



Formal design, verification and implementation of robotic controller software via RoboChart and RoboTool

Wei Li^{1,2} · Pedro Ribeiro³ · Alvaro Miyazawa³ · Richard Redpath² · Ana Cavalcanti³ · Kieran Alden² · Jim Woodcock³ · Jon Timmis⁴

Received: 19 July 2022 / Accepted: 20 April 2024 / Published online: 5 July 2024
© The Author(s) 2024

Abstract

Current practice in simulation and implementation of robot controllers is usually undertaken with guidance from high-level design diagrams and pseudocode. Thus, no rigorous connection between the design and the development of a robot controller is established. This paper presents a framework for designing robotic controllers with support for automatic generation of executable code and automatic property checking. A state-machine based notation, RoboChart, and a tool (RoboTool) that implements the automatic generation of code and mathematical models from the designed controllers are presented. We demonstrate the application of RoboChart and its related tool through a case study of a robot performing an exploration task. The automatically generated code is platform independent and is used in both simulation and two different physical robotic platforms. Properties are formally checked against the mathematical models generated by RoboTool, and further validated in the actual simulations and physical experiments. The tool not only provides engineers with a way of designing robotic controllers formally but also paves the way for correct implementation of robotic systems.

Keywords Formal verification · Robotic controller · State machine · Domain specific language

1 Introduction

Many autonomous systems are deployed in critical applications. A key challenge in developing such systems is ensuring they operate as expected. It is often the case that the software for a robotic system is validated during, or after development, and typically experimentally. Even then, only a small subset of their possible behaviours can be validated, given the difficulties in testing such complex systems. It is very hard to ensure that exhaustive testing has been undertaken or that

adequate coverage with respect to testing has been achieved (Lewis, 2009).

Formal verification is the use of mathematical reasoning to ensure correct execution of computing systems and show that the system satisfies its desired properties or specifications (Ray, 2010). It has been used to verify a variety of robotic systems such as self-driving cars (Bringsjord & Sen, 2016), assistant robots (Webster et al., 2016), UAVs (Webster et al., 2011) and swarming robots (Rouff et al., 2007; Winfield et al., 2005).

Finite state machines are often adopted in the design of robot controllers (Park et al., 2013; Bjerknes & Winfield, 2010; Chen et al., 2015), which execute the program to operate a robot. Here, by controller we refer to the software that implements high-level decision making, reacting to inputs from sensors or embedded software, and producing outputs to be realised by actuators and or low-level control algorithms. In the design stage, often a state machine diagram guides the development of the software controller, and is validated using simulation, but no rigorous connection between the design and the implementation is established. This makes it hard to maintain the consistency between high-level controller designs and low-level implementations. In

✉ Wei Li
fd_liwei@fudan.edu.cn

Pedro Ribeiro
pedro.ribeiro@york.ac.uk

¹ Academy for Engineering and Technology, Fudan University, Shanghai 200043, China

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

³ Department of Computer Science, University of York, York YO10 5GH, UK

⁴ Department of Computer Science, Aberystwyth University, Aberystwyth SY23 3FL, UK

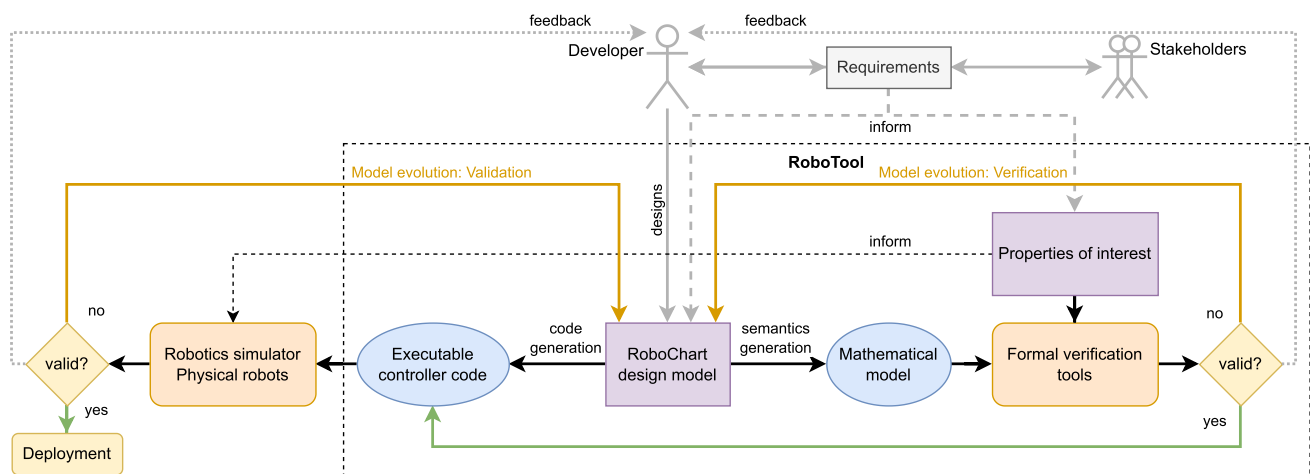


Fig. 1 A development workflow starting from a RoboChart design model and properties of interest, both informed by requirements. Artifacts are indicated by ovals: a mathematical model suitable for analysis using formal verification tools; and executable controller code suitable

for loading into a robotics simulator or deployment onto physical robots. Property verification, via formal approaches and simulation, guides the model evolution. The dashed box defines the scope of the activities carried out using RoboTool

order to build a rigorous connection between design and implementation of autonomous robotic systems, we explore the use of a top-down model-driven approach, focusing on high-level specification and automatic code generation from software design models (Ken, 2002) that are verified to meet functional requirements. We explore the role formal verification can play in the development of the software for a robotic controller at an early stage, even before simulation and deployment are considered, and whether a systematic approach to code generation from design models is feasible, so that a typical iterative development can be model-based, rather than code-centric. We propose an approach, and a tool, that practitioners can use for rapid iterative design supported by: (a) automated formal verification; (b) automatic code generation for simulation. The former allows a developer to get confidence at an early stage that a design is sound, while the latter can help eliminate errors that could otherwise be introduced in the process of manually implementing, and revising, simulation code.

To this end, we use RoboChart (Miyazawa et al., 2019), a state-machine based notation which supports design and verification of the software for robotic controllers. RoboChart takes inspiration from UML, a commonly used modelling language in software development (Bergenti & Poggi, 2000). It is specialised for robotics, and provides support for modelling the architecture of robotic systems, including facilities to model time and probabilities. By adopting the Z (Woodcock & Davies, 1996) type system and its mathematical toolkit, RoboChart supports rich data modelling primitives, such as, sets, abstract datatypes and mathematical functions, for modelling the software at a high-level of abstraction, which is key for exploiting automated verification tech-

niques. While the work we describe here concerns only the (discrete-timed) software and its execution, work is ongoing to complement, and connect, the software with models of physical robots (Miyazawa et al., 2020) and simulation scenarios. Distinctively from other modelling languages, RoboChart has a formal semantics tailored for automatic verification of (discrete) timed functional properties of the software, namely via model-checking, an approach that can provide counter-examples when a violation is identified.

Figure 1 outlines a development workflow using RoboChart that comprises design, formal verification, and simulation of robotic controllers. From a RoboChart design a mathematical model is automatically generated, which is used to assess conformance against properties of interest using formal verification tools. If the result is not satisfactory, then the model is evolved until all properties of interest are satisfied. Then, executable code can be automatically generated for simulation or deployment. Similarly, properties of interest can be assessed against observed behaviour in simulation and deployment. Likewise, if the results are unsatisfactory then the model is evolved, avoiding the need for manually modifying code, an endeavour that could lead to the introduction of unintended problems in an implementation. To support this workflow, we have developed RoboTool,¹ a set of Eclipse plugins that include a graphical editor for constructing RoboChart design models, textual editors for stating properties of a design, including absence of livelocks and deadlocks, but also application-specific behavioural properties. Importantly, RoboTool provides integration with formal verification tools, for example, making reports of verification

¹ <https://robostar.cs.york.ac.uk/robotool/>.

results directly available within the tool. This facilitates the construction and validation of RoboChart models, as well as the automatic generation of simulations and mathematical models, contributing to the implementation of safe robotic controllers.

In Li et al. (2016), we demonstrate the usage of a preliminary version of RoboChart together with the automatic generation of C++ code for a small subset of RoboChart. The generated code is validated using two case studies of multi-robot systems in simulation. The simulation adopts the model-viewer-controller (MVC) pattern (Gamma et al., 1994). Here, we present a richer version of RoboChart, covering a broader set of constructs, such as composite states, and its event-based paradigm that admits both synchronous and asynchronous communications. Moreover, we propose a new software architecture that is closer to that of the formal semantics. This closer correspondence facilitates the definition of compositional rules mapping elements of a RoboChart design model to code, and paves the way for verifying the correctness of the code generator against the (formal) mathematical model. We also report on evidence obtained by model-checking that traces collected from executing generated simulations are in conformance with the formal semantics. The details of the formal semantics and automatic property checking using the related tools are discussed in Miyazawa et al. (2018, 2019), while in this paper we emphasise the simulation and deployment of robotic applications.

In this paper, we demonstrate the application of RoboChart and its related tool through a case study of a robot performing an exploration task. The robot shares many similarities with other robotic systems. For example, the robot needs to explore the environment autonomously while avoiding collisions. It also needs to meet certain critical properties such as being responsive to a command.

In summary, the main contributions of this paper are as follows:

- Illustrates the role automatic formal verification techniques can play in the iterative model-based development of robot controllers.
- Proposes a novel software architecture as a target for automatic generation of object-oriented code from RoboChart design models. In contrast with previous work, it covers a richer set of constructs and is closer to the formal semantics.
- Reports on evidence that our code generation approach preserves functional behaviour from design through simulation and deployment.

The remainder of the paper is organised as follows. Section 2 presents related work. Section 3 introduces RoboChart, its overall structure and a simple example showing key

features of the language. Section 4 presents the software architecture involved in executing a RoboChart model, showing how the elements in a RoboChart model are mapped to the different elements of the software architecture. Section 5 presents the RoboChart model for the case study of the exploration robot. In Sect. 6, we discuss the verification of properties of the robotic system and its validation via simulation and hardware experiments. Section 7 concludes and discusses future work.

2 Related work

RoboChart is a domain-specific language (DSL), dedicated to robotic applications. It supports a model-driven engineering approach, which focuses on creating and exploiting models to provide high-level descriptions of a system. It also provides automatic code generation for simulation and deployment. DSLs have been widely developed in robotics either for general robotic platforms (Schlegel et al., 2009; Dhouib et al., 2012; Klotzbucher & Bruyninckx, 2012) or a particular robotic platform (Schultz et al., 2007; Baumgartl et al., 2013). A survey on DSL for robotics is available in Nordmann (2016). RoboChart is distinctive to these works in its support for mathematical proof of (timed) behavioural properties of design models.

Work in Dhouib et al. (2012) presents a general robotics DSL, RobotML. The aim of RobotML is to support modelling of robotic systems and automatic generation of platform-independent code. RobotML was defined as a UML profile, with the main advantage of describing different components (such as robotic architecture and environment) of a robotic system at a functional level along with their communication protocol. RobotML can generate code for different target platforms, such as OROCOS,² that provides a complete library for robotic applications, and MORSE.³

Work by Schlegel et al. (2009) describes a DSL that also adopts a UML-based component framework. Components are realised through a set of communication patterns such as *request/response* and *publish/subscribe*, which define the visibility of components. A tiered approach, covering abstract and concrete software component models is further explored in Schlegel et al. (2021). Behavioural aspects are characterised by blocks, whose semantics is deferred to outside documentation or reference implementations. Tool support is discussed in Stampfer et al. (2016). In Baumgartl et al. (2013), a DSL is developed for robot pick-and-place applications. The authors demonstrate their approach through a grasp task in which the robot needs to pick and grasp a particular object in an open environment. The key components

² <https://www.oroocos.org/ocl>.

³ <https://morse-simulator.github.io/>.

such as pick and path planner are specified in the language. Additionally, it provides partial support for modelling robot hardware. These works only provide a modelling framework while no verification is performed.

In Schultz et al. (2007), a language is developed to program self-assembling robots. The authors propose a role-based language that allows the programmer to define the behavioural roles of each component independently from the concrete physical structure of the robots. A language for verifying multi-robot systems is developed in Harbin et al. (2021), which supports the systematic robustness analysis of multi-robot systems in simulation. However, automatic executive code is not generated while it can generate middleware for Gazebo/ROS.

In these works, the semantics of the domain specific languages are not formally specified, and there is no support for formal verification of a model designed using such languages. RoboChart is a DSL for designing and verifying robotic controllers based on state-machines, with a well defined semantics to support generation of mathematical models for formal verification as well as generation of platform-independent code for simulation and deployment.

It should be noted, however, that formal verification is supported in other languages. A comprehensive survey on formal specification and verification of autonomous robotic systems is presented in Luckcuck et al. (2019). It highlights the challenges and importance of integrating formal methods for verification. The most prominent technique employed is model checking, namely using temporal logic and state transition systems. For example, verification by model checking is also considered for GenoM Foughali et al. (2016). Models are translated to a mathematical notation called Fiacre, which is close to the input language of the Petri Net model checker TINA. Verification focuses on schedulability. Another approach uses a mathematical notation, BIP, for deadlock checking and schedulability analysis (Abdellatif et al., 2012). As opposed to RoboChart, GenoM is an executable language; models, for example, include C code.

The language in Fleurey and Solberg (2009) is used to model the adaptive architecture of an exploration robot. Automatic generation of mathematical definitions supports the use of model checking and other proof techniques to identify optimal configurations. Unlike RoboChart, behavioural properties are not the focus. What we present here is an approach for modelling, simulation and property checking of robotic systems.

There are other works focusing on developing verification techniques for certain specific components of robotic systems. In Kortik and Shastha (2021), a novel representation and verification technique for ROS components using a linear logic theorem prover is presented. The system model can be either statically extracted by a ROS based static analysis framework or dynamically extracted once all system

components are running. Similarly, Santos et al. (2021) also extracts models from ROS executable code for the purpose of static analysis and model checking of (untimed) behavioural properties, defined via a DSL (Carvalho et al., 2020), over message-passing between nodes given loose behavioural specifications that must be manually constructed for each node. In contrast, in our framework we adopt a model-driven approach that starts from a high-level design model rather than from code.

In Cardoso et al. (2020), components of a NASA rover simulation are verified using complementary methods: the high-level Beliefs-Desires-Intention (BDI) agent is verified using the Agent Java Path Finder (AJPF) model checker (Bordini et al., 2008); the environment interface is verified using Dafny (Leino, 2010); and a model of action nodes, following the publish-subscribe paradigm of ROS, is verified using the FDR model checker (Gibson-Robinson et al., 2014). The emphasis is on component verification with formal models guiding the generation of runtime monitors. Modelling of architectural patterns in RoboChart can be tackled using the companion DSL called RoboArch (Barnett et al., 2022), which has been applied in conjunction with CorTEX (Caliskanelli et al., 2021), a software framework targeted at the nuclear industry.

In Colledanchise et al. (2021) Behavior Trees are formalised via program graphs, akin to extended finite state machines, but without considering time primitives directly in BTs. Instead, a master tick generator is considered for runtime verification of properties, such as deadlines, expressed in a fragment of Timed Propositional Temporal Logic (TPTL) (Alur & Henzinger, 1994). The notion of a clock generator is similar in nature to the approach adopted in UML-RT (Selic, 1998) with timers. UML (OMG Unified Modeling Language, 2015) has a simple notion of time that is not adequate for modelling timed properties. Another profile, UML-MARTE (Selic & Grard, 2013) supports the specification of logical, discrete and continuous time through the notion of clocks. Support for specification of deadlines and duration of behaviours is largely focused on particular behavioural instances via sequence and time diagrams. RoboChart, on the other hand, provides facilities to directly specify budgets and deadlines over transitions and actions.

In Bourbough et al. (2021) an assurance case is developed for an inspection rover, using a combination of models written in AADL (Feiler & Gluch, 2012), Simulink and Event-B (Abrial, 2010). Functional requirements are stated using FRET (Giannakopoulou et al., 2020), with semantics given in Linear Temporal Logic (LTL) for analysis of Lustre (Caspi et al., 1987) models generated from Simulink via CoCoSim (Bourbough et al., 2020), that supports code generation from discrete dataflow models. Kind2 (Champion et al., 2016) is used to verify system-level properties, while some components are verified using Event-B. In contrast, RoboChart

adopts a self-contained, hierarchical, state machine-based software component model. It includes support for synchronous and asynchronous communications, a well-defined action language, preconditions and postconditions, and time primitives encompassing budgets and deadlines, and is tailored for verification using model checkers (Miyazawa et al., 2019; Ye et al., 2022) and theorem provers (Foster et al., 2018). Related work reported in (Murray et al., 2022) has exploited the combination of RoboChart and Simulink for co-verification of system-level properties of an ABB paint robot high-voltage controller, while other case studies are reported in Cavalcanti et al. (2018). Here, we focus on model-based development guided by verification, and code generation for simulation and deployment from RoboChart design models.

3 RoboChart

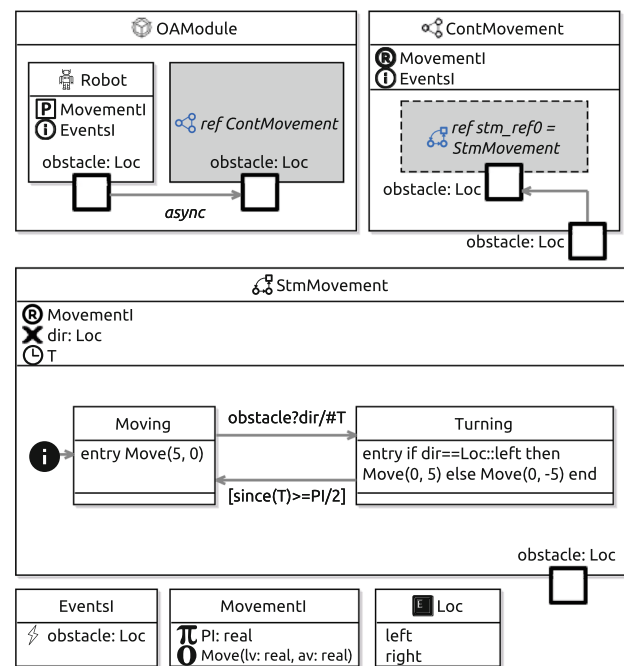
In this section, we first present a simple example, illustrating how a RoboChart model is constructed. We then present the metamodel of RoboChart and its key constructs.

3.1 Obstacle avoidance example

A simple example of a RoboChart design model for a ground robot that can sense obstacles and perform turns is shown in Fig. 2. These capabilities are captured in RoboChart by a robotic platform (indicated by Ⓜ), that is a record of the services (events, operations and variables) available to the software. In this case, the platform Robot provides (indicated by Ⓟ) an interface MovementI, with a constant PI (indicated by π) and an operation Move, that takes a linear and angular velocity, and uses (indicated by Ⓢ) an interface EventsI with an event obstacle (⚡) that carries a value of type Loc to indicate the position (left or right) of an obstacle relative to the robot.

The top-level element of the model is a module called OAModule, that completely defines the behaviour of the software by associating the platform Robot with a software controller ConstMovement, that we discuss next. They are connected via the event obstacle and communicate asynchronously, as indicated by the keyword *async*. Importantly, the interface MovementI required by the controller (indicated by Ⓢ) is provided by the platform. Interfaces enable the definition of self-contained components, while RoboTool uses this information to check that all services required by a component, such as, operations and variables, are provided in the context in which they are used.

The controller ContMovement contains a reference to the state machine StmMovement, that we discuss next. The state machine specifies the actual behaviour of the software, such that by default the robot moves forward until an obstacle is detected, and then either turns left or right. It requires the



Legend:

- Ⓢ Used interface Ⓜ Module X Variable
- Ⓢ Required interface Ⓜ Controller π Constant
- Ⓟ Provided interface Ⓜ Statemachine ⚡ Event

Fig. 2 RoboChart design model for a simple obstacle avoidance robot

interface MovementI, and declares a variable *dir* (indicated by X) of type Loc and a clock *T* (Ⓢ). It contains two states (Moving and Turning), and an initial junction (black circle symbol Ⓢ) with a transition leading to state Moving, whose entry action calls the operation Move with a linear velocity to move the robot forward. If an obstacle is found, the clock is reset (via #T) and state Turning is entered. In Turning, the robot starts turning with an angular velocity based on the location of the obstacle by calling the operation Move. Then, if the time elapsed since the clock was reset has passed the threshold $\text{PI}/2$, it transitions to state Moving again.

3.2 Metamodel of RoboChart

As shown in Fig. 3, a RoboChart model can contain packages, modules, robotic platforms, controllers, state machines, type declarations, and interfaces. For a complete account of the metamodel of RoboChart, see (Miyazawa et al., 2018, 2019).

A RoboChart package (RCPackage) groups components used for designing a robotic system. It is similar in nature to a package in Java or other programming languages and provides scope for the model: packages can import other packages. If no package is declared for a model, then global visibility is assumed. As shown in Fig. 3, a RCPackage can

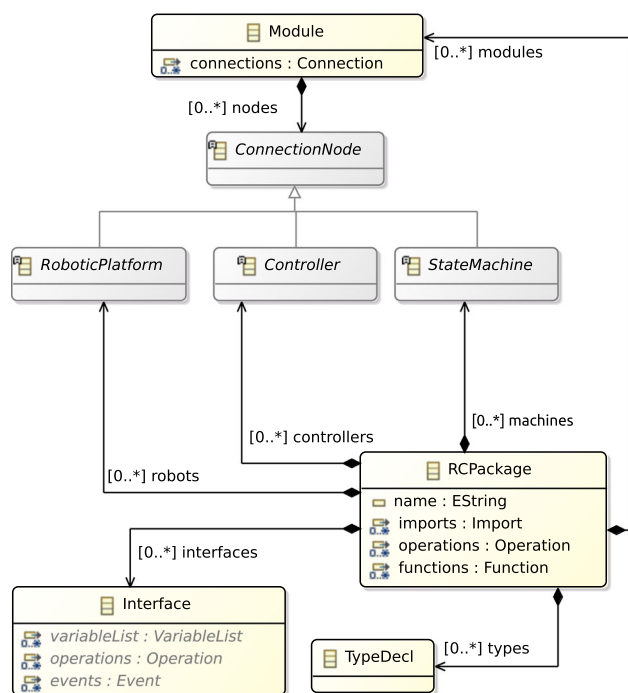


Fig. 3 RoboChart metamodel

include declarations of modules, robotic platforms, controllers, state machines, etc.

A **Module** in RoboChart contains a complete account of a robotic control system. It associates a **RoboticPlatform** with particular **Controller(s)** to specify the behaviours of a robot. The **RoboticPlatform** or **Controller** can be defined inside a **Module**, or outside, in which case it is associated to a **Module** via references. A **RoboticPlatform** provides an abstract description of the hardware in terms of its events, variables and operations. For example, in the example in Fig. 2, the module **OAModule** is specified by a robotic platform that defines an event **obstacle** and provides an interface **MoveIntl** declaring an operation **Move**. **Robotic Platform** and **Controller(s)** interact with each other using events or shared variables. An event may have a type, if it represents an interaction with a sensor or actuator that communicates values. A **Controller** is composed of one or more (potentially interacting) state machines.

A **RoboChart StateMachine**, similarly to UML statecharts, includes (initial) junctions, states, and transitions. Unlike UML, however, there are no parallel regions or inter-level transitions, that can make compositional reasoning challenging. A state can have entry, during, and exit actions. The entry action of a state is executed when the state is being entered, and is followed by the execution of the during action while that state is active. When a state is exited, its exit action is executed. A transition can be triggered by an event guarded by a condition. If an action is associated with the transition, this action is also executed before the state machine enters

the target state or junction. A junction is a decision point, and at least one of its transitions should be enabled at any time. A state can be composite: it contains other states that define behaviour while in that state. In this way, states can be nested inside states and nesting can be arbitrarily deep.

Operations and Events can be grouped in an **Interface** in order to facilitate the reuse of common elements. A state machine can require an interface in order to use the elements grouped in that interface. An operation can either be declared in an interface without implementation (in this case, only the operation signature is provided) or defined by the user using a state machine in a package. In the former case, the user needs to define an implementation for the operation in simulation and deployment. For the latter, the code of the operation is automatically generated. This facilitates modular design.

Mathematical Functions can be defined in RoboChart. The aim of using a function is to perform some necessary calculation, for example, calculating the value of an angle that the robot needs to turn when detecting an obstacle. It also provides a common functionality of which different components can make use. A function can be specified by a precondition and a postcondition.

VariableList is composed of **Variable(s)**, which can be defined using pre-defined types or user-defined types. Primitive types include common types found in programming languages such as `int` and `real`. RoboChart adopts the Z (Woodcock & Davies, 1996) type-system and its mathematical toolkit, so it supports rich types such as sets, enumerations, and data types.

RoboChart also includes time constraints. A clock can be defined inside a state machine to record the passage of time. For example, the primitive `since(T)` yields the time elapsed since the most recent clock reset `#T`. For full details of RoboChart, refer to (Miyazawa et al., 2018).

4 RoboChart software

This section presents the RoboChart software architecture and how a model is executed.

4.1 Software architecture

The RoboChart notation is implemented in RoboTool as a collection of integrated Eclipse plug-ins using the Eclipse Modelling Framework (EMF), Xtext⁴ and Sirius.⁵ The RoboChart metamodel is implemented using EMF. Xtext, a framework for developing domain-specific languages, is used to define a textual language, suitable as a human-readable

⁴ <https://www.eclipse.org/Xtext/>.

⁵ <https://www.eclipse.org/sirius/>.

serialisation of RoboChart models, which are otherwise usually created via a graphical editor implemented using Sirius. The tool implements validation rules, as Xtext check methods, that are automatically checked as a model is being constructed. This ensures models are typed correctly and well-formed. In addition to enforcing scoping rules, the tool also checks, for example, that all operations required by a component are provided in context where that component is used.

For code generation from RoboChart, we use Xtend,⁶ a flexible dialect of Java, that provides a rich templating language suitable for defining methods that map RoboChart constructs to code in a target programming language. This approach has previously been used (Miyazawa et al., 2017) to implement the functions that define the formal semantics of RoboChart models, so that, for instance, the methods can be visually inspected for correspondence with the mathematics.

We adopt object-oriented programming in the software development, as this paradigm supports modular design of the RoboChart software in a structured way and facilitates the reusability of the generated code. Figure 4 shows a class diagram that depicts the RoboChart software architecture.

The architecture is formed by classes that implement key features of the RoboChart language. The two classes *HardwareComponent* and *Interface*, at the top of Fig. 4, in purple colour, need to be extended to create a robotic application. The *Module* fully defines the behaviours of the robot and the assumptions about the *Robotic Platform*. Information about the environment is obtained via the events in the robotic platform, which must be supplied as platform dependent code. We adopt the Sense→Plan→Act paradigm in robotics. In the RoboChart software architecture this is realised via methods *Sense*→*Execute*⁷→*Actuate*, that is, the robot senses the environment, executes the controllers, and then performs actions. The *Sense* and *Actuate* methods are grouped into the *Robotic Platform* class, and the *Execute* method is contained in the *Module* class.

The *HardwareComponent* class, which includes the methods *Sense* and *Actuate*, provides an abstraction of the robot hardware (sensors and actuators). The events and operations are declared in the *Interface* class, which inherits from the class *HardwareComponent*. The *Robotic Platform* class inherits the *Interface* classes.

The *Module* class contains an instance of the *Robotic Platform* class and *Controller* classes. The latter holds instances of the *State Machine* classes where the decision-making mechanism of the robot is defined. Note that currently we only support the simulation of one controller and one state

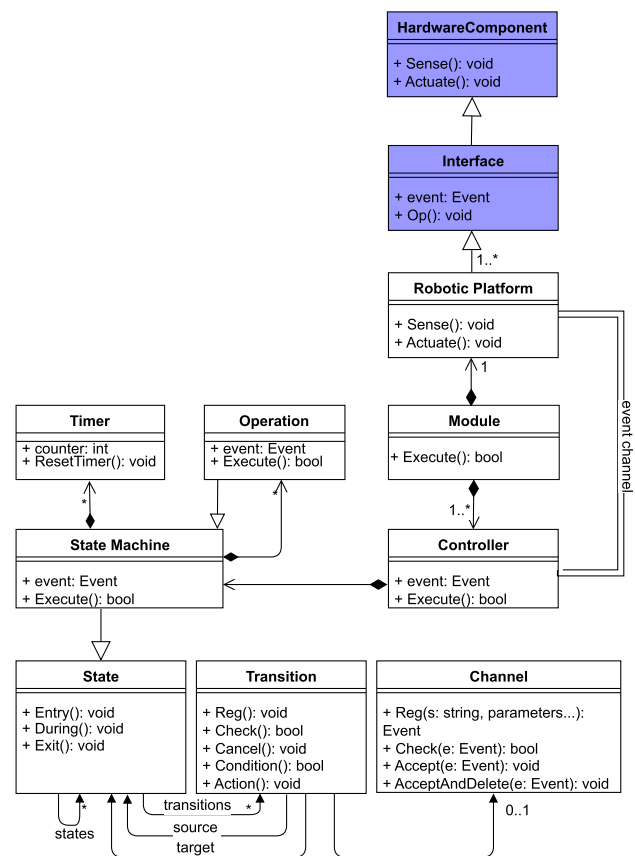


Fig. 4 Class diagram of the RoboChart software architecture used to implement a design model. Classes *HardwareComponent* and *Interface* need to be extended to create an application, while the others are fully generated

machine; however, states machines can have composite states or call operations whose behaviour is defined by another state machine.

The communication between the robotic platform and controllers is via events implemented by an event channel. This is inspired by the communication mechanism of the process algebra Communicating Sequential Processes (CSP) (Roscoe, 2011), which is the basis of the formal semantics of RoboChart (Miyazawa et al., 2019). In the architecture, the communication between different components, namely a *Controller*, including its *State Machine*, and a *Robotic Platform*, is via message passing. A message can be sent over a channel, and another component is able to receive the message sent over a channel to which it has a reference. The motivation for using channels is to facilitate the proof of correctness of the code. A channel acts as an intermediate buffer for bridging the communication of two components, that is, the *sender* and *receiver* are not aware of each other. Each sender only needs to send information into the channel via events and the *Channel* class is responsible for handling

⁶ <https://www.eclipse.org/xtend/>.

⁷ This terminology is chosen to reflect the execution of a wide range of state machines, which may not necessarily implement a planner in the traditional sense.

the asynchronous or synchronous (blocking) communication between the receiver and sender.

Concretely, in RoboTool the software architecture is generated from a RoboChart design model as a C++ API, with concrete code implementing the classes of Fig. 4, such as *Channel*, *State*, and *Event* classes.

In the following, we describe how elements (in bold) of the RoboChart metamodel are mapped to classes of the software architecture.

Module: For every Module in a RoboChart model, a *Module* class is defined that contains instances of the classes that implement its robotic platform, controllers and events. It has an *Execute* method that is used for driving the simulation.

Interface: The *Interface* class contains the operations as methods. It has references to the *Event* class. Variables are generated as attributes.

Robotic Platform: The *RoboticPlatform* class inherits all of the provided and defined *Interface* classes. If the events and variables are declared inside the robotic platform, they are generated as attributes of the *RoboticPlatform* class with types that implement their RoboChart counterparts. It contains constructor and destructor methods, and *Sense* and *Actuate* methods. We observe that the declaration of operations is grouped in the *Interface* class.

Controller: The *Controller* class acts as an intermediate mechanism that connects the robotic platform with the state machine via the event channel. Therefore, this class holds a reference to the robotic platform and state machine, and all of the events defined. Further work will deal with parallel simulations involving several controllers, potentially with several machines.

State Machine: In the implementation, we treat a state machine as a composite state without any outgoing transitions. Therefore, the class *StateMachine* inherits from the *State* class, which can have sub-states. The *State* class also has several instances of the *Transition* class, each of which holds a reference to a source and a target *State* class.

Operation: Each operation defined by a state machine is generated as an *Operation* class, which can be referred by a *State Machine* class. If no definition is given, the operation is generated as a method that is to be implemented by the user.

Event: For a defined event in RoboChart, a corresponding event channel class is declared. This class is then instantiated inside the *Module* class. The name of each defined event needs to be different, which is ensured by the well-formedness rules of RoboChart as implemented by RoboTool.

Clock: If a clock is defined in a state machine, a *Timer* class is generated. It has an attribute *counter*, indicating the elapsed time, and methods such as *IncreaseCounter* and *ResetTimer*. Note that only a single *Timer* class is generated, for each clock defined there will be a corresponding instance of the *Timer* class. The timer is used as a service of

Algorithm 1 Software Architecture Execution

```

1: initialise module, robotic platform, controller
2: and state machine
3: repeat (for each control cycle)
4:   Sense()
5:   while (Execute(s)) do
6:     end while
7:   Actuate()
8: until (program terminates)

```

the state machine, which means the state machine can access the *counter* of a *Timer*.

Function: A function in RoboChart is generated as a plain function, which can be invoked by different classes.

TypeDecl: A primitive data type is directly mapped into a primitive data type in the target programming language. For example, the primitive type *real* corresponds to *double* in the C++ implementation of the software architecture, whereas a record type is generated as a pre-defined class that has all of the fields of that type.

4.2 RoboChart software execution

In this section, we describe how the software architecture shown in Fig. 4 is implemented.

Algorithm 1 shows the execution flow of the software architecture. Before executing the control loop (line 3), the classes of *Module*, *Robotic Platform*, *Controller* and *State Machine*, as depicted in Fig. 4, are instantiated and initialised (lines 1–2). As already mentioned, overall, the implementation of the architecture follows the pattern *Sense*→*Execute*→*Actuate*, which is repeated (lines 4–7) in each control cycle. In *Sense* (line 4), the robot senses the environment, and registers any detected events in the channel. In *Execute* (line 5), the state machine is executed repetitively until no more transitions are enabled (i.e., this method returns false). In *Actuate* (line 7), the parameters of actuators are set. In every control cycle of the state machine, all of the available transitions of the state machine's current active state (including its sub-states) are checked. Note that if a transition is taken, the transitions of the target (active) state will be checked. The execution of the state machine proceeds to the next control cycle only if there are no enabled transitions for the active state.

Algorithm 2 shows how the state machine is executed in each control cycle. Execution of child states is performed recursively. A state has four possible statuses: *s_enter*, *s_active*, *s_exit* and *s_inactive*. States are initially set as *s_inactive*, whereas a state machine's status is *s_active*. We describe each of the cases as follows.

When the state is in the *s_enter* status (line 2), its entry action is executed (line 3). If the state has any child, its child's initial state (*s.InitialChild* on line 4) is set into *s_enter* status,

Algorithm 2 Execute(s)

```

1: switch s.status do
2:   case s_enter
3:     Execute entry action of s
4:     SetStatusEnter(s.InitialChild)
5:     Execute(s.InitialChild)
6:     SetStatusActive(s)
7:     return true
8:   case s_active
9:     while (TryExecuteChildStates(s)) do
10:      end while
11:     if TryTransitions(s) == false
12:       Execute during action of s
13:       return false
14:     end if
15:     return true
16:   case s_exit
17:     SetStatusExit(s.ActiveChild)
18:     Execute (s.ActiveChild)
19:     Execute exit actions of s
20:     SetStatusInactive(s)
21:     return false;
22: end switch

```

and it is executed (line 5). After that, the current status is set into *s_active* via the method *SetStatusActive* (line 6).

If the state is in the *s_active* status (line 8), it first tries to execute its child states using the method *TryExecuteChildStates* (line 9). If any transitions of the child state are enabled, *TryExecuteChildStates* is executed again. After that, it tries its own transitions (line 11). If no transition is enabled, that is *TryTransitions* returns false, the during action of that state is executed (line 12).

Finally, when the state is in the *s_exit* status (line 16), its active child (*s.ActiveChild*) if any, is set into *s_exit* status (line 17), followed by execution of the child, and then execution of its exit action (line 19). After that, the current state is set into *s_inactive* status (line 20). Note that for the current framework, we assume that the entry, exit and transition actions terminate and fit within a single step of the simulation.

Algorithm 3 shows how the transitions of an active state are executed. The *Transition* class has two methods: *Check* and *Condition*. Every time a *Check* method is called (line 2), it invokes the *Check* method of the *Channel* class. This method checks whether the triggering event for the transition has occurred or not. The *Condition* method is used to check whether the expression of the guard of the transition is *true* or *false*. Only when both the triggering event occurs and the condition is met, the transition is enabled. After the transition is taken, the status of the source state is set to *s_exit* and is executed (lines 3–4), followed by the *Action* method of the transition itself (line 5). The status of the target state (*t.target*) is set to *s_enter* (line 6).

Algorithm 3 TryTransitions(s)

```

1: for each transition t ∈  $\mathcal{T}(s)$  do
2:   if t.Check() and t.Condition():
3:     SetStatusExit(t.source)
4:     Execute(t.source)
5:     Action(t)
6:     SetStatusEnter(t.target)
7:     return true
8:   end if
9: end for
10: return false

```

5 Modelling an exploration robot in RoboChart

In this section, we explore the usage of RoboChart through modelling an exploration robot. It can be seen as an extension of the obstacle avoidance example presented earlier in Sect. 3.1, whereby besides exploring the environment autonomously while avoiding obstacles, it needs to satisfy additional functional requirements. Certain safety properties should be established before the robot is deployed. For example, one such property is that the robot should stop immediately if a ‘reset’ command is detected. If the battery level of the robot is low, it is required to return to its charging station.

The requirements for the designed robotic system are as follows:

- R1: If a command is detected, then the robot should stop.
- R2: Once the charging station is found, the robot should flash its LEDs and stop. After that, the robot can still accept a command.
- R3: The robot should avoid obstacles, or reverse, when an obstacle is detected, while no command is detected. Odometer readings can occur internally.
- R4: After an obstacle is detected, an odometer is read immediately.

In order to meet the requirements mentioned above, we designed a RoboChart model. This model includes a robotic platform *VCRobot* and a controller *VCController*. The controller communicates with the robotic platform via events. The controller’s behaviour is defined by a state machine *Movement*. The state machine models the robot behaviour, which performs searching by default when no ‘command’ is detected. While the robot is searching, it performs exploration while trying to avoid the obstacles and once the battery is low, the robot moves towards the goal location (charging station).

Fig. 5 The RoboChart model including definition of module (VModule), robotic platform (VCRobot), interfaces (Events, OperationsSig and OperationsDef) and the enumerated type Loc

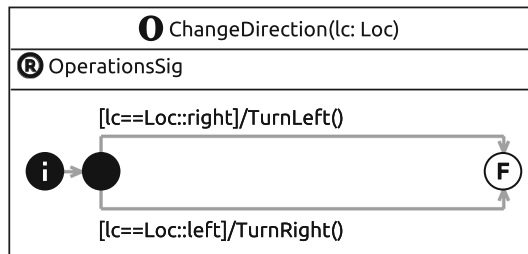
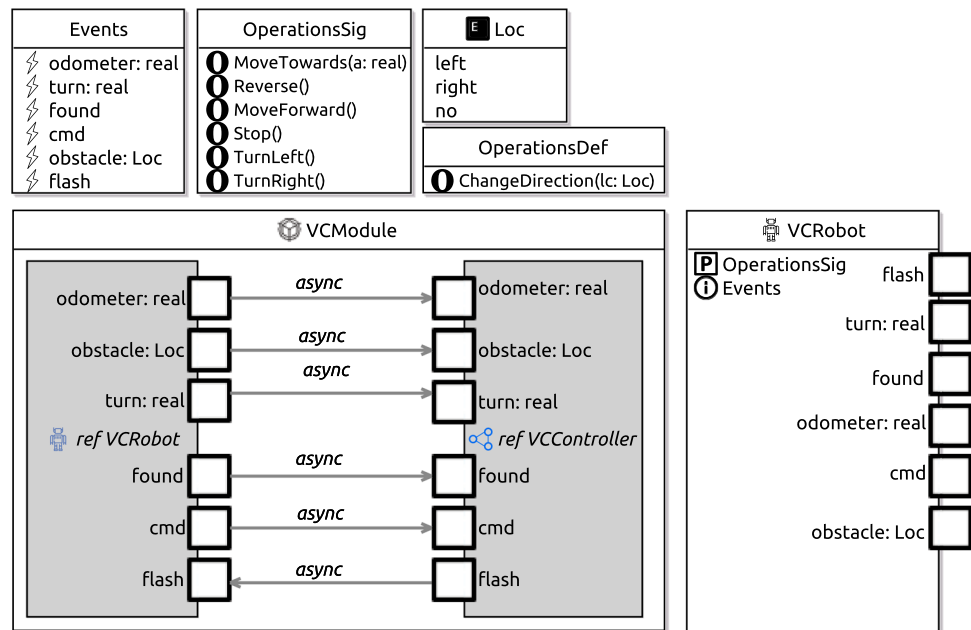


Fig. 6 The *ChangeDirection* operation defined in RoboChart. Its behaviour is defined by a state machine, which calls *TurnLeft* or *TurnRight* depending on the value of *lc* and then terminates

As shown in Fig. 5, a module called *VModule* is defined. It has a reference to a robotic platform, *VCRobot* and its controller, *VController*. Events used between *VCRobot* and *VController* are grouped in an interface, *Events*. The events are connected in the module, indicating the flow of information between components. For example, the real value associated with the event *turn* is always passed from the *VCRobot* to the *VController*. In order to get the odometer reading from the robot, an odometer event of type *real*, is defined. The flash event is used by the controller to cause the robot's LEDs to flash. In RoboChart, the communication between robotic platform and controller is always asynchronous (marked by the symbol *async*). The operations that the robotic platform provides are declared in the interface *OperationsSig*. These operations are assumed to terminate and take negligible time.

The operation *ChangeDirection* in the *OperationsDef* interface, however, is a software operation defined using a state machine (shown in Fig. 6). It calls *TurnLeft* or *Turn*

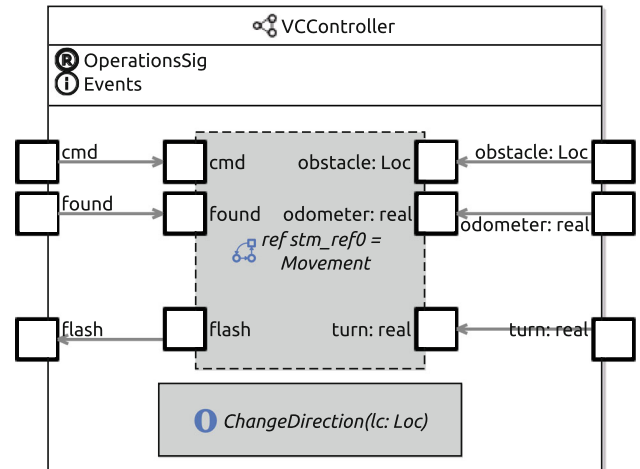


Fig. 7 The *VController* defined in RoboChart. It refers to a state machine *Movement* and defines its communication with the controller via events. It also requires the platform operations in *OperationSig* and brings into context the definition of the software operation *ChangeDirection*

Right depending on the value of parameter *lc* and then terminates.

Figure 7 shows the robotic controller *VController*. It requires the interface *OperationSig*, brings into context the operation *ChangeDirection* (required by *Movement*), and uses the events in interface *Events* for interaction with the state machine *Movement* that defines the decision-making mechanism controlling the robot directly. Note that the controller can refer to one or several (potentially interacting) state machines. In this case study, only one state machine is used. The event information is passed between the robotic platform

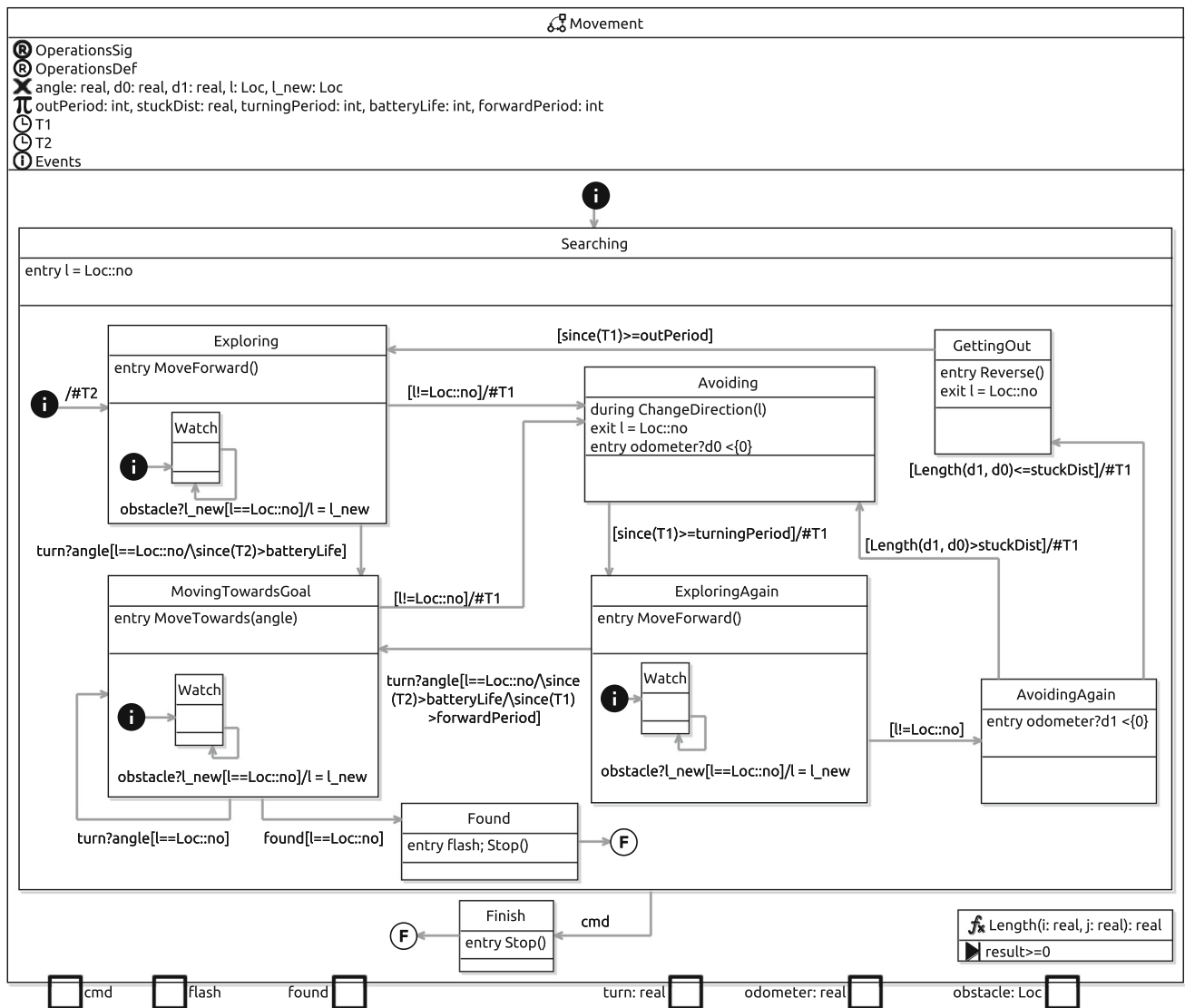


Fig. 8 The Movement state machine defined in RoboChart. It defines the behavior of the robot

Table 1 Summary of CSP operators

Symbol	Name	Description
<i>SKIP</i>	Skip	Terminates immediately without any side effects
<i>STOP</i>	Deadlock	Refuses all interactions, but does not change the state
$P \parallel [cs] \parallel Q$	Parallel composition	Runs P and Q in parallel synchronising on events in cs
$\{e\}$	Channel set	Set of all possible events associated with channel e
$c \rightarrow P$	Prefix	Synchronise on channel c and then behave like P
$P; Q$	Sequential composition	Behaves like P , and once P terminates, behaves like Q
$P \sqcap Q$	Internal choice	Non-deterministically chooses between behaving like P or Q
$P \Theta_{cs} Q$	Exception	Behaves like P , until P raises an event in cs , at which point, it behaves like Q
$P \setminus cs$	Hiding	Runs P with events in cs hidden
$P \blacktriangleright d$	Deadline	Requires P to terminate within d time units

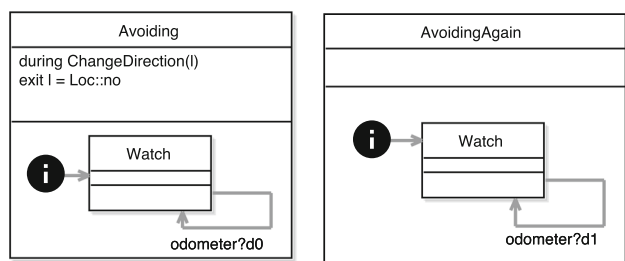


Fig. 9 States Avoiding and AvoidingAgain of an earlier version of SMovement

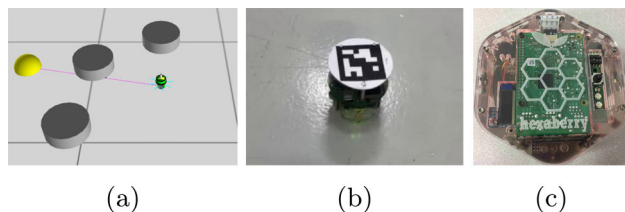


Fig. 10 The simulated and physical scenarios for evaluating the model designed in Sect. 5: **a** simulation; **b** physical e-puck robot; **c** physical hexaberry robot

VCRobot, the controller VController and the state machine Movement. The arrow of connection between events indicates direction. For example, the information of the turn event is passed from the controller to its state machine, Movement. The event flash is passed from the state machine to the controller and then to the robotic platform to light the robot's LEDs.

Figure 8 shows the state machine Movement. It requires the interfaces OperationsSig and OperationsDef. It also defines local variables (e.g., angle) and constants (e.g., outPeriod). The variable angle is used for the robot to turn. The variables d0 and d1 are used to record the odometer readings of the robot. The variables l and l_new of enumeration type Loc are used to record the direction of the obstacle relative to the robot. The state machine also uses the interface Events. Two clocks, T1 and T2, are declared to specify time associated with transitions. In particular, T1 is used to record the turning period of the robot, and T2 is used to record the battery life of the robot.

The state machine Movement consists of two top-level states, Searching and Finish. The initial state is Searching, as indicated by the transition from the initial junction. We need to ensure the robot never misses the command of a human operator according to the specification. Therefore, the cmd event leading to the Finish state is used to trigger the transition from the state Searching to Finish. In the Finish state, the robot stops, by calling stop, and then transitions to a final state.

Searching is a composite state. In this state, there is an entry action, where the location of the obstacle, as recorded by l, is set to Loc::no. The variable l is of type Loc, an enumerated type with constants left, right and no, indicating the existence of the obstacle and its direction, or the absence of an obstacle. When the state machine enters the Exploring state, its entry action is executed: MoveForward is called. While the robot is in the Exploring state, its sub-state Watch monitors for obstacles using the event obstacle. If an event obstacle is received (obstacle?l_new), its value is recorded in l_new, and then this is assigned to l as part of the transition's action ($l = l_new$).

Once an obstacle is detected (either left or right), the transition from Exploring to Avoiding is enabled, the clock (#T1) is reset, the state machine enters the state Avoiding, and the odometer reading (d0) is updated immediately. In this state, the robot attempts to avoid the obstacle by changing the robot's direction as part of the during action that calls operation ChangeDirection. We recall it is defined by a state machine as shown in Fig. 6. ChangeDirection takes the location l of the obstacle as a parameter, and if the obstacle is to the right of the robot, turns left; otherwise, it turns right by calling the appropriate operations TurnLeft or TurnRight, respectively.

In the Avoiding state, we assume that the robot needs a certain time (turningPeriod) to rotate a certain angle. After the robot has finished turning and no cmd event is detected, the state machine switches to the state ExploringAgain in which the robot continues exploring the environment. If an obstacle is detected again, it enters the state AvoidingAgain, where the odometer reading (d1) is updated immediately. When the robot is in state AvoidingAgain, the outgoing transitions are only triggered by a condition that decides whether the difference of two consecutive odometer readings is larger than a certain threshold (stuckDist). If so, we assume that the robot does not get stuck and the state machine transitions to Avoiding; otherwise, the robot is stuck and the state machine enters the GettingOut state. In this case, the robot performs a Reverse action. After a certain time, defined by a constant outPeriod, the state machine enters the Exploring state.

If a turn event is detected, and no obstacle has been detected (guarded by the condition: $l \neq \text{loc::no}$) and a certain amount of time (batteryLife) has passed, the state machine takes the transition to the MovingTowardsGoal state, where the robot turns towards the charging station by calling the operation MoveTowards with a specific angle, obtained from the robotic platform via the turn event. The behaviour is implemented by the operation MoveTowards of the robotic platform.

During this process, if no obstacle has been detected ($l \neq \text{Loc}::\text{no}$), and the robot finds the source, it enters the state Found. It then sends a flash event to the robot, which causes the robot to flash its LEDs.

6 Results

In this section, we report on the results of analysis, simulation and deployment of the system described above. In particular, we examine whether the requirements are met using three different approaches: property checking using model checking techniques, validation via simulations, and validation through experiments using physical robots. A video demonstrating how RoboTool is used, as well as model checking, simulation, and physical experiments using the RoboChart model is included in the supplementary material.

6.1 Property checking via formal verification

RoboChart has a formal semantics defined using *tock*-CSP (Baxter et al., 2022), a timed dialect of the process algebra CSP (Roscoe, 2011). Systems and their components are defined using processes, which interact with each other and their environment via atomic and instantaneous events. CSP provides a large number of operators that can be used to model complex concurrent systems. Table 1 presents the operators that we use for property checking in this work. At the core of CSP is the notion of refinement whereby a process P is refined by Q , written $P \sqsubseteq Q$, exactly when every behaviour of Q is also a behaviour of P , that is, given a specification P we can replace it with an implementation Q .

6.1.1 General properties

Before implementing a model designed in RoboTool, it is customary to make sure that a system satisfies general properties, such as being free of deadlocks and livelocks. General properties of each component can be checked by loading an automatically generated file with suffix `_coreassertions.csp` into the model checker FDR (Gibson-Robinson et al., 2014). Alternatively, FDR can be executed directly from RoboTool, and the results of the verification can be examined in a report that is generated automatically. Another option is using the RoboChart assertion language. For example, to check that the Movement state machine is deadlock free, we can write a property as follows.

assertion A0: Movement is deadlock-free

In our example, we are interested in two properties: deadlock free and livelock free. This means, for example, that the state machine does not contain deadlocks or infinite loops

that could preclude it from responding to inputs. Verification using FDR shows that our model satisfies these properties. This is a guarantee that we obtain without any extra effort, beyond writing the model itself. Verification of these properties at design time is useful, given that identifying the cause of deadlocks and livelocks in a simulation, or at deployment, could be non-trivial.

6.1.2 Task-specific properties

We also use FDR to verify the four requirements presented in Sect. 5 by stating them as refinement assertions. In the following, we show how these are formally specified using *tock*-CSP.

Property R1 For the first property, we want to verify that the robot responds to a command. That is, if a *cmd* event occurs, the robot stops. This is formalised in CSP below.

$$R1 = \text{CHAOS}(\Sigma) \ominus_{\{cmd\}} (\text{StopOp} ; \text{STOP})$$

The process $R1$ initially can perform any event nondeterministically any number of times as defined by $\text{CHAOS}(\Sigma)$, where Σ is the set of all events. Once the event *cmd* occurs, the behaviour is defined by *StopOp*, the process that models the call of operation Stop, sequentially composed with *STOP*, the CSP process that deadlocks, that is, it cannot engage in any event but allows time to pass. We omit the definition of *StopOp*, which requires the operation Stop to terminate immediately. We observe that this is not a deadline on the behaviour of stopping the robot, but rather a deadline on the software operation indicating that it takes a negligible amount of time.

To verify that a RoboChart model satisfies a particular requirement encoded as a CSP process P , we state that $P \sqsubseteq Q$, where Q is the CSP semantics of the RoboChart model. In the case of requirement R1 we check that $R1$ is refined by the CSP semantics of VModule, defined via a process *System*, as $R1 \sqsubseteq \text{System}$. This can be specified using RoboTool's assertion language as follows:

assertion A2: VModule refines R1

We use a particular semantic model of CSP for the refinement check that takes into account liveness and safety, that is, every trace of events performed by *System* is permitted by $R1$ (safety), and at any point in a trace of *System* where an event is accepted in $R1$ then it must also be accepted in *System* (liveness). For each assertion, FDR provides an answer indicating whether the refinement holds, and if it fails a counter-example is presented showing a trace of events after which some undesired behaviour is observed. For property R1 the refinement succeeds.

Property R2 The second property that we want to verify is that when the robot finds the charging station, as indicated

Fig. 11 A trajectory of the **a** simulated e-puck robot, **b** physical e-puck robot and **c** hexaberry robot executing the RoboChart model

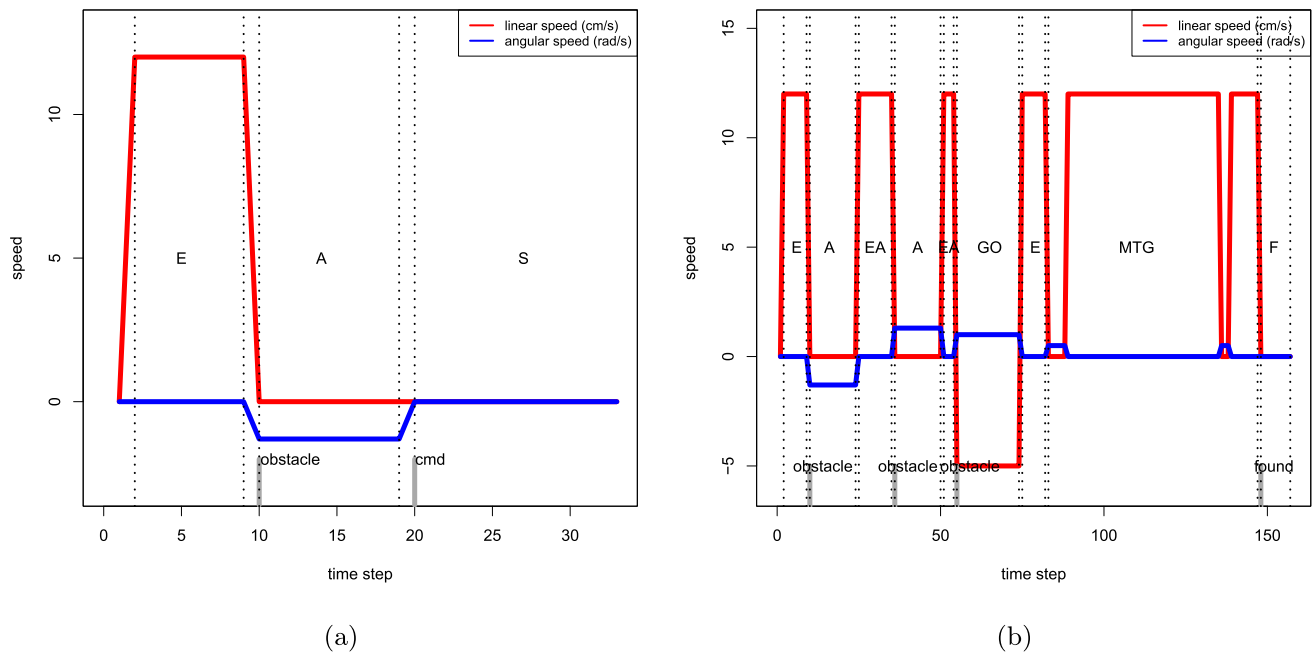
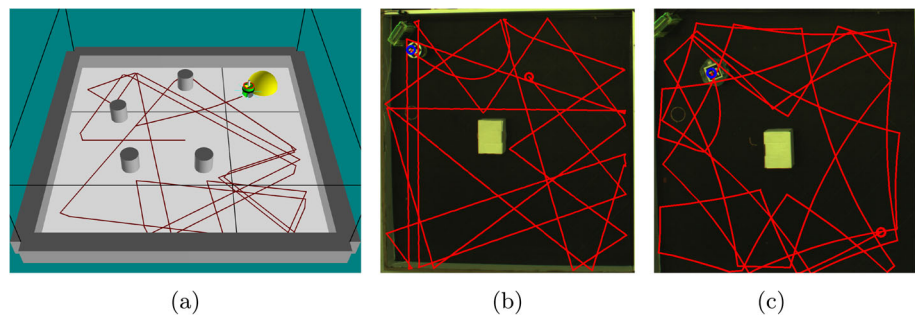


Fig. 12 Two experimental trials **(a)** and **(b)** of simulation

by the event *found*, then it flashes and stops. We formalise this requirement as follows.

$$R2 = \left(\text{CHAOS}(\Sigma) \ominus \{ \text{found} \} \right. \\ \left. (\text{flash} \rightarrow \text{StopOp}; \text{cmd} \rightarrow \text{StopOp}; \text{STOP}) \right)$$

Similarly to *R1*, initially *R2* can engage in any event nondeterministically any number of times. Once *found* takes place then it engages in event *flash*, and afterwards behaves as *StopOp*, the process that models the operation *Stop*, as already discussed, and then also accepts a *cmd* event. The fact that it may *Stop* twice is harmless. This property is also satisfied as checked by FDR.

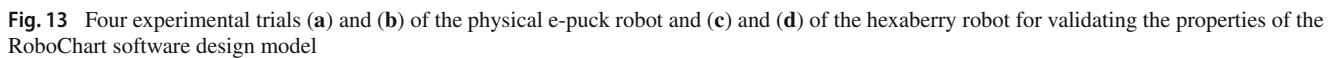
Property R3 The third property requires that if an obstacle is detected while no command is issued, either on the left or the right, via the parametrised event *obstacle*, then the robot should either avoid it, by changing direction, or reverse. We formalise this property as *R3* below, while the conditionality on the absence of *cmd*, and the possible internal odometer

readings, is considered in the refinement check that we define next.

$$R3 = \left(\text{CHAOS}(\Sigma) \ominus \{ \text{obstacle.left}, \text{obstacle.right} \} \right. \\ \left. ((\text{ChangeDirectionOp} \sqcap \text{ReverseOp}); R3) \right)$$

The process *R3* initially accepts any event nondeterministically. Once either *obstacle.left* or *obstacle.right* happen, *R3* behaves as either *ChangeDirectionOp* or *ReverseOp*, the processes that model the call to operations *ChangeDirection* or *Reverse*, and then recurses to behave as *R3*. Here the nondeterminism captures the fact that both behaviours are possible following the detection of an object and indeed a correct implementation could perform either, or even opt between the two behaviours each time an object is detected. The refinement in this case is stated as follows.

$$R3 \sqsubseteq (\text{System} | [\{ \text{cmd} \}] \text{SKIP}) \setminus \{ \text{odometer} \}$$



Property R4 The fourth requirement states that, while no command is issued, if an obstacle is detected, then the odometer is read. It is specified as follows.

Similarly to *R3*, initially we have that *R4* behaves as *CHAOS*(Σ), and can be interrupted by events *obstacle.left* or *obstacle.right* to behave as the process that requires *odometer?* x , with any value x , to take place immediately, specified using the deadline operator (\blacktriangleright), and then recurses as *R4*.

6.1.3 Model evolution

In this section we discuss the evolution of the state machine *SMovement*, using the results of analysis performed using FDR to illustrate problems that were encountered during modelling.

In Fig. 9 we reproduce states *Avoiding* and *AvoidingAgain* of an earlier version of *SMovement*, documented in Ribeiro and Li (2019). We observe that the transitions into and out of these states are the same as in Fig. 8, and so are omitted. Differently, however, these are composite states where the odometer is read on a self-transition of a substate named *Watch*. In this version *R3* is not satisfied without considering further assumptions, because, as defined, it is possible for the system to keep requesting odometer readings indefinitely and refuse to call *Reverse* or even accept *cmd*. This is not just undesirable, but misses an important assumption about the real-world that we did not account for in the verification: there cannot be an infinite number of events in a finite time.

Adjusting the verification by bounding the number of odometer readings in a finite amount of time is a possibility that we pursued. However, further analysis revealed that *R4* was also not satisfied. Analysis using FDR produced a counter-example, whereby after calling *MoveForward* and detecting an obstacle a call to *ChangeDirection* could happen before an odometer reading. Careful consideration of this example revealed that a stale value of variable *d1*, potentially not initialised, could be used when transitioning out of state *AvoidingAgain*. This is unlike the specification of the correct model in Fig. 8, where an odometer reading is always requested upon entering states *Avoiding* and *AvoidingAgain*. This led to the development of a second version of *SMovement*, as documented in Ribeiro and Li (2019).

In addition, because no deadline was specified on the reading of odometer the verification of *R4* using FDR revealed that it was possible for this input to be delayed. In the version presented in Fig. 8 the odometer reading in states *Avoiding* and *AvoidingAgain* has a deadline to enforce that it must be completed immediately. Results obtained with FDR helped to refine the model to obtain a version that meets the requirements. For our case study, the verification of each property took a couple of seconds on a desktop machine (Intel i3-8109U processor). Model-checking is a useful technique to guide roboticists in the development of a satisfactory RoboChart design model.

6.2 Simulation validation

In this section, we validate the three requirements of the RoboChart model in Sect. 5 using simulations.

6.2.1 Simulation platform

To simulate the RoboChart model shown in Fig. 5, we use a 3D simulator, ARGoS (Pinciroli, 2012). It has a built-in model of several ground robots such as e-puck (Mondada et al., 2009) and foot-bot (Ducatelle et al., 2011), and flying robots such as eye-bot (Ducatelle et al., 2011). In simulation, we exercise the RoboChart model using the light and proximity sensors located around the body of the e-puck. The details of the robots are described in Sect. 6.3.1. Figure 10a shows a scenario for evaluating the designed model. In simulation, the charging station is modelled using a light source (yellow). The robot performs a phototaxis behaviour when moving towards the light source.

6.2.2 Simulation experiments

In simulation, the robot is initially located in the centre of a square-sized arena of $1.5\text{ m} \times 1.5\text{ m}$. The obstacles are randomly distributed within the arena. The light source simulating the charging station is fixed in one corner of the arena.

To run the simulation with a particular robot, the generated code was completed with platform-dependent configurations. For example, in the class of *HardwareComponent*, the API of e-puck (sensors and actuators) needs to be inserted⁸. The operations such as *Actuate* corresponding to different robots also need to be adjusted. For example, for the e-puck, the C++ implementation of the operation *Actuate* is defined as follows:

```
void Operations::Actuate():@ // reverse kinematicsdouble
lv,rv;lv=(2*:@:linearSpeed:@-
:@:angularSpeed:@*5.2)/2;rv=(2*:@:linearSpeed:@
+:@:angularSpeed:@*5.2)/2;:@:wheels_actuators:@-
->:@:SetLinearVelocity:@(lv,rv);
```

The C++ source code for the complete simulation targeting ARGoS is available online.⁹

Figure 11a shows the trajectory of the simulated e-puck robot in one of the trials. Figure 12a and b show another two trials of the simulation experiments, which show that the properties have been met. We record the speed of the robot in each state and the point where the events occur. The letters showing in each region separated by dotted lines represent the state (with shortened name) in which the state machine stays. The gray line in the figure marks the point where the event occurs. Figure 12a validates the requirement *R1* in which the robot responds to the *cmd* immediately and stops. In this figure, the robot first explores the environment by moving

⁸ A detailed description of how to construct an ARGoS simulation using the generated code can be found in: <https://liwei-cn.gitbooks.io/robochart-simulation-framework/content/>

⁹ https://robo-star.cs.york.ac.uk/case_studies/exploration/Exploration-V8-gen-argos.zip

forward (with a fixed linear speed and zero angular speed). At time step 10, an obstacle is detected and the robot turns on the spot (with zero linear speed and a negative angular speed). While the robot is still turning, a *cmd* is detected at time step 20 and the robot stops immediately (with zero linear and angular speed). The requirements *R2* and *R3* are validated in Fig. 12b. The robot avoids the detected obstacles and stops when the ‘charging station’ is found. We observe that because the behaviour is deterministic, these trials are adequate to validate the software controller and are representative of other runs, namely in that they cover similar sequences of behaviours of interest.

6.3 Hardware validation

In this section, the requirements of the RoboChart model in Sect. 5 are further validated using physical experiments on two robotic platforms.

6.3.1 Robotic platforms

E-puck Robot: The e-puck, shown in Fig. 10b, is a miniature, differential wheeled mobile robot. The e-puck’s diameter and height are approximately 7.4 cm and 5.5 cm, respectively, and its weight is approximately 150 g. The e-puck is equipped with a directional camera located at its front. The e-puck’s processor is a Microchip dsPIC microcontroller. The Linux board equipped on the robot provides a convenient way for programming the robot using an object oriented language such as C++. The Linux board and microcontrollers communicate via UART.

Hexaberry Robot: The hexaberry robot, which is shown in Fig. 10c, is a Raspberry Pi based robot developed in the York Robot Lab. It carries 6 sensor boards containing various sensors (proximity sensors, TOF laser sensors, etc.) around its body, and a front camera. The robot uses standard I2C protocols to access sensor data. The robot also has an on-board LCD for display and debugging.

6.3.2 Physical experiments

As the e-puck is equipped with a Linux board, the simulation code for ARGoS can be run on the board without any modification. For the hexaberry robot, the code is directly run on the Raspberry Pi to control the robot. The operations and sensor readings still need to be customised for different robotic implementations such as the operation *MovingTowards*. We observed that due to the interference of the infrared beacon, the readings of the proximity sensors of the e-puck and hexaberry robots are noisy. As a result, the robot would perform obstacle avoidance behaviour even without detecting genuine obstacles. We filtered the beacon readings by synchronizing

the frequency of the beacon and the robots’ control cycle, thus minimizing the influence of random noise.

Figure 11b and c show the trajectory of the e-puck and hexaberry robots executing the generated implementation in two of the trials. Figure 13a–d show four trials of the physical experiments, and the traces are similar to those of the simulation experiments. Due to the reality gap (Koos et al., 2013), for different platforms, certain parameters need to be adjusted. For example, the parameter *turningPeriod* for simulation and each physical robot are set to different values in order to perform sufficient obstacle avoidance behaviour. A RoboChart design model contributes to narrowing the gap by providing a systematic approach for developing a software controller. When the behaviour observed is not as expected, a designer can tweak parameters of a model or iterate the design, like we have done, rather than changing the controller code directly. Code implementing the platform services is clearly isolated in this architecture. If it is also relevant for design and analysis, then the robotic platform abstraction needs to be made more concrete and more of its behaviour incorporated into the RoboChart model. This modelling challenge is part of ongoing work on integrating design models with those of physical platforms (kinematics, dynamics, sensors and actuators), some of which we briefly discuss in the conclusions. Related, we anticipate uncertainties could be modelled via the use of probabilistic junctions in a design, or, by considering models that incorporate non-deterministic behaviour, for example, where values range over intervals.

7 Conclusion

In this section we conclude by summarising our contributions, highlighting limitations of our approach, and sketching paths for future work.

7.1 Summary

In this paper, we have explored the application of a model-based framework for modelling software controllers of autonomous robots, combined with the automatic generation of C++ code as well as automatic property checking to guide the design. The properties of the software controller are verified using the automatically generated formal semantics at design time, and are further validated in both simulations and physical robots.

Our vision is to reduce the gap between the high-level modelling and low-level implementation of robotic controllers through the use of formal methods and automatic code generation. This could contribute to providing evidence of safety when the robotic systems are to be deployed in real-world scenarios, especially in safety-critical domains like military and aerospace. Proofs of properties before starting a

simulation or deployment could potentially reduce the testing time as mistakes can be found earlier in the design stage. For example, it is feasible to check that designs are free from deadlocks, an issue that is harder to pinpoint at a later stage.

For modelling, RoboTool provides a graphical editor for constructing RoboChart design models, which is useful to support practitioners in their use of diagrammatic state machines. RoboChart has a formal semantics, so models have a precise account of their behaviour suitable for analysis using formal verification tools. Our extension caters for the automatic generation of code, with only minimal effort required to target a particular robotic platform, suitable for conducting experiments via simulation and deployment. User studies to gauge the usability of our prototype implementation in RoboTool, alongside a broader evaluation of our tools and techniques, including by application to further case studies, are future work.

7.2 Limitations

Because a RoboChart model accounts only for the software, physical aspects concerning the robot and its interaction with the environment need to be captured separately. This is in contrast, for example, with the proposition in Colledanchise and Ögren (2017) where BTs are also used for capturing continuous phenomena. Our experience (Murray et al., 2022) indicates that this separation of concerns can be exploited for the co-verification of system-level properties. Further work is required to consider all these aspects in a unified model-based framework and provide formal guarantees. Related, RoboChart also has facilities that can help with modelling uncertainty, namely via probabilistic junctions (Ye et al., 2022).

The model-driven approach we propose here targets the early stages in the development of a high-level controller, with verification and simulation supporting a developer's workflow. For verification, RoboTool automatically calculates a *tock*-CSP semantics (Baxter et al., 2022) that can be exhaustively checked with FDR. With this technique there is no need to provide explicit input and output values, however it is limited to instantiating data types over finite discrete domains and can suffer from the well-known state-explosion problem. In our experience, high-level designs, due to their abstraction, suffer less from this problem, and are therefore more amenable to this type of verification technique. RoboChart models, namely featuring probabilistic junctions, can also be analysed using PRISM (Kwiatkowska et al., 2011), which provides, in addition, the facility to employ statistical model-checking techniques.

Currently, the generated controller code is a direct mapping from the elements in RoboChart to various classes in the software architecture. We adopt similar communication mechanisms in the code and the RoboChart semantics. In

particular, we envisage that the CSP model generated from a RoboChart specification will be useful to establish the correctness of the generated code using refinement. Practical verification can be carried out using model checkers like FDR (which also provides a facility to animate the model, and thus perform further validation), as used in this work. Integration with theorem provers is ongoing work: an approach for checking deadlock freedom of RoboChart models that do not make use of timed primitives is discussed in Foster et al. (2018). To validate the correctness of the code generator, we recorded the robot's traces, which record the states the robot stays in and any associated events that may change the evaluation of guards and enable transitions. The traces are then passed back to FDR for verification. If FDR reported a successful validation, this meant that the traces were correct according to the model.

7.3 Future work

Another avenue to be explored is checking of properties written in the positive fragment of LTL, that can be expressed via refinement checks for CSP-based semantics (Lowe, 2008). Expressing properties via sequence diagrams is ongoing work (Windsor & Cavalcanti, 2022).

When model checking task-specific properties, some implicit assumptions may need to be made explicit to make the formal verification feasible. These assumptions may not be captured when specifying the properties using natural languages but are typically assumed by implementations and are necessary when analysing RoboChart models. This is because RoboChart is a modelling language rather than a simulation or execution language. To avoid this ambiguity, we are developing a new language, RoboSim, to capture some of these assumptions in the simulation context. As future work, we will investigate the use of controlled natural language (Cavalcanti et al., 2021) to capture assumptions and requirements such as the ones discussed in this paper. Currently, our code generation targets one state machine only. We aim to support more complex robotic systems consisting of multiple parallel (and interacting) state machines, controllers, and multi-robot systems.

Finally, in future work we will consider the connection of RoboChart design models with models that account for kinematics, dynamics, and properties of sensors and actuators, such as noise. That requires targeting richer formal semantics encompassing hybrid and stochastic aspects, but will allow us to further reduce the reality gap by recording assumptions about the physical platform. Together with models of the scenarios of interest, recording properties of

the arena, we envisage that it will be possible to reason about system-level properties, such as, area-coverage.

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1007/s10514-024-10163-7>.

Acknowledgements The authors would like to acknowledge the support from EPSRC grants EP/M025756/1 and EP/R025479/1, and the Royal Academy of Engineering. We thank James Hilder for assistance with the physical experiments, and the anonymous reviewers for their constructive feedback. The icons used in RoboChart have been made by Sarfraz Shoukat, Freepik, Google, Icomoon and Madebyoliver from www.flaticon.com, and are licensed under CC 3.0 BY <https://creativecommons.org/licenses/by/3.0/>.

Declarations

Conflict of interest Wei Li and Pedro Ribeiro contributed equally to this paper. Wei Li, Pedro Ribeiro and Jon Timmis wrote the main manuscript text and Alvaro Miyazawa prepared Fig. 2–4. All authors reviewed the manuscript. The authors declare that they have no Conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

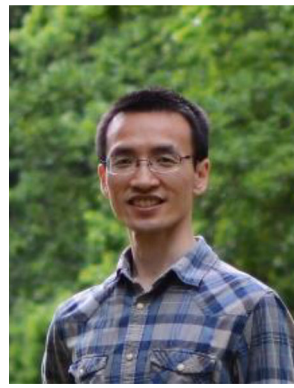
References

- OMG Unified Modeling Language (2015). Object Management Group, www.omg.org/spec/UML/2.5/
- Abdellatif, T., Bensalem, S., Combaz, J., De Silva, L., & Ingrand, F. (2012). Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12), 1563–1578.
- Abrial, J. (2010). *Modeling in event-b-system and software engineering*. Cambridge University Press.
- Alur, R., & Henzinger, T. A. (1994). A really temporal logic. *Journal of the ACM*, 41(1), 181–204.
- Barnett, W., Cavalcanti, A., & Miyazawa, A. (2022). Architectural modelling for robotics: RoboArch and the CorteX example. *Frontiers in Robotics and AI*, 9, 991637.
- Baumgartl, J., Buchmann, T., Henrich, D. and Westfechtel, B. (2013). Towards easy robot programming: Using DSLs, code generators and software product lines. In *Proceedings of the 8th international joint conference on software technologies*, Reykjavík, Iceland. SciTePress, pp 548–554.
- Baxter, J., Ribeiro, P., & Cavalcanti, A. (2022). Sound reasoning in tock-CSP. *Acta Informatica*, 59(1), 125–162.
- Bergenti, F., & Poggi, A. (2000). Exploiting UML in the design of multi-agent systems. In *1st international workshop on engineering societies in the agents world*, Berlin, Germany, Lecture Notes in Computer Science, vol 1972. Springer, p 106–113.
- Bjerknes, J. D., & Winfield, A. F. T. (2010). On fault tolerance and scalability of swarm robotic systems. In *The 10th international symposium on distributed autonomous robotic systems*, Lausanne, Switzerland, Springer Tracts in Advanced Robotics, vol 83. Springer, p 431–444.
- Bordini, R. H., Dennis, L. A., Farwer, B., & Fisher, M. (2008). Automated verification of multi-agent programs. In *23rd IEEE/ACM international conference on automated software engineering*, L'Aquila, Italy. IEEE Computer Society, pp 69–78.
- Bourbough, H., Garoche, P. L., Loquen, T., Noulard, É., & Pagetti, C. (2020). CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models. In *10th European congress on embedded real time software and systems*, Toulouse, France.
- Bourbough, H., Farrell, M., Mavridou, A., Slijo, I., Brat, G., Dennis, L. A., & Fisher, M. (2021). Integrating formal verification and assurance: An inspection rover case study. In *13th international symposium on NASA formal methods, virtual event, lecture notes in computer science*, vol 12673. Springer, pp 53–71.
- Bringsjord, S., & Sen, A. (2016). On creative self-driving cars: Hire the computational logicians, fast. *Applied Artificial Intelligence*, 30(8), 758–786.
- Caliskanelli, I., Goodliffe, M., Whiffin, C., Xymitoulas, M., Whitaker, E., Verma, S., Hickman, C., Minghao, C., & Skilton, R. (2021). CorteX: A software framework for interoperable, plug-and-play, distributed, robotic systems of systems. In A. Cavalcanti, B. Dongol, R. Hierons, J. Timmis, & J. Woodcock (Eds.), *Software engineering for robotics*. Springer.
- Cardoso, R. C., Farrell, M., Luckcuck, M., Ferrando, A., & Fisher, M. (2020). Heterogeneous verification of an autonomous curiosity rover. In *12th international symposium on NASA formal methods, Moffett Field, CA, USA, Lecture Notes in Computer Science*, vol. 12229. Springer, pp 353–360.
- Carvalho, R., Cunha, A., Macedo, N., & Santos, A. (2020). Verification of system-wide safety properties of ROS applications. In *IEEE/RSJ international conference on intelligent robots and systems*. IEEE, pp 7249–7254.
- Pilaud, D., Halbwachs, N., & Plaice, J. A. (1987). Lustre: A declarative language for programming synchronous systems. In *Conference record of the fourteenth annual ACM symposium on principles of programming languages* (pp. 178–188). Germany. Association for Computing Machinery: Munich.
- Cavalcanti, A., Miyazawa, A., Ribeiro, P., & et al. (2018). RoboStar case studies. https://robostar.cs.york.ac.uk/case_studies/
- Cavalcanti, A., Baxter, J., Carvalho, G. (2021). Roboworld: Where can my robot work? In *19th International conference on software engineering and formal methods, virtual event, lecture notes in computer science*, vol. 13085. Springer, pp 3–22.
- Champion, A., Mebsout, A., Stickel, C., & Tinelli, C. (2016). The Kind 2 model checker. In *28th International Conference on Computer Aided Verification*, Toronto, ON, Canada, Lecture Notes in Computer Science, vol. 9780. Springer, pp 510–517.
- Chen, J., Gauci, M., Li, W., Kolling, A., & Gros, R. (2015). Occlusion-based cooperative transport with a swarm of miniature mobile robots. *IEEE Transactions on Robotics*, 31(2), 307–321.
- Colledanchise, M., & Ögren, P. (2017). How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on Robotics*, 33(2), 372–389.
- Colledanchise, M., Cicala, G., Domenichelli, D. E., Natale, L., & Tacchella, A. (2021). Formalizing the execution context of behavior trees for runtime verification of deliberative policies. In *IEEE/RSJ international conference on intelligent robots and systems*. IEEE, pp 9841–9848.
- Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., & Ziane, M. (2012). RobotML, a domain-specific language to design, simulate and

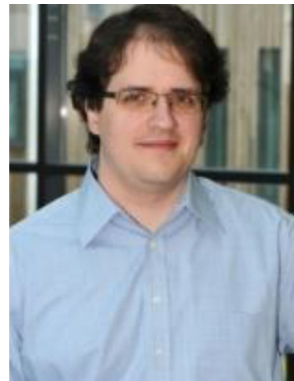
- deploy robotic applications. In *Third international conference on simulation, modeling, and programming for autonomous robots*, Tsukuba, Japan, Lecture Notes in Computer Science, vol. 7628. Springer, p 149–160.
- Ducatelle, F., Di Caro, G. A., Pinciroli, C., & Gambardella, L. M. (2011). Self-organized cooperation between robotic swarms. *Swarm Intelligence*, 5, 73–96.
- Feiler, P. H., & Gluch, D. P. (2012). *Model-based engineering with ADL: An introduction to the SAE architecture analysis & design language*. Addison-Wesley.
- Fleurey, F., & Solberg, A. (2009). 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*, Denver, CO, USA, *Lecture Notes in Computer Science*, vol. 5795. Springer, pp 606–621.
- Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., & Woodcock, J. (2018). Automating verification of state machines with reactive designs and Isabelle/UTP. In *15th international conference on formal aspects of component software*, Pohang, South Korea, *Lecture Notes in Computer Science*, vol. 11222. Springer, pp 137–155.
- Foughali, M., Berthomieu, B., Dal Zilio, S., Ingrand, F., & Mallet, A. (2016). Model checking real-time properties on the functional layer of autonomous robots. In *18th International conference on formal engineering methods*, Tokyo, Japan, *Lecture Notes in Computer Science*, vol. 10009. Springer, pp 383–399.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley.
- Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., & Shi, N. (2020). Formal requirements elicitation with FRET. In *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, live studies track, and poster track co-located with the 26th international conference on requirements engineering: foundation for software quality Pisa*, Italy. p 2584.
- Gibson-Robinson, T., Armstrong, P., Boulgakov, A., & Roscoe, A. W. (2014). FDR3—A modern refinement checker for CSP. In Ábrahám, E., Havelund, K. (Eds) *20th international conference on tools and algorithms for the construction and analysis of systems*, Held as Part of the European Joint Conferences on Theory and Practice of Software, Grenoble, France, *Lecture Notes in Computer Science*, vol 8413. Springer, pp 187–201.
- Harbin, J., Gerasimou, S., Matragkas, N., Zolotas, A., & Calinescu, R. (2021). Model-driven simulation-based analysis for multi-robot systems. In *24th International conference on model driven engineering languages and systems* (pp. 331–341). Japan. IEEE: Fukuoka.
- Kent, S. (2002). Model driven engineering. In *3rd international conference on integrated formal methods*, Turku, Finland, *Lecture Notes in Computer Science*, vol 2335. Springer, pp 286–298.
- Klotzbucher, M., & Bruyninckx, H. (2012). Coordinating robotic tasks and systems with rFSM statecharts. *Journal of Software Engineering for Robotics*, 2(13), 28–56.
- Koos, S., Mouret, J., & Doncieux, S. (2013). The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, 17(1), 122–145.
- Kortik, S., & Shastha, T. K. (2021). Formal verification of ROS based systems using a linear logic theorem prover. In *IEEE international conference on robotics and automation*, Xi'an, China. IEEE, pp 9368–9374.
- Kwiatkowska, M. Z., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan G, Qadeer S (eds) *Computer Aided Verification—23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Proceedings*, *Lecture Notes in Computer Science*, vol 6806. Springer, pp 585–591.
- Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *16th International conference on logic for programming, artificial intelligence, and reasoning, Dakar, Senegal, Lecture Notes in Computer Science*, vol. 6355. Springer, pp 348–370.
- Lewis, W. E. (2009). *Software testing and continuous quality improvement* (3rd ed.). Auerbach Publications.
- Li, W., Miyazawa, A., Ribeiro, P., Cavalcanti, A., Woodcock, J., & Timmis, J. (2016). From formalised state machines to implementation of robotic controllers. In *The 13th international symposium on distributed autonomous robotic systems*, Natural History Museum, London, UK, *Springer Proceedings in Advanced Robotics*, vol. 6. Springer, pp 517–529.
- Lowe, G. (2008). Specification of communicating processes: Temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3), 277–294.
- Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., & Fisher, M. (2019). Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5), 1–41.
- Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., & Timmis, J. (2017). Automatic property checking of robotic applications. *2017 IEEE/RSJ international conference on intelligent robots and systems* (pp. 3869–3876). Canada. IEEE: Vancouver, BC.
- Miyazawa, A., Cavalcanti, A., Ribeiro, P., Li, W., Woodcock, J., & Timmis, J. (2018). RoboChart reference manual. Tech. rep., University of York, <https://robostar.cs.york.ac.uk/publications/techreports/reports/robochart-reference.pdf>
- Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., & Woodcock, J. (2019). RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18, 3097–3149.
- Miyazawa, A., Cavalcanti, A., Ahmadi, S., Post, M., & Timmis, J. (2020). RoboSim physical modelling reference manual. Technical report, University of York, <https://www.cs.york.ac.uk/circus/publications/techreports/reports/physmod-reference.pdf>
- Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klapotcz, A., Magnenat, S., Zufferey, J. C., Floreano, D., & Martinoli, A. (2009). The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions*, Castelo Branco, Portugal. IPCB, Instituto Politécnico de Castelo Branco, pp 59–65.
- Murray, Y., Sirevåg, M., Ribeiro, P., Anisi, D. A., & Mossige, M. (2022). Safety assurance of an industrial robotic control system using hardware/software co-verification. *Science of Computer Programming*, 216(102), 766.
- Nordmann, A., et al. (2016). A survey on domain-specific modeling and languages in robotics. *Journal of Software Engineering for Robotics*, 7(1), 75–99.
- Park, H. W., Ramezani, A., & Grizzle, J. W. (2013). A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. *IEEE Transactions on Robotics*, 29(2), 331–345.
- Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., & Birattari, M. (2012). ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems. *Swarm intelligence*, 6, 271–295.
- Ray, S. (2010). *Scalable techniques for formal verification*. Springer.
- Ribeiro, P., & Li, W. (2019) Case-study: exploration. https://robostar.cs.york.ac.uk/case_studies/exploration/
- Roscoe, A. W. (2011). *Understanding concurrent systems, texts in computer science*. Springer.
- Rouff, C. A., Hinchey, M. G., Pena, J., & Ruiz-Cortes, A. (2007). Using formal methods and agent-oriented software engineering for modeling NASA swarm-based systems. *2007 IEEE Swarm Intelligence Symposium* (pp. 348–355). USA. IEEE: Honolulu, Hawaii.

- Santos, A., Cunha, A., & Macedo, N. (2021). The high-assurance ROS framework. In *3rd IEEE/ACM international workshop on robotics software engineering*. IEEE, pp 37–40.
- Schlegel, C., Hassler, T., Lotz, A., & Steck, A. (2009). Robotic software systems: from code-driven to model-driven designs. In *14th international conference on advanced robotics* (pp. 1–8). Germany. IEEE: Munich.
- Schlegel, C., Lotz, A., Lutz, M., & Stampfer, D. (2021). Composition, separation of roles and model-driven approaches as enabler of a robotics software ecosystem. In *Software engineering for robotics*. Springer, p 53–108.
- Schultz, U. P., Christensen, D. J., & Stoy, K. (2007). A domain-specific language for programming self-reconfigurable robots. In *Proceedings of the 2007 workshop on automatic program generation for embedded systems*. ACM, Salzburg, Austria, pp 28–36.
- Selic, B. (1998). Using UML for modeling complex real-time systems. Lecture notes in computer science. In F. Mueller & A. Bestavros (Eds.), *Languages, compilers, and tools for embedded systems*. Springer.
- Selic, B., & Grard, S. (2013). *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Morgan Kaufmann Publishers Inc.
- Stampfer, D., Lotz, A., Lutz, M., & Schlegel, C. (2016). The smartmdsd toolchain: An integrated mdsd workflow and integrated development environment (ide) for robotics software. *Journal of Software Engineering for Robotics (JOSER)*, 7(1), 3–19.
- Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K. L., Dautenhahn, K., & Saez-Pons, J. (2015). Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Transactions on Human-Machine Systems*, 46(2), 186–196.
- Webster, M., Fisher, M., Cameron, N., & Jump, M. (2011). Formal methods for the certification of autonomous unmanned aircraft systems. In: 30th International Conference on Computer Safety, Reliability, and Security, Naples, Italy, Lecture Notes in Computer Science, vol 6894. Springer, pp 228–242.
- Windsor, M., & Cavalcanti, A. (2022). RoboCert: Property specification in robotics. In A. Riesco & M. Zhang (Eds.), *Formal methods and software engineering*. Springer.
- Winfield, A. F., Sa, J., Fernandez-Gago, M. C., Dixon, C., & Fisher, M. (2005). On formal specification of emergent behaviours in swarm robotic systems. *International Journal of Advanced Robotic Systems*, 2(4), 363–370.
- Woodcock, J. C. P., & Davies, J. (1996). *Using Z—Specification, refinement, and proof*. Prentice-Hall.
- Ye, K., Cavalcanti, A., Foster, S., Miyazawa, A., & Woodcock, J. (2022). Probabilistic modelling and verification using RoboChart and PRISM. *Software and Systems Modeling*, 21(2), 667–716.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Wei Li received the B.Eng. degree in automation and the M.Eng. degree in control science and engineering from the Harbin Institute of Technology, China, in 2009 and 2011, respectively, and the Ph.D. degree in 2016 from the University of Sheffield, UK. After being a research associate at the University of York, UK, he is currently an associate professor with the Academy for Engineering and Technology, Fudan University, China. His research interests include robotics and computational intelligence, and especially embodied intelligence, evolution and learning, self-organized/swarm systems.



Pedro Ribeiro is a Lecturer in Computer Science at the University of York, UK. Previously, he was a Research Fellow in the School of Physics Engineering and Technology and before that a Research Associate. He completed his PhD in CS on the topic of Angelic Processes. His research interests span the breadth of the engineering lifecycle for robotics, including design and development of domain-specific notations and their formal semantics, model-based testing and verification using automated proof techniques. He is a member of the RoboStar Centre for Excellence in Software Engineering for Robotics, and a founding member of Formal Methods Europe's communications committee.



Alvaro Miyazawa is a lecturer at the Department of Computer Science of the University of York. Having completed BSc in Computer Science at the University of Sao Paulo and doctoral research at the University of York, his main research interests are formal semantics and refinement for domain-specific languages and graphical notations and the development of verification strategies to support high levels of automation in program verification. He has applied and developed formal techniques in various fields, including systems engineering, safety-critical real-time systems, and robotics. Currently, his research focuses on modelling, testing, simulation and verification for robotics.

Richard Redpath received his PhD in Electronic Engineering from the University of York in 2019, and he is a Research Associate in the Intelligent and Adaptive Systems group at the Department of Electronic Engineering, University of York, working on model-based approaches for the development of robot controllers.



Ana Cavalcanti is a Professor at the University of York, UK, and holds a Royal Academy of Engineering Chair in Emerging Technologies. In that role, she is Director of the RoboStar centre on Software Engineering for Robotics. She previously held a Royal Society Industry Fellowship, which provided her with the ideal opportunity to understand and contribute to the practice of formal methods working with QinetiQ. Her main scientific achievements have been on the design and justification of

sound program design and verification techniques. She has covered theoretical and practical integration with industry-strength technology. Her work provides support for graphical notations popular with engineers, and for main-stream programming languages. It also supports high degrees of automation to enable usability and scalability. She has chaired the Programme Committee of leading conferences, and been a member of numerous Programme Committees. Currently, she is the Chair of the Formal Methods Europe Board.



Jon Timmis is currently Vice-Chancellor at Aberystwyth University and Professor of Intelligent and Adaptive Systems. Jon graduated in Computer Science from Aberystwyth University and went on to study a PhD at Aberystwyth in the area of artificial intelligence, with a focus on the immune system. Over the last 25 years, his research has focused on the intersection of immunology, computational and mathematical modelling, machine learning, robotics and swarm robotics. Jon is a previous recipient of a Royal Society-Wolfson Research Merit Award and a Royal Academy of Engineering Enterprise Fellowship. Jon co-founded a company to commercialise his research in modelling and simulation applied in the pharmaceutical industry in 2014.

simulation applied in the pharmaceutical industry in 2014.



Kieran Alden is a research fellow in intelligent and adaptive systems with the Department of Electronic Engineering, University of York. He conducts interdisciplinary research that aims to increase confidence in predictions generated by computer models of biological systems, through the development and application of novel techniques that understand and quantify the relationship between the model and the biological system that model is designed to capture. He is a member of the IEEE.



Jim Woodcock holds the Anniversary Chair in Software Engineering at the University of York in the UK. He is an elected fellow of the UK Royal Academy of Engineering and a chartered engineer. He is a professor of computer science at Aarhus University in Denmark. He is a professor of cyber-physical systems, a distinguished researcher at Southwest University in China, and a Chinese Academy of Sciences President's international fellow and distinguished scientist. He is an elected

fellow of the Asia-Pacific AI Association. He is editor-in-chief of ACM's Formal Aspects of Computing journal and of Cambridge's Research Directions: Cyber-Physical Systems.