RoboStar technology: a roboticist's toolbox for combined proof, simulation, and testing

Ana Cavalcanti, Will Barnett, James Baxter, Gustavo Carvalho,
Madiel Conserva Filho, Alvaro Miyazawa, Pedro Ribeiro, Augusto Sampaio

Abstract Simulation is favoured by roboticists to evaluate controller design and software. Often, state machines are drawn to convey overall ideas and used as a basis to program tool-specific simulations. The simulation code, written in general or proprietary programming languages, is, however, the only full account of the robotic system. Here, we present the RoboStar technology, a modern approach to design that supports automatic generation of simulation code guaranteed to be correct with respect to a design model, and complements simulation with model checking, theorem proving, and automatic test generation for simulation. Diagrammatic domain-specific, but tool-independent, notations for design and simulation use state machines, differential equations, and controlled English to specify behavior. We illustrate the RoboStar approach using an autonomous vehicle as an example.

Ana Cavalcanti
University of York, UK e-mail: Ana.Cavalcanti@york.ac.uk

Will Barnett
University of York, UK e-mail: wb689@york.ac.uk

James Baxter
University of York, UK e-mail: James.Baxter@york.ac.uk

Gustavo Carvalho
Universidade Federal de Pernambuco, Brazil, e-mail: ghpc@cin.ufpe.br

Madiel Conserva Filho
Universidade Federal de Pernambuco, Brazil, e-mail: mscf@cin.ufpe.br

Alvaro Miyazawa
University of York, UK e-mail: Alvaro.Miyazawa@york.ac.uk

Pedro Ribeiro
University of York, UK e-mail: Pedro.Ribeiro@york.ac.uk

Augusto Sampaio
Universidade Federal de Pernambuco, Brazil, e-mail: acas@cin.ufpe.br

1 Introduction

Advances in electronics and mechatronics are facilitating exciting applications of robotics. To realise their potential, however, we need to be able to ensure that robots do not fail in a way that can cause harm: a robot strong enough to help an elderly person out of a chair, for example, is strong enough to hurt that person.

Although many factors are involved in establishing the trustworthiness of a robotic system, software poses a key challenge for design and assurance. Full verification is beyond the state of the art due to the complexity of models and properties. Lack of customized techniques and tools means that, despite the very modern outlook of the applications, the current practice of software engineering for robotics is outdated.

What is routine in many engineering disciplines, that is, the use of models, tools, and techniques justified by mathematical principles, called formal methods, is becoming feasible for software developers. Main players in industry like Microsoft, Amazon, and Facebook have started using formal methods [78].

Several domain-specific languages for robotics are available in the literature [56], but, by far and large, their focus is support for programming and simulation. Modern verification techniques have not been widely explored, with a few notable exceptions [30, 1, 25, 38]; some are covered in this book, notably in Chapters 4, 5, 7, 8, 11, 12, and 13. Applications of general-purpose formal techniques have shown the value that they can add to robotics. Due to lack of specialization and difficulties with automation, however, the cost involved and scalability achieved do not indicate a clear prospect of wide practical application.

The RoboStar framework for modelling, verification, simulation, and testing of mobile and autonomous robots uses three domain-specific languages: RoboChart [54], RoboSim [13], and RoboWorld. RoboChart includes a subset of UML-like state machines, a customized component-model, and primitives to specify timed and probabilistic properties. RoboChart is an event-based notation for design; RoboSim is a matching cycle-based diagrammatic notation for simulation. RoboSim also includes block diagrams enriched to specify physical and dynamic behaviors of robotic platforms. RoboWorld uses controlled English to specify assumptions about the environment and the platform. It complements RoboChart.

RoboChart, RoboSim, and RoboWorld provide a solid foundation to deal with software engineering for robotics. They are notations akin to those in widespread use but enriched to enable the use of modern design and verification techniques.

RoboChart, RoboSim, and RoboWorld can be used to generate automatically mathematical models. In the RoboStar approach, these models are hidden from practitioners, but can be used to prove properties of designs and simulations, consistency between them, and generate tests. The RoboStar testing approach is covered in Chapter 11. So far, we have had experience with the model checkers FDR [35] and PRISM [42], and the theorem prover Isabelle [55]. The RoboStar work with probabilistic modelling and PRISM is the topic of Chapter 13.

RoboChart and RoboSim have an associated Eclipse-based tool, called RoboTool[1], which supports graphical modelling, validation, and automatic generation of CSP [69] and reactive modules scripts, simulation code, and tests. It is integrated with FDR4 and PRISM, ARGoS [61], and CoppeliaSim [68]. A variety of plug-ins are available.

By ensuring that a RoboChart and RoboWorld design and a RoboSim simulation are consistent, the RoboStar framework guarantees that properties established by analysis of the design are preserved in the simulation. So, problems revealed by simulation are design problems, not problems in the coding of the simulation itself.

RoboChart, RoboSim, and RoboWorld complement approaches that cater for a global view of the system architecture (like that in Chapter 3) by supporting modelling and verification of the components, covering interaction, time, and probabilistic properties. It also complements work on deployment of verified code.

In this chapter, we describe the RoboStar technology in detail and illustrate its application to modelling, verifying, and simulating control software for an autonomous vehicle. In the next section, we give an overview of the RoboStar technology, including the notations and techniques already available, and those that we envisage as part of our vision for Software Engineering for robotics. Our running example is presented in Section 3. Section 4 presents a RoboChart model for its control software, and Section 5 presents a RoboSim model. Section 6 discusses environment modelling in RoboChart and RoboSim. Finally, Section 7 discusses related work, and Section 8 summarizes the RoboStar vision and agenda for future work.

2 RoboStar vision

The RoboStar approach to Software Engineering for robotics is presented in Figure 1. For modelling designs, we envisage the combined use of RoboChart to model control software, and a controlled natural language, RoboWorld, under development to capture assumptions about the platform and environment. In the proposed workflow, a RoboChart model is the starting point, as indicated by the label (1) in Figure 1. In the next section, we present a RoboChart model for our example in Section 4, where we also give more details about the RoboChart notation.

RoboTool supports the creation and editing of RoboChart diagrams, and the automatic generation of CSP and PRISM models: labels (2) in Figure 1. RoboTool also provides a simple notation that uses controlled English, potentially mixed with CSP processes, to define assertions capturing properties of interest. These assertions can be checked using FDR4 or the PRISM model checker. We plan to enrich this language to cater for a readable account of properties specified in English or diagrammatically (using sequence diagrams, for example).

It is possible, of course, to generate automatically other mathematical models for RoboChart. Currently, we are considering the integration of UPPAAL [6]. A crucial point, however, is that these models are consistent with each other. The

[1] www.cs.york.ac.uk/robostar/robotool/

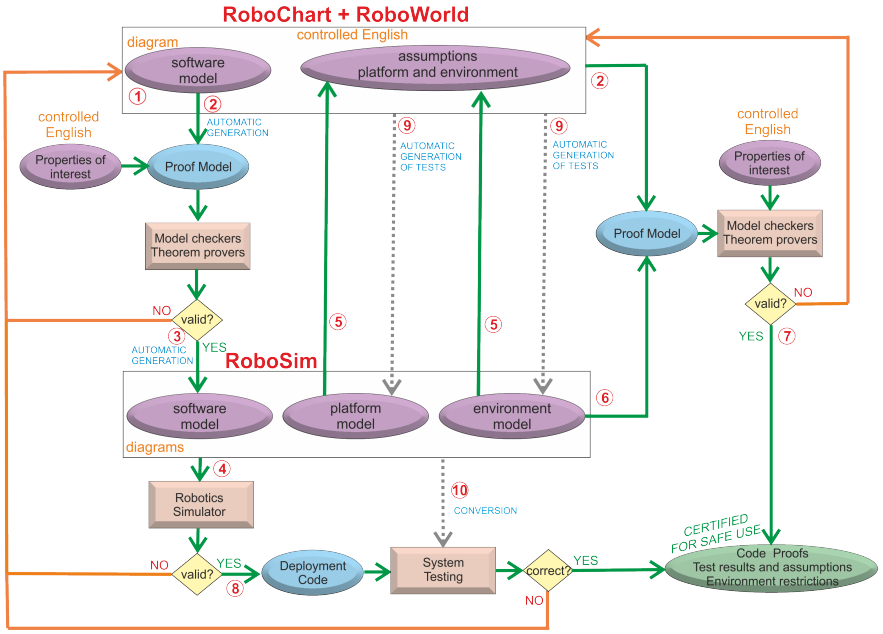


Fig. 1 Idealized workflow using RoboStar technology

definitive semantics of RoboChart is given by the CSP model, and our vision is one of integration of techniques and tools (model checkers and theorem provers) via the justification of soundness based on semantics. It would not be useful, if, for instance, deadlock checks carried out with different tools could give different results.

RoboTool generates two CSP models. The first gives an untimed semantics to RoboChart and ignores the time primitives. The definitive semantics uses the tock dialect of CSP, in which a special event *tock* marks the passage of discrete time. Probabilistic modelling is captured via a semantics given using reactive modules. Consistency between the probabilistic and timed semantics is ongoing work. An early approach that combines CSP and PRISM is available [24].

For theorem proving based on the CSP semantics, an initial approach is described in [27, 28, 29]. Automatic generation of Isabelle theories for proof is ongoing work. Use of Isabelle is our current route to address the issues of scalability that we can expect with the use of model checking. High levels of automation are still possible as evidenced by [2], given the use of proof models that are automatically generated.

In the RoboStar approach, initial proofs to check the models can validate them by establishing core properties, like deadlock and livelock freedom, for example. Checks for the presence or absence of nondeterminism can also often reveal modelling problems. These checks are automatically generated by RoboTool.

If any of the properties of interest do not hold, the RoboChart model should be changed (unless, of course, further work reveals that the property is not actually relevant). For a property that does not hold, a model checker provides an example

that illustrates the problem, and the relationship between such examples and the diagrams is simple. The examples can inform how the model should be changed, and the iterative process of debugging of a model is much cheaper than that of debugging a program. Automation of the generation of proof models makes this iterative process of validation of the RoboChart model using the properties of interest cheap. An animator would help, though, and it is part of our agenda for future work. Such a tool allows tracing the diagram based on a sequence of updates to variables, calls to operations, or occurrence of events indicated by a model checker.

Once we are convinced that enough validation has taken place, it then makes sense to consider simulation of the model. For cyber-physical systems, in general, and robotics systems, in particular, it is often the case that we do not have a full specification of behavior. Such a specification is normally highly dependent on the platform and environment. Simulations can, therefore, be very useful to validate the design. Moreover, simulations are core to current practice.

A simulation requires a cyclic account of the design model. This is what is provided by a RoboSim model, which can be automatically generated from a RoboChart model with guaranteed consistency: label (3) in Figure 1.

For a simulation, however, we also need a physical and behavioral model of the platform and of the environment. RoboSim includes a block diagram notation to describe these models. A domain-specific language for robotics based on XML, namely, SDF (Scene Description Format)[2] provides inspiration and domain knowledge. RoboSim block diagrams, however, afford readability, modularity, and extensibility to models. Moreover, from such diagrams, it is possible to generate SDF documents for use in robotics simulators. This is ongoing work.

A RoboSim block diagram specifies a particular platform. Roboticists routinely write such descriptions, either using an XML-based notation like SDF, or using tool-specific graphical or programming facilities. So, RoboSim block diagrams improve usability and do not require a significant change in the language used by practitioners.

For a design, however, a model for a particular platform or scenario is too specific. Instead, we need an account of operational requirements: assumptions that are made of the platform and environment, and ensure proper behavior of the robot. A proper account of such requirements are often neglected by practitioners. The RoboStar vision is, therefore, to provide support for their description based on the RoboSim block diagrams: label (5) in Figure 1. The RoboStar approach is for the requirements to be captured in RoboWorld, a controlled natural language under development.

Just like for RoboChart, RoboTool can generate automatically CSP models for RoboSim: label (6) in Figure 1. That model has a dual role: justify the soundness of an automatically generated RoboSim model with respect to an original RoboChart model, and proof of properties of interest just like for RoboChart.

Soundness requires consideration of the design model given some assumptions. For example, a RoboChart state machine that is in a state with a transition triggered by an event obstacle, for instance, reacts instantaneously to the detection of such an obstacle (by the platform). A simulation (whether described by RoboSim or not), on

[2] sdformat.org

the other hand, is a cyclic mechanism, where events are only observed and handled at sample times characterized by a cycle period. If an obstacle is detected in between sample times, it is ignored until the next sample time. So, consistent behavior only happens if we assume that events do not happen at all between sample times.

With these assumptions, proof of properties of a RoboSim simulation is not needed if the simulation is automatically generated. On the other hand, a RoboStar developer may well decide to write a RoboSim simulation directly, rather than start from a design model. In that case, proof of properties for RoboSim is useful.

Simulation may reveal problems, in which case, given the high level of automation, the right and cost-effective way to proceed is to update the RoboChart model, rerun proofs of properties, and regenerate the simulation. There is no need to deal with the (low-level) simulation code, and there is value in keeping models up to date.

If the simulation (eventually) suggests that the expected behavior is ensured by the design, a proof can add value by confirming that the property holds. Running simulations is a form of testing, and cannot be used to guarantee properties. Proofs provide evidence of the quality of the design (label (7) in Figure 1) in addition to the evidence provided by running the simulations. On the other hand, attempting to prove properties before checking behavior via simulation can lead to wasted effort, if the simulation can reveal that the property does not hold.

Simulation code for the control software can often be used as a basis for deployment: label (8) in Figure 1. For deployment, system testing, considering both the target platform and environment, if possible, is necessary. This is essential, for example, to confirm that the operational requirements are satisfied, or that errors are not introduced in the (generation of the) deployment code.

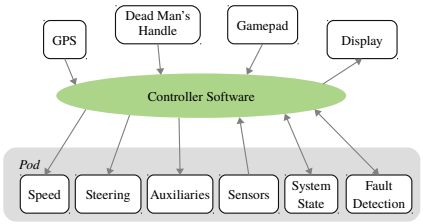
The RoboStar approach to testing is the automatic generation of tests from RoboChart models: label (9) in Figure 1, and the conversion of simulation tests to deployment tests: label (10) in Figure 1. More information about testing from RoboChart models is given in Chapter 11; conversion is in our agenda for the future.

In a simulation, we control the whole robot and the environment. In deployment, since we test the controller on the actual hardware, we typically have less control over the software and only limited observability of its state. (It is possible to instrument the controller to record, for instance, values of variables and the execution path, and to add a probe to control nondeterministic and probabilistic choices. Instrumentation can, however, be inappropriate if we are interested in timing or probabilities; we end up testing a program that is not quite the controller.) In deployment tests, we control just the initial state of the robot, often cannot control the entire environment, and have even weaker observational power. So, in each case, we have different notions of test case. Our approach uses conversion to ensure traceability of tests. In this way, if a deployment test fails, and it corresponds to a simulation test that passed, we gather information about the system under test and the simulation.

If a deployment test fails, automation of the whole approach, including the automatic generation and traceability of tests, encourages update of models. Development and change efforts are concentrated on diagrammatic and controlled English artefacts, rather than low-level code. This increases productivity, lowering costs, and enables the production of evidence of quality as well production of the code itself.

Fig. 2 The autonomous pod, and its inputs and outputs.

(a) 

(b) 

GPS	The current latitude and longitude of the pod.
Dead man's handle	Status of the safety driver's speed limiting control.
Gamepad	Manual commands for pod control.
Display	Instructions to the user of the autonomous pod system.
Pod speed	The speed the pod currently maintains.
Pod steering	The steering angle the pod currently maintains.
Pod auxiliaries	The state of the pod's peripheral devices, such as indicators and horn.
Pod sensors	The state of the pod's various internal sensors, such as bump sensors or seat occupancy.
Pod system state	Indication of the status of the pod embedded controller: drive disabled, manual-drive enabled, or autonomous-drive enabled.
Pod fault detection	A seed-key response to the pod's embedded controller to enable fault detection.

Next, we illustrate the RoboStar technology in Figure 1 in the example of an autonomous vehicle. We focus on the mature components of the framework.

3 Autonomous vehicle

Our case study has been described in [34]. It is a fully autonomous vehicle whose software has been developed by the UK Connected Places Catapult[3]. The pod has been developed as part of the LUTZ Pathfinder project: see Figure 2(a). Our focus is on the Basic Autonomous Control System (B-ACS), which is implemented using C++ and the ROS (Robot Operating System) middleware.

The B-ACS directs the pod around a predetermined route, specified by a sequence of latitude and longitude points. It ensures the pod follows the route and keeps within the speed limit for the current location. Reaction to obstacles, pre-emption, and mitigation of hazardous situations is carried by a safety driver, who can override autonomous control by limiting the pod's speed or stopping the pod altogether.

Figure 2(b) is an overview of the B-ACS inputs and outputs, corresponding to onboard sensors and actuators, extra sensors added to the pod, and interactions with the embedded controller (shown in the shaded area). They are described in Figure 2.

[3] Formerly, the Transport Systems Catapult.

Table 1 Uses of the ROS nodes, message definitions, and classes in each group of nodes of the pod software controller.

Group of nodes	Description
b-acs	Generation and processing of demand messages that guide the pod around a predetermined path. The messages specify the speed and steer values required for the pod to follow the path, based on its current location. Processing involves geofencing, enforcing speed limits, and validity checks, for example.
lutz	Interfacing with the pod to receive and manage its state, translating demands into control messages, carrying out the fault detection protocol, and generating user instructions to guide the safety driver.
data logging	Recording sensor data for later evaluation.

The use of ROS strongly influences the structure of the controller software. It is composed of many modules executed concurrently as an individual process known as a ROS node. The ROS nodes typically communicate with each other using asynchronous messages via a publish/subscribe mechanism provided by ROS.

The behavior of the pod's controller is determined by the function of each ROS node and the messages communicated among them. We describe all these nodes and their relationship in [5]. The original documentation identifies three groups of nodes called b-acs, lutz, and data logging. They are characterized in Table 1.

The nodes for data logging are for evaluation; they are not central to the application and not considered here. More information about them is available at [5].

Figure 3 depicts the nodes of the b-acs group and the messages communicated between them (via the publish/subscribe ROS mechanism). (The nodes of Figure 3 represent each ROS node and the edges represent the messages that the nodes communicate.) Groups of ROS nodes are indicated in the figure by a shaded node with a dashed outline. A description summarising the behavior of the b-acs nodes we model in the next section can be found in Table 2. A full list is in [5].

4 RoboChart model

This section presents a RoboChart model for the pod controller. Section 4.1 outlines the overall structure of the model. The following sections present the robotic platform and the controllers. Section 4.2 covers the RoboChart module and its robotic platform, and Section 4.3 covers the controller for the group of nodes b-acs and their state machines. The data types used in the RoboChart model reflect those used in the source code; they are all defined in [5]. Section 4.4 discusses verification.

Fig. 3 The ROS nodes of the b_acs group, adapted from Transport Systems Catapult specifications

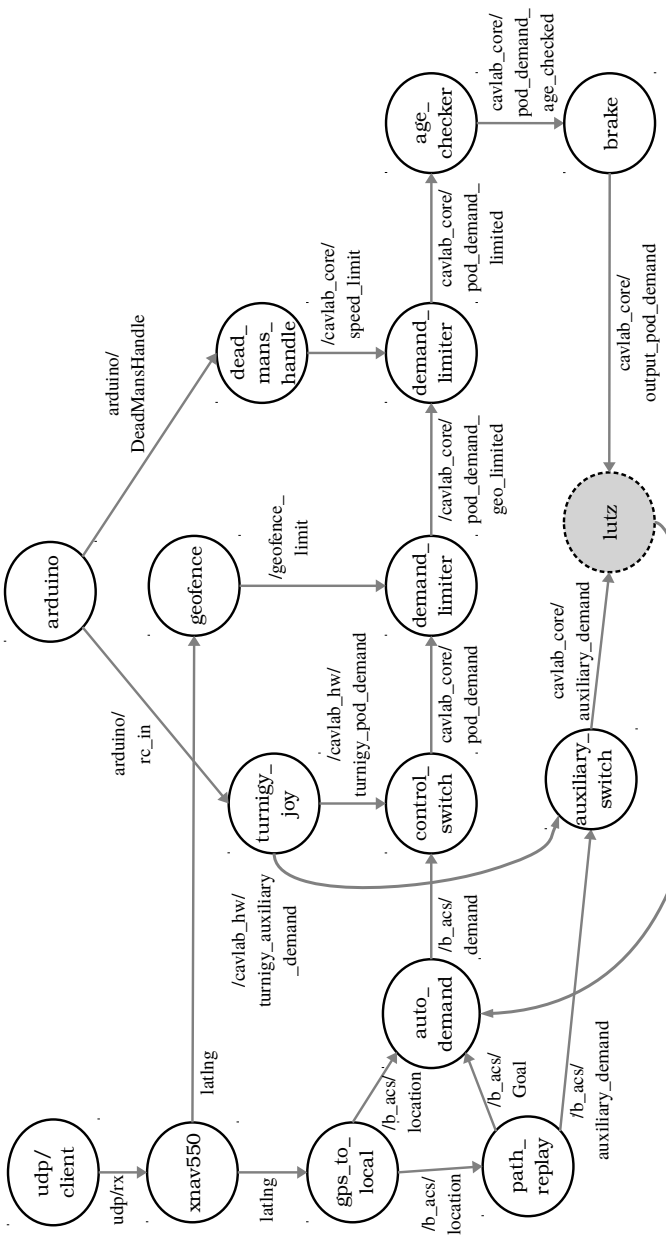


Table 2 b_acs ROS nodes

Node	Description
turnigy_joy	Converts remote control into messages for the pod and its peripheral devices.
dead_mans_handle	Converts dead man's handle messages to publish speed-limit messages.
gps_to_local	Translates the pod's global location sensor information to publish local 2D location messages.
path_replay	Publishes an intermediary goal towards the route's destination using the pod's local location. The goal contains the target local location, curvature, maximum speed, and duration describing the path to take. The route to the destination is defined using a configuration file containing a sequence of global locations and auxiliary state.
auto_demand	Receives a goal, the pod's local location, and speed from the pod state to create and publish a demand message. The demand message contains the specific steer and speed values to meet the goal.
control_switch	Receives multiple pod demand inputs and publishes only the highest priority non-abdicating input. The highest-priority input can abdicate, allowing lower-priority inputs to be published.
geofence	Receives the global location of the pod and publishes the speed limit for the current location. The speed limit for rectangular areas are configured using a file containing a sequence of two pairs of latitude and longitude points, each with an associated speed limit.
demand_limiter	Adjusts the speed of a demand based on a speed limit.
age_checker	Checks the age of a demand; demand messages that are older than a defined time have their speed reduced. The speed reduction increases the older the demand message is, until the speed is zero. The age-checked demand with appropriately adjusted speed is published.

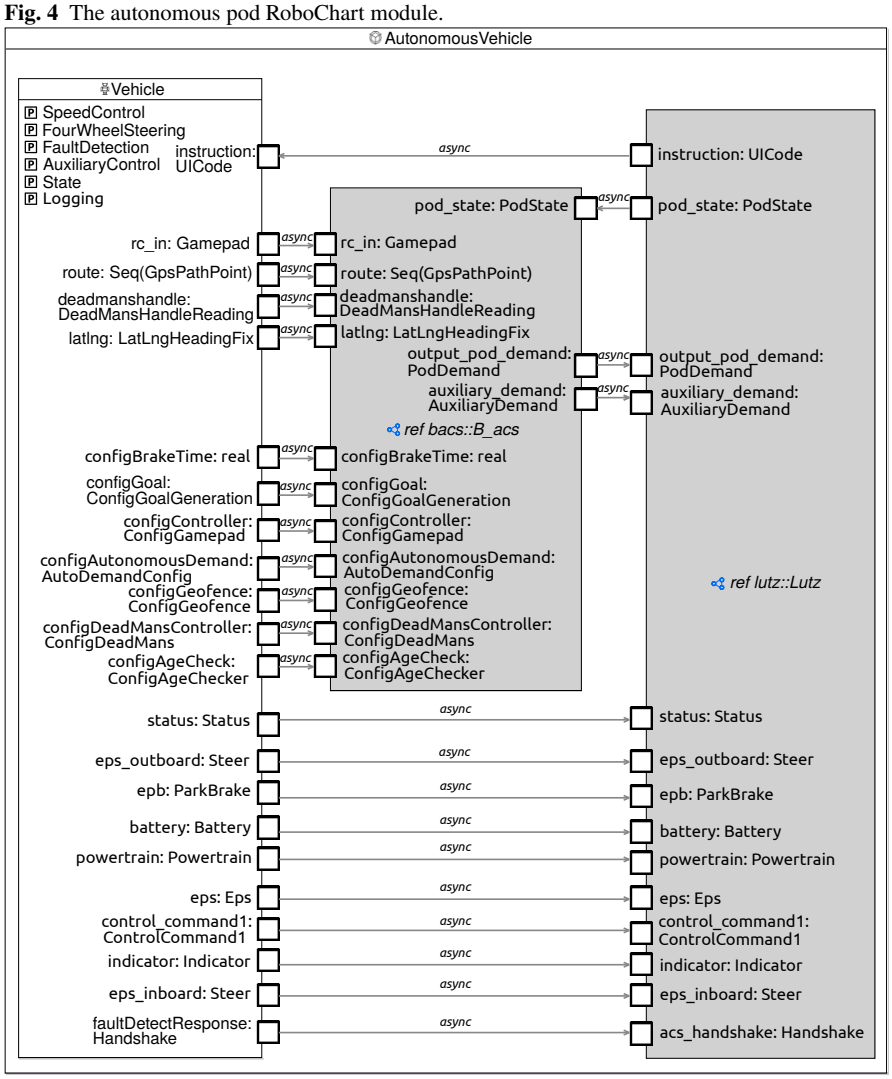
4.1 Overall structure

A RoboChart model is defined by a module, which specifies a robotic platform, and one or more (software) controllers. The module AutonomousVehicle for the pod system is shown in Figure 4. In this example, there are, besides the robotic platform Vehicle, two controllers B-acs and Lutz. Connections between a controller and the platform are always asynchronous. In our example, all connections are asynchronous, whether they are with the platform, or between controllers.

The robotic platform of a RoboChart model captures a representation of its sensors and actuators via variables, events, and operations available to the controllers. Changes to the values of the variables, occurrences of the events, and calls to these operations define the observable interactions between the robot control software and its environment. For the pod, the robotic platform represents the Vehicle, including all of the extra sensors in Figure 2. Section 4.2 defines the robotic platform for the pod and details the analysis of the inputs and outputs for the controller software.

Because RoboChart controllers can describe concurrent behavior, they can represent either the individual ROS nodes of the pod or higher-level functionality provided by the groups of nodes in Table 1. Representing each node as a controller would mean that there are more controllers in the module, making it more difficult to understand.

Fig. 4 The autonomous pod RoboChart module.



Representing functionally related groups of ROS nodes as controllers emphasizes the coupling between related areas of functionality. This means, however, that ROS nodes are represented as RoboChart machines, and so their behavior is predominately sequential. For our example, this is not an issue because the nodes of the pod are sequential. In addition, in general, a node that has a parallel implementation can be modelled by a group of machines inside a controller.

The controller for the group b-acs that determines the autonomous behavior of the pod is in Section 4.3. The rest of the model is discussed in [5].

Table 3 Mapping from the pod inputs and outputs to the RoboChart robotic platform.

System input	Name in the model
Sensors	status eps_outboard epb battery powertrain eps indicator epsInboard
Fault detection	faultDetectResponse
System State	control_command1
GPS	latlng
Dead man's handle	deadmanshandle
Gamepad	rc_in
Configuration files	configGoal configController configAutonomousDemand configGeofence configDeadMansController

System output	Name in the model
Speed	setSpeed setParkingBrake
Steering	setFrontSteering setRearSteering
Auxiliaries	setAuxiliaries
System state	requestState
Fault detection	sendHandshakeResponse
Display	display

4.2 Robotic platform

The pod sensors are modelled as inputs and the actuators as outputs of the controller software; Figure 2 shows these inputs and outputs. Table 3 lists the names of the corresponding elements used in the RoboChart robotic platform (see Figure 4).

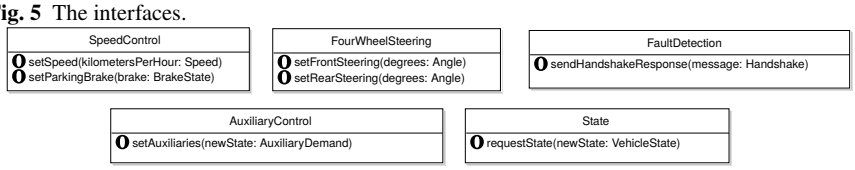
Besides the inputs and outputs of Figure 2, there are configuration files that record ROS parameters for the nodes. They are stored in a server accessible by the nodes and can be modified at runtime. So, the parameters are a form of input represented in the robotic platform. For example, configDeadMansController communicates a record whose fields specify minimum and maximum ranges for analogue inputs; the `DeadMansHandle` node uses this parameter to validate its inputs.

The inputs provide information about a robot's state or environment, therefore, they are modelled as events. Outputs alter the state of the system; this means that they can be mapped to variables, events, or operations.

For our example, outputs that significantly affect the state of the system, for instance, the movement of the pod, are modelled as operations. Because the display does not affect the state of the system, it is modelled as an event.

For ease of reference, variable, events, and operations can be grouped in interfaces. Figure 5 shows the interfaces of our model. They group the operations, and are provided by the platform and required by the controllers, as described next.

Fig. 5 The interfaces.



4.3 B_acs controller

To define the B_acs controller, the behavior of the nodes in Figure 3 has to be captured. Some of them are modelled via the abstraction provided by the robotic platform. The UDP/client node provides connectivity between the several devices that comprise the robotic platform, and are abstracted away by the definition of the platform as a single component. The xnav550 and arduino nodes translate low-level sensor data into ROS messages captured as input events in the platform: location, safetyDriverInput, and remoteControl. All other nodes contribute to the functionality of the controller and so are modelled as state machines.

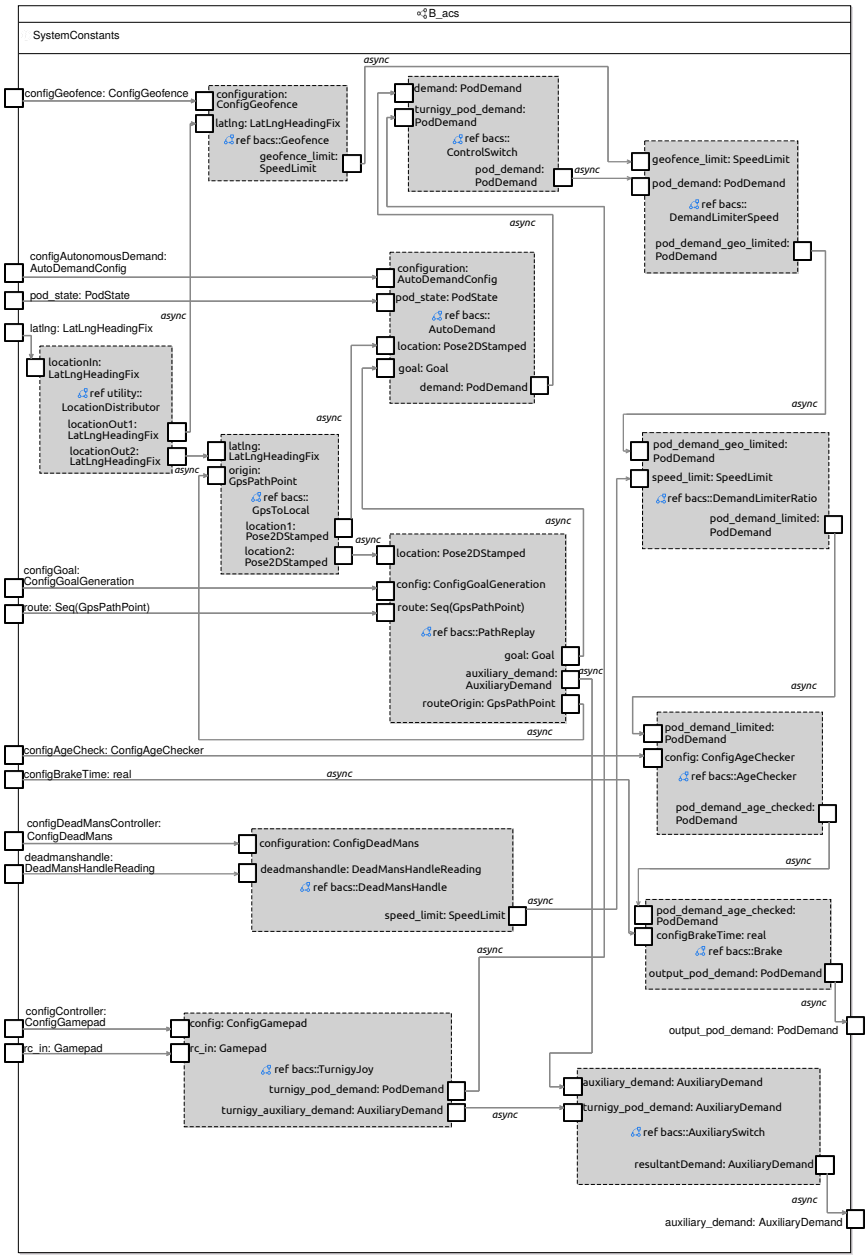
Figure 6 depicts the B_acs controller: its machines and connections. The messages used for communication between nodes are captured by events of their corresponding machines. Their connections correspond to ROS topics, which are used to communicate the messages via the publish and subscribe ROS mechanism. So, overall, the connections mirror the edges in Figure 3. Given the nature of the publish and subscribe mechanism, the connections are all asynchronous.

The structure of the messages are modelled using RoboChart data types and fields. For example, the `LatLngHeadingFix` message consists of three doubles representing a latitude, a longitude, and a heading; this can be represented using a RoboChart data type with three fields of type real. These message types are used to define the types of the events that represent the communications via these messages.

The states of the RoboChart state machines are determined by control flow analysis of the source code for the corresponding ROS nodes. As a small example, we show in Listing 1 pseudocode that is representative of the implementation of the dead_mans_handle node. The corresponding state machine is shown in Figure 7.

The method `deadMansHandleNode()` (line 7) is the node's constructor; it gets configuration information from the ROS parameter server (lines 8 and 9), and subscribes and publishes to the topics used in the node (lines 11 and 12). In the state machine DeadMansHandle, the check for availability of new parameter values, carried out by the calls to getParam, is captured by a communication via a configuration event. The subscribed and published topics are represented by the events deadmanshandle and speed_limit. The type ConfigDeadMans of configuration is a record with three fields: two corresponding to the variables `rangeMin` and `rangeMax` (lines 1 and 2), and a third boolean field retrieved to indicate whether a new configuration is available. The types of deadmanshandle and speed_limit are primitive (basic) types corresponding to those in the code.

Fig. 6 The B_acs controller



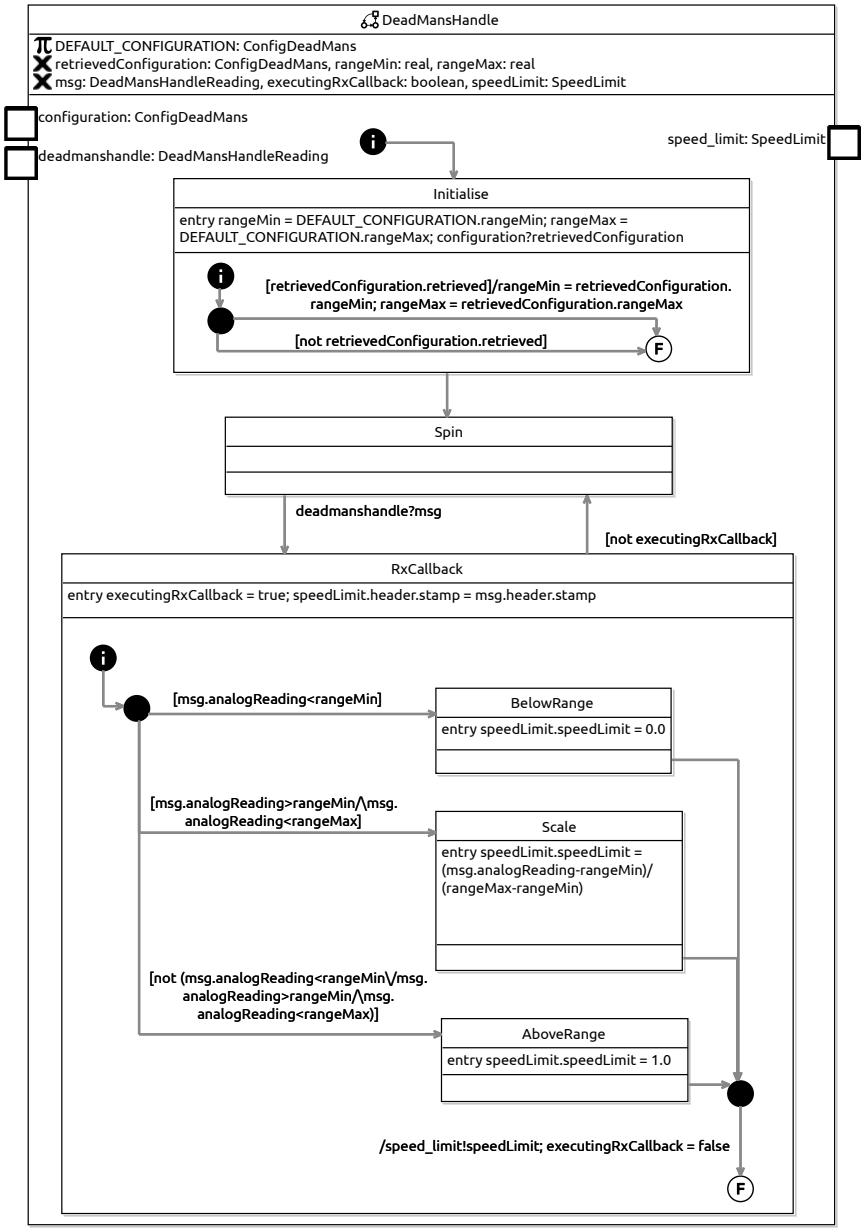
Listing 1 Pseudo code for the ROS node dead_mans_handle.

```
1  rangeMin 100
2  rangeMax 500
3
4  sub  //Instance of ros::Subscriber class
5  pubSpeedLimit  //Instance of ros::Publisher class
6
7  deadMansHandleNode(){
8      nodeParameters.getParam("rangeMin", rangeMin);
9      nodeParameters.getParam("rangeMax", rangeMax);
10
11     sub = node.subscribe("arduino/deadMansHandle", 10, rxCallback
       , this)
12     pubSpeedLimit = node.advertise("cavlab_core/speed_limit", 10)
13 }
14
15 rxCallback( deadMansHandleReading & msg ){
16     speedLimit  //Instance of SpeedLimit class
17
18     speedLimit.header.stamp = msg.header.stamp;
19
20     if (msg.analogReading < rangeMin){
21         speedLimit.speedLimit = 0.0;
22     } else if (msg.analogReading > rangeMin && msg.analogReading
       < rangeMax){
23         speedLimit.speedLimit = (msg.analogReading - rangeMin) /
       (rangeMax - rangeMin);
24     } else{
25         speedLimit.speedLimit = 1.0;
26     }
27
28     pubSpeedLimit.publish(speedLimit)
29 }
```

When the `deadMansHandleNode()` node constructor is executing (lines 7-13), the node can be considered to be initialising. So, the first state in the RoboChart model is a composite Initialise state. On entry to Initialise, DEFAULT_CONFIGURATION parameter values are assigned to the variables rangeMin and rangeMax (lines 1 and 2). Next, an input retrievedConfiguration is taken via configuration (lines 8 and 9). The machine in Initialise captures the behavior of getParam. If the call returns true, that is, retrievedConfiguration.retrieve holds, rangeMin and rangeMax are updated. Otherwise, the input is ignored.

The ROS nodes in the pod control system have either a periodic or a aperiodic control flow. In both cases, after the initialization, the `spin()` method is invoked. It simply calls the ROS `ros_spin()` method, which blocks handling asynchronously received messages on subscribed topics by calling the corresponding callback methods defined by the node. So, the callback methods define the node's behavior. In Listing 1, the callback method rxCallback is in lines 15-29.

Fig. 7 The DeadMansHandle machine.



In the RoboChart model, from the Initialise state, a transition (without a label) moves immediately to the state Spin. This captures the behavior as the node waits for

a message msg. When it arrives (via the event deadmanshandle), a transition leads to a (composite) state RxCallback that models the callback method.

A boolean variable executingRxCallback is used to ensure that the state RxCallback is not left until the behavior corresponding to the execution of the method is finished. So, upon entry of RxCallback, this variable is set to true, and the only transition out of RxCallback has a guard that requires it to be false.

In the machine in RxCallback, the initial junction leads to a junction that corresponds to the if-else structure in `rxCallback` (lines 20-26). The guard in each transition coming out of the junction matches those in lines 20, 22, and 24 (which has an implicit condition). Each target state has entry actions that match the assignments in the code (lines 21, 23, and 25). From each state, there is an immediate transition to a junction with a single transition to a final state. The action of that transition communicates the output via the event speed_limit, corresponding to the publish statement in line 28 of Listing 1. The transition action also updates executingRxCallback so that a transition out of RxCallback leads back to Spin.

Other machines are presented in [5]. We now briefly discuss their verification.

4.4 Verification

The RoboChart model of the autonomous vehicle enables both core and user-specified timed and untimed properties of the controller software to be verified automatically. The core properties that are automatically generated include: deadlock and livelock freedom, determinism, termination, and reachability. Other properties of interest can be, for the moment, defined as CSP processes that are verified by refinement checking against the calculated semantics.

RoboTool supports the specification of the core properties to be verified utilising an assertion language that uses controlled English [53]. For example, to check the core property that the DeadMansHandle machine is deterministic, the corresponding statement written using the assertion language is as follows.

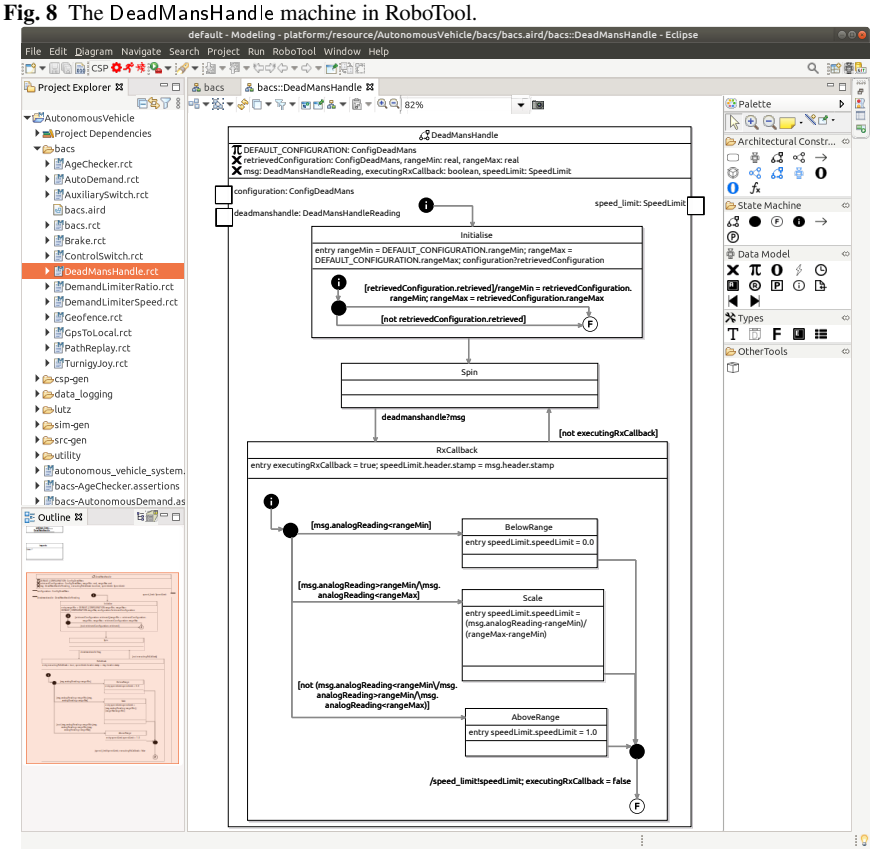
assertion DMH_1**:**
bacs::DeadMansHandle::DeadMansHandle **is deterministic**

The assertion keywords are indicated in boldface and specify the property to be checked. DMH_1 is a user-defined label for easy identification of the property. The next part of the **assertion** statement is a fully qualified name of a component (module, controller, or machine) from the RoboChart model. Figures 8 and 9 show RoboTool and a report generated in the verification of the DeadMansHandle machine.

For the DeadMansHandle machine, an important property, which we name DMH_INOUT, is that for every input received from deadmanshandle an output is generated via speed_limit; this can be expressed as shown below in CSP.

The expression defines a CSP process *OFEI* (Output For Every Input) that offers a choice of events corresponding to the three events of the DeadMansHandle machine. (Fully qualified names have been truncated for readability.)

Fig. 8 The DeadMansHandle machine in RoboTool.

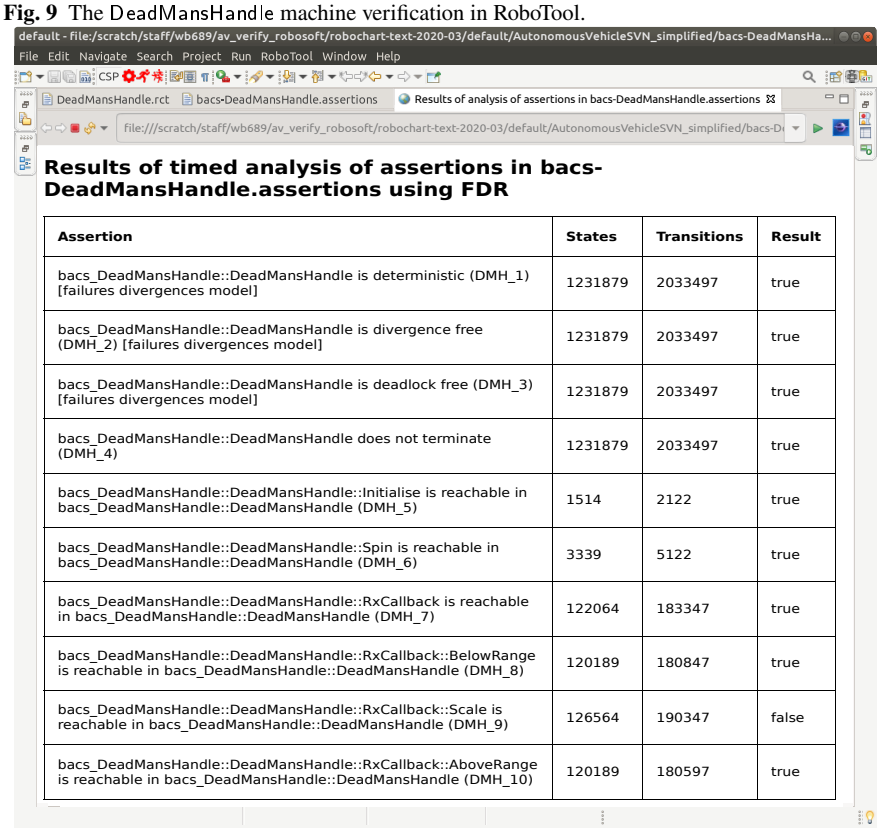


$$
\begin{aligned}
&\mathit{OFEI} = \\
&\quad \mathit{configuration.in?currentConfiguration} \longrightarrow \mathit{OFEI} \\
&\quad \Box \\
&\quad \mathit{speed\_limit.out?speedLimit} \longrightarrow \mathit{OFEI} \\
&\quad \Box \\
&\quad \mathit{deadmanshandle.in?msg} \longrightarrow \mathit{speed\_limit.out?speedLimit} \longrightarrow \mathit{OFEI}
\end{aligned}
$$

The first two choices consider occurrences of the events *configuration*, which is an *in*put, and *speed_limit*, an *out*put. In these cases, *OFEI* recurses, placing no constraints on the behavior of the machine being verified. The final choice is an *in*put via *deadmanshandle*, which is immediately followed by a *speed_limit out*put event. This describes the desired property DMH_INOUT shown below.

The assertion language can be used to specify refinement checks against the defined CSP expressions and instances of RoboChart components. The **assertion** DMH_INOUT specifies a refinement check and the **model** to use for verification; in our example, we consider just the **traces** of the process.

Fig. 9 The DeadMansHandle machine verification in RoboTool.



default - File:/scratch/staff/wb689/av_verify_robosoft/robochart-text-2020-03/default/AutonomousVehicleSVN_simplified/bacs-DeadMansHa...

File Edit Navigate Search Project Run RoboTool Window Help

DeadMansHandle.rct | bacs-DeadMansHandle.assertions | Results of analysis of assertions in bacs-DeadMansHandle.assertions

file:///scratch/staff/wb689/av_verify_robosoft/robochart-text-2020-03/default/AutonomousVehicleSVN_simplified/bacs-D

Results of timed analysis of assertions in bacs-DeadMansHandle.assertions using FDR

Assertion	States	Transitions	Result
bacs_DeadMansHandle::DeadMansHandle is deterministic (DMH_1) [failures divergences model]	1231879	2033497	true
bacs_DeadMansHandle::DeadMansHandle is divergence free (DMH_2) [failures divergences model]	1231879	2033497	true
bacs_DeadMansHandle::DeadMansHandle is deadlock free (DMH_3) [failures divergences model]	1231879	2033497	true
bacs_DeadMansHandle::DeadMansHandle does not terminate (DMH_4)	1231879	2033497	true
bacs_DeadMansHandle::DeadMansHandle::Initialise is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_5)	1514	2122	true
bacs_DeadMansHandle::DeadMansHandle::Spin is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_6)	3339	5122	true
bacs_DeadMansHandle::DeadMansHandle::RxCallback is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_7)	122064	183347	true
bacs_DeadMansHandle::DeadMansHandle::RxCallback::BelowRange is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_8)	120189	180847	true
bacs_DeadMansHandle::DeadMansHandle::RxCallback::Scale is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_9)	126564	190347	false
bacs_DeadMansHandle::DeadMansHandle::RxCallback::AboveRange is reachable in bacs_DeadMansHandle::DeadMansHandle (DMH_10)	120189	180597	true

assertion DMH_INOUT**:**
bacs::DeadMansHandle::DeadMansHandle **refines** OFEI **in the traces model**

Similar processes and assertions can be used to verify that the value of the output speed limit must always be less than or equal to a given maximum (assertion DMH_OUT_BELOW_MAX) and the out speed limit must always be greater than or equal to zero (DMH_OUT_ABOVE_MIN). For that, we define processes similar to OFEI, but restrict the values of the outputs that are produced to the valid sets.

For abstraction, RoboChart models can contain undefined types, constants, and functions that must be defined to verify the properties specified. For model checking, RoboTool generates preliminary instantiations for all of the undefined elements and core types. These instantiations need to be tailored appropriately to the domain of the system, noting that a large cardinality of definitions of sets leads to models that are complex and require significant resources to verify. The instantiation file used for verification of the DeadMansHandle state machine is shown in Listing 2. To prevent RoboTool from overwriting customized instantiations, `not` is appended to the end of a description to suppress regeneration of the corresponding definition.

Listing 2 DeadMansHandle instantiations.

```
1 -- generate real not
2 nametype core_real = { -2..2}
3
4 -- generate
5 -- const_bacs_DeadMansHandle_DeadMansHandle_DEFAULT_CONFIGURATION
      not
6 const_bacs_DeadMansHandle_DeadMansHandle_DEFAULT_CONFIGURATION =
    (-1, 1, false)
```

Table 4 The verification results for the DeadMansHandle machine. Legend: ① Deterministic, ② Divergence freedom, ③ Deadlock freedom, ④ Does not terminate, ⑤ All states are reachable, ⑥ DMH_INOUT, ⑦DMH_OUT_BELOW_MAX, ⑧ DMH_OUT_ABOVE_MIN

	Property								
State Machine	①	②	③	④	⑤	⑥	⑦	⑧	**Note**
DeadMansHandle	✓	✓	✓	✓	✓	✓	✓	✓	
DeadMansHandle	✓	✓	✓	✓	✓	✓	✓	✓	Timed

Types have been represented by minimal sets, for example, `core_real` (line 2) ranges from -2 to 2. These values can be used to represent data domains of the DeadMansHandle machine; for example, we can have 2 to represent values above the maximum range, 1 for the that maximum, 0 for values in range, and so on. The FDR model checker does not support real numbers, so, for verification, we need to use integers, and may need to make approximations in the model. To the best of our knowledge, there are no model checkers for FDR that deal with real numbers.

The results obtained in verifying properties for the aperiodic DeadMansHandle machine are summarized in Table 4. Further verification, associated with the generation of a simulation, is discussed in the next section.

5 Simulation

As well as verification, a RoboChart model can also be used as a basis to develop a simulation. It is, of course, possible to develop a simulation from scratch, as it is usually the case. Even in this scenario, there is still value derived from using a RoboChart model for guidance because it is precisely described in an organized notation. As already mentioned, with the use of RoboTool, it can be guaranteed that the model is valid (well typed, all operations are declared, all states are connected, and so on) and core properties can be checked automatically. So, a RoboChart model is a high-quality starting point for further work on coding.

In addition, it is possible to check whether the RoboChart model can be accurately described by a simulation at all. A simulation is an iterative mechanism; in each cycle, the inputs are read, the data is processed, the outputs are produced, and then time is

advanced. So, developing a simulation corresponding to a RoboChart model requires scheduling the processing in the state machines in cycles. This may not be possible if, for example, the RoboChart model requires at some point two urgent calls to the same operation. It is not possible to call the same operation twice in a simulation cycle, and, so, if both calls are urgent, that timed behavior cannot be reproduced in a simulation. Such a RoboChart model needs to be revisited.

An automated schedulability check can reveal such a problem. It consists of verifying whether, in the presence of the assumptions normally made when writing a simulation, the RoboChart model does not deadlock. The assumptions relate the events, variables, and operations of the robotic platform of a RoboChart model to the inputs and outputs read in a simulation cycle. They also enforce the restriction on operation calls. Absence of deadlock in the presence of these assumptions ensures that there is no behavior of the RoboChart model that cannot be implemented in the simulation paradigm that reflects the assumptions.

For our example, DeadMansHandle state machine, this check passes. It can be carried out automatically by RoboTool using the assertion below.

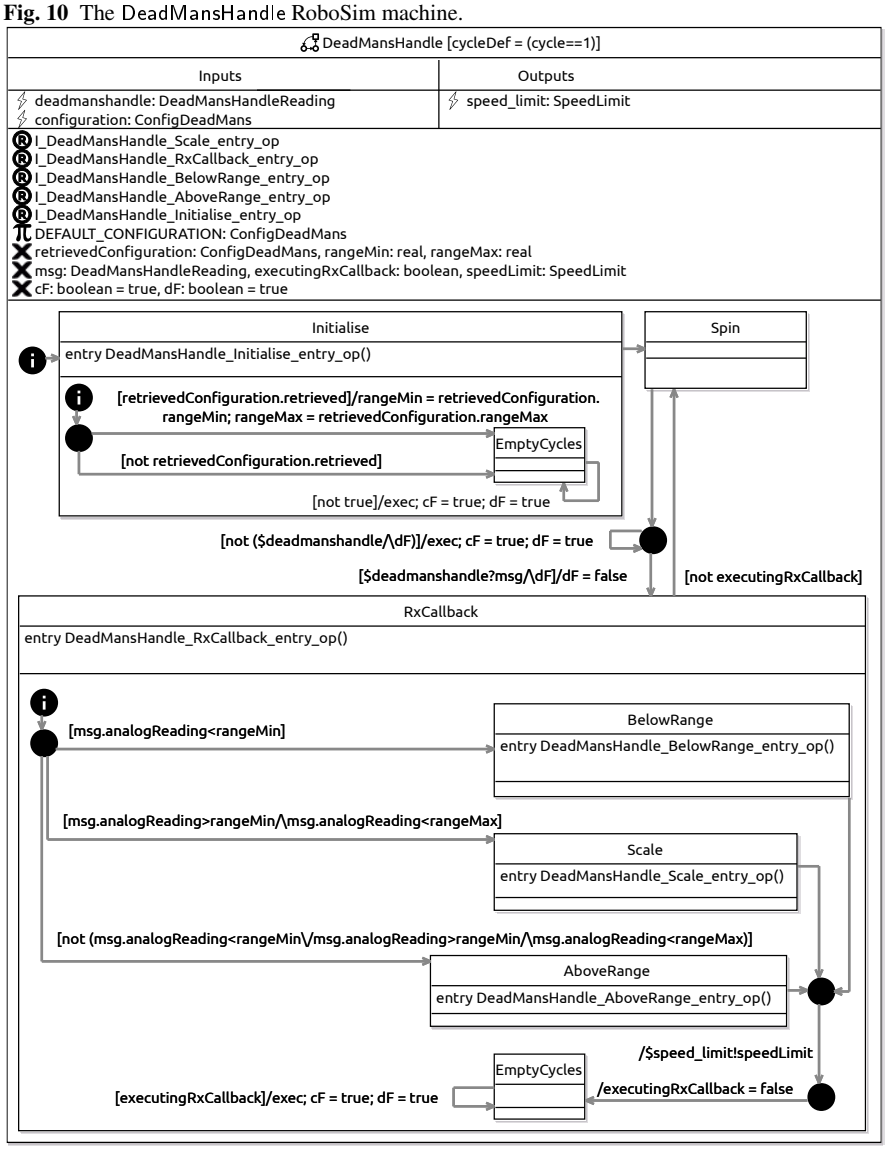
simulation DMHSim **of**
  bacs::DeadMansHandle::DeadMansHandle **{ cycleDef** cycle == 1 **}**
assertion Schedulable**:** DMHSim **is schedulable**

The **simulation** clause defines a simulation specification from a RoboChart component. In this example, we have a specification DMHSim based on DeadMansHandle. The specification requires the definition of a cycle, which is given in the **cycleDef** clause and embeds the simulation assumptions mentioned above. The **cycleDef** clause specifies the value of a variable cycle; in our example, it is 1. The actual **assertion**, called Schedulable in the example, requires that DMHSim **is schedulable**. Given that the pod model is constructed to reflect a (cyclic) implementation, it is not a surprise that the machines are schedulable in simulations.

If the RoboChart model is schedulable, we can generate simulations automatically. For constrained RoboChart models, it is possible to generate a C++ simulation for ARGoS. The gap between event-based control flow embedded in a design state machine and cycle-based control flow of a simulation, however, is very large. To bridge this gap and produce an artefact that describes the cycle-based mechanism faithfully, we have developed RoboSim. This is also a diagrammatic notation, with support for modelling, validation, and verification in RoboTool.

In what follows, we present two aspects of RoboSim. In Section 5.1, we describe a RoboSim module corresponding to the RoboChart module in Figure 4. This is an account of the simulation of the control software; it is called a d-model (for data model). We focus our discussion on the simulation of the state machine DeadMansHandle. In Section 5.2, we describe the RoboSim support to describe physical models, called p-models, and illustrate our ideas using a simplified version of the pod vehicle. Finally, Section 5.3 discusses our approach to generate simulation code from RoboSim models for use with robotics simulators.

Fig. 10 The DeadMansHandle RoboSim machine.



5.1 RoboSim: d-model

RoboSim has the same structure of modules, controllers, and machines as RoboChart. For our example, we use the same module and controller definitions in RoboChart to define the RoboSim simulation, except only that the RoboSim versions define a value for the length of the simulation cycle. For the machines, however, different models

give a cycle-based account of the behavior. We present in Figure 10 a RoboSim machine corresponding to DeadMansHandle in Figure 7. The RoboSim machine has the same name but defines the cycle in a cycleDef clause next to the name.

In a RoboSim machine, there is only one event (in the RoboChart sense) called exec, available without declaration. It controls the cyclic flow of the simulation. The processing phase of the simulation is defined by the machine, using the inputs, and the event exec to indicate when processing is finished and outputs can be provided.

In each cycle, inputs are read to determine whether the corresponding events have happened and, if so, the values that are provided as input. In the definition of the machine, if needed, this information is available. For our example, the inputs are deadmanshandle and configuration, declared as events in the RoboChart model, but used in RoboSim differently. Rather than as a trigger, we use $deadmanshandle?msg as a boolean in a guard, which is true if the input deadmanshandle event has happened. In this case, the input value is recorded in the local variable msg.

The transition whose guard includes $deadmanshandle?msg is part of the translation of the transition from Spin to RxCallback in the RoboChart model. In the first cycle, like in the RoboChart model, the machine starts in Initialise and proceeds to Spin, and from there to a junction. In the junction, there is a decision characterized by the guards of the outgoing transitions. If $deadmanshandle?msg is true, the simulation machine moves to the state RxCallback, as in the RoboChart machine.

The guard uses also a local boolean variable dF initialized to true. Our model-transformation strategy for generating a RoboSim model from a RoboChart model uses such variables to record whether an associated input event, in this case, deadmanshandle has been referenced. It ensures that only one reference is possible in each cycle. This is to match the structure of the RoboChart model, where multiple references to an event denote different occurrences of that event. In RoboSim, a different occurrence can take place only in a future cycle.

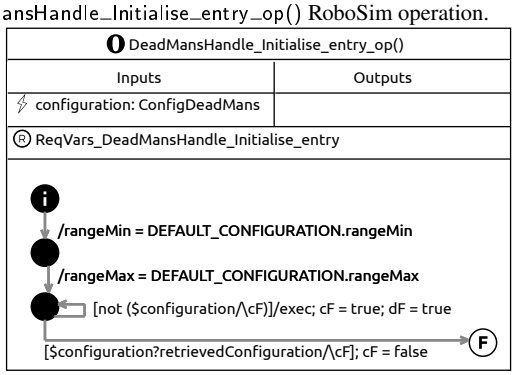
Another transition from the junction is for when deadmanshandle does not happen. In RoboChart, the machine remains in Spin waiting to be interrupted by the event. In RoboSim, exec is raised to allow the simulation to progress to the next cycle (when it then might be the case that the event happens). In this extra transition, all boolean variables (dF and cF, for $configuration) are reset to true. No variable is associated with the reference to $deadmanshandle, so no variable is updated.

In the translation, all state actions (that is, entry, during, and exit actions) become calls to operations that capture the original action in the RoboChart model. In our example, for instance, we have an operation DeadMansHandle_Initialise_entry_op() for the entry action of Initialise. Its definition is shown in Figure 11.

This transformation to include operation calls is part of an initial normalization phase in the model transformation. It removes or reduces complexity in the potentially rich structure of the RoboChart model to simplify translation. For the state actions, normalization ensures that they all have the same form: an operation call. In the actions themselves, structure defined by sequence (;) and conditionals (if-then-else), for example, is also removed in favour of exclusive use of states and junctions.

Based on the variables (including constants), events, and operations referenced in the original action, an interface is defined and required in the corresponding operation

Fig. 11 The DeadMansHandle_Initialise_entry_op() RoboSim operation.



definition. The interface ReqVars_DeadMansHandle_Initialise_entry required in DeadMansHandle_Initialise_entry_op() collects retrieveConfiguration, rangeMin, rangeMax, cF, and dF, and the constant DEFAULT_CONFIGURATION.

Each primitive action (assignment or input) in the entry action of Initialise occurs as an isolated transition action in DeadMansHandle_Initialise_entry_op(). This results from the normalization. In the case of the input, similar to what is done in DeadMansHandle, an extra transition with event exec allows the cycles to proceed.

The structures of the machines for Initialise and RxCallback are very similar to those of the original RoboChart machines. Of note is the fact that, during processing, outputs are defined, but become visible to other machines, controllers, and the platform only when exec occurs. So, in the RoboChart machine for RxCallback, the output via speed_limit is the trigger of the transition to the final state. In the RoboSim machine, the output occurs as a transition action $speed_limit!speedLimit. The use of $speed_Limit, instead of simply speed_limit, indicates that this is an output that is only actually visible when the processing phase of the simulation terminates.

The final states of the machines for Initialise and RxCallback are replaced with a new state EmptyCycles. This is because, in general, the final state of a machine of a composite state needs to explicitly raise the exec event to allow the cycles to proceed. So, instead of the special final state, the translation uses a new state with an extra transition that has an exec action. This transition is guarded by the negation of the guards of the transitions that leave that composite state.

In the example, there is only one outgoing transition from Initialise and one from RxCallback. In the case of Initialise, the negation of its outgoing transition is just false; in the case of RxCallback, it is executingRxCallback. In both cases, the extra transitions are never taken because their guards never hold. For RxCallback, as soon as its machine finishes, the outgoing transition is enabled because of the action executingRxCallback = false, and the state machine DeadMansHandle exits RxCallback. In general, however, this is not the case.

Generating code for a simulation from a RoboSim module is much more direct than from a RoboChart module. In the case of RoboSim, cyclic behavior is already identified. If the RoboSim model is verified against, or generated automatically, from

a RoboChart model, we can be certain that its properties are preserved (given the assumptions characteristic of the cyclic control flow of a simulation). For example, we know that the outputs produced are as specified by the RoboChart model for the inputs provided, and time budgets and deadlines are preserved.

For a simulation, however, as well as an account of the control software, we also need a model of the physical robot. This is discussed in the next section.

5.2 RoboSim: p-model

RoboSim has a block-diagram notation that allows us to describe robotic platforms by characterising their physical properties and behaviors using systems of differential-algebraic equations. Figure 12 presents a RoboSim specification of a (simplified) physical model (p-model) for the pod in our running example.

A diagram for a p-model defines blocks to represent links (that is, rigid bodies), joints, sensors, and actuators, as well as blocks to represent some of their properties. The diagram defines a tree structure that specifies a containment relationship between elements. The root of the tree represents the physical component as a whole. Its children are blocks representing links (or parts) of that physical component. Links may contain junctions; links and junctions may contain sensors and actuators; and so on. In Figure 12, the tree represents a physical model for the pod called Vehicle.

This simplified model for the vehicle omits its body (just for conciseness). What is modelled is the frame and the wheels, as shown in Figure 13. This p-model contains several links. A link called frame has three bodies, namely, front, column, and rear. These are the bars that define the H shape in Figure 13. Since frame is a link, its multiples bodies are pieces of a single rigid component.

The containment association is represented by connections between blocks with a closed lozenge on the side of the containing block. They can be annotated with the position coordinates (x, y, and z), and orientation (roll, pitch, and yaw) of the contained block. These are defined with respect to the frame of reference of the containing block. For the p-model block, by convention, the position and orientation are (0,0,0) and (0,0,0). In Figure 12, the front and rear bars are positioned by identifying their y coordinates as 0.975m to the front and to the back (-0.975m) of the column. They have the same orientation as the vehicle.

The wheels are also links of the Vehicle. In Figure 12, they are left_front_wheel, right_front_wheel, left_rear_wheel, and right_rear_wheel, defined as parts. Each part is an instance of a p-model defined separately by another block diagram. In this example, the p-model is Wheel; it is shown in Figure 14. Each Wheel has a link wheelL with a body wheelB defined as a Cylinder.

The left_rear_wheel and right_rear_wheel are connected by an axle, whose body is a Cylinder. The connection to the wheels are fixed: represented by solid lines.

Two more parts, left_kingpin and right_kingpin, are instances of a KingpinAssembly that can be used to turn a wheel. As shown in Figure 14, this is a Box with a Revolute joint kingpin_joint, whose axis is the wheel_axis defined as a parameter to

Fig. 12 The pod p-model.

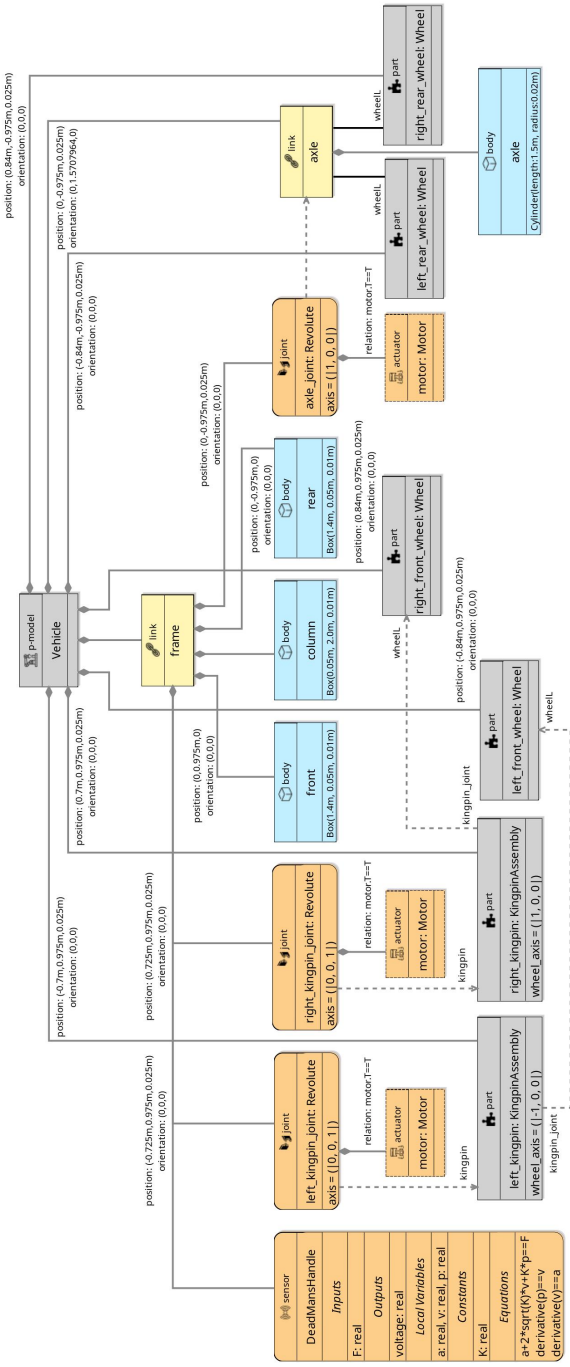


Fig. 13 The pod in CoppeliaSim.

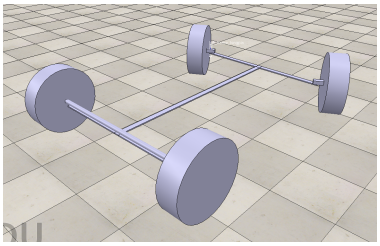
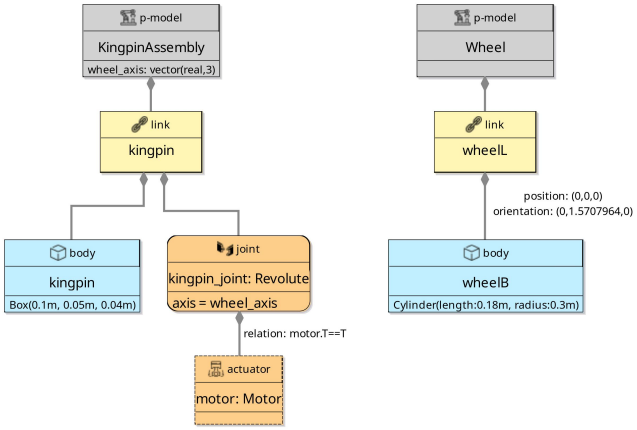


Fig. 14 The physical models for the parts used in the pod.



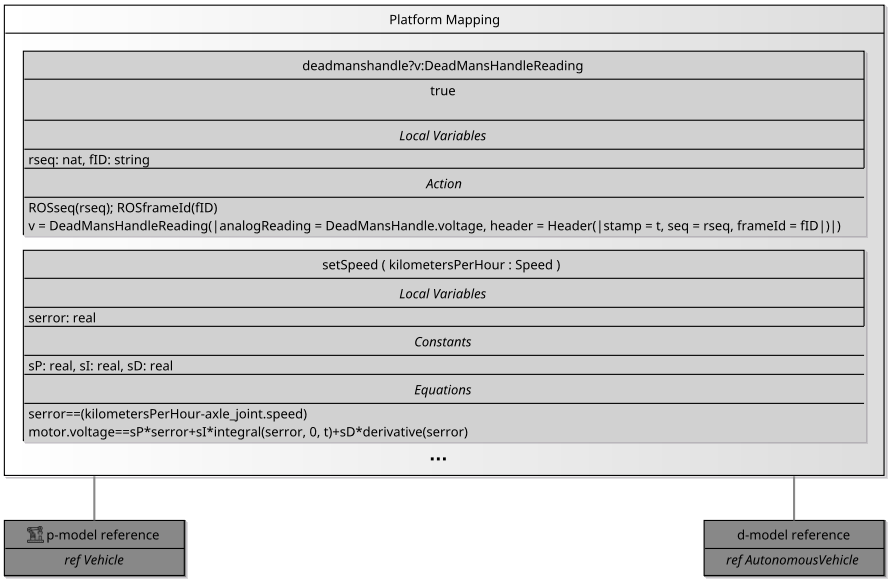
KingpinAssembly. In the definitions of left_rear_wheel and right_rear_wheel, the values of this parameter are specified for the wheel associated with the kingpin.

Every joint is contained in a link, and has a flexible connection, identified by a dashed arrow, to another link. The joint kingpin_joint of a KingpinAssembly is contained in its link. Its flexible connection is specified when defining a part. In Figure 12, the flexible connections are to the wheels. Connections to and from a part are annotated with the elements of the part that are being connected. In the case of both left_kingpin and right_kingpin, the kingpin_joint is flexibly connected to the link wheelL (of left_front_wheel and of right_front_wheel).

The frame has three Revolute joints: left_kingpin_joint and right_kingpin_joint, flexibly connected to the links in left_kingpin and right_kingpin to turn the front wheels, and axle_joint, flexibly connected to the axle for the rear wheels. The behavior of a Revolute joint, and others, is defined as part of a library, using a system of differential-algebraic equations. Sensors and actuators can also have their behavior defined in this way, and the library includes a number of such definitions. In our example, all joints contain a motor, an actuator defined in a library.

As an example, we show a sensor DeadMansHandle contained in the frame. Its input is the force F applied to the handle, and its output is a voltage. The equations at

Fig. 15 The platform mapping for the pod.



the bottom of the DeadMansHandle block, use local variables a, v, and p, to record the acceleration, velocity, and position of the handle, and the spring constant K. When the driver releases the handle, it goes back to the initial position.

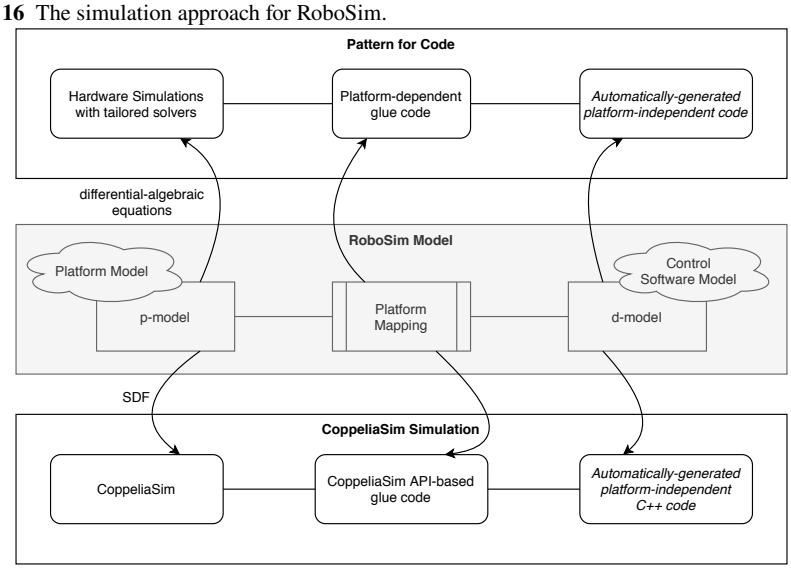
A more realistic physical model for the pod includes a description of its body and properties such as its weight and shape, which have an effect on its motion. Moreover, the simple steering mechanism based solely on kingpin joints is not what is used in the pod. It uses two sets of rack-and-pinion steering mechanisms, although only the front set is used by the control software. The functionality of the simplified steering component presented here is all that is needed for the control software.

To give an account of the behavior of the robot, considering both its control software, defined by a d-model, and its physical platform, defined by a p-model, we need to combine these models. Their connection is via the variables, events, and operations defined in the d-model for the platform. These elements identify the visible behavior of the software. So, to make the connection, we need to define them using elements of the p-model: outputs of sensors and inputs of actuators.

This definition is provided via a platform mapping. For our example, a sketch is presented in Figure 15. The mapping box connects a d-model, in our example, AutonomousVehicle, to a p-model, here, Vehicle. In the box, we define each variable, event, or operation of the connected d-model. In Figure 15, we present just two such definitions: for the event deadmanshandle and for operation setSpeed.

For events, a condition specifies when they happen. In our example, the condition for deadmanshandle is just true, indicating that this event is always available. The Action defines the communicated value v, using two Local Variables rseq and fID

Fig. 16 The simulation approach for RoboSim.



to record elements of a ROS header defined by built-in platform software operations called ROSseq and ROSframeId. These variables are used to define the record of type DeadMansHandleReading that is assigned to v. It records the voltage output by the sensor DeadMansHandle in the p-model, and, in the header, the time t in which deadmanshandle occurs and the message is communicated. The operation setSpeed is defined by Equations that define a PID that reaches the speed kilometersPerHour.

To validate a p-model, and to execute the simulation as a whole, we need code for a robotics simulator. Generation of such code is discussed next.

5.3 Simulation code

Figure 16 summarizes our approach to simulation of RoboSim models. As already mentioned, a RoboSim model (light-grey box in Figure 16) is composed of three distinct components. The *p-model* is a block diagram that describes the physical platform in terms of links, joints, sensors, actuators, and their equations. The *d-model* is a RoboSim module that specifies the control software in terms of variables, events, and operations of the platform. The *platform mapping* describes how they are interpreted in terms of variables (continuous flows) of the physical model.

Accordingly, a simulation derived from a RoboSim model consists of three components, matching the structure of the RoboSim model; see Figure 16. In general, the p-model needs to be used to produce a system of differential-algebraic equations that is passed to an off-the-shelf or tailor-built solver to simulate the continuous behaviors of the system. The d-model needs to be used to automatically generate

platform-independent code. Finally, the platform mapping needs to be used to generate solver-dependent interface code that bridges the gap between the control software and platform simulations, passing data back and forth between them.

This general pattern can be adopted to develop a simulation from scratch, but can also be instantiated for an existing simulator to streamline the simulation process. Our approach to generating simulations relies on existing simulators. We have experience with ARGoS and CoppeliaSim. Such robotics simulators embed domain knowledge, such as multi-body physics notions like links and joints. They also provide the means to simulate their behaviors efficiently through the use of various physics engines.

Figure 16 describes the instantiation of our pattern for use with CoppeliaSim. The d-model is used in the same way, with the generated code suitable for any simulator that adopts the used programming language. For CoppeliaSim, we use C++. The p-model does not need to be used to provide the equations for simulation. Abstractions such as link and joints are available in CoppeliaSim and can be imported via SDF, which can be used as an input to various robotics simulators. Finally, the platform mapping is used to produce an interface that implements the platform's variables, events, and operations used by d-model in terms of the API of CoppeliaSim.

With a high level of automation, fixing problems found during simulation costs much less. Tests for use with the simulation are discussed in Chapter 11.

6 Environment modelling

Another core component of a robotic system is the environment in which it operates. For RoboSim, as a simulation language, we need a notation to characterize a particular scenario (or a collection of specific scenarios). We envisage the use of a block diagram, like for a p-model definition, to specify an e-model, that is, an environment model. Like for a p-model, such a diagram defines the physical elements of the scenario and their behavior. An environment mapping needs to describe how elements of the scenario are perceived and affected by the sensors and actuators.

For a RoboChart model, the possibility of defining only a particular scenario is too restrictive. In a design model, we need, instead, to identify the assumptions about the environment and the robotic platform that need to be satisfied to ensure the proper behavior of the robotic system. The assumptions are the operational conditions.

When documented, if at all, these assumptions are commonly expressed in natural language. In the RoboStar technology, these assumptions, which abstract general properties of any valid environment and platform, are also written in natural language, but using controlled English. For natural-language processing (NLP), we need to manage the trade-off between unconstrained text and automation capabilities.

Some NLP techniques, such as the one described in [14], do not restrict the text and, thus, can be considered to deal with fully natural languages. In general, they are built on artificial intelligence techniques, and rely on a large corpus of sentences to train the underlying models to be capable of processing new unseen text. This approach is not suitable for RoboStar, since we do not have a large corpus of envi-

ronment and platform assumptions; many times, they are not properly documented, and left as part of the roboticists tacit knowledge.

At the other end of the spectrum of NLP techniques, we have very constrained languages, such as that in [21]. They require loss in naturalness by considering fragments of formal definitions and programming concepts. The imposed structure, however, favours text processing automation even without a corpus of examples.

In the middle of this spectrum, we have approaches that seek for a compromise between naturalness and constrained writing [77, 46, 11]; we seek this compromise. We have devised RoboWorld, a controlled natural language (CNL) for the specification of environment and platform assumptions with a precise semantics. From the controlled English, we can automatically generate CSP scripts of models that support the consideration of these assumptions within the RoboStar technology.

RoboWorld is defined using the Grammatical Framework (GF) [65], a special-purpose functional programming language for developing grammars. It supports the complexities found in different natural languages, such as word inflections and agreement between elements of a sentence. In Section 6.1 we comment on the syntax of RoboWorld, followed by an overview of its semantics in Section 6.2.

6.1 RoboWorld syntax

RoboWorld is defined with abstract and concrete grammars. The abstract grammar defines the types of assumptions that can be described in the language. It can be seen as a metamodel of the supported controlled English. Differently, the concrete grammar relates the metamodel with actual English sentences.

The grammars establish that a RoboWorld specification defines assumptions about the world, including the environment and the platform, and mapping information. The mapping explains how the world influences and is influenced by the values of the variables, events, and operations of the platform of a RoboChart module. The concept is similar to that of a platform mapping described in Section 5.2.

Here, we illustrate the syntax of RoboWorld via an example presented in Figure 17. It specifies some assumptions and part of the mapping for the autonomous pod. The environment assumptions highlight that the arena is two-dimensional, the ground is flat, and that we have obstacles in some locations of the arena.

To exemplify a mapping definition, we show how the input event deadmanshandle can be defined. The English description defines when the event occurs, and the value communicated. The assumption is slightly more abstract than the definition in the platform mapping in Figure 15, since it does not impose any restriction on values of the header that are not relevant for the software. The RoboChart module for the pod reflects in many ways the fact that we have a ROS application, so it is to be expected that features of ROS are assumed (like the format of messages).

In general, we can be more abstract if it is convenient. For example, if an application has an input event obstacle, the RoboWorld mapping can specify “The event obstacle happens when the robot is less than one meter from a location in which

Fig. 17 RoboWorld assumptions for the pod.

```
## World assumptions ##
The arena is two-dimensional.
The gradient of the ground is 0.
There are entities called obstacle.
Some of the locations contain an obstacle.

## Mapping definitions ##
** Output event definitions **
** Input event definitions **
The event deadmanshandle is always available, and it communicates a record
whose field analogReading records the voltage obtained from the
DeadMansHandle, and whose field header is a record whose field stamp
records the time the event occurs.
** Operation definitions **
When the operation setSpeed(x) is called, the speed of the robot is set
to x km/h.
** Variable definitions **
```

there is an object". In this case, there is no reference to a particular sensor, and a variety of technologies can be used to satisfy such an assumption.

In Figure 17, the definition of the operation `setSpeed` explains how it affects the robotic platform. Here, we have another example of a very abstract definition, where the use of motors that is needed to achieve the required speed is not mentioned. This is in direct contrast with the platform mapping in Figure 15.

As already said, RoboWorld has a precise formal semantics given in CSP. As we explain in the next section, in the semantics, there are concepts that are application independent, whereas others are derived directly from the controlled English specification, such as that presented in Figure 17.

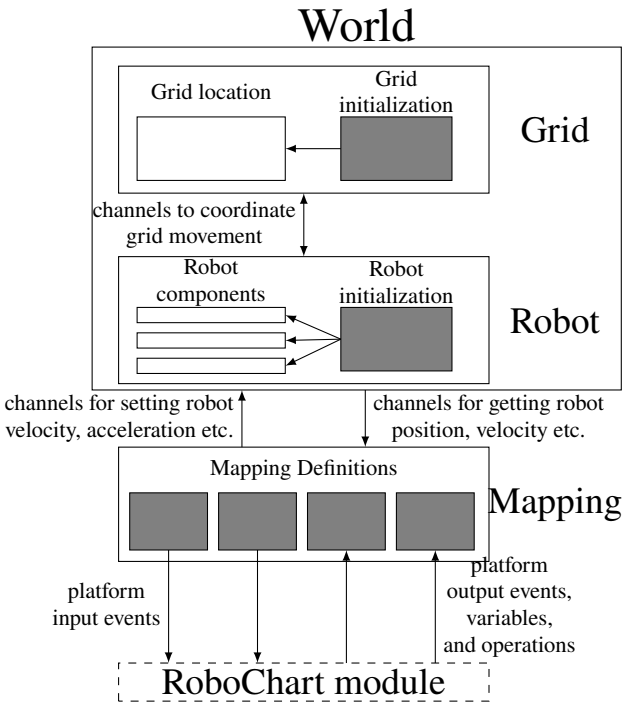
6.2 RoboWorld semantics

A CSP process that captures a RoboWorld specification consists of two components, shown in Figure 18: a process that captures the world assumptions, and another for the mapping. The process that captures the whole robotic system behavior also includes the CSP definition for the RoboChart module.

CSP processes communicate via channels. In the case of a process for a RoboChart module, there are channels to represent each of the variables, events, and operations of the robotic platform. In the system process in Figure 18, these channels are used for communication with the mapping process. A process for a RoboChart module, which defines control software, does not communicate directly with the world process.

The mapping process captures how the variables, events, and operations of the control software affect and are affected by the world. To specify that, the mapping needs to set and get information about the world. So, the mapping process commu-

Fig. 18 The structure of the RoboWorld CSP semantics.



nicates with the world process via get and set channels for properties like position, velocity, and acceleration of the robot, for example. The world process describes the layout and behavior of objects in the world.

The parts of the CSP process that are application-dependent and generated from the RoboWorld description are indicated in Figure 18 by grey shading. Roughly speaking, as probably expected, the world process is generated from the world assumptions, and the mapping process is generated from the mapping definitions.

The world process has two components: the grid and the robot processes. The grid process models a 2D or 3D arena as a grid of locations. With that, we can represent the position of each of the entities in the environment, including the robot itself. The world assumptions define the initial positions and properties of each of the entities on the grid, in terms of application-independent location processes that manage the movement and placing of entities on the grid.

The robot process has several components for managing the robot's dynamic behavior, such as movement and carrying of objects. These components are application-independent. For example, the processes that manage the robot's movement compute its path, but the robot is moved by communication with the grid process. The world assumptions are used to generate an initialization process that defines the movement components required, and specifies initial values for the robot's position and velocity.

The mapping includes a process for each mapping definition, specifying how the variables, events, and operations of the module affect and are affected by the behavior of the robot. For an input event, information about the robot, such as its position, is obtained from the world process and the event generated when the conditions for its occurrence specified in the mapping definition are met. For an output event or operation, the mapping process gets information from the module and communicates with the world to get and set values, such as the robot velocity and acceleration, as specified in the mapping definition. The mapping-definition processes are combined in parallel so that each input and output is handled separately.

We automate the translation of RoboWorld descriptions to CSP using the GF Java API. We use GF to parse the input text, and Java code to traverse the AST and generate the CSP script. RoboWorld embeds primitive notions of 2D and 3D arenas, regions, areas, objects (with a physical body), and entities (like gas and light), and so on. More examples are required to enrich RoboWorld further.

7 Related work

Early on, existing mathematical techniques [20, 38] have been applied to robotics. Model-checking techniques are available for many general-purpose languages. The goal of the RoboStar technology, however, is customization to produce simple domain-specific languages for practitioners [18, 60, 10], and with tool support for graphical modelling, and optimizations in the semantics and verification that do not apply in general. Nordmann *et al.* [56] suggest that domain-specific languages for robotics like RoboChart and RoboSim are becoming popular. Our work is distinctive in its use of mathematical models for verification.

There are several general languages for architectural and behavioral modelling: SysML [58], AADL [23], and Focus [9], and others. For SysML, a comprehensive semantics in a CSP-like language is available [43]. The RoboSim block diagrams used to specify physical models are based on SysML block diagrams.

UML (and its derivatives) have been used in various application domains, including safety-critical systems. There are many formalizations of UML, but, in general, covering subsets of UML. There are tailored semantic domains [8], and applications of existing techniques: graph transformations [41], CSP [66, 16], and others.

The AutoFocus [74] approach caters for the whole development process, from informal textual specification to code. This tool chain is similar to RoboTool in its goals. On the other hand, where AutoFocus targets embedded software with behavior defined by automata or functions, RoboTool focuses on robotic applications with behavior defined by state machines. Verification in AutoFocus uses theorem proving with Isabelle/HOL; similar goals are explored in [27] for RoboTool. 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.

In this book, other domain-specific approaches to modelling and verification of robotic systems are presented. In Chapter 1, we have an approach to deal with the challenges of creating product lines for robotic systems that uses a few domain-specific languages. RoboChart and RoboSim can be useful in that context to specify functionality, physical elements such as sensors, and (non-functional) time requirements. They would complement, rather than replace, the use of the domain-specific notations that deal with feature modelling and can support use of verification.

Model-based and component-based development is at the heart of the very ambitious RobMoSys framework presented in Chapter 2. That effort proposes a modelling approach for development of robotic software based on loosely connected components. It puts these forward as a basis for the collaborative construction of a base of reusable resources developed and used by a variety of stakeholders. For description of behaviour, they use data sheets, an abstraction mechanism . RoboChart (or RoboSim) could be used in conjunction with data sheets to provide a layer of formality while maintaining abstraction, something that cannot be achieved with code. A challenge is to map the concepts in RoboChart and RoboSim related to the abstraction of the robotic platform and of the environment, since these are not present in RobMoSys.

Chapter 8 also reports on a very successful approach based on a domain-specific language called GenoM. Like RoboStar notations, GenoM covers architectural design, concurrency, control of events, and verification by translation to existing formal notations and tools. 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, but indeed requiring extra modelling effort from users.

SafeRobots [63] is a general component-based framework in which components have a data-flow architecture. OCL is adopted for definition of properties, but specification of behavior is via code from libraries rather than state machines.

MontiArcAutomaton [67] comprises an ADL based on components and connectors that allows extension with component-behavior modelling languages. There is support for use and integration of multiple modelling languages and code generators, and for heterogeneous target platforms. RoboChart, as a language based on components and connectors, could be integrated with this setting.

FlexBE [70] is a behavior engine for ROS that enables human operators to specify and observe a robot's behavior and intervene at runtime by pausing or modifying it. Behaviors are specified by hierarchical state machines with actions implemented in Python. Similar, but more abstract, models can be developed in RoboChart for verification using shared variables and multiple state machines. FlexBE's tool does not support formal verification. Thus our approach is complementary.

MissionLab [19] supports end users in specifying behavior as mission plans in military applications. A wizard allows the definition of tasks, environment, the possibility of presence of enemies, and other parameters. Behavior is defined using simple state machines. Verification is not mentioned, but usability studies indicate ease of use. Such studies for RoboChart and RoboSim are not available yet.

SPECTRA allows modelling of behavior and environment assumptions using patterns [48] of LTL with efficient synthesis algorithms [7]. This requires discrete data type abstractions [50]. Time constraints cannot be directly specified, and so the

model needs to account for the target cyclic paradigm. Evidence [49] suggests that modelling of realistic environments and traceability are challenging.

Mauve [36] supports component-based models with interfaces defined by constants, operations, and ports, but not shared variables like RoboChart and RoboSim. Behavior can be defined just by code or simple textual state machines. Specifications, however, can use a contract language based on temporal logic and observation points of the code or machine. Code generation is for Orocos [73] platforms, with an optimized WCET analysis used to ensure schedulability. Time properties are derived from this analysis, rather than specified like in RoboChart.

The work in [25] is for an adaptive architecture; the verification enables identification of optimal configurations based on various proof techniques including model checking. Verification of behavioral properties, however, is not the focus.

Orccad [20, 38] is a notation for modelling, simulation, and programming, with verification (of timed properties) based on the translation of models into formal languages like for RoboChart and RoboSim. Orccad models are formed of tasks defined by control laws, combined by procedures defined by reactive programs. The combined use of RoboChart with control laws is addressed in [12], and that approach, based on modern co-simulation standards [26], can be used for RoboSim.

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 refinement. Comparable UPPAAL models require additional states, interleaved automata, and state invariants. Ongoing work is exploring a RoboChart and a RoboSim semantics using timed automata for property verification.

A real-time extension of UML statecharts called Hierarchical Timed Automata (HTA) is proposed in [15]. Roughly speaking, HTA is timed automata with hierarchy and history, but no operations. In [15], HTA are translated to timed automata for use with UPPAAL. Some of the restrictions on UML are similar to those of RoboChart, but some impose severe constraints on data, guards, and use of events. On the positive side, the target UPPAAL timed automata remain decidable.

UML [37] has a simple notion of time and little support to model timed properties. On the other hand, UML-MARTE [72] is a profile with support for logical, discrete and continuous time using clocks. Clock constraints may be specified using CSSL [47], and a constraint solver [17] can find solutions for deployment. Specification of deadlines and time budgets is through sequence and time diagrams. While it is possible to define budgets for a particular behavior, it is not possible to define timed constraints in terms of transitions and states. Limited support for UML-MARTE is available in the freely available Papyrus tool [32], an Eclipse plugin for UML.

UML-RT, an extension to UML, focuses on the architectural description of systems using the notions of capsules, ports, and protocols. Capsules encapsulate state machines, while communication between capsules is via ports and defined by protocols. A timing protocol can act as a timer by raising timeouts [71], but it is not obvious how deadlines can be specified directly on UML-RT state machines.

A CSP-based semantics for UML-RT is defined in [64], but it does not cover time. An extension to UML-RT [3] has a timed semantics defined using CSP+T [79], an extension of CSP that supports the timing of events. CSP+T is the inspiration for the work in [3], where annotations are added to record the occurrence time of events and constrain the occurrence time of subsequent events. Some RoboChart and RoboSim time primitives are similar, we have a richer set of primitives.

General-purpose simulation frameworks, such as Simulink and Stateflow [51, 52], 20-sim[4], and Modelica [31] are in widespread use. They can be used in robotics, but roboticists often describe state machines using an informal notation [59, 62, 75] before writing optimized code (in C or C++, for instance) for a simulator tailored for robotics. When there are complex control laws involved, the general simulators are useful. This is, however, not the case in many applications, and the flexibility of code-based simulations enable the development of more efficient simulations.

Compared to Stateflow, which is a statechart simulation language, RoboSim is, on one hand, much more restrictive, but, on the other hand, the cycle of a RoboSim model can be more flexibly defined. The occurrence of events or the structure of the machine does not implicitly define the behavior in each cycle. Moreover, in Stateflow, in each cycle, the machine is potentially executed several times, once for each event that has occurred. RoboSim adopts the approach of reactive simulators, where the machine is executed just once, when all events are normally considered. We expect, however, that it is possible to define a pattern for Stateflow models to allow their verification following the RoboSim approach.

Robotics simulators vary in their coding language. Webots [57] and CoppeliaSim (previously called V-REP) [68] provide (different) graphical interfaces. Webots adopts a human-readable customized notation; in CoppeliaSim, several general languages are available. ARGoS and Enki (home.gna.org/enki/) are programmed using different C++ libraries. The Microsoft Robotics Developer Studio (www.microsoft.com/robotics) has environments and platforms that can be programmed in VPL or C#. Player/Stage [33] provides a device server, and clients can be programmed using popular languages. MASON [45] and BREVE [39] adopt the agent paradigm; BREVE adopts a custom language or Python, and MASON, Java.

None of these simulators adopts a diagrammatic notation like RoboSim to specify simulation code. Moreover, there is no portability between them. The RoboStar vision is that a RoboSim model can be used for automatic generation of code for such simulators. We have illustrated the results for CoppeliaSim.

RobotML [18] is a domain-specific language for robotics based on UML. It has support for automatic generation of platform-independent code, but reasoning about non-functional properties is envisaged although not available yet.

In the same vein, rFSM [40] is a domain-specific language for simulation and deployment but does not have a formal semantics. There no support for analysis of models, either in isolation or in relation to designs, like we have for RoboSim.

[4] www.20sim.com/

RoboFlow [4] is a programming language with operational semantics. This formal semantics provides a clear way to define sound tools, but there is no support for reasoning about RoboFlow models in relation to designs.

ArmarX and Rafcon are programming languages for robotics based on state machines, but without formal semantics [76, 10]. Some of their restrictions, like the absence of inter-level transitions, are similar to those of RoboChart and RoboSim, and ultimately can facilitate the provisioning of reasoning facilities.

In summary, what is distinctive about the RoboStar technology are small and controlled domain-specific languages. The architectural pattern that they embed can guide roboticists in developing models; the same is not true of open and general languages. The restrictions of RoboChart and RoboSim simplify their semantics and facilitate verification. Beyond support for verifying desirable properties of individual models, we have a conformance notion for a simulation with respect to a more abstract design model. More than new notations, the RoboStar technology provides a modelling and verification approach for simulation of robotic applications that can be useful in the context of all notations based on state machines above.

A comprehensive survey on formal specification and verification in robotics [22, 44] highlights model checking as the most prominent verification approach in the literature. As illustrated here, RoboChart supports verification by model checking. Our long-term plan, however, is the use of theorem proving to deal with larger models. Ongoing and future work will explore combination of verification approaches.

8 Final considerations and future work

Current practice in robotics is normally based on standard state machines [18, 60, 10, 76], without formal semantics, to specify the robot controller only. The state machine that gives an abstract account of the robot controller guides the development of a simulation, but no rigorous connection between them is established. For implementation in a robotic platform, ad hoc adjustments are normally required to cater for the reality gap between the simulation and actual environment. Numerous iterations of (re)development and testing, tool dependency, and low-level programming are prevalent, with an impact on cost, maintainability, and reliability.

RoboStar technology addresses the issues of principled modelling, verification, sound simulation generation, and testing. With domain-specific languages, and support for automatic generation of artefacts, and verification, it enables significant advances in the practice of software engineering for robotics.

Reactive robotics simulators do not normally generate code specifically for deployment. Instead, simulation code is often reused after changes, because the API for simulation and for deployment are different, and simulation code is based on a cyclic executive. (This is a simple programming pattern for single-processor architectures; it cannot easily cope with multiple processors, heterogeneous architectures, and mixed criticality.) There is potential loss of properties observed via simulation: because of the possibility of changes introducing errors, and of the reality gap.

In future work, we will develop a domain-specific language for modelling deployment (layered) architectures and code, and a library of architectures. For each of them, we will define how to derive code automatically from a RoboChart or RoboSim model. Automation will ensure preservation of properties. The code will use multiple processors, and have components with differentiated levels of guarantee: hard results for the high-criticality, and probabilistic guarantees for the low-criticality components. Fault models will justify adequacy of the approach to fault tolerance. Monitors will enable update of the deployment, simulation, and design models.

RoboWorld provides the basis for further work on identification of additional environment concepts of relevance to particular areas of application and categories of robotic platforms. In Chapter 7, the authors quite rightly state that a mathematical model cannot capture the physical world. Our approach with RoboWorld is to capture the assumptions that are necessary to prove properties of interest. These assumptions are operational conditions that, currently, are, at best, left implicit.

CorteX, described in Chapter 10, provides support for principled programming. It is a middleware designed to deal with the maintainability challenges faced by large-scale long-running applications, typical of those in the nuclear industry. Code generation for CorteX from RoboChart or RoboSim models is a very interesting avenue for future work. CorteX is equipped with validation support based on testing, and complementarity of RoboStar technology is promising.

Our vision is a 21st-century toolbox for robot-controller developers. In this toolbox, a developer can find unambiguous diagrammatic notations to specify models for the environment, the robotic platform, and the controller. For commonly used environments and robotic platforms, the toolbox includes a range of ready-made models. Because these models are precise, there is no scope for misunderstanding and, most importantly, the toolbox includes techniques for desirable properties of the models: deadlock freedom, speed limits, and so on.

Since the technique for validation that robot controller developers favour nowadays is simulation, in the 21st-century toolbox, there are tools for automatic generation of these simulations. The ingenuity of the developer is now focused in the optimization of the simulation and of the associated deployed code. Because the languages used for simulation and programming are high-level, the results are tool independent, and can be deployed in a variety of robotic platforms.

With the 21st-century toolbox, the costly cycles of iterations of design and testing, with problems found very late, even just at deployment time, are reduced. Moreover, the developer can demonstrate that the controller produced satisfies essential properties. Software for mobile and autonomous robots is cheaper and more reliable.

Acknowledgements The B-ACS work has been done as part of the CAVlab project in 2017 - 2018. The team involved includes Dave Barnett, Servando German Serrano, Ujjar Bhandari, Nastaran Shatti, and Alan Peters. Zeyn Saigol is proposing and developing the B-ACS work as a suitable autonomy verification case study. All members of the RoboStar group (www.cs.york.ac.uk/robostar/) have contributed directly or indirectly to the vision described here. Our work is funded by the Royal Academy of Engineering under Grant No CiET1718/45, and by the UK EPSRC (Engineering and Physical Sciences Research Council) under Grants No EP/M025756/1 and EP/R025479/1.

References

1. T. Abdellatif, S. Bensalem, J. Combaz, L. deSilva, and F. Ingrand. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578, 2012.
2. M. M. Adams and P. B. Clayton. Cost-Effective Formal Verification for Control Systems. In K. Lau and R. Banach, editors, *ICFEM 2005: Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer-Verlag, 2005.
3. K. B. Akhlaki, M. I. C. Tunon, J. A. H. Terriza, and L. E. M. Morales. A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. *Science of Computer Programming*, 65(1):41–56, 2007.
4. S. Alexandrova, Z. Tatlock, and M. Cakmak. Roboflow: A flow-based visual programming language for mobile manipulation tasks. In *IEEE International Conference on Robotics and Automation*, pages 5537–5544, 2015.
5. W. Barnett. Architectural Data Modelling for Robotic Applications. Technical report, 2019.
6. G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *3rd International Conference on the Quantitative Evaluation of Systems*, pages 125–126. IEEE Computer Society, 2006.
7. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911 – 938, 2012. In Commemoration of Amir Pnueli.
8. M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML - Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.
9. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag, 2001.
10. S. G. Brunner, F. Steinmetz, R. Belder, and A. Domel. Rafcon: A graphical tool for engineering complex, robotic tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3283–3290, 2016.
11. G. Carvalho, A. L. C. Cavalcanti, and A. C. A. Sampaio. Modelling Timed Reactive Systems from Natural-Language Requirements. *Formal Aspects of Computing*, 28(5):725–765, 2016.
12. A. L. C. Cavalcanti, A. Miyazawa, R. Payne, and J. Woodcock. Sound simulation and co-simulation for robotics. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 173–194. Springer International Publishing, 2017.
13. A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.
14. D. Chen and C. Manning. A fast and accurate dependency parser using neural networks. In *Conference on Empirical Methods in Natural Language Processing*, pages 740–750. Association for Computational Linguistics, 2014.
15. A. David, M. O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, pages 218–232. Springer Berlin Heidelberg, 2002.
16. J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2-3):118–145, 2003.
17. J. DeAntoni and F. Mallet. *Objects, Models, Components, Patterns*, chapter TimeSquare: Treat Your Models with Logical Time, pages 34–41. Springer, 2012.
18. S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. *Simulation, Modeling, and Programming for Autonomous Robots*, chapter RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pages 149–160. Springer, 2012.
19. Y. Endo, D. C. MacKenzie, and R. C. Arkin. Usability evaluation of high-level user assistance for robot mission specification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 34(2):168–180, 2004.

20. B. Espiau, K. Kapellos, and M. Jourdan. *Formal Verification in Robotics: Why and How?*, pages 225–236. Springer London, 1996.
21. M. Esser and P. Struss. Obtaining Models for Test Generation from Natural-Language like Functional Specifications. In *International Workshop on Principles of Diagnosis*, pages 75–82, 2007.
22. M. Farrell, M. Luckcuck, and M. Fisher. Robotics and integrated formal methods: Necessity meets opportunity. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods*, volume 11023 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2018.
23. P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
24. M. S. Conserva Filho, R. Marinho, A. C. Mota, and J. C. P. Woodcock. Analysing robochart with probabilities. In T. Massoni and M. R. Mousavi, editors, *Formal Methods: Foundations and Applications*, pages 198–214. Springer, 2018.
25. F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 606–621. Springer-Verlag, 2009.
26. FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. https://www.fmi-standard.org, 2014.
27. S. Foster, J. Baxter, A. L. C. Cavalcanti, A. Miyazawa, and J. C. P. Woodcock. Automating Verification of State Machines with Reactive Designs and Isabelle/UTP. In K. Bae and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, pages 137–155, Cham, 2018. Springer.
28. S. Foster, A. L. C. Cavalcanti, S. Canham, J. C. P. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Theoretical Computer Science*, 802:105 – 140, 2020.
29. S. Foster, Y. Nemouchi, C. O'Halloran, K. Stephenson, and N. Tudor. Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In *8th International Conference on Formal Methods in Software Engineering*. ACM, 2020. To appear.
30. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal Methods and Software Engineering*, pages 383–399. Springer, 2016.
31. P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.
32. S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, chapter 19 Papyrus: A UML2 Tool for Domain-Specific Language Modeling, pages 361–368. Springer, 2010.
33. B. Gerkey, R. T. Vaughan, and H. Andrew. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *11th International Conference on Advanced Robotics*, pages 317–323, 2003.
34. S. German, A. Peters, D. Barnett, U. Bhandari, and N. Shatti. Connected and Autonomous Vehicles Laboratory (CAVLab) - An accessible facility for development and integration of CAV thecnologies. In *ITS World Congress*, 2018.
35. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.
36. N. Gobillot, C. Lesire, and D. Doose. A modeling framework for software architecture specification and validation. In D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, pages 303–314. Springer International Publishing, 2014.
37. Object Management Group. OMG Unified Modeling Language, 2015.
38. K. Kapellos, D. Simon, M. Jourdant, and B. Espiau. Task level specification and formal verification of robotics control systems: State of the art and case study. *International Journal of Systems Science*, 30(11):1227–1245, 1999.

39. J. Klein. BREVE: a 3D Environment for the Simulation of Decentralized Systems and Artificial Life. In *8th International Conference on Artificial Life*, pages 329–334. The MIT Press, 2003.
40. M. Klotzbucher and H. Bruyninckx. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *Journal of Software Engineering for Robotics*, 2(13):28–56, 2012.
41. S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In M. Butler, L. Petre, and K. SereKaisa, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.
42. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.
43. L. Lima, A. Miyazawa, A. L. C. Cavalcanti, M. Cornélio, J. Iyoda, A. C. A. Sampaio, R. Hains, A. Larkham, and V. Lewis. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, 16(3):1–28, 2017.
44. M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. Formal specification and verification of autonomous robotic systems: A survey. *CoRR*, abs/1807.00048, 2018.
45. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.
46. B. Luteberget, J. J. Camilleri, C. Johansen, and G. Schneider. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In A. Cimatti and M. Sirjani, editors, *Software Engineering and Formal Methods*, pages 87–103. Springer International Publishing, 2017.
47. F. Mallet. Clock constraint specification language: specifying clock constraints with UML/-MARTE. *Innovations in Systems and Software Engineering*, 4(3):309–314, 2008.
48. S. Maoz and J. O. Ringert. GR(1) Synthesis for LTL Specification Patterns. In *10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 96–106. Association for Computing Machinery, 2015.
49. S. Maoz and J. O. Ringert. Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In *4th Workshop on Synthesis*, 2015.
50. S. Maoz and J. O. Ringert. On the Software Engineering Challenges of Applying Reactive Synthesis to Robotics. In *1st International Workshop on Robotics Software Engineering*, pages 17–22. Association for Computing Machinery, 2018.
51. The MathWorks,Inc. *Simulink*. `www.mathworks.com/products/simulink`.
52. The MathWorks,Inc. *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.com/products`.
53. A. Miyazawa, P. Ribeiro, A. L. C. Cavalcanti, W. Li, J. Timmis, and J. C. P. Woodcock. RoboChart and RoboTool: Modelling, Verification and Simulation for Robotics. Technical report, University of York, Department of Computer Science, York, UK, 2020. Available at `www.cs.york.ac.uk/circus/RoboCalc/robosim/robosim-reference.pdf`.
54. A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.
55. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.
56. A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede. A survey on domain-specific modeling and languages in robotics. *Journal of Software Engineering for Robotics*, 7(1):75–99, 2016.
57. M. Olivier. Webots^{TM}: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.
58. OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012.
59. H. W. Park, A. Ramezani, and J. W. Grizzle. A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. *IEEE Transactions on Robotics*, 29(2):331–345, 2013.

60. I. Pembeci, H. Nilsson, and G. Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 168–179. ACM, 2002.
61. C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.
62. C. A. Rabbath. A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. *Journal of Intelligent & Robotic Systems*, 70(1):233–253, 2013.
63. A. Ramaswamy, B. Monsuez, and A. Tapus. Saferobots: A model-driven framework for developing robotic systems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1517–1524, 2014.
64. R. Ramos, A. C. A. Sampaio, and A. C. Mota. A Semantics for iUML-RT Active Classes via Mapping into *Circus*. In *Formal Methods for Open Object-based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 99–114, 2005.
65. Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, 2011.
66. H. Rasch and H. Wehrheim. Checking consistency in UML diagrams: Classes and state machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2003.
67. J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *Journal of Software Engineering for Robotics*, 6(1):33–57, 2015.
68. E. Rohmer, S. P. N. Singh, and M. Freese. V-rep: A versatile and scalable robot simulation framework. In *IEEE International Conference on Intelligent Robots and Systems*, volume 1, pages 1321–1326. IEEE, 2013.
69. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
70. P. Schillinger, S. Kohlbrecher, and O. von Stryk. Human-robot collaborative high-level control with application to rescue robotics. In *IEEE International Conference on Robotics and Automation*, pages 2796–2802, 2016.
71. B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.
72. B. Selic and S. Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers Inc., 2013.
73. P. Soetens and H. Bruyninckx. Realtime hybrid task-based control for robots and machine tools. In *2005 IEEE International Conference on Robotics and Automation*, pages 259–264, 2005.
74. M. Spichkova, F. Hölzl, and D. Trachtenherz. Verified system development with the autofocus tool chain. In *Workshop on Formal Methods in the Development of Software*, 2012.
75. T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. L. Grixa, F. Ruess, M. Suppa, and D. Burschka. Toward a Fully Autonomous UAV: Research Platform for Indoor and Outdoor Urban Search and Rescue. *IEEE Robotics Automation Magazine*, 19(3):46–56, 2012.
76. M. Wachter, S. Ottenhaus, M. Krohnert, , N. Vahrenkamp, and T. Asfour. The ArmarX Statechart Concept: Graphical Programing of Robot Behavior. *Frontiers in Robotics and AI*, 3:33, 2016.
77. C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal. Automatic generation of system test cases from use case specifications. In *International Symposium on Software Testing and Analysis*, pages 385–396. Association for Computing Machinery, 2015.
78. J. C. P. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.
79. J. J. Zic. Time-constrained Buffer Specifications in CSP + T and Timed CSP. *ACM Transactions on Programming Languages and Systems*, 16(6):1661–1674, 1994.