Automatic Property Checking of Robotic Applications

Alvaro Miyazawa[1], Pedro Ribeiro[1], Wei Li[2], Ana Cavalcanti[1] and Jon Timmis[2]

***Abstract*— Robot software controllers are often concurrent and time critical, and requires modern engineering approaches for validation and verification. With this motivation, we have developed a tool and techniques for graphical modelling with support for automatic generation of underlying mathematical definitions for model checking. It is possible to check automatically both general properties, like absence of deadlock, and specific application properties. We cater both for timed and untimed modelling and verification. Our approach has been tried in examples used in a variety of robotic applications.**

I. Introduction

Robotic software is typically very complex. Customised architectural designs and development processes have been proposed to master complexity [1], [2]. Associated with that, domain-specific languages (DSL) have been presented for modelling [3], [4] and simulation [5], which, along with testing, is the favoured method of validation and verification. Here, we focus on validation of low-level designs, as opposed to decision-making or planning-level control, to *complement* simulation efforts. While simulation can reveal properties of specific scenarios, proof can establish such properties. While simulation is appealing in design exploration, proof can provide assurance that key properties are always valid.

Our approach to modelling controllers is based on a profile (that is, restriction) of the very popular UML [6] called RoboChart. At their core, RoboChart models are defined by state machines, a notation widely accepted in the embedded-software industry. In this approach, the behaviour of a robot is characterised by a state, in which it may execute a particular operation and react to events from its environment. Reaction typically leads to a change of state.

RoboChart also includes elements to structure models to foster reuse and modularity. These structuring facilities embed the notions of robotic platforms and their controllers, with synchronous or asynchronous communications. Distinctively, RoboChart also has constructs to specify time properties: budgets and deadlines for operations and events.

The RoboChart notation is designed with two main goals: (1) adopt the style of modelling used by the robotics literature and (2) enable automatic generation of mathematical definitions for proof. Several DSL for robotics use state machines [5], [4], [7], some supported by C code [8] to define simulations. RoboChart targets graphical modelling, rather than simulation or programming, and validation by proof of (possibly timed) properties. On the other hand, it is possible to generate simulations from RoboChart models [9].

A tool, called RoboTool[1], provides a graphical editor for RoboChart models, and generates automatically mathematical definitions that capture their behaviour. RoboTool is integrated with a model checker [10], a tool that uses the mathematical definitions to prove general properties, like freedom of deadlock, or specific properties, like impossibility of ignoring a particular event in given circumstances, for example. When testing a system or a model, via simulation, for instance, we cannot cover all scenarios that can play out. A model checker, on the other hand, explores exhaustively all the states of the model to ensure that the property of interest is satisfied. Any correct implementation of the model also has that property. If, however, the property does not hold, a model checker provides a counterexample.

RoboTool is provided as a set of Eclipse plugins implemented using the Xtext[2] and Sirius[3] frameworks. Xtext automatically generates plugins that implement a parser, and provides mechanisms for the implementation of validators, type checkers, and code generators. Sirius supports the definition of graphical representations, and produces a plugin for construction and visualisation of models. The model checker used by RoboTool is FDR4 [10].

In this paper, we present RoboTool, RoboChart, and their use to validate models for several applications: chemical detectors [11], and a transporter [12]. Fig. 1 shows the robot from [11], used here as a running example: a tele-operated chemical detector. It has a sensor at the front to identify changes in the chemical composition of air over time. Upon detection of a harmful gas, it indicates via a yellow light the presence of an anomaly, or, depending on the nature and concentration of the gas, via a red light, a siren, and a flag to mark the location. The robot uses a main processor to detect gas and accept commands from the operator to move, and a micro-controller to manage the light, the siren, and the flag.

The rest of the paper is structured as follows. In the next section, we discuss related work. RoboChart and RoboTool are explained in Section III. Property checking is discussed in Section IV. Our examples are the subject of Section V. We conclude and propose future work in Section VI.

II. Related work

A recent survey on DSL for robotics is available [13]; it seems to indicate an increase in the interest and use of DSL. In most works, the models enable code generation for execution or simulation. Although this is possible for RoboChart [9], our focus is generation of a mathematical

*This work was supported the EPSRC.
[1]Department of Computer Science, University of York, UK
[2]Department of Electronic Engineering, University of York, UK

[1]Available at `www.cs.york.ac.uk/circus/RoboCalc`.
[2]`https://eclipse.org/Xtext/`
[3]`www.eclipse.org/sirius`

Fig. 1. Chemical detector from [11]. We discuss automatic proof of some of its key properties in Section IV.

model for property checking. Closer to our work are the languages that address architectural design and programming, dealing with concurrency and control of events, targetting capability building. While there is a large number of such proposals, there is a limited amount of results on mathematical verification. We present some below.

Verification by model checking is also considered for GenoM [14]. Models are translated to a mathematical notation called Fiacre, which is close to the input language of the Petri Net based model checker TINA. Verification focusses on schedulability, while functional properties are not yet pursued. Another approach uses an alternative mathematical notation, called BIP, for deadlock checking and schedulability analysis [15]. As opposed to RoboChart, GenoM is an executable language; models, for example, include C code.

The language in [16] 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. Behavioural properties are not the focus.

In [17], a pioneering effort defines Orccad, for modelling, simulation, and programming. Translation to two different notations allows verification of timed behavioural properties. Orccad models a system in terms of tasks defined by control laws, combined by procedures defined by reactive programs.

In summary, most of the DSL for robotics in the literature are not associated with a technique for proof of properties. As far as we know, the works that present such a technique, do not focus on behavioural properties. This is what we address.

There are, of course, many general-purpose languages for which support for model checking is available, like C, for example; it is also possible to apply model checking to Simulink models. Early efforts on verification for robotics apply existing mathematical techniques as they are [17]. What we present here is a customised approach for modelling and property checking of robotic systems. Customisation allows us to use a simple language akin to what is already used by practitioners [5], [4], [7], [18], with friendly support for graphical modelling, and optimisation in the checks.

III. RoboChart and RoboTool

To prove properties of a robotic system, we need an unambiguous description of its behaviour. We cannot rely on an informal use of diagrams or natural language. Practitioners often use state machines [19], [20], [21]. We adopt this approach via RoboChart, and with tool support, via RoboTool, enforce precise use of notation with fixed meaning, and scalability. A complete description of RoboChart is in [25].

In RoboTool, we can define packages, like in Java, but the definitions in a package are graphical. The model of a robot can be composed of several packages, one of which must have the definition of a module, which models the robotic system as a whole by identifying a platform and one or more software controllers. The platform and the controllers themselves may be defined in other packages. A controller definition identifies the resources of a platform (variables, event, and operations) that it requires, and so is an independent component. A module connects controllers to a platform that provides the resources they require.

A platform specifies the requirements on the hardware via the definition of variables, events, and operations representing in-built facilities it must provide. To satisfy these requirements, it is often necessary to implement procedures that use the sensors and actuators. For instance, using a sensor that signals the presence of an obstacle, we can write a procedure that detects a threshold-crossing to raise an event for the state machine. Similarly, using actuators for wheels, a wrapper operation can be written to provide functionality to move the robot at a given speed. Although not required, operations of the platform may be defined in RoboChart by state machines that use events and variables of the platform.

Fig. 2 presents the module for our running example, the chemical detector. The platform is called SensorVehicle. It has no variables, but provides (P) the operations incr() and decr() to increment and decrement the speed of the vehicle. Since we do not further define these operations, we do not require a variable to record speed. The operations are grouped in an interface Speed provided (P) by the module, and defined in the same package. Interfaces group variables and operations either provided or required by a component.

SensorVehicle also provides events. The event gas corresponds to the chemical sensor that detects gas. Its input indicates that it gets information about the air composition. The event command corresponds to an input from a joystick-like interface to control the vehicle; its type Command indicates that it carries values of that type. The events flag, siren, and light correspond to actuators that control the mechanisms to alert to a change in background levels of gases; finally, turn corresponds to the communication to the hardware of a real value defining a degree and associated movement of the wheels to turn the vehicle by that degree.

A controller is specified by one or more parallel state machines. A controller typically corresponds to a processing unit in the architecture of the robot. Parallel state machines, on the other hand, are used to structure specification of functionality, and do not necessarily indicate parallelism in the software. For instance, the chemical detector is controlled by an on-board computer, so, we define two controllers shown in Fig. 2. MainComputer is the on-board computer processor concerned with gas detection and movement, and

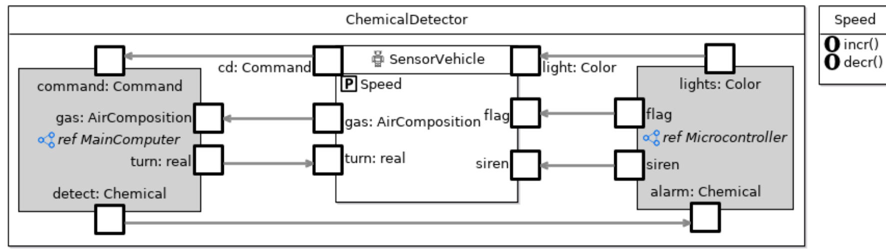

Fig. 2. Package System.rct for the chemical detector, showing the module ChemicalDetector and the interface Speed.

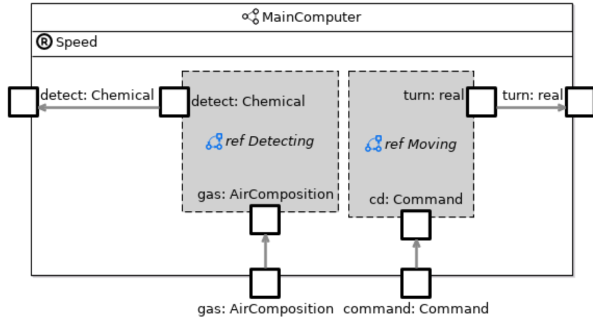

Fig. 3. The controller MainComputer for the on-board computer of the chemical detector. It is defined by two state machines, Detecting and Moving, and requires the interface Speed and four events (see Fig. 2).

the Microcontroller handles the signalling of gas.

The definition of the MainComputer, shown in Fig. 3, uses two state machines: Detecting and Moving. The first is concerned with gas detection, and the second with moving the robot. These are independent pieces of functionality.

A module also defines how communication flows between the controllers and the platform. The communication between controllers is either synchronous (default) or asynchronous. Asynchronous communication is indicated by the keyword async and is often used to model the interaction between distributed controllers. In our example, however, communication between the controllers is synchronous.

The RoboChart state machines are deliberately kept simple. The class diagram in Fig. 5 defines their structure. UML facilities like parallel states, inter-level transitions, and history junctions, which prevent mathematical definitions that support compositional and scalable automatic reasoning are left out. An example state machine is presented in Fig. 4; it defines the behaviour of the Microcontroller.

As shown in Fig. 5, a state machine, defined by the class StateMachineBody has a context, defining the variables, events, and operations it can use. A state machine can also define clocks used to define time restrictions. Importantly, a state machine is a NodeContainer: it contains nodes, that is, States and Junctions, and the Transitions between them. A Junction is like a State, but a Transition must be taken immediately from a junction. The Initial state is a Junction. A State can have Actions: entry, during, and exit actions, executed when the machine enters, is at, or exits the State.

The language used to define actions is precisely defined by a Statement. Besides standard facilities, like assignments and triggering events, it includes also time primitives to define budgets and deadlines. This is in line with the informal practice adopted in the robotics literature [22], [23]. Operations are assumed to take no time unless a budget is specified. Budgets can be precise, or define an interval of time. So, we have no implicit time properties: the time that may be taken by each operation is explicitly defined. On the other hand, events can happen at any time unless a deadline is specified: a maximum amount of time that can pass between the event being offered and it being accepted by the environment.

For instance, when the state DangerousGas is entered (see Fig. 4), the micro-controller raises an event light for the platform to turn it red, another event siren to sound the alarm, and then waits for a period defined by a constant flagTime, before raising an event flag, which must be accepted by the platform within a deadline defined by a constant dF ($<\{$dF$\}$). The use of wait(flagTime) defines a budget of flagTime time units for the actual operation of dropping a flag. The deadline $<\{$dF$\}$, however, specifies that the platform can only take up to another dF time units to complete the operation. In summary, we use the event flag to represent the *completion* of the actual drop of the flag by its actuator: we record that this operation takes at least flagTime units, but no more than flagTime+dF. An alternative model could represent the interaction with the flag actuator as an operation with a budget between flagTime and flagTime+dF time units using wait([flagTime,flagTime+dF]). The fact that we do not define a budget for the operations to set a light colour or sound a siren indicates that they can take a negligible amount of time. Without a deadline, however, we impose no restrictions on how long it can be.

A Transition connects a source and a target State. It can have a Trigger: an event that needs to occur to enable the transition. It can also have two Expressions. One is a boolean condition, called a guard: a transition can only be taken when it is true. The other is a deadline on the Trigger event, if it is present. The deadline establishes the maximum amount of time that can pass before the Trigger can take place, unless that machine takes a different Transition before that. Finally, a Transition can have a ClockReset, which allows conditions to refer to the most recent time a clock C has been reset through the expression since(C). Examples of the use of a clock are provided in Section V-C.

RoboTool implements the classes defined in Fig. 5 and

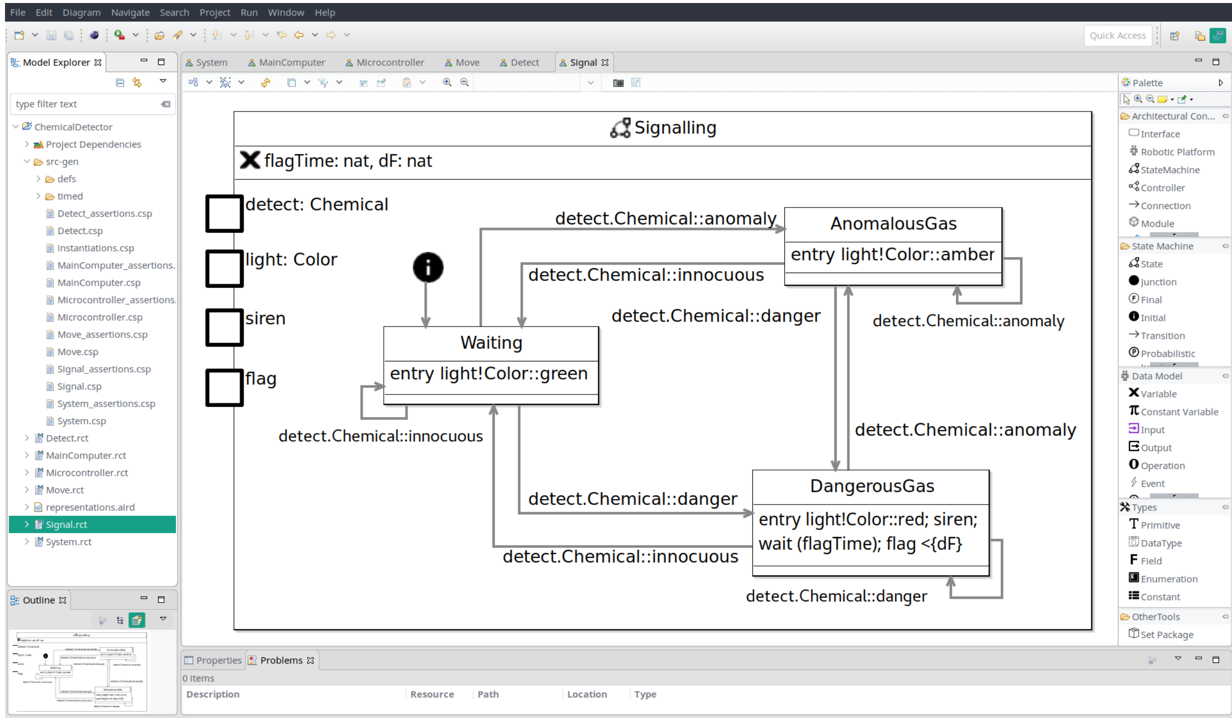

Fig. 4. Signalling state machine that defines the behaviour of Microcontroller in RoboTool. The panel on the left shows all packages of this example. The central panel shows the diagrammatic version of the state machine in package Signal.rct. On the right panel we have the RoboTool palette of elements that compose a RoboChart model. At the bottom panel, any problems with validation are listed.

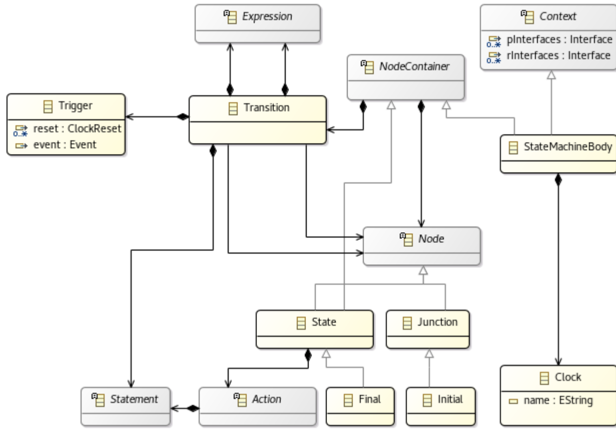

Fig. 5. A class diagram of the RobChart state machines.

others to cater for modules and controllers. Most importantly, as shown on the left panel in Fig. 4, in a directory src-gen, it generates automatically mathematical definitions in .csp files. We discuss these in the next section.

IV. Property Checking

Mathematical definitions for a RoboChart model describe restrictions on the order and availability of the events and operations of the platform. In effect, they characterise the observable behaviour of the robot, which is captured by these events and operations. For example, in the chemical detector they constrain the order and availability of the flag drop and of calls incr() to the speed increase operation (as well as of the other events and calls to operations of SensorVehicle).

In these definitions we use what is called a process algebra, particularly, CSP [24]. In this notation, the definitions are a sequence of equations, each defining a process. A CSP process characterises behaviour by defining order and availability of events, therefore, we define both events and operation calls of RoboChart as CSP events.

In RoboTool, the src-gen folder contains a subfolder defs with automatically generated mathematical definitions for each module, controller, state machine, and operation in a separate .csp file. For each of these, we also have a file with suffix _assertions.csp, including checks of deadlock, livelock, nondeterminism, and timelock. A timelock arises when a process refuses to let time pass, indicating some deadline is infeasible. There are also assertions to find unreachable states, which could stem, for example, from unsatisfiable conditions on transitions. Finally, for each package, there is a file with suffix _defs.csp containing CSP definitions of the types in the package. These files are used to construct automatically the definitions of the packages.

For each package, src-gen includes a .csp file importing the corresponding _defs.csp file, and the files with the definitions and checks of the modules, controllers, state machines, and operations used in that package. Therefore, we avoid circular imports and reimports, forbidden in CSP.

Proof of properties should use the .csp file for a package in src-gen; by clicking on such a file, the CSP model checker FDR4 is invoked by RoboTool. With one click in FDR,

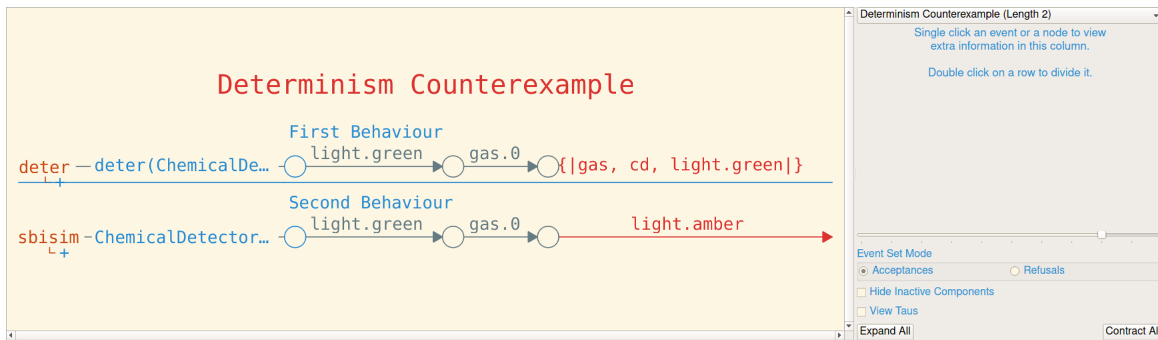

Fig. 6. Checking the module ChemicalDetector. It is not deterministic, as shown by the counterexample.

we trigger the check of the core properties for the module, controllers, and state machines.

Determinism is not essential: it is perfectly valid to define nondeterministic models. A typical example is a model in which more than one transition without a guard can be enabled in a particular state, and either at least two of these transitions are associated with the same event or one of them has no event, and they lead to different states. Proof of properties of such a model needs to consider all the possible behaviours arising from nondeterminism.

We note that events between controllers are not visible by a user. So, transitions associated with these events are like transitions without events and may also lead to nondeterminism. This reflects a system view of the robot: its visible behaviour is that of its platform, not the internal communications of its software components. This means that any property proved is respected by a correct implementation that, for example, uses just one controller or a different set of controllers. The RoboChart model specifies behaviour that can be implemented using a different software architecture.

The chemical detector, for instance, is nondeterministic, as indicated by FDR in Fig. 6. The assertion for determinism fails and we get a counterexample. It shows that, ChemicalDetector, after the event light.green, which is part of the entry action of the first state of Signalling (see Fig. 4) and a gas reading, it is able to accept a command via cd, or another gas reading, or set the light to green again. ChemicalDetector is, however, also able to set the light to amber. We have a nondeterminism in that the robot may accept or refuse to set the light to green. It arises because we do not fully specify the algorithm for gas detection, and the communication between the controllers (see Fig. 2) is not visible to the robot; it is just part of the protocol that governs the internal interaction of the controllers.

The .csp files for the packages only have the imports; the actual definitions are in the files in the defs directory. The definition for a module's behaviour is the conjunction of definitions of the controllers. The platform does not exhibit any prescribed behaviour: it only defines the observable events and operations. If there is any communication between the controllers, as is the case in the chemical detector, the connected events between them, need to be identified.

For illustration, we sketch the definition of the ChemicalDetector module below.

```
ChemicalDetector =
  (MainComputer[[...]]
    [|{|Microcontroller_alarm|}|]
   Microcontroller[[..]])
  \ {|Microcontroller_alarm|}
```

It includes the processes that define the controllers `MainComputer` and `Microcontroller`, and establishes that they communicate via an event `Microcontroller_alarm`. We note that, as shown in Fig. 2, in the module they are connected via an event detect in the MainComputer and alarm in the Microcontroller. Above, these events are identified by calling both of them `Microcontroller_alarm`. Moreover, because, as already mentioned, communications between controllers are not visible to the users of the robot, this event is hidden using `\ {|Microcontroller_alarm|}` in CSP.

Identification of events is achieved via renaming, indicated by `[[...]]` above. These renamings establish the connections in the module. For example, the omitted renaming pair `MainComputer_turn <- turn` identifies the `turn` events of MainComputer and the platform. This is needed because each controller, state machine, or operation, is an independent component, with an independent definition. So the events called turn in the platform and in MainComputer are different and independent; connection is achieved in the module. Like in the case of detect and alarm, even events with different names, but of the same type, can be connected.

Time aspects of RoboChart models are defined using a discrete-time version of CSP that marks passage of time using the CSP event `tock`. For example, the budget expressed in RoboChart as wait(flagTime) ; flag is defined by a sequence of as many `tock` events as specified by flagTime followed by the CSP process defining the statement flag. Deadlines are encoded by failing to offer `tock` after they elapse. Time is uniform across CSP processes.

Before verifications are carried out, all constants, like flagTime need to be given a value. Types also need to be given finite definitions. A file called instantiations.csp contains default definitions for all of them as simple equalities (like `flagTime = 2` and `nametype nat = {0..2}`. They can be changed, if needed, by editing this file.

Finally, RoboTool generates also optimised versions of the processes. These optimised processes are the versions used in the assertions: processes with suffix _O. Using the optimisation facilities of FDR, we can obtain CSP processes with the same number of states of the RoboChart state machines and are, therefore, confident in scalability.

Besides the core checks automatically generated, we can check application-specific properties. For example, for the chemical detector, we can show that commands result in an event turn, or calls incr() or decr(). We can also prove that every gas event must eventually be followed by a light event, but up to two gas events may take place before a light event must be raised. A file with suffix _assertions.csp is generated for each package, where the relevant checks can be written and kept. Neither instantiations.csp nor the _assertions.csp files are regenerated, once they are updated, to avoid losing any definitions and checks.

For efficiency, besides verifying a whole module, we can verify the processes for individual controllers, state machines, or operations. Our experience with the chemical detector and other larger and realistic examples is presented in the following section. A complete description of how all the mathematical definitions are generated is in [25].

V. Examples

In the following sections, we discuss the results of the analysis of various examples. For each of them, we have identified requirements, expressed them in either RoboChart itself or CSP, and carried out proof. Details are in [26] and models in www.cs.york.ac.uk/circus/RoboCalc.

A. Tele-operated chemical detector

The original model was developed in-house, and checked with the robot developers. Analysis of the original model uncovered a few modelling mistakes and contributed to the development of the corrected model discussed in Section III.

In our verification, with the core checks, we observed that the state machine for Detecting the gas was deterministic. Although the algorithm that detects the gas is deterministic, because we do not specify it in RoboChart, we must consider that it is possible for that algorithm to generate any valid output. Therefore, Detecting cannot be deterministic. Further review identified that the event detect was erroneously used in the triggers of transitions out of the state of Detecting in which the analysis is carried out. This meant that a component interacting with Detecting could choose the value communicated via detect. This is undesirable, since detect is an output. Instead, we need to have transitions without a trigger, but with an action that defines the value communicated. In this way, the output communicated via detect cannot be determined by another component, but only after a transition is chosen. As explained above, this choice is nondeterministically made by Detecting because the transitions do not have a trigger nor a guard. This is an easy modelling mistake to make, and affects the quality of the model, but can be checked automatically.

More interestingly, we verified that the requirement "the chemical detector shall always accept a gas read" was not satisfied. The verification showed that there was a deadlock after the trace [light.green, gas]; this suggested investigating extensions of this trace and led to the discovery that [light.green, gas.a, light.green], for any air composition a, was not a valid trace. An evaluation of the model based on this trace led to the observation that the state Waiting of Signalling (see Fig. 4) did not accept the event detect.innocuous. Similarly, there were also missing transitions in the states AnomalousGas and DangerousGas. Simulations and programs based on the original model would, therefore, have mistaken behaviour.

Verification of the above requirement uses the following deadlock assertion in CSP.

```
assert ChemicalDetector_O
       [|{|cd|}|]
       STOP :[deadlock free]
```

The process analysed behaves like the optimised module (Fig. 2) process ChemicalDetector_O, but has all interactions on cd blocked (by the CSP process STOP). So, this requires that, even if the human operator gives no further commands, the gas reads is still possible.

We also verified that the requirement "whenever the siren is triggered, the flag should be dropped within t time units" can be satisfied as long as $\mathsf{flagTime} + \mathsf{dF} \leq \mathsf{t}$, that is the deadline $\mathsf{flagTime} + \mathsf{dF}$ for dropping the flag is less than t. Requiring that "whenever a dangerous gas is detected the flag is dropped within a specified time" is not possible unless deadlines on the events light and siren, which take place before the flag event, are specified.

B. Autonomous chemical detector

In this section, we discuss the verification of an autonomous version of the chemical detector example, which is currently being prototyped. In its model, movement involves a random walk to search for gas and autonomous obstacle avoidance. A complete account is in [26].

For the analysis of gas, in this version, we use a function analysis that takes the input from four gas sensors and determines whether there is a gas of interest or not. In this case, we declare the function analysis and its type, but do not specify it further, since we do not plan to verify this algorithm. We also use a function location, which defines an angle for the robot to turn, based on the intensity of gas indicated by each of the sensors. We define this function precisely, but use conditions to define implicitly the value of its output: the angle of the sensor giving the highest intensity. For model checking, just like for simulation, in both cases, we need CSP specifications of these functions.

Simple definitions, like analysis(gs) = noGas, are automatically generated. Here, RoboTool chooses an element of the return type, in this case noGas, and defines analysis as the constant function whose value is that element for all arguments. A more useful definition can be provided using equations. For example, we use the following definition.

```
analysis(<>) = noGas
analysis(<g>^gs) =
  if (GasSensor_c(g) == 0
      and analysis(gs) == noGas)
  then noGas else gasD
```

The input is an array of gas sensors. For the empty array <>, the result is `noGas`. For an array whose first element is the sensor `g` followed by the elements in the array `gs`, if `g` gives a 0 reading (`GasSensor_c(g) == 0`) and the `analysis` of the other sensors `gs` indicates `noGas`, then the result is `noGas`. Otherwise, the `analysis` indicates that gas has been detected: `gasD`. This definition can be added in instantiations.csp to replace the default definition.

In this example, the robot stops once it reaches the gas source and drops the flag. A system that terminates is not deadlock free, but we need to check that it deadlocks just when it terminates. So, we check for deadlock freedom, not of ChemicalDetector_O, but its conjunction with a process that, once the event `flag` happens, goes on forever without deadlocking. The overall process no longer terminates, so if it is not deadlock free, then the robot has a unwanted deadlock, that is, a deadlock that arises not due to termination. We indeed found such a deadlock in our model, which uncovered missing transitions in a state machine.

Another application-specific requirement is that every command to move the robot (resume, stop, or turn) leads to a reaction by the robot, before another command is issued. The verification of this property failed. The counterexample showed that the problem is that, if the main computer does not find gas, it keeps sending the command resume to the micro-controller, so that it continues a random walk. It was, therefore, possible that the micro-controller is continuously interrupted by commands resume and not have the opportunity to actually carry out the random walk. This revealed a hidden assumption of the modeller: the gas analysis is much slower than the random walk operation, and so this cannot happen. The model needed to be enriched with time information to record this assumption.

The reason two resume events can happen in sequence is because the operation randomWalk is called in the during action, which means the potential reaction to resume can immediately be interrupted by another resume event. This violation is due to missing timing information regarding the occurrences and processing of gas readings.

Because of the number of sensors, the number of states in this example is significant. Although we could carry out our verification, we were motivated to consider optimisations of the RoboChart model. A common modelling pattern is the use of a variable to record the result of a function call, for example, in an entry action, and subsequent use of that variable, instead of the function call, in further actions and transitions from the same state. If that variable is global in the RoboChart model, it becomes part of the state of its CSP process and slows down the model checking. We have changed our original model to remove these variables, and proved that the two models are equivalent, using their

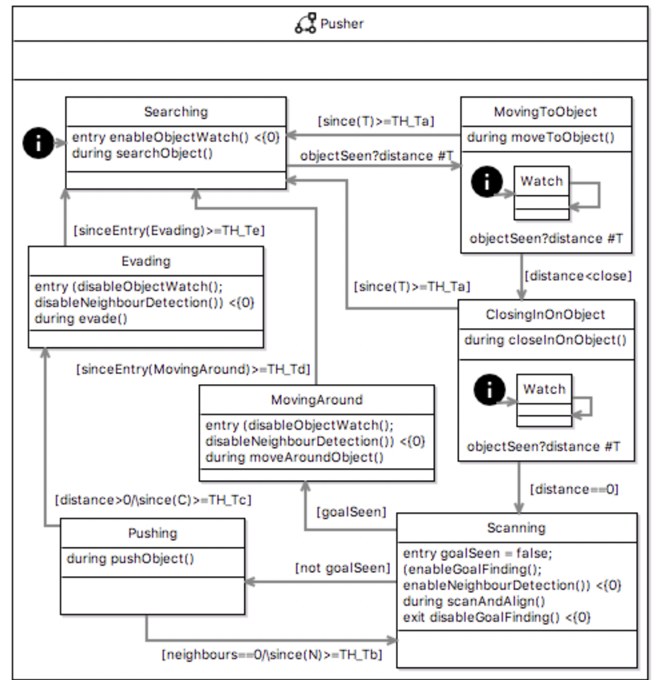

Fig. 7. Transport swarm state-machine.

definitions generated by RoboTool. The time taken to carry out the proofs does not change much, but the time to compile the CSP process for the optimised model is radically shorter.

C. *Transporter*

In this section, we summarize our analysis of individual robots in an occlusion-based transport swarm used to push tall objects towards a goal. The RoboChart model shown in Fig. 7 is derived from the description in [12], and a complete account of this case study can be found in [26]. This is a richer example where we make full use of the timed primitives of RoboChart. For instance, in this example the robot resumes Searching whenever timed thresholds related to the last time an object was seen elapse. We model this by resetting (#T) clock T whenever an objectSeen is triggered, thus allowing transitions from states MovingToObject and ClosingInOnObject to be constrained relative to #T using an expression comparing since(T) and a constant TH_Ta.

The main controller was found to be nondeterministic due to cases where several transitions are enabled at the same time, particularly some with no trigger but guarded by timed conditions. For example, the transition from the state MovingToObject to the state Searching, when enabled competes with the self-transition in the sub-state Watch of MovingToObject. To eliminate this nondeterminism, competing transitions need to have a negated timed constraint, so that whenever a transition with no trigger, but constrained by time is enabled, no other transitions are enabled.

We also found that it was possible for a critical operation scanAndAlign, called within a state Scanning, and used for the robot to align itself alongside an object before pushing it, to be interrupted immediately. This potentially erroneous

behaviour stems from an implicit assumption in [12] revealed by our verification, where Scanning should not be exited before some significant amount of time elapses.

VI. Conclusions

We have presented RoboTool and its notation, RoboChart. They enable automatic generation of mathematical definitions that support automatic proof of key properties of robotic controllers. We have applied this technology to several examples. Besides those discussed here, we have models for the alpha algorithm, a humanoid, the relay chain algorithm for AUV in [27], and other smaller examples. Our work complements others in the literature. For example, [28] presents a graphical notation to depict component-based architectures of software product lines for robotics; RoboChart could be employed to define the behaviour of the components in these models. Another state-machine notation for robotics is in [29]; many of its decisions on including or excluding UML and statechart features are similar to those in the design of RoboChart, and so our approach to proof may be useful for those models.

When compared to other robotic notations, RoboChart is distinctive in its treatment of time properties (budgets and deadlines) and support for proof. The checks have been very efficient. For the untimed analysis, the most expensive in terms of time was the autonomous chemical detector. The timed analysis takes longer because the extra `tock` events leads to an explosion in the number of states of models of the machines. This is because each extra time step becomes a new state in the CSP model.

In continuing with our work, we will pursue extensions to deal with probabilistic properties and physical models using results on probabilistic UML [30] and PRISM for model checking [31], as well as hybrid models, with continuous variables, to model the platform and environment [32].

Acknowledgement

We thank Abdulrazaq Abba, Augusto Sampaio, Jim Woodcock and anonymous referees for good suggestions. This work is funded by the EPSRC (EP/M025756/1).

References

[1] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. Jonsson, "Verification of autonomous systems for space applications," in *IEEE Aerospace Conference*, 2006, pp. 1–11.

[2] N. Hochgeschwender et al, "A model-based approach to software deployment in robotics," in *IROS*, 2013, pp. 3907–3914.

[3] S. Alexandrova, Z. Tatlock, and M. Cakmak, "Roboflow: A flow-based visual programming language for mobile manipulation tasks," in *ICRA*, 2015, pp. 5537–5544.

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

[5] S. Dhoui et al, *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012, ch. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pp. 149–160.

[6] O. M. Group, "OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1," August 2011. [Online].

[7] S. G. Brunner et al, "Rafcon: A graphical tool for engineering complex, robotic tasks," in *IROS*, 2016, pp. 3283–3290.

[8] A. Mallet et al, "Genom3: Building middleware-independent robotic components," in *2010 ICRA*, 2010, pp. 4627–4632.

[9] W. Li, A. Miyazawa, P. Ribeiro, A. L. C. Cavalcanti, J. C. P. Woodcock, and J. Timmis, *From formalised state machines to implementation of robotic controllers*, ser. Springer Tracts in Advanced Robotics. Springer, 2016.

[10] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3 A Modern Refinement Checker for CSP," in *TACAS*, 2014, pp. 187–201.

[11] J. A. Hilder et al, "Chemical detection using the receptor density algorithm," *IEEE TSMC*, vol. 42, no. 6, pp. 1730–1741, 2012.

[12] J. Chen, M. Gauci, and R. Gross, "A strategy for transporting tall objects with a swarm of miniature mobile robots," in *ICRA*, 2013, pp. 863–869.

[13] A. Nordmann et al, "A survey on domain-specific modeling and languages in robotics," *JSER*, vol. 7, no. 1, pp. 75–99, 2016.

[14] M. Foughali etal, "Model checking real-time properties on the functional layer of autonomous robots," in *ICFEM*, 2016.

[15] T. Abdellatif et al, "Rigorous design of robot software: A formal component-based approach," *Robotics and Autonomous Systems*, vol. 60, no. 12, pp. 1563–1578, 2012.

[16] F. Fleurey and A. Solberg, "A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems," in *MoDELS*. Springer-Verlag, 2009, pp. 606–621.

[17] K. Kapellos, D. Simon, M. Jourdant, and B. Espiau, "Task level specification and formal verification of robotics control systems: State of the art and case study," *International Journal of Systems Science*, vol. 30, no. 11, pp. 1227–1245, 1999.

[18] M. Wachter et al, "The armarx statechart concept: Graphical programing of robot behavior," *Frontiers in Robotics and AI*, vol. 3, p. 33, 2016.

[19] H. W. Park, A. Ramezani, and J. W. Grizzle, "A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking," *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 331–345, 2013.

[20] C. A. Rabbath, "A finite-state machine for collaborative airlift with a formation of unmanned air vehicles," *Journal of Intelligent & Robotic Systems*, vol. 70, no. 1, pp. 233–253, 2013.

[21] T. Tomic et al, "Toward a fully autonomous uav: Research platform for indoor and outdoor urban search and rescue," *IEEE Robotics Automation Magazine*, vol. 19, no. 3, pp. 46–56, 2012.

[22] W. Liu and A. F. T. Winfield, "Modeling and optimization of adaptive foraging in swarm robotic systems," *International Journal of Robotics Research*, vol. 29, no. 14, pp. 1743–1760, 2010.

[23] G. Pini et al, "Task partitioning in a robot swarm: Object retrieval as a sequence of subtasks with direct object transfer," *Artificial Life*, vol. 20, no. 3, pp. 291–317, 2014.

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

[25] A. Miyazawa et al, "RoboChart: a State-Machine Notation for Modelling and Verification of Mobile and Autonomous Robots," University of York, Department of Computer Science, York, UK, Tech. Rep., 2016.

[26] A. Miyazawa, A. Cavalcanti, P. Ribeiro, W. Li, and J. Timmis, "RoboCalc Case Studies," 2016. [Online]. Available: www.cs.york.ac.uk/circus/RoboCalc/case-studies/

[27] B. Naylor et al, "The Relay Chain: A Scalable Dynamic Communication link between an Exploratory Underwater Shoal and a Surface Vehicle," 2014.

[28] D. Brugali and L. Gherardi, *HyperFlex: A Model Driven Toolchain forDesigning and Configuring Software Control Systems for Autonomous Robots*. Springer, 2016, pp. 509–534.

[29] M. Klotzbucher and H. Bruyninckx, "Coordinating Robotic Tasks and Systems with rFSM Statecharts," *JSER*, vol. 2, no. 13, pp. 28–56, 2012.

[30] D. N. Jansen, H. Hermanns, and J.-P. Katoen, "A Probabilistic Extension of UML Statecharts," in *FTRTFT*, ser. LNCS, vol. 2469. Springer, 2002, pp. 355–374.

[31] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: a hybrid approach," *STTT*, vol. 6, no. 2, pp. 128–142, 2004.

[32] S. Foster and J. C. P. Woodcock, "Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL," in *Concurrency, Security, and Puzzles*, ser. LNCS, vol. 10160. Springer, 2017, pp. 39–64.