrow Memory\_DTP(\ldots)$. The guard is re-evaluated on every memory update, and the result restricts the occurrence of the transition in the semantics of GoToNest by making the communication on *internal*.*t*2 available or not.

The process *S* for a state gives an independent and compositional account of its behaviour. A state in a hierarchical machine can potentially model a significant component of the system [15]. Our approach facilitates the definition of a refinement technique for state machines like that in [67] for SysML, and enables compositional verification of states. For

instance, if a state S is shown to be refined by a state T, any state machine that contains S is guaranteed to be refined by the same state machine with T substituted for S. Compositionality is particularly relevant for us in the long term, as we are interested in refinement techniques to support proof of correctness of software.

The models just described can be automatically generated by our RoboTool presented in Sect. 5.1. For that, RoboTool implements the transformation rules that formalise our semantics, which we describe next.

4.3 Formalisation

The semantics of RoboChart is formalised by a set of functions from RoboChart to CSP models. The main function $[\![\_]\!]_{\mathcal{M}}$ is for modules; it is shown in Fig. 13. It takes a value of the type *Module* (see Fig. 7) as an argument, used in a where clause to determine its robotic platform (*rp*), controllers (*ctrls*), connections (*cons*), the set of asynchronous connections not involving the platform (*asyncs*), and the set of events of asynchronous connections (*evasyncs*). The set *asyncs* of asynchronous connections is defined by a set comprehension as the set of all connections c of the module m, such that c is asynchronous ($c.async$) and none of its ends ($c.from$ and $c.to$) is the platform. The set *RoboticPlatform* is the syntactic category of models of robotic platforms defined in the metamodel. The set *asyncs* is used to calculate the set *evasyncs* of identifiers for the events associated with asynchronous connections. This set includes the identifier of the source ($c.efrom$) and target ($c.eto$) events of the connections. These unique identifiers are determined by the function *eventId*.

Set comprehensions are often used in our semantics to define sets succinctly. The notation for set comprehensions we use is that adopted in Z. For example, $\{x : T \mid P(x) \bullet f(x)\}$, where x is a variable name, T is a set, P is a predicate, and f is a function, denotes the set of values obtained by applying the function f to all values x in T such that $P(x)$ is true. Both the predicate and function parts of a set comprehension can be omitted, in which case they are interpreted as the value *true* and the identify function, respectively.

The result of $[\![\_]\!]_{\mathcal{M}}$ is a value of type CSPProcess that represents a CSP process. This value defines a process via the constructor functions *hiding*, *exception*, *generalisedParallel*, and *replicatedInterleave*, as well as several functions described later: *buffer*, *modMemory*, and so on. Definitions such as that of $[\![\_]\!]_{\mathcal{M}}$ in Fig. 13 are hard to read due to the usage of CSP constructors as functions; for this reason, we present our semantic functions here in a more readable style as rules. For the definition of $[\![\_]\!]_{\mathcal{M}}$, for instance, we have Rule 1 instead.

In each rule definition, we identify the name of the function, its parameters, and return type in the header, and specify

$$
\begin{array}{l}
[\![m : Module]\!]_{\mathcal{M}} = \\
\quad hiding(\\
\quad\quad exception(\\
\quad\quad\quad hiding(\\
\quad\quad\quad\quad generalisedParallel(\\
\quad\quad\quad\quad\quad replicatedInterleave(\{c : asyncs \bullet buffer(c)\}), \\
\quad\quad\quad\quad\quad generalisedParallel(\\
\quad\quad\quad\quad\quad\quad modMemory(m), \\
\quad\quad\quad\quad\quad\quad composeControllers(m, ctrls, cons), \\
\quad\quad\quad\quad\quad\quad memoryChannels(m) \\
\quad\quad\quad\quad\quad), \\
\quad\quad\quad\quad\quad chanSet(evasyncs) \\
\quad\quad\quad\quad), \\
\quad\quad\quad\quad hiddenModuleChannels(m) \\
\quad\quad\quad), \\
\quad\quad\quad skip, \\
\quad\quad\quad \{end\} \\
\quad\quad), \\
\quad\quad \{end\} \\
\quad) \\
\\
\text{where} \\
\quad ctrls = \langle x : m.controllers \rangle \\
\quad cons = m.connections \\
\quad asyncs = \\
\quad\quad \{c : cons \mid \\
\quad\quad\quad c.async \wedge \{c.from, c.to\} \cap RoboticPlatform = \emptyset \\
\quad\quad \} \\
\quad evasyncs = \{c : asyncs \bullet eventId(c.eto)\} \cup \\
\quad\quad\quad\quad\quad \{c : asyncs \bullet eventId(c.efrom)\}
\end{array}
$$

Fig. 13 Semantic function for Modules

the function in the body of the rule. Our meta-notation is straightforward, and we underline its terms, which are used to specify the CSP processes (for example, the **where** clauses), and use standard mathematical italic font for CSP terms (for example, *Skip*).

The result of applying $[\![\_]\!]_{\mathcal{M}}$ to our running example, that is, the module Foraging in Fig. 5, is the process *Foraging* defined in the previous section. It is worth mentioning, however, that, for clarity, we have made some simplifications to the *Foraging* definition. First, since there are no asynchronous interactions between controllers in Foraging, the replicated interleaving over connections in asyncs in Rule 1 is over the empty set, and omitted. Additionally, since none of the controllers in this example terminates, the interruption $\Theta_{\{end\}}$ and hiding $\setminus \{end\}$ are redundant and also omitted. We explain these elements of Rule 1 in more detail next.

The semantic function buffer takes an asynchronous connection c as argument and defines a process that models a buffer of size one. It always accepts an input, possibly overriding a previously buffered value, and provides an output, if available. The input and output channels match those for c in evasyncs; they associate the buffer specifically with c. The rule that defines buffer and some other rules omitted here

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

$$\Big(\Big(\Big(\Big(\Big(\Big(\Big(\Big|\!\Big|\!\Big|\ c : \text{asyncs} \bullet \text{buffer(c)}\Big) \ [\![\text{evasyncs}]\!]\ \Big(\text{modMemory(m)} \ [\![\text{memoryChannels(m)}]\!]\ \text{composeControllers(m, ctrls, cons)}\Big)\Big) \setminus \text{hiddenModuleChannels(m)}\Big) \ \Theta_{\{end\}}\ Skip\Big) \setminus \{end\}$$

where

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

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

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

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

Rule 2. Composition of controllers

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

if #ctrls = 1 then

  renamingController(m, head ctrls, cons)

else

  renamingController(m, head ctrls, cons) [[connevts]] composeControllers(m, tail ctrls, cons)

where

$\text{connevts} = \text{renCtrlEvts(m, head ctrls, cons)} \cap \bigcup \{\text{c} : \text{tail ctrls} \bullet \text{renCtrlEvts(m, c, cons)}\}$

are in [96], and are implemented in RoboTool, presented in Sect. 5.1.

In Rule 1, we have one buffer for each asynchronous connection in asyncs. The buffers are combined in interleaving. If asyncs is empty, the interleaving reduces to *Skip*, the process that terminates immediately without any interaction. The interleaving is in parallel with the processes for the platform memory and for the controllers, synchronising on the events in evasyncs.

The function modMemory takes a module and defines a memory process for it, that is, a process to hold its platform's variables. The set memoryChannels(m) includes the channels used for interaction with this memory process, used for communication with the controllers.

The function composeControllers (Rule 2) takes a module, a sequence of controllers, and a set of connections, and defines the parallel process that composes the controller processes. Finally, hiddenModuleChannels takes a module and determines the set of channels used internally by that module's process.

The parallelisms and hiding of the CSP events identified by hiddenModuleChannels(m) in Rule 1 formalise the account of a module process in the previous section. To deal with termination, however, that process is composed with an exception operator that leads to *Skip* whenever the *end* event occurs. An exception $P\ \Theta_A Q$ is a process that behaves like P until an event in the set A occurs, when it behaves like Q.

The parallel composition of controllers in a sequence ctrls is calculated recursively as shown in Rule 2. It constructs the parallel process that composes the semantics of the first controller (head ctrls) renamed according to its connections, and the process for the remaining controllers (tail ctrls) defined via a recursion. The renaming applied to a controller process is given by renamingController(m, c, cons), which defines the process for the controller c, and the renaming to capture the connections cons. The parallel processes synchronise on the events in connevts, which contains the connection events common to the first and the remaining controllers after renaming. They are determined by the function renCtrlEvts(m, c, cons), which considers the renaming effected by renamingController(m, c, cons).

The definition of renamingController(m, c, cons), omitted here, uses the function $[\![\_]\!]_C$, which gives the semantics of a controller c. It is defined similarly to the semantics of a module as shown in Rule 3. The memory process ctrlMemory(c) is composed in parallel with composeMachines(c, ms, cs), which is the parallel composition of the processes for c's state machines, synchronising on the events in lvars ∪ rvars. These include the writing events (*set*) for local variables (identified by allLocalVariables(c)) and the writing events (*set_Ext*) for the required variables (allRequiredVariables(c)). Similarly, the memory and controller processes also synchronise on the events in lconsts ∪ rconsts, which include the writing events (*set*) for the local and required constants. The parallel composition does not include buffers because connections

Rule 3. Semantics of controllers $[\![\text{c : ControllerDef}]\!]_{\mathcal{C}}$: CSPProcess =

$$((\text{composeMachines(c, ms, cs)} \;[\![\; \text{lvars} \cup \text{rvars} \cup \text{lconsts} \cup \text{rconsts} \;]\!]\; \text{ctrlMemory(c)}) \setminus (\text{lvars} \cup \text{rvars} \cup \text{lconsts})) \; \Theta_{\{end\}} Skip$$

where

$\text{ms} = \langle \text{x : c.machines} \rangle$
$\text{cs} = \text{c.connections}$
$\text{lvars} = \{\!|\, \text{v : allLocalVariables(c)} \bullet set\_\text{vid(v)} \,|\!\}$
$\text{rvars} = \{\!|\, \text{v : allRequiredVariables(c)} \bullet set\_Ext\_\text{vid(v)} \,|\!\}$
$\text{lconsts} = \{\!|\, \text{v : allLocalConstants(c)} \bullet set\_\text{vid(v)} \,|\!\}$
$\text{rconsts} = \{\!|\, \text{v : allRequiredConstants(c)} \bullet set\_\text{vid(v)} \,|\!\}$

Rule 4. Controller memory ctrlMemory(c : ControllerDef) : CSPProcess =

let $Memory(\text{vars}) \mathrel{\widehat{=}}$
$\quad \Box\, v : \text{lvars} \bullet set\_\text{vid(v)}?x \rightarrow (\fatsemi\, m : \text{rmachines(v)} \bullet set\_Ext\_\text{vid(v, m)}!x \rightarrow Skip);\ Memory(\text{vars[name(v) := x]})$
$\quad \Box$
$\quad \Box\, v : \text{rvars} \bullet set\_Ext\_\text{vid(v)}?x \rightarrow (\fatsemi\, m : \text{rmachines(v)} \bullet set\_Ext\_\text{vid(v, m)}!x \rightarrow Skip);\ Memory(\text{vars[name(v) := x]})$
within
$\quad \text{constInit(c)};\ Memory(\text{varvalues})$

where

$\text{ms} = \text{c.machines}$
$\text{lvars} = \text{allLocalVariables(c)}$
$\text{rvars} = \text{requiredVariables(c)}$
$\text{vars} = \langle \text{v : rvars} \cup \text{lvars} \bullet \text{name(v)} \rangle$
$\text{varvalues} = \langle \text{v : rvars} \cup \text{lvars} \bullet \text{initial(v)} \rangle$
$\text{rmachines} = \check{}\,\text{v} \bullet \{\text{m : ms} \mid \text{v} \in \text{requiredVariables(m)}\}$

Rule 5. Semantics of state machines $[\![\text{stm : StateMachineDef}]\!]_{\mathcal{STM}}$: CSPProcess =

$$\left(\left(\begin{array}{l} \left(\text{initialisation(stm)} \;[\![\; \text{flowevts} \;]\!]\; \text{composeStates(ss, stm)} \right) \setminus \{\!|\, enter, entered, exit, exited \,|\!\} \\ \quad [\![\text{getsetChannels(stm)} \cup \text{trigEvents(stm)}]\!] \\ \text{stmMemory(stm)} \end{array} \right) [\![\text{renameTriggerEvents(stm)}]\!] \setminus \text{getsetLocalChannels(stm)} \cup \{\!|\, internal \,|\!\} \right) \Theta_{\{end\}} Skip$$

where

$\text{flowevts} = \bigcup \{\text{x : } SIDS \setminus \text{states(stm)};\ \text{y : states(stm)} \bullet \{\!|\, enter.\text{x.y}, entered.\text{x.y}, exit.\text{x.y}, exited.\text{x.y} \,|\!\}\}$
$\text{ss} = \langle \text{x : stm.nodes} \mid \text{s} \in \text{State} \rangle$

between machines are always synchronous. The *set* events of the local variables (collected in lvars) and local constants (in lconsts), and the *set_Ext* events of the required variables (in rvars) are then hidden, with termination accounted for as in the semantics of modules: by capturing the event *end* through an exception Θ and terminating.

The memory process for a controller is specified by Rule 4; it differs from that for a robotic platform in that it accepts *set_Ext* events for each required variable, which are used to propagate updates to shared variables. Since a controller memory is never read directly, this process does not accept *get* events.

The function ctrlMemory(c) defines a parameterised recursive process *Memory*(vars) that at each step reads a value through one of two types of channels: *set* for local variables and *set_Ext* for required variables. The parameters vars are the names name(v) of both local variables (lvars) and required variables (rvars). For each of them, depending on their nature (local or required), *Memory*(vars) offers, in a choice, a different communication. For a variable v in lvars, it accepts a value through the channel *set*_vid(v); here vid(v) specifies the qualified name of v. Next, *Memory*(vars) propagates sequentially ($\fatsemi$) the received value x to all machines m in the controller that require v using the channel *set_Ext*_vid(v, m). The sequence of such machines is determined by rmachines(v); the order in this sequence is arbitrary. The name $vid(v, m)$ is the qualified name of v in the machine m. Finally, *Memory*(vars) recurses with argu-

ment name(v) as x (vars[name(v) := x]). For a variable v in rvars, the memory process accepts through *set_Ext*_vid(v) a value being propagated from a platform, and propagates it to the machines, as for local variables. The controller memory is defined by an interleaving constInit(c) of *set* events for constants of the controller, followed by the instantiation of *Memory*(vars) with arguments varvalues that define the initial values initial(v) for the variables v in the memory.

The definition of composeMachines is similar to that of composeControllers in Rule 2, and omitted here. It uses the semantic function for a state machine, specified by Rule 5. The definition in Rule 5 follows the pattern in Rules 1 and 3: the semantics of the component (state machine, here) is composed in parallel with a memory process defined by stmMemory(stm).

There are two main differences here. First, the semantics of a machine stm is itself the parallel composition of an initialisation process initialisation(stm) and the parallel composition of the state processes defined by composeStates(⟨s : stm.nodes | s ∈ State⟩, stm). The arguments here are a sequence containing the nodes of stm that are a state, as opposed to a junction, and stm itself, which defines how those states are used. Second, the memory process stmMemory(stm) accepts not only *get*, *set*, and *set_Ext* events, but also triggers of the transitions of the machine to support the evaluation of guards. So, the synchronisation set includes *get* and *set* events for the machine variables (and constants), local and required, defined by getsetChannels(stm), and the transition trigger events, defined by trigEvents(stm), including events of the special channel *internal* used for transitions without trigger. The memory process stmMemory(stm) is defined in Rule 12. We rename the transition trigger events (renameTriggerEvents(stm)) to remove the transition identifiers, and hide the events (in getsetLocalChannels(stm)) that use the channels *internal*, *get*, or *set* for local variables or constants.

The initialisation process initialisation(stm) defines the semantics of the sequences of transitions from the initial junction to a state. We note that it is possible that, from the initial junction, there can be several transitions to other junctions before a state is reached. The function composeStates is specified in Rule 6.

The initialisation and states processes synchronise on the set flowevts, which contains *enter*, *entered*, *exit*, and *exited* events. This synchronisation plays two roles. First, it allows initialisation(stm) to request machine states to be entered. For that, it uses events *enter*.x.y and *entered*.x.y, where x is the machine identifier, and y is a state of the machine. Second, the synchronisation blocks the possibility of any other machine or states outside stm requesting states of stm to be entered or exited. For that, flowevts includes events whose first parameter x is any identifier in *SIDS*, but not in the set states(stm) of identifiers for the states of stm. *SIDS* includes the machine identifier itself, and that caters for requests from initialisation(stm). By including the identifiers of all other machines and of states not in stm, which are not used in initialisation(stm), the synchronisation blocks them. They are allowed by composeStates because the semantics of the states is agnostic to the context, including the machine and any sibling states, that can request their entering and exiting. The second parameter y of the events of flowevts is an identifier in states(stm), since it is the states of stm that can be requested to be entered and exited.

Composition of states is defined by Rule 6; it takes a sequence of states ss and their node container p. In Rule 5, the node container is the state machine. In Rule 8, where composeStates is used to define the semantics of a composite state, the container is that state.

The definition of composeStates(ss, p) follows a pattern similar to that used for controllers (Rule 2) . The restricted semantics (defined later by Rule 10) of the first state of ss (head ss) is composed in parallel with the composition of the semantics of the rest of the sequence of states (tail ss) calculated recursively, synchronising on the set cflowevts of their common flow events. This set is calculated similarly to the set connevts in Rule 2, and consists of the intersection of the set of events obtained by applying the function flowEvents to the first state, and the union of the sets obtained by applying flowEvents to the rest of the states.

The function flowEvents is given by Rule 7. It takes a state s and a node container p as parameters, and returns a value of type ChannelSet that represents a set of CSP events. The channel set returned by the function flowEvents contains the *enter*, *entered*, *exit* and *exited* events that represent requests or acknowledgements from s or to s. One of the two parameters y of each event is the identifier id(s) of s, and the other is the identifier of one of the children of its container p, that is, s itself or one of its siblings. These events allow s to request any of its siblings to enter or exit, and any of the sibling states to request s to enter or exit.

Rule 6 uses a restricted semantics of states, also used to give semantics of substates. We present here first the semantics of (composite) states in Rule 8, and then the restricted semantics for a state in a container in Rule 10. The semantics of simple states is similar and simpler.

Rule 8 defines $[\![s]\!]_{\mathcal{S}}$ as a parallelism of two processes. The first, *Inactive*, models the intrinsic behaviours of the state, that is, its actions and transitions. The second parallel process is composeStates(⟨x : states(s)⟩, s); it accounts for the semantics of the children of s in a similar fashion as the semantics of state machines, controllers, and modules handle the semantics of their components. The parallel processes synchronise on the events in the set *flowtrigevts*, defined in Rule 9 by the function flowTriggerEvents, with the flow events (those in flowevts) hidden. The set flowtrigevts

Rule 6. Composition of states composeStates(ss : Seq(State), p : NodeContainer) : CSPProcess =

```
if #ss = 1 then
  restrictedState(p, head ss)
else
  (restrictedState(p, head ss) [| cflowevts |] composeStates(tail ss, p)) \ cflowevts
where
  cflowevts = flowEvents(head ss, p) ∩ ⋃{s : tail ss • flowEvents(s, p)}
```

Rule 7. Flow events flowEvents(s : State, p : NodeContainer) : ChannelSet =

$$\bigcup \{x : states(p);\ y : \{id(s)\} \bullet \{|enter.y.x, entered.y.x, exit.y.x, exited.y.x, enter.x.y, entered.x.y, exit.x.y, exited.x.y|\}\}$$

Rule 8. Semantics of composite states $[\![s : State]\!]_{\mathcal{S}}$: CSPProcess =

let

$Inactive \mathrel{\widehat{=}} enter?o : sids.id(s) \rightarrow Activating(o)$

$Activating(o) \mathrel{\widehat{=}} [\![s.entry]\!]_{Action};\ initialisation(s);\ entered.o.id(s) \rightarrow ([\![s.during]\!]_{Action};\ Stop)\ \triangle$

$$\left(\begin{array}{l} \Box\ t : transitionsFrom(s) \bullet [\![t, s, false]\!]_{\mathcal{T}}^{Inactive,Activating} \\ \Box \\ \Box\ e : Event \bullet \begin{array}{l} \text{if (e.type == null) then eventId(e)?}x : tids \rightarrow exit;\ Inactive \\ \text{else eventId(e)?}x : tids?y \rightarrow exit;\ Inactive \end{array} \end{array}\right)$$

within

$(Inactive$ [| flowtrigevts |] composeStates($\langle$x : states(s)$\rangle$, s)) \ flowevts

where

flowtrigevts = flowTriggerEvents(s)

flowevts = $\bigcup${x : $SIDS \setminus$ states(s); y : states(s) •{|$enter.x.y, entered.x.y, exit.x.y, exited.x.y$|}}

sids = $SIDS \setminus$ {id(s)}

exit = $exit?as$: sids.id(s) $\rightarrow$ exitSubstates(s); $[\![s.exit]\!]_{Action}$; $exited.as$.id(s) $\rightarrow$ $Skip$

tids = $TIDS \setminus$ tIDS(s)

is used to restrict the flow and trigger events to ensure we have a compositional semantics for states as explained before.

The first parallel process *Inactive* specifies the initialisation of s using *enter* . The communication on *enter* is a request originating from any other state, with identifier o in sids, for s to be entered. The definition of sids ensures that the request can come from any state (or machine), but not s. A request from s itself is handled by the process for the transitions from s. After the request, the initialisation proceeds to the second process, *Activating*, which takes o as argument.

Activating(o) executes the entry-action process of s, that is, $[\![s.entry]\!]_{Action}$, requests initialisation of the machine in s, if any, using initialisation(s), indicates that s has finished entering using *entered*, and executes the during-action process $[\![s.during]\!]_{Action}$, while offering the possibility of interruption by a transition process. For a during action, its semantics ($[\![s.during]\!]_{Action}$) is composed with *Stop*. The transition processes are offered in external choice; there are two groups: (1) transitions that start in s and (2) transitions that can interrupt s if present in an ancestor state, including those without trigger, modelled using semantic *internal* events.

A transition t in the first group transitionsFrom(s) is given semantics by $[\![t, s, false]\!]_{\mathcal{T}}^{P,Q}$. Besides t, this function takes as arguments the source of t, which is s here, a boolean indicating whether that source is an initial node, which is *false* here, and processes P and Q as parameters. These model the source state of t, when inactive, in the case of P, and after entering has been requested, in the case of Q. If t exits the state, $[\![t, s, false]\!]_{\mathcal{T}}^{P,Q}$ proceeds to P. If t is a self-transition, $[\![t, s, false]\!]_{\mathcal{T}}^{P,Q}$ proceeds to Q. In Rule 8, these arguments are the processes *Inactive* and *Activating*. The semantics of transitions is given by Rule 11.

The second group contains all possible transitions that could appear in any ancestor state. In the semantics, the set of all trigger events contains all pairs e.tid, where e

Rule 9. Synchronisation events between parent state and substates flowTriggerEvents(s : State) : ChannelSet =

$$(\{\!|e : Event;\ t : TIDS \bullet e.t|\!\} \setminus \text{substatesTriggers(s)}) \cup \bigcup\{x : SIDS \setminus \text{states(s)};\ y : \text{states(s)} \bullet \{\!|enter.x.y, entered.x.y, exit.x.y, exited.x.y|\!\}\}$$

is any event, that is, an element of Event, which includes the *internal* events, and tid is any identifier from a set *TIDS* of valid transition identifiers. Those for the transitions of s and its substates, determined by the function tIDS(s), cannot identify transitions of ancestors of s. So, we only need to consider transitions with identifiers in the set tids = *TIDS* \ tIDS(s). The name of the trigger event is that determined by eventId(e). Parameters *y* of typed events, that is, those for which e.type is not null, are modelled as parameters of the CSP channel eventId(e). After the trigger event, the state can be exited, as defined by the process exit.

The process exit waits for a synchronisation on the channel *exit*, requests and waits for the active substate, if any, to exit, using the process exitSubstates(s), executes its exit-action process $[\![\text{s.exit}]\!]_{\mathcal{A}ction}$, and indicates completion of the exiting process through *exited*. After executing the process exit, *Activating*(*o*) recurses to *Inactive* so that s accepts new requests to be entered.

The synchronisation set between *Inactive* and the composition of substates is given by Rule 9. It specifies the events used by a parent state s to interact with its children. This set includes all the pairs of events *e* and transition identifiers *t*, that is, trigger events of the semantics, except those for transitions of the substates of s, defined by substatesTriggers(s). Additionally, flowTriggerEvents(s) includes the flow events *enter*, *entered*, *exit*, and *exited*, where the first parameter x does not identify a child of s: it is in SIDS \ states(s), and the second parameter y identifies one of those substates. As illustrated in the previous section, the result is that requests to enter or exit a substate by a non-sibling state can come only from the parent state.

Rule 10 gives the restricted semantics of a state s, which captures its behaviour when used in a given node container (machine or parent state) p. It is discussed in Sect. 4.2 and used in Rule 6.

As explained previously, the semantics of a state is compositional and offers not only its own transitions, but also any transition that could possibly belong to one of its ancestors to account for the behaviour of composite states. When composing the process for a state s into the semantics of a container p, information about the identifiers of the transitions of the sibling states becomes available. Since exit from a state cannot be requested due to such a transition, we block the events in the process for s for transitions with those identifiers. This is achieved by synchronising that process with *Skip* on all possible transitions not from or in s, defined by all_other_transitions_S, except those corresponding to actual transitions in p: in the set all_transitions_PS of transitions from s and its siblings.

The set of all transitions wholly contained within a state s is given by allTransitions(s); it determines the transitions between the substates of s (at any depth), but not the transitions starting at s itself. With the set transitionsFrom(s), on the other hand, we get exactly the transitions that start at s. These functions are used to define the set of identifiers tidsfromwithin of the transitions from and within s, which is used to specify the set all_other_transitions_S. In the definition of all_transitions_PS, the event of a transition t of p is obtained by t.trigger.event as defined in the metamodel.

The semantics of transitions is given by Rule 11. It takes a transition t, a node container origin, a boolean value initial, and processes P and Q. This function is called recursively to cover all the transitions in a flow starting in a state or initial junction and finishing in a state. The parameters origin and initial record the starting point of the flow, that is, the source of the first transition, and whether or not it is an initial junction. If the source is a state, P and Q model it: P, when it is inactive, and Q, after entering has been requested.

Since it is called recursively, Rule 11 distinguishes the possible starting points src of a transition: a state, an initial junction, or just a junction (that is not initial). In each case, it produces a slightly different process.

If it is a state, the process consists of a communication given by the semantics $[\![\text{t.trigger}]\!]^{\text{id(t)}}_{\mathcal{T}rigger}$ of the trigger (potentially non-existent, leading to an event that uses the channel *internal*), and a process that exits the state and moves on to the target of the transition. This process consists of a communication on *exit* to indicate that exiting of src has started, exitSubstates(src) to request and wait for the active child, if any, to exit, the process $[\![\text{src.exit}]\!]_{\mathcal{A}ction}$ for the exit action, an indication through *exited* that the exiting is finished, the process $[\![\text{t.action}]\!]_{\mathcal{A}ction}$ for the transition action, and, finally, $\text{compileTarget(tgt, src, false)}^{\text{P,Q}}$, a process that captures the execution of the target. For example, if that target is a state, it is a request to enter that state. If it is a junction, we have a recursive call to deal with the transitions from that junction, and so compileTarget has the same parameters declared in Rule 11 itself.

Rule 10. Restricted semantics of states restrictedState(p : NodeContainer, s : State) :CSPProcess =

$[\![s]\!]_{\mathcal{S}}$ [| all_other_transitions_S \ all_transitions_PS |] *Skip*
where
tidsfromwithin = {t : transitionsFrom(s) ∪ allTransitions(s) • id(t)}
all_other_transitions_S = {e : Event; tid : TIDS \ tidsfromwithin • eventId(e).tid}
all_transitions_PS = {e : Event; tid : TIDS • eventId(e).tid} \ {t : allTransitions(p) • eventId(t.trigger.event).id(t)}

Rule 11. Semantics of transitions $[\![$t : Transition, origin : NodeContainer, initial : boolean$]\!]^{P,Q}_{\mathcal{T}}$: CSPProcess =

if src ∈ State
$[\![$t.trigger$]\!]^{id(t)}_{\mathcal{T}rigger}$ → *exit*.id(src).id(src) → exitSubstates(src); $[\![$src.exit$]\!]_{\mathcal{A}ction}$; *exited*.id(src).id(src) → $[\![$t.action$]\!]_{\mathcal{A}ction}$;
compileTarget(tgt, src, false)^{P,Q}
else if src ∈ Initial
internal.id(t) → $[\![$t.action$]\!]_{\mathcal{A}ction}$; compileTarget(tgt, parent(src), true)^{P,Q}
else if src ∈ Junction
internal.id(t) → $[\![$t.action$]\!]_{\mathcal{A}ction}$; compileTarget(tgt, origin, initial)^{P,Q}
where
src = t.source
tgt = t.target

Rule 12. State machine memory stmMemory(stm : StateMachineDef) : CSPProcess =

let *Memory*(vars) ≙
□ *v* : lvars • *get*_vid(v)!name(v) → *Memory*(vars) □ *set*_vid(v)?*x* → *Memory*(vars[name(v) := x])
□
□ *v* : rvars • (*get*_vid(v)!name(v) → *Memory*(vars) □ *set*_vid(v)?*x* → *Memory*(vars[name(v) := x])
□
*set_Ext*_vid(v)?*x* → *Memory*(vars[name(v) := x]))
□
□ *v* : allConstants(stm) • *get*_vid(v)!name(v) → *Memory*(vars)
□
□ *t* : allTransitions(stm) • memoryTransition(t); *Memory*(vars)
within
constInitSTM(consts, stm,*Memory*(varvalues))
where
rvars = requiredVariables(stm)
lvars = allLocalVariables(stm)
consts = ⟨v : allConstants(stm) • v⟩
vars = ⟨v : rvars ∪ lvars • name(v)⟩ ⌢ ⟨v : consts • name(v)⟩
varvalues = ⟨v : rvars ∪ lvars • initial(v)⟩ ⌢ ⟨v : consts • name(v)⟩

The process when the source is an initial junction is slightly simpler, as there is no trigger, and no state to exit. In this case, the process accepts a synchronisation on the channel *internal* (corresponding to the empty trigger) with the identifier of the transition (id(t)) as a parameter, followed by the execution of the transition action and compileTarget(tgt, parent(src), true)^{P,Q}. Here, the second argument is the parent of src. This argument is used to request the target state to be entered, and if the source is an initial node, that request comes from the parent state (or machine). The semantics when the source is a regular junction is similar to that of initial junctions, except that the parameters origin and initial of the rule are passed on directly to compileTarget.

The parameters origin and initial are used by the function compileTarget to identify whether the sequence of transitions whose semantics is being defined starts in an initial state, and whether that sequence forms a cycle returning to the initial state. In the first case, the sequence of transitions does not lead to exiting a state (unlike regular transitions), and must lead to the execution of the remaining behaviours (for

instance, executing the during action) of the state that contains the initial junction. In the second case, the sequence of transitions must lead to the behaviour specified by the second process Q given as parameter. This is needed to guarantee that after a cycle, the source state is not waiting to be activated, as it has already been entered by the final transition of the cycle. The parameters P and Q are the continuation processes used to build the mutually recursive processes of a state (Rule 8).

The memory process of a machine is defined by Rule 12. Like Rule 4, this rule defines a recursive parameterised process (*Memory*(vars)). At each step it offers, in choice, *get*_vid(v) and *set*_vid(v) events for each local variable of the state machine (lvars), events *get*_vid(v), *set*_vid(v) and *set_Ext*_vid(v) for each required variable (rvars), *get*_vid(v) events for each local and required constant (allConstants(stm)), and processes memoryTransition(t) offering the events of each of the transitions t of the machine. The choices involving events *set*_vid(v) and *set_Ext*_vid(v) result in recursive calls where the parameter for the variable is replaced with the received value, and all other choices lead to recursive calls with unchanged parameters.

Like in Rule 4, the values of the loose (local and required) constants $c_1, c_2, \ldots$, that is, those whose values are not determined in the machine, are read via *set* channels. Here, however, as defined by the function constInitSTM (omitted) they are read in an (arbitrary) order $set\_c_1?c_1 \rightarrow set\_c_1?c_2 \rightarrow \ldots$. This defines a scope where the names $c_1, c_2, \ldots$ are defined. In addition, in a let-expression, the constants whose values are defined in the machine are declared locally with their values. All constant names are used as arguments in $Memory(\ldots, c_1, c_2, \ldots)$ to define the values of the constants in the call to the process *Memory*. Unlike the platform and controller memory processes, here we need to record the values of the constants as parameters to make them available via *get* channels to the machine.

As previously said, state machines can also be used to define operations. In this case, the semantics of the operation is a parameterised process that takes the arguments of the operation as parameters. Apart from that, the process that defines the semantics is exactly that of a state machine presented above (Rule 5).

State machines may contain actions, which are composed of statements. Statements may contain expressions, and these expressions must be evaluated before the statement is executed. Since variables are recorded in a memory process, the values used in a statement must be first read from the memory, and a local context must be created where the statement can be executed. Rule 13 specifies this behaviour. It takes a statement s, and uses a function readState to construct a process that reads a set of variables from the memory creating a local context, and executes the statement in that context. The set of variables is calculated using the function usedVariables, and the basic semantics of a statement is given by the function $[\![\_]\!]_{Statement}$. The semantics for statements, expressions, triggers, and so on is standard and omitted here, but is in [96].

Next, we present RoboTool, which implements the semantic rules above to calculate fully automatically a CSP model for a RoboChart diagram.

5 Verification and validation

The well-formedness conditions and semantics defined in the previous sections allow us to both validate and verify our models. In particular, verification can be performed using the CSP model-checker FDR [41]. To that end, some level of automation is necessary. This has been achieved in the form of a prototype tool called RoboTool, which we discuss in the next section. Section 5.2 further expands on the use of model-checking technology for the verification of properties. Finally, in Sect. 5.3, we discuss some examples.

5.1 Tool support

RoboTool[6] is a set of Eclipse[7] plug-ins implemented using the Xtext[8] and Sirius[9] frameworks.

Modelling We have used the *Eclipse Modeling Framework* (EMF) to implement the metamodel presented in Sect. 3.2 as a basis to generate a textual editor using Xtext and a graphical editor using Sirius. The textual notation is used exclusively as an internal representation specific to RoboTool. A different implementation of RoboChart might choose a different representation.

Figure 14 shows RoboTool with a number of diagrams open. The RoboTool window has four areas: model explorer (top-left), outline (bottom-left), graphical editor (top-right), and properties/problems (bottom-right).

The area for the graphical editor itself is divided into two parts: the diagram canvas (left) and the tool palette (right). RoboChart diagrams are constructed in the diagram canvas using tools from the palette; additionally, diagrams can be edited by double-clicking (for example, to edit names and actions), or by right-clicking elements and selecting actions from the menu.

The graphical editor constructs a RoboChart model and automatically verifies all the well-formedness conditions described in Sect. 3.3 and [96], and type compatibility in expressions and statements. We describe the implementation of these checks below.

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

[7] www.eclipse.org.

[8] www.eclipse.org/xtext.

[9] www.eclipse.org/sirius.

Rule 13. Semantics of statements in context $[\![s : \text{Statement}]\!]_{StatementInContext} : \text{CSPProcess} =$

$$\text{readState}(\text{usedVariables}(s), [\![s]\!]_{Statement})$$

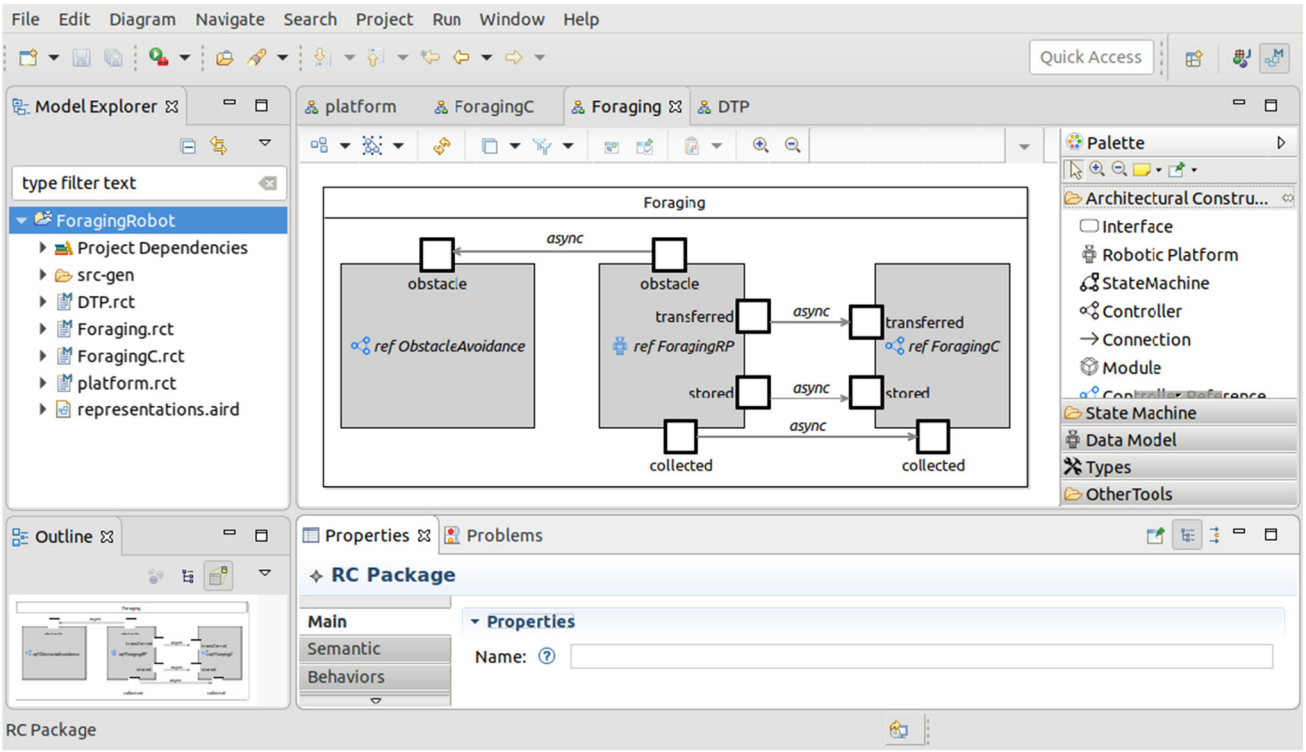

Fig. 14 RoboChart textual and graphical editors

```
@Check
def junctionWFC1(Junction j) {
  if(j instanceof Initial) return
  val parent = j.eContainer as NodeContainer
  if (parent.transitions.filter[t|t.source == j].size == 0) {
    error(
      'A junction in ' + parent.name + ' should have at least one outgoing transition',
      RoboChartPackage.Literals.NAMED_ELEMENT__NAME,
      'transitionFromJunction'
    )
  }
}
```

Fig. 15 Implementation of the well-formedness condition J1 for junctions

Validation The well-formedness conditions are implemented through the validation mechanism provided by Xtext. Each condition is associated with one or more validation rules implemented by a method written in Java or Xtend[10] and annotated with `@Check`. Figure 15 shows the implementation of the well-formedness condition J1 for junctions (see Sect. 3.3).

The method in Fig. 15 first checks if its argument `j` is an initial junction (`j instanceof Initial`). If so, the condition being implemented does not apply as there is a stronger condition for initial junctions (condition J4 in Sect. 3.3). Next, the method recovers the container `parent`, that is, the state or machine that contains the junction, obtains the set `parent.transitions` of transitions fully contained in the parent and restricts this set, through the method

[10] A dialect of Java (www.eclipse.org/xtend/).

`filter`, to those transitions `t` whose `source` is the junction. Finally, it checks the `size` of this set, and if it is empty, it produces an error using the method `error`. This method takes a textual description of the error, a literal that indicates the element of the abstract syntax tree to which the error should be attached (in this case, the name of the junction) and an identifier for the type of error.

The close correspondence between the RoboChart well-formedness conditions and the validation rules implemented in Xtext provides validation for both the language and our conditions. Violation of such rules not only produces error information, but also prevent the generation of the semantics of invalid models. Adding or removing validation rules to cater, for instance, for different semantic models or applications (like code generation) is a simple exercise.

While most of the well-formedness conditions can be checked automatically, conditions such as *The guards of the transitions out of a junction must form a cover* (see condition (2) for junctions) cannot be checked syntactically, requiring the use of automatic theorem provers or SMT solvers. Nevertheless, violations of such conditions does not prevent the generation of semantics that can be processed with CSP tools, and that semantics can be used to check well-formedness in these cases. For instance, the violation of the condition mentioned above leads to a deadlocked junction, which can be identified via a deadlock check using FDR.

Our validation rules additionally check for type correctness based on the type system of the Z notation [99]. Libraries for the data types modelled in the Z toolkits (sets, functions, and so on) are under development.

Semantics generation RoboTool calculates both the untimed and timed semantics (presented in the next section) tailored for use with FDR, that is, using the ASCII version of CSP called CSP-M, to support automatic verification of properties. For example, Fig. 16 shows the implementation of Rule 6 in Sect. 4.3.

While the rules are essentially the same, there are some superficial differences. Our formalisation identifies meta-notation using colour and underline, but in our implementation the meta-notation is the language Xtend. The target language constructs are encoded as strings enclosed in triple quotes, with in-line meta-expressions identified by the use of guillemets («»).

In addition, in the code generator, instead of building one monolithic CSP process that yields the overall semantics of a module, we declare a number of named processes for each of the main components: modules, controllers, state machines, states, and so on. This is important to improve the performance of verification via model checking in FDR and readability. For example, in Fig. 16, instead of calling the function restrictedState used in Rule 6, we call directly a process whose definition is given by the implementation of restrictedState. The processes declared by restrictedState are named `S_name_R`, where `name` is the name of the state defined by «`id(state)`» in Fig. 16. The prefix `S` indicates that the process models a state, and the suffix `R` indicates that this is the restricted semantics of the state.

One consequence of using process declarations is that values that would otherwise be available throughout the monolithic process defined by Rule 1 must be passed as parameters. For example, in Fig. 16, if the machine defines an operation, any parameters need to be passed on to the processes `S_name_R` that defines the state. This is achieved through the method `parameterisation` of a state's state machine (obtained via the method `connectionNode`).

Constants are specials kinds of variables, and in our formal semantics, they are held in the memories of the relevant components. As an optimisation, the semantics generated by RoboTool handles constants using parameters. For example, a constant declared by the robotic platform gives rise to a global CSP declaration of that constant, which is passed as parameter to the processes for the controllers that require the constant, which in turn pass it as a parameter to all its state machines that require the constant as well. A similar treatment is given to constants defined in a controller. Constants defined in a machine are also declared globally. Careful choice of names avoids clashes in global declarations: we use qualified names based on the modules, controllers, or machines that define the constants.

The set of events cflowevts used in the parallel composition and in the hiding in Rule 6 is calculated in Fig. 16 using an extra method `getSyncSet` that takes the head and tail of the sequence of states. The semantics generated by RoboTool uses declarations of channel sets as well as processes to structure definitions.

Verification tools (for CSP) may impose extra requirements on the RoboChart model or on its semantics. For example, a model checker such as FDR requires a concrete version of CSP with a fixed syntax and additional restrictions. One such restriction requires that, in general, sets used in the model (for example, in iterated operators such as the external choices and interleavings used in Sect. 4) are finite. Additionally, aspects such as declaration of channels and uniqueness of names must be taken into account when calculating a semantic model that targets such tools. To generate CSP models that can be analysed using FDR, aspects such as these, which are not particularly relevant to the formal semantics, must be dealt with. In what follows, we briefly describe our solutions to some of these issues.

In RoboChart, names must be unique within a specific scope, but can be the same in different components. In our running example, for instance, the obstacle event is declared both in the platform ForagingRP (Fig. 2) and the controller ObstacleAvoidance (Fig. 5). In our semantics implementation, RoboChart events give rise to global

Fig. 16 Implementation of Rule 6

```
def composeStates(Iterable<State> nodes) {
  if (nodes.size == 1) {
    val state = nodes.head
    '''S_«id(state)»_R«state.connectionNode.parameterisation»'''
  } else {
    val state = nodes.head
    val tail = nodes.tail
    '''
      (S_«id(state)»_R«state.connectionNode.parameterisation»
        [|«state.getSyncSet(tail)»|]
      «tail.compileSubstates»)\«state.getSyncSet(tail)»
    '''
  }
}
```

channel declarations; modules, controllers, and machines do not define separate scopes in the CSP model. So, we use fully qualified names to guarantee uniqueness. The events obstacle, for instance, yield two channel declarations *ForagingRP_obstacle* and *ObstacleAvoidance_obstacle*, incorporating the names of the components where the events are declared, thus guaranteeing that the names are unique. We also avoid name clashes between events used just by the semantics; one such event is *enter*, for example.

While CSP-M supports a form of scoping known as modules, these are relatively new and not used in the untimed semantics. As we discuss in the next section, modules are used in the timed semantics to reuse the untimed semantics in the timed context as well as to avoid name clashes between the two semantics.

In the semantics in Sect. 4, events such as *enter*, *entered*, *exit*, and *exited* have type *SIDS* and can be used in the semantics of any state machine. *SIDS*, however, is an infinite set of state identifiers, and cannot be used in CSP-M. To achieve finiteness, for each state machine, we calculate the sets of identifiers that actually appear in that machine and define a machine-specific set of identifiers. For instance, the set of identifiers for the machine DTP is called *DTP_SIDS* and contains seven identifiers: *DTP*, for the machine, and, for the states, *Exploring*, *GoToNest*, *WaitForTransfer*, *Neighbourhood*, *GoToSource*, and *GoToNestDirectly*. In this way, we guarantee finiteness, but require events such as *enter* to be declared with different types for each of the machines, thus requiring the use of unique fully qualified names. We take a similar approach to limit the set of transition identifiers (*TIDS*).

While sets such as *SIDS* and *TIDS* can be replaced with finite sets using solely information available in the model, other potentially infinite sets such as int and real are more difficult to handle. This is due to the fact that, in general, it is not possible to find automatically a subset of these sets that is finite and covers all possible values used in the specification. To calculate the semantics of a model that uses such infinite types, our implementation defines simple default representations as sets of two or three elements of the original set, and adapts the primitive operations to check for containment of the results to avoid type errors.

This adaptation allows the semantics to be checked by FDR, but may be oversimplifying the model. For this reason, the definitions of the types and operations are all recorded in a single file `instantiations.csp` in the generated semantics. This facilitates both inspection and modification of the abstractions, if needed. In this way, it is also possible to perform verifications of the model with different abstractions. Verification of properties of the model using the original infinite types cannot in general be performed using model-checking techniques (although there are model checkers, such as nuXmv [18], which support some forms of infinite state), but, alternatively, can be verified via theorem proving. Technology for verification of RoboChart models via theorem proving is currently under development, but already presents some encouraging results [34].

The implementation in Xtend validates our semantics: matches the rules and functions closely, and ensures coverage of the metamodel and well-typedness of the definitions. Also, RoboTool has enabled the construction and verification of several examples that provide further evidence of adequacy of the semantics.

5.2 Model checking

Besides the semantics, RoboTool generates, for every module, controller, and machine of a model, a set of assertions to check general properties such as termination, deadlock freedom, divergence freedom, and determinism. These assertions refer to the generated semantics and can be automatically checked using FDR.

As mentioned, FDR is a refinement model checker. While it differs from temporal logic model checkers, it does exhaustively analyse a model to check that it satisfies a given specification. The notion of satisfaction is refinement, and the specification is also a CSP process. This allows us to compare different RoboChart models, supporting a devel-

```
assertion DTP does not terminate (R1)
assertion Foraging is deadlock-free (R2)
assertion Neighbourhood is deterministic (R3)
```

Fig. 17 Assertions for the foraging example

```
csp ObstacleFree csp-begin
  ObstacleFree = DTP [|{|obstacle|}] STOP
csp-end

assertion ObstacleFree is deadlock-free (R1)
```

Fig. 18 CSP block in `.assertions` file

opment process in which models are made more and more concrete (closer to an implementation), and their correctness is established by refinement checks. Lowe [54] investigates the interplay between temporal logic and refinement in the context of process algebras such as CSP. That work shows that the bounded, positive fragment of linear temporal logic can be checked using refinement.

RoboTool can also process assertion files (with extension `.assertions`) that define properties of particular interest for verification. The definitions are described using a tool-independent simple form of controlled English. Figure 17 shows an assertion file for our running example, and the assertion language is fully specified in [96]. In Fig. 17, we have three assertions named R1, R2, and R3 specifying that the machine DTP does not terminate, that the module Foraging is deadlock free, and that the state Neighbourhood is deterministic. A `.assertions` file can be used to produce a report that records the result of the verification. A full account of the syntax of `.assertions` files is in [97].

The advantage of using a `.assertions` file, instead of the assertions automatically generated, is that we can avoid the need to interact directly with FDR. RoboTool includes a facility to use a `.assertions` file to generate a report with the results of the verification based on the names of the assertions. Moreover, the assertions allow the reporting to be tailored for the application. For example, the RoboTool assertions that are automatically generated check for termination, and indicate failure if the machine does not terminate. In our example, we do not expect DTP to terminate, and so we include the negation of that assertion in the `.assertions` file. The report generated, therefore, shows no failures.

A `.assertions` file can also contain blocks of CSP that specify processes or custom assertions. One such block is shown in Fig. 18. It declares a process called `ObstacleFree` defined as `DTP` with the event `obstacle` deadlocked. The assertion `R1` then checks that the process `ObstacleFree` is deadlock free: even if there is no obstacle ever, the machine does not deadlock.

Since RoboChart models may contain loose constants, given types, and undefined functions, but FDR cannot cope

```
csp Instantiations csp-begin
      nametype nat = {0..10}
      nametype real = {0,10}
      ...
      DTP_e = 1
      ...
      d((x1,y1),(x2,y2)) = abs(x1-x2)+abs(y1-y2)
      ...
csp-end
```

Fig. 19 CSP Instantiations block

with such elements, instantiations must be provided. To produce a complete CSP specification, RoboTool provides default instantiations in the file `instantiations.csp`. Loose constants are instantiated with a default value determined by the type of the constant, given types are instantiated as nat, and functions are instantiated as constant functions returning the default value of the return type of the function. User-defined values for the instantiations can be defined in a `.assertions` file using a CSP block called `Instantiations` as illustrated in Fig. 19. This block modifies the default instantiations for types such as `nat`, constants such as `DTP_e`, and functions such as `d`, which calculates the distance between two points.

The semantics hides channels as soon as possible, and this has a positive effect on the performance of FDR as the compression functions tend to be more effective on systems with many internal actions [85, p. 176]. The use of process declarations in RoboTool, as already mentioned, also contributes positively. In addition, to optimise the verifications, RoboTool generates definitions that make extensive use of semantic-preserving compression functions available in FDR.

In the next section, we present examples of models and verifications carried out using RoboTool.

5.3 Case studies

We have used RoboChart and RoboTool to model a few examples from the robotics literature [10,26,40,44,70]. In this section, we focus on three case studies for which we have formalised various requirements and used our semantics to verify them. These case studies and the results of their analyses are available online [84].

Chemical detector Our first example [44] is a tele-operated chemical detector that receives commands to move forward and turn, and analyses the air to detect potentially dangerous gases. Upon detection, the robot flashes a light, and drops a flag to mark the location of the gas source. This model includes two controllers, one with a single state machine, and the other with two non-interacting state machines. The

Table 4 Summary of verifications using FDR

Example	States	Transitions	Compilation (s)	Deadlock (s)	Longest check (s)
Chemical detector	70	237	0.07	0.17	0.17
Alpha algorithm	15	46	0.16	0.12	0.12
Autonomous detector	47	702	0.5	0.05	0.06

machines use features of the core RoboChart notation, such as, simple states, transitions, junctions, and events.

Alpha algorithm Our second example is the alpha algorithm studied in [26], which is widely used in swarm robotics. This algorithm uses communication facilities of the robotic platform to estimate the number of other robots in the neighbourhood. Depending on that number, the algorithm decides whether the robot is in an aggregate of robots or not. If it is not, the alpha algorithm uses facilities of the robotic platform to turn the robot around and move towards the aggregate. Otherwise, the algorithm effects a random turn to avoid the collapsing of the aggregate. Our RoboChart model for the alpha algorithm has two controllers, each defined by a single state machine. The machines feature synchronous and asynchronous communications, the full range of time primitives, and local variables.

Autonomous detector The third case study in Table 4 is an autonomous version of the chemical detector above. This model expands on the gas detection mechanism and replaces the controller that treats movement requests with a controller that performs a random walk with obstacle avoidance. This example uses features of RoboChart such as asynchronous communications, and variables in all components.

For each example above, Table 4 gives the size of the underlying labelled transition systems (number of states and transitions) after compression, as well as the time FDR takes to compile the CSP model, to check it for deadlock, and the longest time to check a requirement. The compilation time is that FDR takes to build the labelled transition system from the CSP model. It is presented to emphasise the adequacy of the RoboChart semantics for analysis with FDR.

The number of states and transitions of the uncompressed labelled transition system is often large. Through judicious structuring and hiding of events, the semantics can exploit compression functions available in FDR to radically reduce that number. Our experiments, including the three case studies we present here, indicate that the sizes of the compressed labelled transition systems are proportional to the numbers of states in the RoboChart model. We observe, however, that a number of factors can affect the overall size, including, for example, the instantiation of types. Further details about the verifications, including the requirements, models and results are in [84].

No doubt, further evaluation is necessary to gauge the effectiveness of model-checking RoboChart semantic models. We are, however, encouraged by the results with compression, which are related to the structure of the models. In any case, our plan in the long term is to use theorem proving based on the UTP theory for CSP.

6 Time in RoboChart

In this section, we describe the support for modelling and verification of timed properties using RoboChart. The semantics presented in Sect. 4 ignores all time primitives of RoboChart. Here, we show how that semantics can be extended to deal with them. In the next Sect. 6.1 we define the timed constructs, and in Sect. 6.2 we formalise their semantics. Support in RoboTool for model checking is discussed in Sect. 6.3.

6.1 Notation

In RoboChart timed properties are modelled directly in state machines, a practice commonly seen across the robotics literature. We consider, for example, the state machine DTP of the foraging example (Fig. 2) where the clock T is used to control the time available for transferring objects and exploring the neighbourhood. For instance, the time allowed for completing a transfer is bounded by guarding an outgoing transition of the state WaitForTransfer with since(T)>waitDeadline, so that if the event transferred does not occur within waitDeadline time units relative to the previous reset (#T) of T then the operation GoTo(nest) is called.

Unless specified, operations are not assumed to terminate and can take any amount of time to complete. For example, the GoTo operation declared in the interface MovementI is not further specified, and so can take an arbitrary amount of time. (Since it is used in a during action, it can be interrupted by the state's outgoing transitions, though.) In the interest of predictability, RoboChart requires time properties to be specified explicitly, including those of operations.

Timed budgets and deadlines are recorded over actions and events. To illustrate these, we introduce in Fig. 20 the state machine Movement, which is part of the model of an autonomous chemical detector considered in our case studies [44]. Here, a composite state Stoppable has a transition to the state Found triggered by the event stop to stop the robot by calling the operation move(0,0) with linear and angular

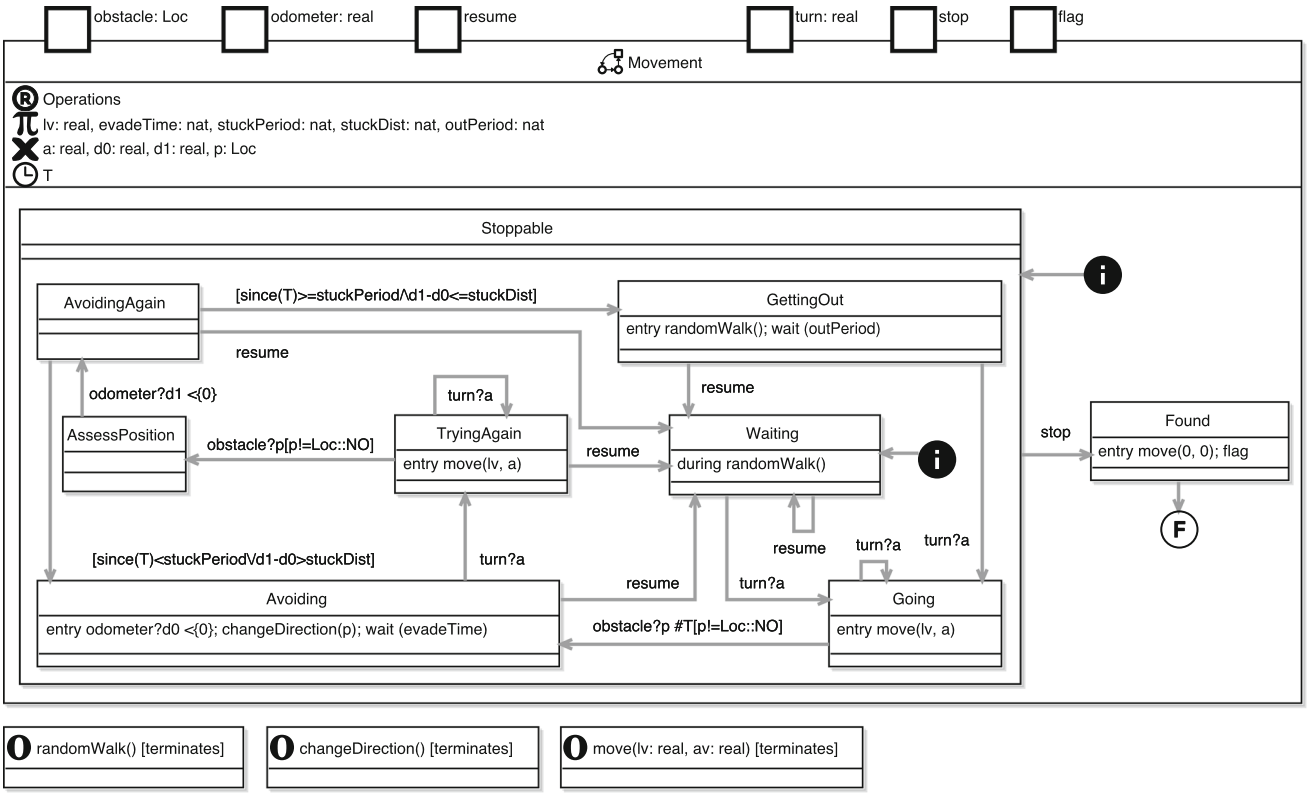

Fig. 20 State machine Movement of the autonomous chemical detector from [44]

velocity set to zero, and then drop a flag by raising the event flag.

Within the state Stoppable the robot, initially in the state Waiting, starts performing a randomWalk() and can be interrupted by either the event resume to keep calling randomWalk(), or the event turn?a to transition to the state Going and turn. In this state, it can continuously handle the event turn?a to repeatedly call move(lv,a), with a particular linear (lv) and angular velocity (a), or handle obstacle?p. In this case, it resets a clock T and transitions to the state Avoiding when p is not equal (!=) to Loc::NO. Here, Loc is an enumerated type whose definition we omit, but whose value NO indicates absence of obstacles.

In Avoiding, with odometer?d0 <{0} the robot requests a read from the odometer with a deadline zero and stores the value in the local variable d0, calls the operation changeDirection(p), and waits for evadeTime time units, before being able to resume and transition to the state Waiting, or turn and transition to the state TryingAgain. In that state, it calls the operation move, and can then resume, turn, or accept an obstacle event and transition to AssessPosition. It then must receive an input via the event odometer?d1 with deadline zero and transition to AvoidingAgain. We observe that the reading is now recorded in a variable d1 unlike in the state Avoiding. This allows transitions out of the state AvoidingAgain to be constrained relative to the difference d1-d0. If at least stuckPeriod time units have been spent trying to avoid the obstacle and a distance less than or equal to stuckDist has been covered, then the robot transitions to GettingOut, otherwise it resets the clock T and transitions to Avoiding. In GettingOut, a randomWalk() is carried out for outPeriod time units, before the robot can be commanded to resume or turn.

Both deadlines, over the action odometer?d0<{0}, and over the trigger odometer?d1 <{0}, impose requirements on the platform regarding the availability of sensor information represented by the event odometer over time. In Avoiding, a budget of evadeTime units is specified to allow the behaviour of changeDirection to complete before accepting a resume or turn event. We observe that in this particular example operations like changeDirection and move are specified as terminating and consuming no time, thus the budget evadeTime is used as an indication of the time required for the actual behaviour of changing direction to occur. A similar budget is provided for the effect of the operation call randomWalk() in the state GettingOut.

A robotic platform could meet the timed requirements of Movement, for example, by allowing an odometer reading to take place at any time. A deadline, however, can also be a requirement on availability of events of another state machine, possibly in a different controller. In that case it may not be obvious whether the complete module meets all

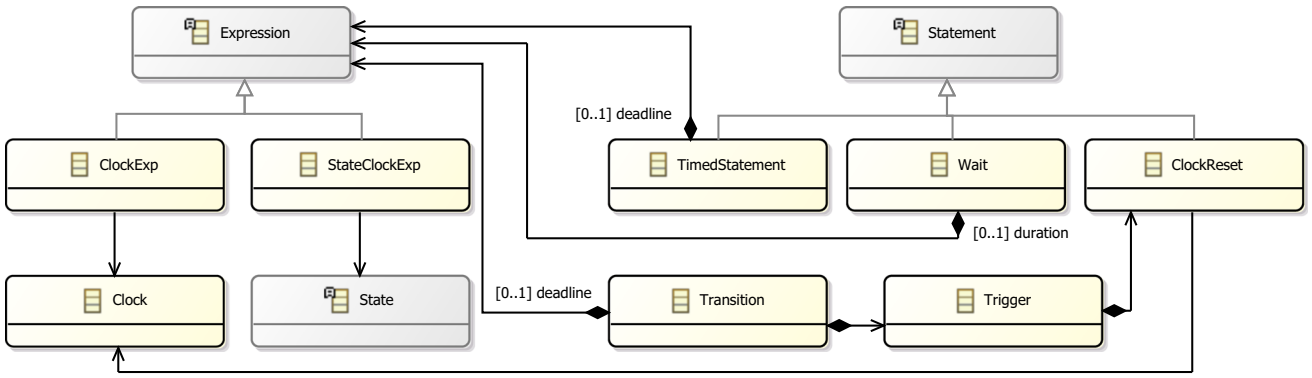

Fig. 21 Metamodel of time primitives of RoboChart

timed requirements. We have encountered instances of this situation, for example, in the model for the alpha algorithm, where two controllers interact asynchronously via events. As we discuss later in Sect. 6.3 our timed verification approach for RoboChart can help identify such scenarios.

We next describe how the time primitives illustrated above are captured in the RoboChart metamodel (Sect. 6.1.1) and present the well-formedness conditions that apply to them (Sect. 6.1.2).

6.1.1 Metamodel

In Fig. 21, we provide a simplified view of the metamodel components that support modelling of timed aspects. A summary of the time primitives is included in Table 5. They are expressions and statements that are omitted in the previous Figs. 6, 7 and 8.

As previously discussed we have a notion of Clock that allows transitions to be guarded by time expressions that define constraints relative to the occurrence of other events via the primitives since(C) (which is a ClockExp) and #C (ClockReset), and relative to activation of a state via sinceEntry(S) (a StateClockExp). Because a ClockExp and a StateClockExp are expressions, the metamodel allows their use as part of any expressions. A ClockReset is a statement, but it can also be used as part of a trigger, when the reset and the event trigger are to take place at the same time.

In the next section, we define well-formedness conditions to confine the use of ClockExp and StateClockExp expressions to (well-typed) transition guards. In addition, like in timed automata, we restrict expressions involving clocks to comparing single timed primitives with constant expressions, that is, those whose value depends only on literals or declared constants.

Finally, we also have a timed primitive to impose a deadline d on an action A, a TimedStatement A <{d}, or on a transition trigger e, a Trigger e <{d}, and to specify a budget: Wait. The are two forms of Wait statements: Wait(d), where d is an Expression, and Wait([i,j]), defining a budget between i and j.

6.1.2 Well-formedness

In addition to the conditions of Sect. 3.3, we impose additional restrictions regarding the usage of clocks in expressions and actions as described below. These are in addition to usual type and scope rules. Overall, the well-formedness conditions ensure that clocks are local to machines and can be used only via the time primitives.

Timed expressions

TE1 *Expressions involving* since *and* sinceEntry *are permitted only in transition guards* because clocks can be used only to constrain the availability of transitions using supported expressions, and therefore their value cannot be directly queried.

TE2 *A clock* C *in an expression* since(C) *may reference only a clock declared within the expression's containing state machine*. This is because clocks are not shared across different state machines.

TE3 *A state* S *in an expression* sinceEntry(S) *may reference only a state within the expression's containing state machine*. This is because transitions may not be constrained based on the entering of states of other state machines. Instead, such a scenario can be modelled by state machines interacting via events.

TE4 *Expressions* since(C) *and* sinceEntry(S) *may be compared only with an expression that is constant, and only when using one of the operators* >, <, >=, <=, ==. This restriction is similar to timed automata in that the value of clocks can only be compared with constant expressions, and not with other clocks. We, however, allow such expressions to be combined using boolean operators, as well as boolean-valued functions, possibly resulting in expressions involving more than one clock. Atomic expressions such as naturals are constants, as

Table 5 Time primitives of RoboChart

Syntax	Metamodel element	Description
#C	ClockReset	Resets clock C
since(C)	ClockExp	Time elapsed since the most recent reset of clock C
sinceEntry(S)	StateClockExp	Time elapsed since state S was entered
A <{d}	TimedStatement	Deadline on action A to terminate within d time units
e <{d}	Transition	Deadline on event e to happen within d time units
Wait(d)	Wait	Explicit time budget of d time units
Wait([a,b])	Wait	Nondeterministic time budget of d time units where $a \leq d \leq b$

Table 6 Summary of Timed CSP operators related to time

Symbol	Name	Description
$Wait(d)$	wait	Terminates exactly after d time units
$P \triangle_d Q$	timed interrupt	Initially behaves as P and after exactly d time units behaves as Q
$P \blacktriangleright d$	deadline	Deadline on P to terminate within d time units

are variables declared as constant, and any combination of such expressions using arithmetic operators. More generally, a function application whose arguments are constant is also constant.

Timed statements

TS1 *A clock reset* #C *may only reference a clock within its containing state machine.*

The semantics for models that are well-formed, according to the above conditions, is described next.

6.2 Semantics

In this section, we use the compositional untimed semantics of Sect. 4 as a basis to define a timed semantics for RoboChart. As previously stated, our formalisation targets the UTP, but we use CSP as a front end and, in particular, for modelling time, we use discrete Timed CSP [88] enriched with a notion of deadlines [98]. For the purpose of early validation using FDR, we encode this Timed CSP semantics into a dialect of CSP, namely, tock-CSP, as described later in Sect. 6.3. In Sect. 6.2.1, we briefly introduce Timed CSP. In Sect. 6.2.2, we provide an overview of the timed semantics, which we formalise in Sect. 6.2.3.

6.2.1 Timed CSP

Timed CSP is a timed version of CSP for specification of real-time properties. It has operational and denotational semantics for continuous time, whose models have been extensively studied [23]. Here, we use the discrete-time version of CSP. The operators presented in Table 2 are equally applicable in Timed CSP. Prefixing ($c \rightarrow P$), in particular, allows the implicit passage of time until the point where the environment is ready to engage in the event c, after which it takes place immediately and then behaves as P. Internal and unobservable behaviour is assumed to take place as fast as possible, given that it is not under the control of the environment, and thus respects the principle of maximal progress whereby as much internal behaviour as possible takes place before time can advance.

In addition to the standard operators of CSP (Table 2), we list three timed operators in Table 6 that we use extensively: $Wait(d)$, which waits for d time units before terminating, $P \triangle_d Q$, which is a strict timed interrupt that initially behaves as P and exactly after d time units then behaves as Q, and $P \blacktriangleright d$, which requires P to terminate within d time units. The first two are Timed CSP operators, and the last is a deadline operator. We consider, however, a discrete-time semantics for these operators. It is given in the context of *Circus Time* [91, 98], a timed process algebra based on CSP, and, therefore, Timed CSP, but with a UTP discrete-time model. In our tock-CSP encoding, we define all these operators as well (see Sect. 6.3.2).

6.2.2 Overview

To capture the semantics of the timed constructs of a RoboChart model, we do not need to change the semantics of the module and of the controllers, but that of state machines and actions is enriched. The augmented structure of the timed semantics of a state machine is depicted in Fig. 22. We have an additional process *Clocks* composed in parallel with the machine process.

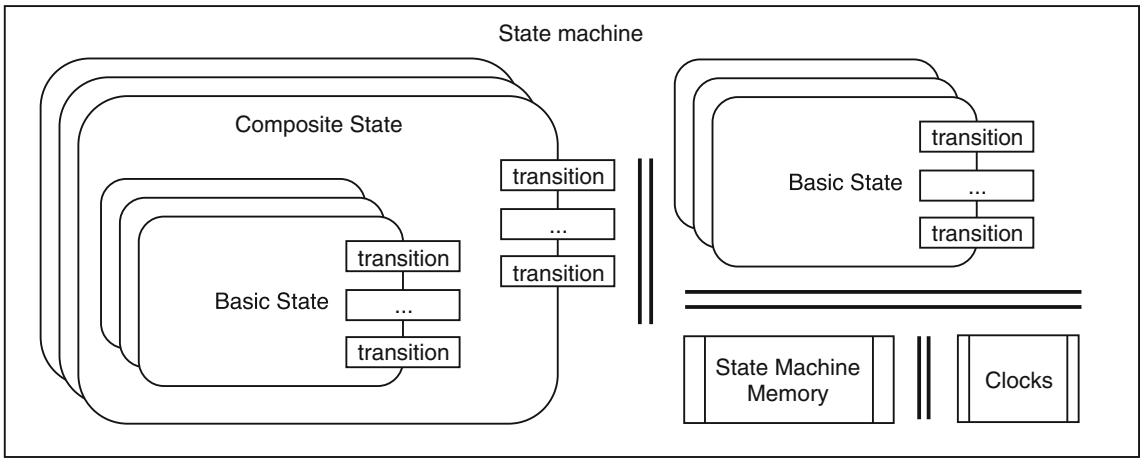

Fig. 22 Structure of the timed RoboChart state machine semantics: parallel lines indicate parallel composition; bordered boxes indicates points of interaction

To illustrate the new structure and some of the additional changes we sketch below the timed semantics of the machine DTP from the foraging example (Fig. 5). Compared to the untimed semantics, we have a similar structure: *Init* and the state processes are composed with the parallel composition of memory and clock processes, denoted here by *Memory_DTP* and *Clocks_DTP*. In this example, *Clocks_DTP* models T, and the parallel composition with *Memory_DTP* represents a memory that can deal with guards that depend on time. The synchronisation set for the outer parallelism includes, in addition to the memory *set* and *get* events, the event $clockReset.T$ that the *Init* and state processes can use to reset the clock T.

The constants, such as, nest provided by the robotic platform, are potentially used in both *Memory_DTP* and *Clocks_DTP*. So, their values are determined using a *set* communication. In this example, none of the constants have their values defined in the machine.

Clocks are not given semantics directly; instead, each expression in a guard that uses since or sinceEntry is modelled explicitly, rather than by the evaluation of clock values. For example, in DTP we have two guards that use the since primitive: since(T)>waitDeadline and since(T)> explorationDeadline. They are encoded by the boolean variables wc_T_1 and wc_T_2 represented in the memory process like all other variables. They are updated in the process *Clocks_DTP* using the channels $setWC\_T_1$ and $setWC\_T_2$. The process denoted by *Memory_DTP* records these two additional variables, which are initially *false*. This initialisation is arbitrary, as the initial value of these variables is determined in *Clocks* and set using $setWC\_T_1$ and $setWC\_T_2$. Guards are still modelled in *Memory_DTP* since they can in general depend on clocks as well as other variables.

$$
\begin{aligned}
&DTP = \\
&\left(\begin{array}{l}
\left(\begin{array}{l}
Init \\
\quad [\![EnterExitChannels]\!] \\
\left(\begin{array}{l} ExploringR \\ [\![Exploring\_S \cap \ldots]\!] \\ \ldots \end{array}\right) \\
\setminus (Exploring\_S \cap \ldots)
\end{array}\right) \setminus (\Sigma \setminus TRoboEvents) \\
\quad [\![\{\!| set\_DTP\_P, \ldots, clockReset.T |\!\}]\!] \\
set\_DTP\_nest?nest \rightarrow \ldots \\
\left(\begin{array}{l} Memory\_DTP \\ [\![\{\!| setWC\_T_1, setWC\_T_2, \ldots |\!\}]\!] \\ Clocks\_DTP \end{array}\right) \\
\setminus \{\!| setWC\_T_1, setWC\_T_2 |\!\}
\end{array}\right) \\
&\quad \setminus \left\{\!\!\left|\begin{array}{l} set\_DTP\_P, get\_DTP\_P, \\ get\_DTP\_P, get\_DTP\_source, \\ get\_DTP\_dist, get\_DTP\_position, \\ clockReset.T \end{array}\right|\!\!\right\}
\end{aligned}
$$

The original hiding of all events (that is, those in the set Σ) except those in *RoboEvents* is changed to consider *TRoboEvents*. This includes all events of *RoboEvents*, plus $clockReset.T$ that can be used by *Init* and the state processes to reset T in agreement with *Clocks_DTP*.

Clocks As previously said, expressions using since(C) or sinceEntry(S) to compare a constant expression with a clock C, explicitly indicated in the case of since(C), or implicitly in the case of sinceEntry(S), are modelled by boolean variables. The value of such a variable is controlled by a "waiting-condition" process that toggles it according to the time passed since a clock reset or state entry. The clock process is defined by the parallel composition of these condition processes.

For example, for the guard of the transition from the state WaitForTransfer to GoToNestDirectly in the machine DTP, namely, since(T)>waitDeadline, we have the following condition process WC_T1. Initially it sets the value of the

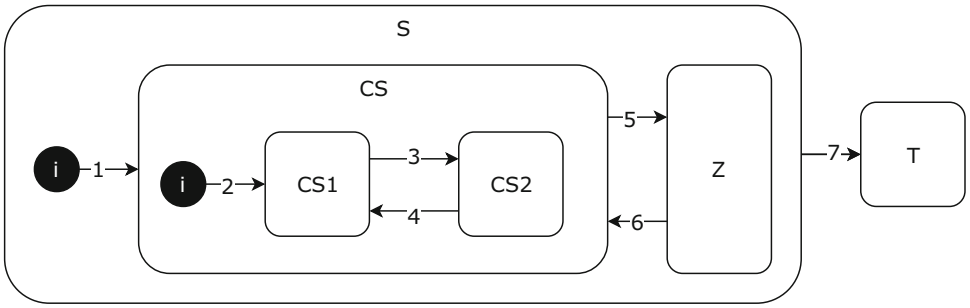

Fig. 23 Example of a partial diagram with composite states S and CS

variable wc_T1 to *false* by synchronising on $setWC\_T_1!false$. This captures the assumption that all clocks are initially set to 0. The type of since(C) and sinceEntry(S) expressions is that of the natural numbers, and in a well-typed model they are compared to expressions of the same type. Here, waitDeadline is a natural constant, and so not smaller than 0.

$$WC\_T1 = setWC\_T_1!false \rightarrow WC\_T1\_monitor$$

After the initialisation, WC_T1 behaves as follows.

$$WC\_T1\_monitor = \left(\begin{array}{l} RUN(\{\!|T_1|\!\}) \triangle_{waitDeadline+1} \\ setWC\_T_1!true \rightarrow RUN(\{\!|T_1|\!\}) \\ \quad WC\_T1\_reset \end{array} \right) \triangle$$

$$WC\_T1\_reset = clockReset!T \rightarrow setWC\_T_1!false \rightarrow WC\_T1\_monitor$$

This process is willing to synchronise indefinitely (as defined by $RUN(\{\!|T_1|\!\})$) on T_1, the CSP event that uniquely identifies the transition from WaitForTransfer to GoToNestDirectly, until exactly $waitDeadline+1$ time units have passed, when it sets wc_T1 to *true* (using the communication $setWC\_T_1!true$) and accepts T_1 indefinitely once more. At any point it can be interrupted by a $clockReset!T$ event as defined by WC_T1_reset. Following the interruption, it sets wc_T1 to *false* and then behaves as $WC\_T1\_monitor$ again.

Synchronisation on $T1$ is used in $WC\_T1\_monitor$ to control when the state machine and the memory processes can synchronise on $T1$. It ensures that they synchronise only once an update to the value of wc_T1 has been considered by the memory process. For that, WC_T1 withdraws the possibility to synchronise on $T1$, while the value is being set via the synchronisation with the memory process on $setWC\_T1$. Once wc_T1 has been set, $T1$ is offered again, thus allowing the state machine and memory processes to synchronise on $T1$ (if that is possible as determined by the memory process). Without this synchronisation, it would be possible for the machine and memory processes to enable the transition identified by $T1$, ignoring the synchronisation on $setWC\_T1$ to update the value of its guard.

The complete clock process is then defined as the parallel composition of every waiting-condition process for every expression involving since or sinceEntry in every transition of a state machine. For example, in the case of *DTP*, it is the process named *Clocks* below.

$$\alpha_{WC\_T1} = \{\!|T_1, setWC\_T_1, clockReset.T|\!\}$$
$$\alpha_{WC\_T2} = \{\!|T_2, setWC\_T_2, clockReset.T|\!\}$$
$$Clocks = WC\_T1[\alpha_{WC\_T1} \parallel \alpha_{WC\_T2}]WC\_T2$$

The process *Clocks* composes WC_T1 and WC_T2 using the alphabetised parallel operator, with the set of events used in WC_T1 denoted as α_{WC_T1}, and as α_{WC_T2} for WC_T2. The processes synchronise on the intersection of α_{WC_T1} and α_{WC_T2}. In our example above, this is the $clockReset.T$ event, so that a clock reset is taken into account by both processes. In general, if the guard for a transition has several conditions on a clock, the processes for these conditions synchronise on the CSP event for the transition as well.

Trigger deadlines The semantics of states is largely unchanged from that in Sect. 4, requiring only a small modification to cater for deadlines on transition triggers. Such a deadline is relevant once its associated transition is enabled, that is, the source state is active and the transition's guard is true. For example, in Movement (Fig. 20) the deadline odometer?d1 <{0} on the transition from AssessPosition to AvoidingAgain is enabled whenever AssessPosition is active.

More broadly, we consider the example in Fig. 23, where we have a composite state S with a transition (7) that we suppose to have a trigger deadline. Transition (7) is enabled when its guard is true and S has entered, which, in addition to completing the transitions (1) and (2) and its entry action, requires that the substates CS and CS1 have also entered, and so completed their own entry actions, if any. Whenever the guard of the transition (7) is false, the deadline in (7) is disabled. We observe, however, that the deadline is not disabled even if the transitions (3), (4), (5) or (6) are triggered. To capture this semantics, a state process uses additional CSP events to query the memory process as to whether a transition's guard is enabled.

Below, we show the modified definition of *S_main*.

$$
\begin{array}{l}
S\_main = \\
\quad enter?s!SID \rightarrow \\
\quad \left(\begin{array}{l}
entry; \\
enter!SID!SSID \rightarrow entered!SID!SSID \rightarrow \\
entered!s!SID \rightarrow \\
\quad \left(\begin{array}{l}
((during;\ STOP) \ |||\ triggerDeadlines) \\
\quad \triangle \left(\begin{array}{l} transitions\_of\_S \\ \Box \\ all\_other\_transitions\_S \end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
$$

The only difference compared to the untimed semantics is that, in interleaving with the process that models the *during* action, we compose a process *triggerDeadlines* that models the trigger deadlines.

This process, defined below, controls, for each transition with a deadline, when that deadline should be imposed, and enforces it. The activation and deactivation of the deadline is controlled using *deadline.tid.on* and *deadline.tid.off* events for each transition identifier *tid*, based on the guard of the transition, which is evaluated as part of the state machine's memory process.

Below, *triggerDeadlines* is defined as an interleaving of processes $td(tid, d)$, where the pairs (tid, d) are taken from a set *tDS* of identifiers *tid* and associated deadlines *d* for all transitions from S with a deadline.

$$triggerDeadlines = |||(tid, d) : tDS \bullet td(tid, d)$$

These processes interact with the memory process using *deadline!tid!on* and *deadline!tid!off*, which, based on the guard of the transition *tid*, determine when that transition is enabled or disabled, and the deadline must be enforced or cancelled. We define $td(tid, d)$ below.

$$
\begin{array}{l}
td(tid, d) = deadline.tid.on \rightarrow \\
\qquad\qquad (deadline.tid.off \rightarrow Skip \blacktriangleright d);\ td(tid, d)
\end{array}
$$

This process captures the behaviour when, given the currently active state, enabledness of the transition *tid* depends only on its guard. So, it is determined by the event *deadline.tid.on*. If that event takes place, the deadline is enforced by accepting a cancellation via the event *deadline.tid.off*, which happens if the guard becomes false, but must happen within *d* time units.

Effectively, a deadline can be met in two ways: the guard becomes false according to the synchronisation on *deadline.tid.off* within *d* time units, or a transition out of S takes place in S_{main}, in which case the behaviour of *triggerDeadlines* is interrupted ($\triangle$ in S_{main}). The deadline on the process *deadline.tid*, *off* $\rightarrow$ *Skip* is enough to impose a deadline on all other events offered in choice. So, the possibility to be interrupted by events in *transitions_of_S* and *all_other_transitions* can only be realised within the deadline, before *deadline.tid.off*.

For every transition *tid* with a trigger deadline and a guard g, the memory process offers in choice the events *deadline.tid.on* and *deadline.tid.off* as sketched below.

$$
\begin{array}{l}
Memory\_STM(\ldots) = \\
\quad \left(\begin{array}{l}
\ldots \\
\Box \\
g\ \&\ deadline!tid!on \rightarrow Memory\_STM(\ldots) \\
\Box \\
\neg\ g\ \&\ deadline!tid!off \rightarrow Memory\_STM(\ldots)
\end{array} \right)
\end{array}
$$

Whenever the guard g of a transition identified by *tid* is true, the memory process can synchronise on the event *deadline.tid.on*, and on *deadline.tid.off*, otherwise. If a transition has no guard, then only *deadline.tid.on* is available in the memory process.

Action deadlines As already discussed, the action language of RoboChart supports the definition of deadlines on actions (A <{d}). The semantics of such an action is given directly in terms of our deadline operator ($\blacktriangleright d$).

Next, we formalise the necessary changes to the semantics rules of Sect. 4 to deal with time.

6.2.3 Formalisation

In this section, we formalise the RoboChart timed semantics using discrete Timed CSP augmented with a deadline operator (Table 6). Because we fully reuse the untimed semantics, we adopt many of the rules from Section 4 by considering their result as discrete Timed CSP processes. Since a process defined using purely the CSP operators is a valid Timed CSP process, the syntactic category TimedCSPProcess of the metamodel for Timed CSP includes all the terms of CSPProcess. So, we can specify that the main semantic functions define terms of TimedCSPProcess without extra changes beyond those we have discussed above.

The semantic rules for modules and controllers are unchanged, whereas those for state machines and states are redefined. In addition, we define rules to calculate waiting-condition processes, augment the structure of the state machine memory process, and change the composition of processes for composite states to cater for trigger deadlines. Finally, we give semantics to deadlines over actions and to budgets.

State machines Rule 14 for the timed semantics of state machines is similar to Rule 5 from Sect. 4. The difference is in the memory process: it is defined by a function stmMemory(stm, wcs), which takes an extra parameter

Rule 14. Timed semantics of state machines $[\![\text{stm} : \text{StateMachineDef}]\!]_{STM} : \text{TimedCSPProcess} =$

$$
\begin{array}{l}
\Big(\big(\text{initialisation(stm)} \; [\![\text{flowevts}]\!] \; \text{composeStates}(\langle \text{s} : \text{stm.nodes} \mid \text{s} \in \text{State} \rangle, \text{stm}) \big) \setminus \{\!| \mathit{enter}, \mathit{exit}, \mathit{exited} |\!\} \\
\quad [\![\text{getsetChannels(stm)} \cup \text{trigEvents(stm)} \cup \text{clockResets(wcs)} \cup \text{deadlineEvents(stm)}]\!] \\
\quad \text{constInitSTM(consts, stm,} \; \big(\text{stmMemory(stm, wcs)} \; [\![\text{clockMemSync}]\!] \; \text{stmClocks(wcs)} \big) \big) \setminus (\text{clockMemSync} \setminus \text{trigEvents(stm)}) \Big) \\
[\![\text{renameTriggerEvents(stm)}]\!] \setminus \text{getsetLocalChannels(stm)} \cup \text{clockResets(wcs)} \cup \text{deadlineEvents(stm)} \cup \{\!| \mathit{internal}, \mathit{entered} |\!\} \\
\Theta_{\{end\}} \mathit{Skip} \\
\textbf{where} \\
\quad \text{wcs} = \{\text{t} : \text{allTransitions(stm)} \mid \text{t.condition} \neq \text{null} \bullet \text{t} \mapsto \text{wc(t.condition)}\} \\
\quad \text{clockMemSync} = \{\!| \text{t} : \text{Transition} \mid \text{t} \in \text{dom wcs} \bullet \text{triggerEvent(t)} |\!\} \cup \{\!| \text{v} : \text{allClockVariables(wcs)} \bullet \mathit{setWC}\_\text{vid(v)} |\!\} \\
\quad \text{flowevts} = \bigcup \{\text{x} : \text{SIDS} \setminus \text{states(stm)}; \; \text{y} : \text{states(stm)} \bullet \{\!| \mathit{enter}.\text{x}.\text{y}, \mathit{entered}.\text{x}.\text{y}, \mathit{exit}.\text{x}.\text{y}, \mathit{exited}.\text{x}.\text{y} |\!\}\} \\
\quad \text{consts} = \langle \text{v} : \text{allConstants(stm)} \bullet \text{v} \rangle
\end{array}
$$

Rule 15. Timed state machine memory
$\text{stmMemory(stm} : \text{StateMachineDef, wcs} : \text{Transition} \nrightarrow (\text{Expression, WC})) : \text{TimedCSPProcess} =$

$$
\begin{array}{l}
\textbf{let} \; \mathit{Memory}(\text{vars}) \mathrel{\widehat{=}} \\
\quad \Box \, \mathit{v} : \text{lvars} \bullet \mathit{get}\_\text{vid(v)}!\text{name(v)} \rightarrow \mathit{Memory}(\text{vars}) \; \Box \; \mathit{set}\_\text{vid(v)}?x \rightarrow \mathit{Memory}(\text{vars[name(v)} := \text{x]}) \\
\quad \Box \\
\quad \Box \, \mathit{v} : \text{rvars} \bullet (\mathit{get}\_\text{vid(v)}!\text{name(v)} \rightarrow \mathit{Memory}(\text{vars}) \; \Box \; \mathit{set}\_\text{vid(v)}?x \rightarrow \mathit{Memory}(\text{vars[name(v)} := \text{x]}) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{set\_Ext}\_\text{vid(v)}?x \rightarrow \mathit{Memory}(\text{vars[name(v)} := \text{x]})) \\
\quad \Box \\
\quad \Box \, \mathit{v} : \text{allConstants(stm)} \bullet \mathit{get}\_\text{vid(v)}!\text{name(v)} \rightarrow \mathit{Memory}(\text{vars}) \\
\quad \Box \\
\quad \Box \, \mathit{t} : \text{allTransitions(stm)} \bullet \text{memoryTransition(t, wcs)}; \; \mathit{Memory}(\text{vars}) \\
\quad \Box \\
\quad \Box \, \mathit{v} : \text{cvars} \bullet \mathit{setWC}\_\text{vid(v)}?x \rightarrow \mathit{Memory}(\text{vars[name(v)} := \text{x]}) \\
\quad \Box \\
\quad \Box \, \mathit{t} : \text{allDeadlineTransitions(stm)} \bullet \text{memoryDeadline(t, wcs)}; \; \mathit{Memory}(\text{vars}) \\
\textbf{within} \\
\quad \mathit{Memory}(\text{varvalues}) \\
\textbf{where} \\
\quad \text{rvars} = \text{requiredVariables(stm)} \\
\quad \text{lvars} = \text{allLocalVariables(stm)} \\
\quad \text{consts} = \langle \text{v} : \text{allConstants(stm)} \bullet \text{v} \rangle \\
\quad \text{cvars} = \text{allClockVariables(wcs)} \\
\quad \text{vars} = \langle \text{v} : \text{rvars} \cup \text{lvars} \cup \text{cvars} \bullet \text{name(v)} \rangle \frown \langle \text{v} : \text{consts} \bullet \text{name(v)} \rangle \\
\quad \text{varvalues} = \langle \text{v} : \text{rvars} \cup \text{lvars} \cup \text{cvars} \bullet \text{initial(v)} \rangle \frown \langle \text{v} : \text{consts} \bullet \text{name(v)} \rangle
\end{array}
$$

wcs, a partial function from transitions to pairs defined by applying a syntactic function wc to the guards t.condition of transitions t in allTransitions(stm) with a guard (t.condition $\neq$ null). The first component of wc(t) is t's guard with boolean variables replacing comparisons with since or sinceEntry, and the second is a partial function from expressions to variables. This function, which we formally define in Rule 17, maps each expression that defines a comparison with since and sinceEntry to the boolean variable that replaces it.

We consider, for instance, the machine Movement in Fig. 20 with two transitions whose guards depend on the clock T. For this example, we refer to the transition from AvoidingAgain to GettingOut as T_0. In this case, the set wcs includes the mapping below.

$$
wcs = \left\{ T_0 \mapsto \begin{pmatrix} (\mathsf{b}_0 \wedge \mathsf{d1} - \mathsf{d0} \leq \mathsf{stuckDist}), \\ \{\mathsf{since(T)} >= \mathsf{stuckPeriod} \mapsto b_0\} \end{pmatrix}, \ldots \right\}
$$

T_0 is mapped to a pair: the first component is the guard, namely, $\mathsf{since(T)} >= \mathsf{stuckPeriod} \wedge \mathsf{d1} - \mathsf{d0} \leq \mathsf{stuckDist}$, with the expression since(T)>=stuckPeriod replaced by a fresh boolean variable b_0. The second component is the function that maps the original expression in the guard, since(T)>=stuckPeriod, to the variable b_0.

The memory process stmMemory(stm, wcs) is composed in parallel with the clocks process stmClocks(wcs) synchronising on the events in clockMemSync. The definition of this set is based on the transitions t in the domain of wcs. It includes their trigger events, determined by triggerEvent(t), used by both the memory and clocks processes to control the transitions. We also include *setWC_* events for the boolean variables used to represent expressions involving clocks, as defined by the function allClockVariables(wcs). The *setWC_* events are used by the memory and clocks processes only, so we hide the events in the set difference between clockMemSync and trigEvents(stm), so that the composition of the memory and clock processes synchronises with the state processes on trigEvents(stm), which, we recall, contains the trigger events for all transitions in the machine.

Because stmClocks(wcs) may use constants, the definition of their values is handled by applying the function constInitSTM to the parallel composition of the processes stmMemory(stm, wcs) and stmClocks(wcs), instead of as part of the definition of stmMemory (see Rule 12). Effectively, this composition is a timed memory process that takes into account clocks.

The parallel composition of the processes defined by initialisation(stm) and composeStates(...) is similar to that in Rule 5 for the untimed semantics except that the channel set *entered* is not hidden after this composition so as to allow waiting-condition processes within stmClocks(wcs) that model expressions involving sinceEntry(S) to synchronise on events *entered*.*x*.*S*. Instead, the hiding of *entered* events is scoped further outside the parallel composition together with *internal*.

Compared to Rule 5, the hiding is extended to cover events related to clock resets (clockResets(wcs)) and trigger deadlines (deadlineEvents(stm)). The synchronisation set of this parallel composition with the processes defined by stmMemory(stm, wcs) and stmClocks(wcs) is extended in a similar way. The function clockResets(wcs) defines the set of all events *clockReset*.*C* for every clock C declared in the state machine, and deadlineEvents(stm) the set of all events *deadline*.*tid*.*on* and *deadline*.*tid*.*off*, for every identifier *tid* for a transition with a deadline.

State machine memory Compared to Rule 12 for the memory process in the untimed model, Rule 15 introduces only small changes. As already mentioned, the function stmMemory that it defines has an extra parameter: a partial function wcs from transitions to pairs of expressions and boolean variables. The type WC is that of partial functions Expression $\nrightarrow$ Variable, where an expression comparing since or sinceEntry using one of the allowed operators (defined by well-formedness condition TE4 for timed expressions in Sect. 6.1.2) is mapped to a boolean variable. The parameter wcs is required to formalise the waiting conditions.

In Rule 15, using the extra parameter wcs, we define a set cvars of all boolean variables used for expressions involving clocks, as defined by allClockVariables(wcs). The sequences vars and varsvalues are defined to consider all variables, including those in cvars. Accordingly, the choices in *Memory* are augmented with two replicated external choices. One is over each boolean variable *v* in cvars, to synchronise on *setWC_*vid(v) and recurse with the value of *v* updated. The other is over each transition t with a trigger deadline as defined by the function allDeadlineTransitions(s), synchronising on the process memoryDeadline, which is defined by Rule 16.

For a transition t with a guard (t.condition $\neq$ null), Rule 16 defines a process in which *deadline*.id(t).*on* is available when the guard, $[\![\pi_1(\text{wcs(t)})]\!]_{\mathcal{E}xpr}$, rewritten to take the clocks into account, is true, and the event *deadline*.id(t).*off* is offered whenever it is false. The function π_1 is used to project the first component of the pair mapped from t in wcs. If no guard exists, *deadline*.id(t).*on* is always offered.

Lastly, the function memoryTransition that we use in Rule 15 takes the function wcs as a parameter so that a transition's guard whose expression gives rise to a waiting condition can be defined based on the expression recorded in wcs. The formalisation of both waiting conditions and trigger deadlines is discussed next.

Waiting conditions The function wc used in the definition of Rule 14 is defined by Rule 17. It takes an expression exp and yields a pair, whose first component is the result of replacing every comparison with a clock in exp by a fresh boolean variable, and whose second component is a partial function of type WC.

Rule 17 shows part of the definition of wc for the cases where it is applied to expressions since(C)>=e, since(C)>e, $e_1 \wedge e_2$, and for expressions that do not involve the since and sinceEntry constructs, identified by the pattern _ (underscore), which matches anything. Other cases are similar and omitted here. A complete definition can be found in [96]. For since(C)>=e, the first component of the pair defined by wc is the expression b, a fresh name for a boolean variable, and whose second component is the set where there is a single mapping from the expression since(C)>=e to b. The other cases involving since(C) and sinceEntry(S) are similar. The conjunction of two expressions e_1 and e_2 is defined by considering the application of wc to e_1 and e_2. The first component of the resulting pair is the conjunction of the first components resulting from each application of wc, that is, $\pi_1(\text{wc}(e_1)) \wedge \pi_1(\text{wc}(e_2))$. The second component is the union of the second components resulting from each application of wc, that is, $\pi_2(\text{wc}(e_1)) \cup \pi_2(\text{wc}(e_2))$. Finally, wc(_)

Rule 16. Memory: trigger deadline
memoryDeadline(t : Transition, wcs : Transition ⇸ (Expression, WC)) : TimedCSPProcess

if (t.condition ≠ null) then
  $(\llbracket \pi_1(\text{wcs(t)}) \rrbracket_{\mathcal{E}xpr})$ & $deadline$.id(t).$on \rightarrow Skip \;\Box\; (\neg \llbracket \pi_1(\text{wcs(t)}) \rrbracket_{\mathcal{E}xpr})$ & $deadline$.id(t).$off \rightarrow Skip$
else
  $deadline$.id(t).$on \rightarrow Skip$

Rule 17. Eliciting waiting conditions wc(exp : Expression) : (Expression, WC)

wc(since(C) >= e) = (b, {(since(C) >= e) ↦ b})
wc(since(C) > e) = (b, {(since(C) > e) ↦ b})
...
$\text{wc}(\text{e}_1 \wedge \text{e}_2) = (\pi_1(\text{wc}(\text{e}_1)) \wedge \pi_1(\text{wc}(\text{e}_2)), \pi_2(\text{wc}(\text{e}_1)) \cup \pi_2(\text{wc}(\text{e}_2)))$
wc(_) = (_, ∅)

where b is a fresh identifier

Rule 18. State machine clocks stmClocks(wcs : Transition ⇸ (Expression, WC)) : TimedCSPProcess =

$\Big\|\,(t, e, v) : \{\text{t : Transition, e : Expression, v : Variable} \mid \text{t} \in \text{dom wcs} \wedge (\text{e} \mapsto \text{v}) \in \pi_2(\text{wcs(t)})\}$
  • $[\![\alpha\text{WC(t, e, v)}]\!]$ compileWC(t, e, v)
where
αWC(t, e, v) = {|triggerEvent(t), $setWC$_vid(v)|} ∪ alphaClockReset(e)

Rule 19. Waiting condition compileWC(t : Transition, exp : Expression, v : Variable) : TimedCSPProcess

compileWC(t, since(C) >= e, v) =
let $Reset = clockReset$.id(C) → $setWC$_vid(v)!$false \rightarrow Monitor$
  $Monitor = (RUN(\{|\text{triggerEvent(t)}|\}) \triangle_{\llbracket \text{e} \rrbracket_{\mathcal{E}xpr}} setWC\text{\_vid(v)}!true \rightarrow RUN(\{|\text{triggerEvent(t)}|\})) \triangle Reset$
within
  $setWC$_vid(v)!$false \rightarrow Monitor$
...

Rule 20. Semantics of clock reset $\llbracket$s : ClockReset$\rrbracket_{\mathcal{S}tatement}$: TimedCSPProcess =

$clockReset$.id(s.clock) → $Skip$

Rule 21. Semantics of wait $\llbracket$s : Wait$\rrbracket_{\mathcal{S}tatement}$: TimedCSPProcess =

$\llbracket$s.duration$\rrbracket_{\mathcal{W}ait}$
where
$\llbracket$e : RangeExp$\rrbracket_{\mathcal{W}ait}$ = $\prod n : \llbracket \text{e} \rrbracket_{\mathcal{E}xpr} \bullet Wait(n)$
$\llbracket$e : Expression$\rrbracket_{\mathcal{W}ait}$ = $Wait(\llbracket \text{e} \rrbracket_{\mathcal{E}xpr})$

Rule 22. Semantics of statement deadlines $[\![\text{s : TimedStatement}]\!]_{Statement}$: TimedCSPProcess =

$$[\![\text{s.stmt}]\!]_{Statement} \blacktriangleright [\![\text{s.end}]\!]_{\mathcal{E}xpr}$$

Rule 23. Semantics of composite states $[\![\text{s : State}]\!]_{\mathcal{S}}$: TimedCSPProcess =

let

$Inactive \mathrel{\widehat{=}} enter?o : \text{sids.id(s)} \rightarrow Activating(o)$

$Activating(o) \mathrel{\widehat{=}} [\![\text{s.entry}]\!]_{Action}; \text{ initialisation(s)}; entered.o.\text{id(s)} \rightarrow ([\![\text{s.during}]\!]_{Action}; Stop \;|||\; \text{triggerDeadlines(s)}) \triangle$

$$\left(\begin{array}{l} \Box\, t : \text{transitionsFrom(s)} \bullet [\![t, \text{s}, \text{false}]\!]_{\mathcal{T}}^{Inactive,Activating} \\ \Box \\ \Box\, e : \text{Event} \bullet \text{if (e.type == null) then eventId(e)?}x : \text{tids} \rightarrow \text{exit}; \; Inactive \\ \qquad\qquad \text{else eventId(e)?}x : \text{tids?}y \rightarrow \text{exit}; \; Inactive \end{array} \right)$$

within

$(Inactive \;[\![\; \text{flowtrigevts} \;]\!]\; \text{composeStates}(\langle \text{x : states(s)} \rangle, \text{s}))$

where

flowtrigevts = flowTriggerEvents(s)

sids = $SIDS \backslash \{\text{id(s)}\}$

exit = $exit?x : \text{sids.id(s)} \rightarrow \text{exitSubstates(s)}; \; [\![\text{s.exit}]\!]_{Action}; \; exited.x.\text{id(s)} \rightarrow Skip$

tids = $TIDS \backslash \text{tIDS(s)}$

considers expressions _ not affected; in this case, it defines the pair whose components are _ and the empty function.

Rule 18 defines the clock process stmClocks(wcs) as the replicated alphabetised parallel composition of processes compileWC(t, e, v) specified in Rule 19. Here, t, e, and v are drawn from the set obtained by considering every transition t in the domain of wcs, and expression e mapped to a variable v in the partial function in the second component (π_2) of the pair mapped from t in wcs. These are, therefore, the comparisons e in the guard of t, involving since and sinceEntry constructs, and the variables v used to represent those expressions.

For each compileWC(t, e, v) process, the synchronisation set αWC(t, e, v) contains the trigger events for t, the $setWC\_$ events for v, and the clock reset events for the clocks in e as defined by alphaClockReset(e). The definition of this function, omitted here, considers all possible valid expressions (like the definition of wc in Rule 17). For example, alphaClockReset, when applied to the comparison since(C) >= e, yields the set containing $clockReset.C$, and when applied to an expression involving sinceEntry(S) yields the events $entered$.x.id(S) where x is a valid state identifier.

We sketch Rule 19, which defines compileWC. We show the definition for since(C)>=e; other cases are in [96]. The process compileWC(t, since(C) >= e, v) first sets the value of v in the memory process to *false* using setWC_vid(v)!*false* and then behaves as the process *Monitor*. This synchronises any number of times (using *RUN*) on the trigger for t. After exactly e time units it is interrupted ($\triangle_{[\![\text{e}]\!]_{\mathcal{E}xpr}}$) and synchronises on $setWC_$vid(v)!*true* to set v to *true*, and then offers the trigger once more. At any point it can be interrupted by the event $clockReset$.id(C), as defined by *Reset*, sets v to *false*, and then behaves as *Monitor* again.

Statements A clock reset #C specified in an action, or as part of a trigger, is formalised by Rule 20: we have a prefixing on $clockReset$.id(s.clock), where id(s.clock) identifies the clock that is being reset.

The semantics of a statement s of the form Wait(e) is defined by Rule 21. In this case, we use the semantics of the expression s.duration, which identifies e, as defined by another function $[\![\_]\!]_{\mathcal{W}ait}$. Rule 21 also defines this semantic function, by considering two cases. If e is a range expression [a,b], the semantics is a nondeterministic choice of $Wait(n)$ processes, where n ranges over the values determined by e as defined by its semantics $[\![\text{e}]\!]_{\mathcal{E}xpr}$ as an expression. These are the values in the closed interval [a,b]. The second case in the definition of $[\![\_]\!]_{\mathcal{W}ait}$ considers expressions of any other type, and so the semantics is defined by *Wait* with an argument given by the semantics of e.

Finally, Rule 22 defines a deadline on an action. We use the deadline construct to specify that the process that gives semantics to the action s must terminate within the time specified by the expression s.end.

States As said, the timed semantics of a state is largely similar to the untimed semantics. For composite states, it is in Rule 23. We focus on the changes required to accommodate trigger deadlines and clock expressions defined

Rule 24. Composition of states composeStates(ss : seq State, p : NodeContainer) : TimedCSPProcess =

```
if #ss = 1 then
  restrictedState(p, head ss)
else
  (restrictedState(p, head ss) ⟦ cflowevts ⟧ composeStates(tail ss, p))
where
  cflowevts = flowEvents(head ss, p) ∩ ⋃{s : tail ss • flowEvents(s, p)}
```

Rule 25. Semantics of trigger deadlines triggerDeadlines(s : State) : TimedCSPProcess =

$$\begin{array}{l}\textbf{let}\ \mathit{Deadline}(t) \mathrel{\widehat{=}} \mathit{deadline}.\mathsf{id(t)}.\mathit{on} \rightarrow \\ \qquad\qquad \mathsf{readState(usedVariables(t.end),}\,(\mathit{deadline}.\mathsf{id(t)}.\mathit{off} \rightarrow \mathit{Skip}) \blacktriangleright [\![\mathsf{t.end}]\!]_{\mathcal{E}xpr})\,;\ \mathit{Deadline}(t) \\ \textbf{within} \\ \quad |||\, t : \mathsf{tDS} \bullet \mathit{Deadline}(t) \\ \textbf{where} \\ \quad \mathsf{tDS} = \{\mathsf{t : Transition} \mid \mathsf{t} \in \mathsf{transitionsFrom(s)} \wedge \mathsf{t.end} \neq \mathsf{null}\}\end{array}$$

in transition guards. In interleaving with the semantics of the during action ($[\![\mathsf{s.during}]\!]_{\mathcal{S}tatement}$) we have a process triggerDeadlines(s), whose formalisation we give in Rule 25. The hiding of flowevts in the untimed semantics given by Rule 8 is omitted to allow waiting conditions, defining the semantics of guards that use state clock expressions, such as sinceEntry(S), to observe the *entered* events of a state S. Similarly, and for the same reason, when compared to Rule 6, the hiding of cflowevts is also omitted in Rule 24, defining the composition of states in the timed semantics.

Trigger deadlines The semantics of trigger deadlines for a state s is formalised by Rule 25. It specifies a replicated interleaving of recursive processes for transitions t drawn from a set tDS, containing every transition whose trigger has a deadline (t.end ≠ null). Each recursive process is defined as initially offering the event *deadline*.id(t).*on*, so as to synchronise with the memory process whenever t's guard is true, and then offering to synchronise on *deadline*.id(t).*off* within the time specified by the expression in t.end. If the transition guard becomes false then the process recurses. Here, we use readState (previously introduced in Sect. 4.3) to ensure that the deadline specified by the expression t.end takes into account the value of any variables it may use, as defined by a function usedVariables.

As previously said, RoboTool generates automatically both the untimed and the timed semantics of RoboChart that we have just formalised. We next discuss how RoboTool deals with the timed semantics.

6.3 Verification support

As discussed in Sect. 5, we use FDR for early validation of our semantics via analysis of case studies. For the timed semantics, RoboTool generates automatically models in tock-CSP, an encoding in CSP-M of discrete Timed CSP that uses the event *tock* to mark the passage of time. In Sect. 6.3.1, we describe how RoboTool is extended to calculate the tock-CSP semantics. In Sect. 6.3.2, we discuss support for model checking of timed properties. Finally, in Section 6.3.3 we discuss the application of our technique to case studies.

6.3.1 Tool support

In general, the approach to construction and validation of timed models in RoboTool follows that already discussed in Sect. 5.1. Here, we focus on the changes required to support timed aspects of RoboChart.

The conditions TE2, TE3, and TS1, discussed in Section 6.1.2, are enforced by scoping rules defined using an Xtext scope provider. This is an Xtend class whose methods define how elements of the metamodel, such as a variable or a clock, can be found given their identifiers and the context where they are used. For example, the use of since(C) in an expression gives rise to a recursive look-up in the hierarchy of RoboChart terms that contain that occurrence of since(C) to find the declaration of C within the containing state machine. The metamodel ensures that there is no other point where such a declaration could be. A similar approach is used to implement all scope rules of RoboChart.

The well-formedness conditions TE1 and TE4 are enforced by Xtend methods annotated with `@Check` as already dis-

cussed in Section 5.1. Essentially these methods recursively traverse the model built by Xtext and check that clock expressions, such as since(C) and sinceEntry(S), are contained within a transition's guard, and that such expressions are compared, using only the allowed operators, with constant expressions. Whenever this cannot be ascertained, an error is indicated and the timed CSP-M model is not generated.

Semantics generation We have used Xtext to implement the CSP-M script generator in RoboTool. Namely, we have used the Java dialect Xtend to implement a class with several methods corresponding to the semantic rules presented in this paper as already discussed in Sect. 5.1. Because the timed semantics for many constructs is largely the same as the untimed semantics, the Xtend class for the generator of the timed semantics is implemented as a subclass of that for the untimed semantics. It overrides the methods that implement one of the rules presented in this section.

This reuse is possible because FDR allows CSP-M syntax for timed processes to be written without having to explicitly introduce *tock* events, relying instead on a rewriting function, called a "`Timed` section", which, given an untimed CSP-M script and a function specifying how many tocks should follow a particular event, introduces *tock* events wherever required following the strategy outlined in [88]. For example, we consider below the CSP-M process `P` that offers the events `a` and `b` in an external choice and then recurses.

```
OneStep(_) = 0
Timed(OneStep) {
  P = a -> P [] b -> P
}
```

The application of `Timed(OneStep)` to this fragment, with `OneStep` being defined as 0 for every event (_), is equivalent to the following definition where `tock` is also offered as a choice in `P` to allow time pass, while the process waits for interaction on `a` or `b`.

```
P = a -> P [] b -> P [] tock -> P
```

Similarly, operators like interleaving and parallel composition are transformed to require synchronisation on `tock`, so that a single clock is used for all parallel processes. A complete specification of the `Timed` section facility is available in FDR's manual.[11]

Use of a `Timed` section is very convenient, but there are some technical issues that need to be addressed. Hiding, for example, even when used within a `Timed` section, does not ensure maximal progress. Instead a timed CSP-M process needs to be prioritised explicitly to give internal events priority over *tock*. This can be specified using `prioritise(P,<{},{tock}>)`. The function `prioritise` takes a process `P` as the first parameter, and a sequence `<X_0,...,X_n>` of sets of events as a second parameter where the silent action τ, and $\checkmark$, which signals termination, are implicitly included in `X_0`, so that the events in `X_j` are possible only if those in `X_i` are not available, where `X_i` appears before `X_j` in the sequence. So, `prioritise(P,<{},{tock}>)` ensures maximal progress: time passes in `P` only when there are no internal (τ or $\checkmark$) events available.

We exploit FDR's timed section facility to add *tock* to untimed CSP-M processes by enclosing the generated semantics within a timed section. In every case, we define `OneStep` as the constant function shown above that associates 0 with all events, since they do not consume time themselves. Instead, it is the prefixing on an event that allows time to pass. Prioritisation is defined for the module, controller, and machine processes.

The generator for the timed semantics creates a CSP model that includes all definitions of the untimed semantics to allow analysis using both models. Because we fully reuse the generator for the untimed semantics, however, it is inevitable that the process names are duplicated. In order to avoid this clash we enclose the generated CSP-M syntax in a `module`, a namespace facility that FDR provides to scope declarations. For example, given a CSP-M script file `Movement.csp`, which contains the generated timed semantics for the Movement state machine, we include it within a module named `timed`, declared using the `module` construct.

```
module timed
exports
  include "defs/Movement.csp"
endmodule
```

A CSP-M process with the name `timed::Movement` is then available. It corresponds to the CSP-M process of name `Movement` declared within the CSP-M script file `Movement.csp`, declared after `exports`. We do not follow the same strategy for CSP-M channel names as these are shared between the timed and untimed semantics. Therefore, channel names are declared only by the generator of the untimed semantics.

Process deadline In Sect. 6.2 we use deadlines in the context of discrete Timed CSP. Here, we give the definition of $P \blacktriangleright d$ using tock-CSP.

The deadline for a process P to terminate within d time units is enforced by the parallel composition of P with a process $Wait_u(d)$ that can engage in at most d *tock* events. Moreover, if P terminates, then a fresh event f interrupts the $Wait_u(d)$ process, and thus terminates the parallel composition. For precision, here we use the subscript $_u$ to indicate that P_u is a process where *tock* is not offered implicitly. In this case, for example, the process $Wait_u(d)$ offers exactly a

[11] www.cs.ox.ac.uk/projects/fdr/.

d number of *tock* events and then terminates without offering any more *tock* events. Similarly, $Skip_u$ does not offer *tock* at all, and $f \rightarrow_u P$ does not offer *tock* while f is offered. This distinction is important since the encoding of timed CSP processes $Wait(d)$ and *Skip*, for example, as tock-CSP processes requires that *tock* is offered alongside the possibility to terminate for compositionality. For example, in a parallel composition $P \,[\![\{...\}]\!]\, Q$, termination is only possible when both P and Q terminate, so if we consider P to be *Skip*, then it lets time pass by offering *tock* alongside the possibility to terminate because Q may engage in a number of *tock* events before terminating itself.

$$P \blacktriangleright d = \begin{pmatrix} (P;\ f \rightarrow_u Skip_u) \\ \quad [\![\{tock, f\}]\!] \\ (Wait_u(d) \triangle f \rightarrow_u Skip_u) \end{pmatrix} \setminus \{f\}$$

In this encoding, a deadline is captured by refusing *tock* once it is reached. So, a process that fails to meet its deadline timelocks. Such undesirable behaviour can be identified by checking that *tock* is never refused. Occurrences of the fresh event f used for control are hidden.

Timed interrupt The second operator that is not standard in the Timed CSP and tock-CSP literature is the strict timed interrupt operator ($\triangle_d$) that we use extensively in the semantics of waiting conditions. We define it using *tock*-CSP as follows.

$$P \triangle_d Q = \begin{pmatrix} (P \triangle f \rightarrow_u Skip_u) \\ \quad [\![\Sigma]\!] \\ (RT(d);\ f \rightarrow_u Skip_u) \end{pmatrix} \setminus \{f\};\ Q$$

We have, first of all, a parallel composition of two processes synchronising on every event (Σ): P with the possibility of being interrupted by f, a fresh event; and $RT(d)$ sequentially composed with a prefixing also on f. Finally, f is hidden and there is the sequential composition with Q. The process $RT(d)$ is defined by an external choice of two guarded processes as follows.

$$RT(d) = \begin{pmatrix} d == 0 \,\&\, Skip_u \\ \Box \\ d > 0 \,\&\, \begin{pmatrix} RUN(\Sigma \setminus \{f, tock\}) \triangle \\ tock \rightarrow_u RT(d-1) \end{pmatrix} \end{pmatrix}$$

When d is zero it behaves as $Skip_u$, and thus terminates immediately, and otherwise when $d > 0$ it synchronises on every event in Σ except for f and *tock*, any number of times, but can be interrupted by a *tock* event to behave as $RT(d-1)$. In $P \triangle_d Q$, the process $RT(d)$ ensures that P can only engage in visible events, including *tock*, before a d number of *tock* events occur. After exactly d time units, the synchronisation on f is used to interrupt P and yield control to Q.

We next explain how to use the tock-CSP encoding of our semantics to verify RoboChart models using FDR with the support of RoboTool.

6.3.2 Model checking

In addition to the properties described in Section 5.2, RoboTool generates properties for verification using the timed semantics. All properties already considered for the untimed semantics, except for deadlock freedom, but including termination, divergence freedom, and determinism, are also stated for the timed model. In addition, the following properties are automatically generated: zeno and timelock freedom, reachability for every state in every machine, and proper initialisation of clocks. (All clock expressions in guards are preceded by appropriate clock resets.) As for verification using the untimed semantics, it is also possible to write custom assertions using the timed semantics. These need to be stated as a refinement.

Using the tock-CSP semantics, zeno freedom can be verified by checking for divergence freedom, while timelock freedom is checked by ensuring that *tock* can never be refused. A timelock-free model, however, is not necessarily deadlock free in the sense that it never refuses all events other than *tock*. For example, a process that only ever offers *tock* is timelock free, but not deadlock free in this sense. In general, specifying that a process eventually accepts an event other than *tock*, for use with a model checker like FDR, requires assumptions about the timeliness of events for a particular model [54]. As previously said, such temporal properties are limited to the bounded positive fragment of LTL.

A timelock identifies a scenario where a deadline is potentially not met. As an example, we consider the operation call randomWalk() in the states GettingOut and Waiting of the machine Movement in Fig. 20. As discussed in Sect. 4.2, operation calls are identified by events with suffix *Call* and *Ret*, to indicate an operation call and return, so in this case the call is identified by the events *randomWalkCall* and *randomWalkRet*. The empty and the singleton trace $\langle randomWalkCall \rangle$, for example, are identified as leading to timelocks. The first timelock occurs because once the Waiting state is entered, its during action must start immediately, and the operation randomWalk must be called. The second timelock occurs because a call takes no time, and so, its termination is urgent. Urgency characterises a deadline, and so *tock* is refused. These deadlines can be met only with an assumption that the platform accepts calls to randomWalk

```
assertion Movement is timelock-free (R5)
assertion clock Movement::T is initialised (R6)
assertion Movement is divergence-free (R7)
timed assertion Movement::Found is
      reachable in System (R8)

timed csp Q csp-begin
Q = ...
csp-end
```

Fig. 24 Example of Timed CSP `.assertions` file

at all times, and that this operation terminates without consuming time.

State reachability checks can identify states that are never entered. Because the RoboChart semantics has an explicit model of state activation, we can use $entered.x.\mathsf{id(S)}$ events to check whether a state S can be entered. For this purpose, RoboTool also generates CSP-M processes for each machine, controller, and module, where the *entered* events are visible. In these processes, *entered* events are removed from the hiding in Rule 14 and given priority over *tock* (using the operator `prioritise`), so that their maximal progress is retained even though they are not hidden. The reachability assertion for a state S identified by *sid* is then as shown below, where P is a semantics process (for a state machine, controller, or module).

$$\neg\ Stop_u \sqsubseteq_{FD} P \setminus (\Sigma \setminus \{\!|x : SID \bullet entered.x.sid|\!\})$$

This assertion states that $Stop_u$ is not refined by a (modified) process P giving semantics to a state machine, controller, or module, with only $entered.x.\mathsf{id(S)}$ events visible. The absence of a counterexample indicates that the refinement holds and the assertion is false. So, P can never perform a trace with the event $entered.x.sid$, and so S is not reachable. The existence of a counterexample identifies a trace where S is entered. The possibility for observing *entered* events also allows for animating a state machine, using FDR, while observing its internal control flow.

The tool-independent assertion language described in Sect. 5.2 also caters for timed properties. With that, given a `.assertions` file, like that shown in Fig. 24, RoboTool generates CSP-M assertions for the timed semantics by default as well.

Using the extended language, we can specify if an assertion applies exclusively in the untimed or timed semantics, by prefixing an assertion with `untimed` or `timed`, as shown in Fig. 24. Similarly, custom blocks contained between `csp-begin` and `csp-end` can also be specified as `timed` or `untimed`. In Fig. 24, we have two assertions (`R5` and `R6`) that are only applicable in the timed semantics due to their nature, an assertion `R7` that is checked using both the untimed and timed semantics (because it is not prefixed with `timed`), and `R8` that is checked using the timed semantics only. Assertion `R5` is satisfied if there is no timelock, while assertion `R6` is satisfied if every transition of the machine Movement whose guard uses the clock T can only be triggered after T has been reset at least once. Assertion `R7` is satisfied if there is no divergence, while assertion `R8` is satisfied if the state Found of Movement is reachable in the context of the module System.

Next, we describe the use of RoboTool, and of the timed semantics and assertions that it generates automatically, to verify the examples in Sect. 5.

6.3.3 Case studies

As previously mentioned in Sect. 5.3, we have used RoboChart and RoboTool to model examples from the literature. In Table 7, we show results obtained with FDR to check for divergence freedom in the examples we described in Sect. 5.3, as well as an additional example, but now using the timed semantics. Overall, compared with Table 4, we have an increase in the number of states and transitions visited, mainly due to the use of *tock* to encode time. Despite the increase in complexity, we have a modest increase in the compilation and verification time.

We have also considered more efficient models for the memory processes for state machines, where individual variables are modelled in separate, but parallel, processes that synchronise with a control process. This can yield a reduction in the number of states, which is noticeable when a state machine has several variables or timed conditions, which, as previously discussed, are encoded using boolean variables.

We have also considered the generation of a single waiting-condition process for comparisons that are syntactically equivalent in RoboChart, again yielding efficiency gains in terms of state-space complexity. These optimisations, while valuable from the point of view of making model checking tractable for larger examples, are likely not beneficial for the purpose of theorem proving, which is our long-term aim.

7 Conclusions

We have presented RoboChart, a new notation for modelling robotic systems. It is based on UML state machines, but is distinctive in many ways. It includes the notions of robotic platform, and parallel controllers and machines, synchronous and asynchronous communications, a well-defined action language, pre- and postconditions, and time primitives, all with a formal mechanised semantics for verification and refinement.

Table 7 Summary of verifications using the timed semantics with FDR

Example	States	Transitions	Compilation (s)	Deadlock (s)	Longest check (s)
Chemical detector	504	1623	0.1	0.26	0.26
Autonomous detector	83	611	1.87	0.06	0.07
Alpha algorithm	4427	10,405	4.36	0.37	0.48
Transporter	78	216	15.65	0.07	0.09

We have described the semantics of RoboChart using CSP to enable validation via model checking. It is, however, a front end to a predicative relational semantics using UTP. We are already exploring automatic proof of deadlock freedom using Isabelle/HOL [34,38].

First, we have presented an untimed semantics, in which uses of time primitives are ignored. The timed semantics defines an extension of the untimed model of a RoboChart diagram, whereby timed constructs of CSP capture the budgets and deadlines. It is our long-term aim to formalise the relationship between the timed and untimed models by exploiting the results on compositionality of semantic theories in the UTP.

Early work on statecharts have considered translation to C, Occam, VHDL, and Programmable Logic Arrays [17,27,28]. It is noted the difficulty of translating an intrinsically parallel model to a sequential program. In the case of the translation to Occam, however, we note some similarities with the structure of our model [17]. Processes are used to represent machines, though not states. Pairs of channels are used to request and acknowledge exit from a state. Compositional semantics is, however, not a goal. Inter-level transitions and history junction seem to be considered.

To support work on case studies and validation of our semantics, we have developed RoboTool, a user-friendly tool for editing and verifying RoboChart diagrams. Besides automatic generation of the CSP semantics, RoboTool also supports the automatic generation of C++ implementations for a subset of state machines and controllers. Currently, we are extending the code generation facilities of RoboTool to cover the complete notation, and targeting specific simulators to provide a link between implementations of RoboChart models and the specific API of a simulator. Finally, we are investigating the use of the generated code for deployment using the Robot Operating System (ROS[12]). At a later point, we will address proof of correctness of the generated simulations.

Current work is also considering inclusion of probabilistic choice in RoboChart. Work on probability is available in the UTP [100], and we are pursuing translation to PRISM [49] for verification, with an associated encoding of Markov decision processes in the UTP.

RoboChart can also be enriched with support for modelling the environment and the robotic platforms in more detail. Facilities to be considered will be based on the pragmatic approaches taken by robotics simulators. For verification, however, it is in our plans to take inspiration from hybrid automata [43] and from the UTP model of continuous variables [35].

RoboChart is more restrictive than general architectural languages, such as AADL [32], or even UML [74] and SysML [73]. Architectural models can be defined in RoboChart by identifying the controllers of a robotic system and their connections, and, for each controller, state machines that capture independent functionality or threads of behaviour. By restricting the way in which components can be defined, we simplify the modelling task and the semantics of RoboChart. Currently, we are developing modelling guidelines for RoboChart.

While in principle, it should be possible to verify our examples using other CSP model checkers, we adopt the concrete standard CSP language (CSP-M). Use of tools such as PAT [94] and ProB [50] would require different code generators: PAT accepts a different version of CSP called CSP#, and ProB accepts a subset of CSP-M.

As part of our future work, we plan to explore a variety of other model checkers and SMT solvers, such as UPPAAL [7], PRISM [49], nuXmv [18], Z3 [69] and CVC4 [4], and investigate the use of sequence diagrams for the specification of properties and visualisation of counterexamples produced by FDR. While our approach supports traceability of counterexamples due to a one-to-one correspondence between CSP events and RoboChart variables, events, and operations, it currently requires manual inspection of the output of FDR. Automatic visualisation of counterexamples will further improve the usability of our approach by limiting the situations where knowledge of CSP and FDR are required. Finally, as a further effort in the validation of our semantics, we intend to encode the semantic functions in Isabelle/HOL, and prove various properties including totality and well-typedness with respect to well-formedness conditions. This encoding will also be used in conjunction with Isabelle/UTP in order to support verification of RoboChart models via theorem proving.

Acknowledgements This work is funded by the EPSRC Grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engi-

[12] www.ros.org.

neering. No new primary data was created during this study. We thank James Baxter and Augusto Sampaio for many suggestions to improve RoboChart and this paper. The icons used in RoboChart have been made by Sarfraz Shoukat, Freepik, Google, Icomoon and Madebyoliver from www.flaticon.com, and are licensed under CC 3.0 BY.

References

1. Akhlaki, K.B., Tunon, M.I.C., Terriza, J.A.H., Morales, L.E.M.: A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. Sci. Comput. Program. **65**(1), 41–56 (2007)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Baar, T.: Verification support for a state-transition-DSL defined with Xtext. In: Mazzara, M., Voronkov, A. (eds.) Perspectives of System Informatics, pp. 50–60. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41579-6_5
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 171–177. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: IEEE International Conference on Software Engineering and Formal Methods, pp. 3–12. IEEE Computer Society (2006)
6. Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: 3rd International Conference on the Quantitative Evaluation of Systems, pp. 125–126. IEEE Computer Society (2006)
7. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL–*-a Tool Suite for Automatic Verification of Real–Time Systems. In: Proceedings of Workshop on Verification and Control of Hybrid Systems III, no. 1066 in Lecture Notes in Computer Science, pp. 232–243. Springer, Berlin (1995)
8. Bergstra, J.A., Klop, J.W.: Process theory based on bisimulation semantics. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, pp. 50–122. Springer, Berlin (1989)
9. Berthomieu, B., Vernadat, F.: Time petri nets analysis with TINA. In: 3rd International Conference on the Quantitative Evaluation of Systems, pp. 123–124. IEEE Computer Society (2006)
10. Bjerknes, J.D., Winfield, A.F.T.: On Fault Tolerance and Scalability of Swarm Robotic Systems, pp. 431—444. Springer, Berlin (2013)
11. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012). https://doi.org/10.1016/j.jcss.2011.08.007
12. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML—towards a system model for UML: The state machine model. Technical Report, TUM-I0711, Institut für Informatik, Technische Universität München (2007). http://www4.in.tum.de/publ/papers/TUM-I0711.pdf
13. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, Berlin (2001)
14. Brunner, S.G., Steinmetz, F., Belder, R., Domel, A.: RAFCON: A graphical tool for engineering complex, robotic tasks. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3283–3290 (2016)
15. Brunner, S.G., Steinmetz, F., Belder, R., Domel, A.: Rafcon: A graphical tool for engineering complex. robotic tasks. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3283–3290 (2016)
16. Buchanan, E., Pomfret, A., Timmis, J.: Dynamic Task Partitioning for Foraging Robot Swarms, vol. 9882, pp. 113–124. Springer (2016)
17. Calvez, J.P., Pasquier, O.: Implementation of statecharts with transputers. Microprocess. Microprogram. **35**(1), 133–139 (1992)
18. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Biere, A., Bloem, R. (eds.) 26th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer, Berlin (2014)
19. Cavalcanti, A.L.C., Woodcock, J.C.P.: A Tutorial Introduction to CSP in Unifying Theories of Programming. In: Refinement Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 3167, pp. 220–268. Springer, Berlin (2006). https://doi.org/10.1007/11889229_6. https://www-users.cs.york.ac.uk/~alcc/publications/papers/CW06.pdf
20. Chen, J., Gauci, M., Gross, R.: A strategy for transporting tall objects with a swarm of miniature mobile robots. In: ICRA, pp. 863–869. IEEE (2013)
21. David, A., Möller, M.O., Yi, W.: Formal verification of UML statecharts with real-time extensions. In: Kutsche, R.D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, pp. 218–232. Springer, Berlin, Heidelberg (2002)
22. Davies, J., Crichton, C.: Concurrency and refinement in the unified modeling language. Formal Asp. Comput. **15**(2–3), 118–145 (2003)
23. Davies, J., Schneider, S.: A brief history of Timed CSP. Theor. Comput. Sci. **138**(2), 243–271 (1995)
24. DeAntoni, J., Mallet, F.: Objects, models, components, patterns. In: chap. TimeSquare: treat your models with logical time, pp. 34–41. Springer, Berlin (2012)
25. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: SIMPAR 2012, pp. 149–160. Springer, Berlin (2012)
26. Dixon, C., Winfield, A.F.T., Fisher, M., Zeng, C.: Towards temporal verification of swarm robotic systems. Robot. Auton. Syst. **60**(11), 1429–1441 (2012)
27. Drusinsky, D., Harel, D.: Using statecharts for hardware description and synthesis. IEEE Trans. Comput. Aided Desi. Integr. Circuits Syst. **8**(7), 798–807 (1989)
28. Dutt, N.D., Cho, J.H., Hadley, T.: A user interface for VHDL behavioral modeling. In: Borrione, D., Waxman, R. (eds.) Computer Hardware Description Languages and Their Applications, pp. 407–425. North-Holland, Amsterdam (1991)
29. Endo, Y., MacKenzie, D.C., Arkin, R.C.: Usability evaluation of high-level user assistance for robot mission specification. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **34**(2), 168–180 (2004)
30. Espiau, B., Kapellos, K., Jourdan, M.: Formal verification in robotics: Why and how? In: Robotics Research, pp. 225–236. Springer, London (1996)
31. Farrell, M., Luckcuck, M., Fisher, M.: Robotics and integrated formal methods: necessity meets opportunity. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 11023, pp. 161–171. Springer, Berlin (2018)

32. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional, Reading (2012)
33. Fleurey, F., Solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: International Conference on Model Driven Engineering Languages and Systems, pp. 606–621. Springer, Berlin (2009)
34. Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J.: Automating verification of state machines with reactive designs and Isabelle/UTP. In: Bae, K., Ölveczky, P.C. (eds.) Formal Aspects of Component Software, pp. 137–155. Springer, Cham (2018)
35. Foster, S., Thiele, B., Cavalcanti, A.L.C., Woodcock, J.C.P.: Towards a UTP semantics for Modelica. In: UTP 2016, Lecture Notes in Computer Science. Springer (2016)
36. Foster, S., Woodcock, J.C.P.: Towards verification of cyber-physical systems with UTP and Isabelle/HOL. In: Gibson-Robinson, T., Hopcroft, P.J., Lazic, R. (eds.) Concurrency, Security, and Puzzles—Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday, Lecture Notes in Computer Science, vol. 10160, pp. 39–64. Springer, Berlin (2017)
37. Foster, S., Zeyda, F., Woodcock, J.C.P.: Isabelle/UTP: a mechanised theory engineering framework. In: Naumann, D. (ed.) Unifying Theories of Programming, Lecture Notes in Computer Science, vol. 8963, pp. 21–41. Springer, Berlin (2015)
38. Foster, S., Zeyda, F., Woodcock, J.C.P.: Unifying heterogeneous state-spaces with lenses. In: A.C.A. Sampaio, F. Wang (eds.) Theoretical Aspects of Computing, Lecture Notes in Computer Science, vol. 9965, pp. 295–314 (2016)
39. Foughali, M., Berthomieu, B., Zilio, S.D., Ingrand, F., Mallet, A.: Model checking real-time properties on the functional layer of autonomous robots. In: Formal Methods and Software Engineering, pp. 383–399. Springer, Berlin (2016)
40. Gauci, M., Chen, J., Li, W., Dodd, T., Gross, R.: Self-organized aggregation without computation. Int. J. Robot. Res. **33**(8), 1145–1161 (2014)
41. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3: a modern refinement checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 187–201 (2014)
42. Gobillot, N., Lesire, C., Doose, D.: A modeling framework for software architecture specification and validation. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) Simulation, Modeling, and Programming for Autonomous Robots, pp. 303–314. Springer, Berlin (2014)
43. Henzinger, T.A.: The theory of hybrid automata. In: 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278–292 (1996)
44. Hilder, J.A., Owens, N.D.L., Neal, M.J., Hickey, P.J., Cairns, S.N., Kilgour, D.P.A., Timmis, J., Tyrrell, A.M.: Chemical detection using the receptor density algorithm. IEEE Trans. Syst. Man Cybern. C Appl. Rev. **42**(6), 1730–1741 (2012)
45. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, Upper Saddle River (1985)
46. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
47. Hochgeschwender, N., Gherardi, L., Shakhirmardanov, A., Kraetzschmar, G.K., Brugali, D., Bruyninckx, H.: A model-based approach to software deployment in robotics. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3907–3914 (2013)
48. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An integrated semantics for UML class, object and state diagrams based on graph transformation. In: Butler, M., Petre, L., SereKaisa, K. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 2335, pp. 11–28. Springer, Berlin (2002)
49. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: a hybrid approach. Int. J. Softw. Tools Technol. Transf. **6**(2), 128–142 (2004)
50. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods, pp. 855–874. Springer, Berlin (2003). https://doi.org/10.1007/978-3-540-45236-2_46
51. Li, W., Miyazawa, A., Ribeiro, P., Cavalcanti, A.L.C., Woodcock, J.C.P., Timmis, J.: From formalised state machines to implementations of robotic controllers. In: Groß, R., Kolling, A., Berman, S., Frazzoli, E., Martinoli, A., Matsuno, F., Gauci, M. (eds.) Distributed Autonomous Robotic Systems: the 13th International Symposium, pp. 517–529. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73008-0_36
52. Lima, L., Miyazawa, A., Cavalcanti, A.L.C., Cornélio, M., Iyoda, J., Sampaio, A.C.A., Hains, R., Larkham, A., Lewis, V.: An integrated semantics for reasoning about SysML design models using refinement. Softw. Syst. Model. **16**, 875–902 (2015). https://doi.org/10.1007/s10270-015-0492-y
53. Lima, L., Miyazawa, A., Cavalcanti, A.L.C., Cornélio, M., Iyoda, J., Sampaio, A.C.A., Hains, R., Larkham, A., Lewis, V.: An integrated semantics for reasoning about SysML design models using refinement. Softw. Syst. Model. 1–28 (2015)
54. Lowe, G.: Specification of communicating processes: temporal logic versus refusals-based refinement. Form. Asp. Comput. **20**(3), 277–294 (2008)
55. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. CoRR arXiv:1807.00048 (2018)
56. Mallet, F.: Clock constraint specification language: specifying clock constraints with UML/MARTE. Innov. Syst. Softw. Eng. **4**(3), 309–314 (2008)
57. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., Resnick, M.: Scratch: a sneak preview. In: Second International Conference on Creating, Connecting and Collaborating Through Computing, 2004. Proceedings. pp. 104–109. IEEE (2004)
58. Maoz, S., Ringert, J.O.: Gr(1) synthesis for ltl specification patterns. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 96–106. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786824
59. Maoz, S., Ringert, J.O.: Synthesizing a lego forklift controller in GR(1): A case study. In: Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015., pp. 58–72 (2015). https://doi.org/10.4204/EPTCS.202.5
60. Maoz, S., Ringert, J.O.: On the software engineering challenges of applying reactive synthesis to robotics. In: 2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE), pp. 17–22 (2018)
61. Maoz, S., Ringert, J.O.: Spectra Language and Spectra Tools User Guide (2018). http://smlab.cs.tau.ac.il/syntech/spectra/
62. The MathWorks,Inc.: Stateflow and Stateflow Coder 7 User's Guide. www.mathworks.com/products
63. Menghi, C., Tsigkanos, C., Berger, T., Pelliccione, P., Ghezzi, C.: Property specification patterns for robotic missions. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18, pp. 434–435. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183440.3195044
64. Milner, R.: Communication and Concurrency. Prentice-Hall, Upper Saddle River (1989)
65. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, Cambridge (1999)

66. Miyazawa, A., Cavalcanti, A.L.C.: Refinement-oriented models of Stateflow charts. Sci. Comput. Program. **77**(10–11), 1151–1177 (2012)
67. Miyazawa, A., Cavalcanti, A.L.C.: Formal refinement in SysML. In: Albert, E., Sekerinski, E. (eds.) 11th International Conference on Integrated Formal Methods. Lecture Notes in Computer Science, pp. 155–170. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-10181-1_10
68. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J.: Automatic property checking of robotic applications. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3869–3876 (2017). https://doi.org/10.1109/IROS.2017.8206238
69. de Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78800-3_24
70. Naylor, B., Read, M., Timmis, J., Tyrrell, A.: The Relay Chain: A Scalable Dynamic Communication link between an Exploratory Underwater Shoal and a Surface Vehicle (2014)
71. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Berlin (2002)
72. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in Robotics. J. Softw. Eng. Robot. **7**(1), 75–99 (2016)
73. Object Management Group: OMG Systems Modeling Language (OMG SysML), Version 1.3 (2012). www.omg.org/spec/SysML/
74. Object Management Group: OMG Unified Modeling Language (2015). www.omg.org/spec/UML/2.5/
75. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral aadl models in real-time maude. In: Hatcliff, J., Zucca, E. (eds.) Formal Techniques for Distributed Systems, pp. 47–62. Springer, Berlin, Heidelberg (2010)
76. Park, H.W., Ramezani, A., Grizzle, J.W.: A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. IEEE Trans. Robot. **29**(2), 331–345 (2013)
77. Pembeci, I., Nilsson, H., Hager, G.: 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, pp. 168–179. ACM (2002)
78. Rabbath, C.A.: A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. J. Intell. Robot. Syst. **70**(1), 233–253 (2013)
79. Ramaswamy, A., Monsuez, B., Tapus, A.: Saferobots: A model-driven framework for developing robotic systems. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1517–1524 (2014)
80. Ramos, R., Sampaio, A.C.A., Mota, A.C.: A semantics for UML-RT active classes via mapping into Circus. Formal Methods Open Object-based Distributed Systems, Lecture Notes in Computer Science **3535**, 99–114 (2005)
81. Rasch, H., Wehrheim, H.: Checking consistency in UML diagrams: classes and state machines. In: Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science, vol. 2884, pp. 229–243. Springer, Berlin (2003)
82. Ribeiro, P., Miyazawa, A., Li, W., Cavalcanti, A.L.C., Timmis, J.: Modelling and verification of timed robotic controllers. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods, pp. 18–33. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-66845-1_2
83. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Code generator composition for model-driven engineering of robotics component and connector systems. J. Softw. Eng. Robot. **6**(1), 33–57 (2015)
84. RoboCalc Project: RoboChart Case Studies (2017). www.cs.york.ac.uk/circus/RoboCalc/case-studies/
85. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science. Springer, Berlin (2011)
86. Schillinger, P., Kohlbrecher, S., von Stryk, O.: Human–robot collaborative high-level control with an application to rescue robotics. In: IEEE International Conference on Robotics and Automation, Stockholm, Sweden (2016)
87. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic soft. systems: from code-driven to model-driven designs. In: ICAR 2009, pp. 1–8. IEEE (2009)
88. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley, London (2000)
89. Selic, B.: Using UML for modeling complex real-time systems. In: Mueller, F., Bestavros, A. (eds.) Languages, Compilers, and Tools for Embedded Systems, Lecture Notes in Computer Science, vol. 1474, pp. 250–260. Springer, Berlin (1998)
90. Selic, B., Grard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Morgan Kaufmann Publishers Inc., Burlington (2013)
91. Sherif, A., Cavalcanti, A.L.C., He, J., Sampaio, A.C.A.: A process algebraic framework for specification and validation of real-time systems. Form. Asp. Comput. **22**(2), 153–191 (2010). https://doi.org/10.1007/s00165-009-0119-6
92. Soetens, P., Bruyninckx, H.: Realtime hybrid task-based control for robots and machine tools. In: 2005 IEEE International Conference on Robotics and Automation, pp. 259–264 (2005)
93. Spichkova, M., Hülzl, F., Trachtenherz, D.: Verified system development with the autofocus tool chain. In: Proceedings 2nd Workshop on Formal Methods in the Development of Software (2012). https://doi.org/10.4204/EPTCS.86.3
94. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, pp. 709–714. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-02658-4_59
95. Tomic, T., Schmid, K., Lutz, P., Domel, A., Kassecker, M., Mair, E., Grixa, I.L., Ruess, F., Suppa, M., Burschka, D.: Toward a fully autonomous UAV: research platform for indoor and outdoor urban search and rescue. IEEE Robot. Autom. Mag. **19**(3), 46–56 (2012)
96. University of York: RoboChart Reference Manual. https://bit.ly/2Ooe7RS
97. University of York: RoboTool Reference Manual. https://bit.ly/2QGDbA0
98. Wei, K., Woodcock, J.C.P., Burns, A.: Timed *Circus*: timed CSP with the miracle. In: International Conference on Engineering of Complex Computer Systems, pp. 55–64 (2011)
99. Woodcock, J.C.P., Davies, J.: Using Z-Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River (1996)
100. Zhu, H., Sanders, J.W., He, J., Qin, S.: Denotational Semantics for a Probabilistic Timed Shared-Variable Language. In: UTP 2013. Lecture Notes in Computer Science, vol. 7681, pp. 224–247. Springer, Berlin (2013)
101. Zic, J.J.: Time-constrained buffer specifications in CSP + T and timed CSP. ACM Trans. Program. Lang. Syst. **16**(6), 1661–1674 (1994)

Alvaro Miyazawa is a research associate at the University of York. Having completed B.Sc. in Computer Science at the University of Sao Paulo and doctoral research at the University of York, his main research interests are in formal semantics and refinement for domain-specific languages and graphical notations, and the development of refinement strategies to support high levels of automation in program verification. Currently, his research focuses on modelling, simulation, and verification for robotics.

Pedro Ribeiro is a research associate in the High Integrity Systems Engineering group at the University of York. He holds an MEng in Computer Systems & Software Engineering and a Ph.D. in Computer Science. His doctoral work addressed the treatment of angelic nondeterminism in process calculi. He has previously worked on the timed semantics underpinning Safety-Critical Java applications. Currently, his work focuses on the timed semantics of RoboChart for refinement.

Wei Li received the B.Eng. degree in automation and the M.Eng. degree in control science and engineering from Harbin Institute of Technology, China, in 2009 and 2011, respectively. He also obtained the Ph.D. degree from University of Sheffield, UK, in 2016. He is currently a research associate in the department of electronic engineering, University of York, UK. His research interests include robotics and computational intelligence, and specifically self-organised systems and co/evolutionary machine learning.

Ana Cavalcanti is a Professor at the University of York and a Royal Academy of Engineering Chair in Emerging Technologies. From 2012 to 2017, she was Royal Society Wolfson Research Merit Award holder. In 2003, she was awarded a Royal Society Industry Fellowship to work with QinetiQ on formal methods. She has published more than 150 papers, and chaired the Programme Committee of various well-established international conferences. Her main research interest is in Software Engineering for Robotics. She is currently Chair of the Formal Methods Europe association.

Jon Timmis is Professor of Intelligent and Adaptive Systems at the Department of Electronic Engineering, University of York. He is a previous holder of both a Royal Society Wolfson Research Merit Award and Royal Academy of Engineering Enterprise Fellowship. His research interests are interdisciplinary in nature, and focus on the modelling and simulation of the immune system, the development of evidence-based simulations, and fault tolerance in biologically inspired systems. He is a Senior Member of the IEEE.

Jim Woodcock is Professor of Software Engineering at the University of York. His research interests are in the unification of mathematical theories for cost-effective design of hardware and software components in innovative, safe, and secure cyber-physical systems. His scientific work has enabled him to make significant contributions to the application of mathematical techniques in industry in domains of strategic importance to society. He is a Fellow of the UK's Royal Academy of Engineering.