
Proceedings Paper:
Cavalcanti, A. and Hierons, R.M. orcid.org/0000-0002-4771-1446 (2023) Challenges in testing of cyclic systems. In: Aït-Ameur, Y., Khendek, F. and Méry, D., (eds.) 2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS). 27th International Conference on Engineering of Complex Computer Systems, 14-16 Jun 2023, Toulouse, France. Institute of Electrical and Electronics Engineers (IEEE) ISBN 979-8-3503-4004-4

https://doi.org/10.1109/iceccs59891.2023.00010

Challenges in testing of cyclic systems

Ana Cavalcanti
Department of Computer Science
University of York
York, UK
0000-0002-0831-1976

Robert M. Hierons
Department of Computer Science
University of Sheffield
Sheffield, UK
0000-0002-4771-1446

***Abstract*—The state of practice in design and verification of control software for robotics is code centric. The RoboStar framework supports a model-based approach, providing support for modelling and simulation, and techniques for automatic generation of artefacts. Existing results support test generation using a reactive design model; in RoboStar such models can be described using a diagrammatic notation called RoboChart. Here, we describe the challenges involved in using such tests for execution against simulations or cyclic implementations either automatically generated or custom developed. While it is possible to use a cyclic model to generate tests in the first place, reactive models are akin to those normally used by the community. Moreover, by linking design-based tests to the tests executed against the cyclic mechanisms, we support traceability.**

***Index Terms*—RoboStar, test generation, formal testing, simulation**

I. Introduction

The state of practice of testing in robotics is ad hoc. The modern outlook, however, is on safety-critical, real-time robotic applications operating in an uncertain environment; these features make productive testing difficult, in terms of achieving coverage, of observability, and of reaching reliable conclusions. Recent experience reports [1], [2] recognise these challenges, but indicate the existence of a culture of even not believing in the value of testing. If we are, however, to realise the potential of robotics, for the benefit of our society, trustworthiness, and therefore verification, is paramount.

Two key questions are left unanswered by an ad hoc approach to testing. (1) Effectiveness: if we do not find a fault, what can we conclude? (2) Soundness: if we do find a fault, is it a problem in the robotic system or in the test? Both questions are hard, as they amount to deciding if enough tests have been run, and if feasible inputs are being considered with the right verdict. Model-based testing addresses both questions.

In the model-based testing approach, tests are generated using a model of the robotic system. Such an approach allows much of testing to be automated [3], leading to significant cost reductions [4], and supports strong statements on test effectiveness. In particular, the tester can use a test suite that determines correctness in certain conditions [5].

Previous work has presented support for automatic test generation using RoboChart [6], a domain-specific notation for modelling and verification of control software for robotic systems. RoboChart uses state machines and a simple component model, enriched with timed constructs, to specify a timed platform-independent behavioural model for the software. RoboChart has a formal semantics, defined using a discrete-time variant [7] of the process algebra CSP [8], namely, *tock*-CSP. Support for modelling, model checking, as well as test generation is available via a tool called RoboTool[1].

A RoboChart model characterises the reactive behaviour of a robotic control software in terms of interactions representing services of the robotic platform. Roughly speaking, these are outputs of sensors (inputs to the software) and inputs to actuators (outputs of the software) providing information and affecting the environment of the robotic system. In the *tock*-CSP semantics, we use CSP events to represent the services of the RoboChart model, with the additional special *tock*-CSP event called *tock* representing passage of time.

The *tock*-CSP testing theory [9] defines how to construct tests from minimal forbidden traces of a process, including input and output events and *tock*. RoboTool generates such traces automatically. To illustrate the ideas, we consider a simple ranger robot that moves about in an arena, avoiding obstacles. To model this robot in RoboChart, we can use a RoboChart event obstacle to represent the input from an infrared sensor, for instance, and operations move(l,a) and stop() to represent a motor or set of motors with an API that implements facilities to set the linear and angular velocities l and a of the robot, and to stop it. For such a model, RoboTool can generate traces such as $\langle\, move.out.lv.0, tock, obstacle.in, move.out.lv.0 \,\rangle$, where tags *.in* and *.out* distinguish inputs and outputs. This trace records an immediate operation call move(lv,0) (first event $move.out.lv.0$), followed by the passage of one time unit (*tock*), followed by an input that flags an obstacle ($obstacle.in$). The last event $move.out.lv.0$ is forbidden, since the RoboChart model defines that after an obstacle is detected, we do not set the robot to move(lv,0) in a straight line.

The shape of the tests derived from a forbidden trace is defined by the testing theory. Roughly speaking, the test drives the system under test (SUT), here a robotic control software, via the trace, and then attempts the forbidden event. If the SUT cannot be driven to just before the forbidden event, the

This work has been funded by the UK EPSRC Grants EP/M025756/1, EP/R025479/1, and EP/V026801/2, and by the UK Royal Academy of Engineering Grant No CiET1718/45.

[1] https://robostar.cs.york.ac.uk/robotool/

test is inconclusive. If the SUT can be driven to that point, but the forbidden event does not occur, the test passes. If the forbidden event is observed, the test fails.

The SUT is a proposed implementation of the robotic control software. Such implementations are typically cyclic mechanisms, particularly so in the case of simulations. RoboChart models are akin to those used in the robotics literature [10]–[13]. They are, however, reactive, not cyclic models.

In each cycle, a simulation reads the inputs, corresponding to registers of the sensors, executes computations to process those inputs and calculate outputs, and then writes the outputs to the registers of the actuators. A simulation takes an idealised view of time: input, computation, and output are infinitely fast at the sample times, defined by the period, that is, the size of the cycle, and then no input, computation, or output occurs while the time advances to the next sample time.

Unsurprisingly perhaps, using a reactive design model to judge the correctness of a simulation (or of a cyclic implementation) must be done in a context where the assumptions embedded in the cyclic paradigm are taken into account. For instance, if in the reactive RoboChart model there is an immediate response to the obstacle event, perhaps in the form of a call to the stop() operation, this cannot be necessarily reflected in a simulation. As explained above, a simulation does not perceive any inputs during the quiescence period. Therefore, correctness can only be established if we assume that events only happen at the sample times of the simulation.

We have previously defined (and formalised in *tock*-CSP) a conformance relation that can be used to compare reactive and cyclic models [14]. Our work is in the context of cyclic (simulation) models defined using another diagrammatic domain-specific language, called RoboSim. RoboChart and RoboSim are part of the RoboStar [15] framework for model-based design and verification of control software in robotics.

Just like conformance between a reactive and a cyclic model needs to take into account the assumptions of the cyclic paradigm, test generation and execution also need to be adapted. Tests generated from a RoboChart model cannot be directly used to test a cyclic SUT. For example, the test for the trace presented above starts with an observation of a call move(lv,0). In a simulation, however, such an output cannot be observed until all the inputs are read. So, use of a straightforward implementation of the test always leads to an inconclusive verdict. For the simulation, the test needs to provide all inputs before any output can be observed.

One possible solution is to use RoboSim, rather than RoboChart, to generate tests. There is, however, value in understanding tests results in the context of the design of the software. So, we discuss here the challenges involved in connecting the tests generated using RoboChart to tests that can be run against a simulation or cyclic implementation.

Next section, we present in more detail the discrepancies between reactive and cyclic models, using RoboChart and RoboSim in examples. In Section III, we discuss the possibility of test conversion as an approach to deal with such challenges. In Section IV we indicate an agenda for future work.

II. Reactive and cyclic models

In Fig. 1, we present the RoboChart model for the simple ranger described above. Two small boxes at the top, labelled MovementI and ObstacleI, declare the operations and event representing the services of the robotic platform already described. Control software is modelled in RoboChart using a component called module. In the example, it is represented by the box CMovement on the lower right-hand side corner. A module includes a robotic-platform block, here called FootBot, to represent the services of the platform needed by the software. In the example, FootBot declares MovementI and ObstacleI to record that the visible behaviour of the software is defined in terms of uses of these services.

A robotic platform can be connected to one or more controllers, running in parallel, whose behaviour is defined by one or more state machines, also running in parallel. In our example, we have just the controller Movement which uses the machine SMovement to define its behaviour.

A RoboChart machine is similar in many ways to a UML machine, for instance, but includes, first of all, a context of declarations that define the variables, constants, clocks, events, and operations used by the machine. RoboChart also has a well-defined action language, and imposes simple restrictions, such as absence of inter-level transitions, to enable definition of a tractable compositional semantics. In addition, RoboChart machines can specify time budgets and deadlines.

SMovement starts in the state Moving, where the call move(lv,0) is urgently made upon entry. After that call, SMovement pauses for one time unit, using the action wait(1), to allow time for the robot to start moving. Afterwards, SMovement remains in Moving until an obstacle is detected, when a transition to another state Turning is enabled and immediately taken. Enabled transitions are urgent for predictability. In that transition, a clock #MBC is reset, stop() is called, and the robot pauses for another time unit.

In Turning, we have an urgent call move(0,av), a pause for one time unit, and then for further time until the transition out of Turning is enabled. This transition does not have a trigger, but has a guard, which requires the value of the clock (since(MBC)) to be high enough for the robot to have had time to turn. The transition then leads back to Moving.

Although SMovement has a form of cycle, from Moving to Turning and back, that is not a simulation cycle. First, it has no fixed period: it is triggered by an obstacle detection. In contrast, a simulation cycle determines when sensors are read, and so it needs to be very fast. The RoboChart models completely abstracts away from this cyclic behaviour, and how its actions are allocated along these short cycles.

In addition, events in a simulation are treated as data. For instance, in a state with transitions triggered by two different events e1 and e2, if both events occur, a simulation typically prioritises one of them. This is achieved by handling the event with lower priority only when that of higher priority does not occur. So, it needs to specify behaviour in the absence of an event, something that is not possible in a reactive model.

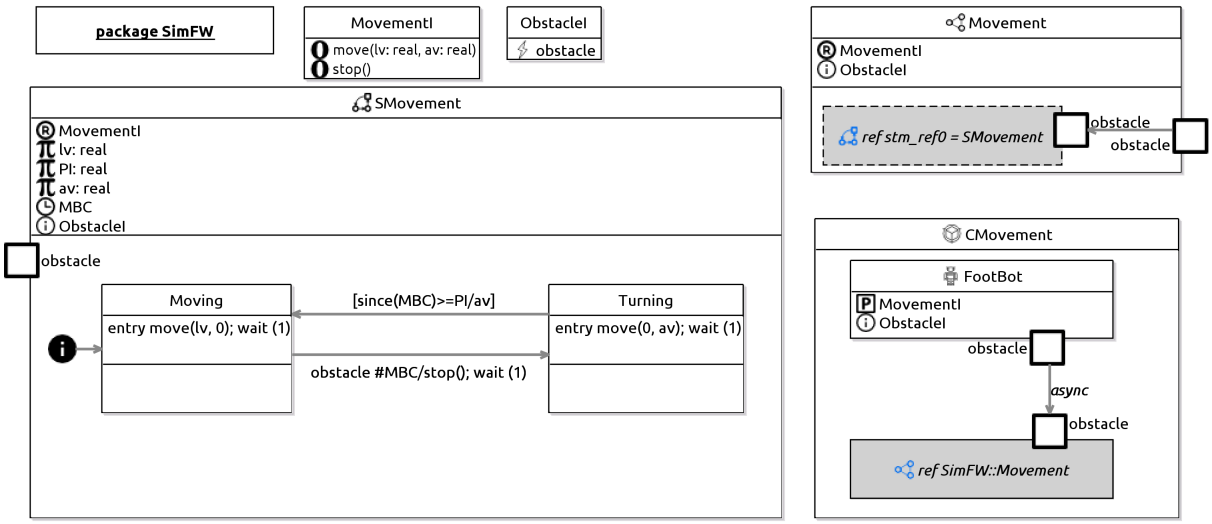

Fig. 1. A RoboChart model for a small ranger robot. It is defined in a package called SimFW.

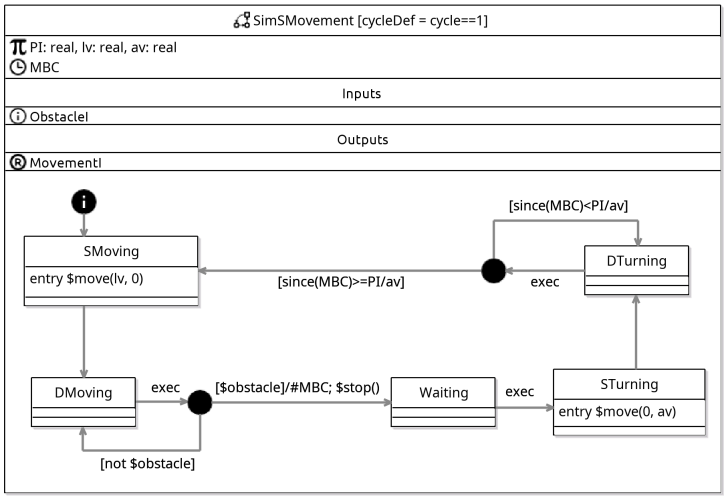

Fig. 2. A RoboSim state machine for the small ranger robot.

To model cyclic mechanisms, RoboStar provides RoboSim, a diagrammatic notation that is similar to RoboChart, but embeds the cyclic paradigm, where event occurrences are captured by Boolean variables, associated with additional variables when they communicate values. The definition of the cycle period and the scheduling for each cycle are explicit in RoboSim models. In Fig. 2, we present the RoboSim machine SimSMovement for our example. The module and controller from the RoboChart model can be adopted almost as they are, except only that in RoboSim they define a period.

In a RoboSim state machine, we also define a period (in a cycleDef clause), the context of variables, constants, and clocks, and then the inputs and outputs separately. For example, in SimSMovement the period is 1, and the declarations indicate obstacle as an input, and the operations are outputs.

A RoboSim machine uses a single event, called exec, to mark the end of the processing phase (at the sample time) of the cycle. It also uses a $ to distinguish the Boolean variables representing events and the calls to operations of the platform. In SimMovement, the control flow starts in the state SMoving, whose entry action calls move(lv,0), and then moves to a state DMoving. There, a transition with an exec trigger goes to a junction. As mentioned, with the exec, the processing phase of the cycle ends, and time then advances to the start of the next cycle. At that point, one of a pair of transitions from the junction is taken, depending on whether $obstacle is true or not, that is, depending on whether, at the start of the cycle, the input obstacle has been flagged as having happened or not. If it has not ($not obstacle), then the transition leads back to DMoving, where another exec event advances the cycle. Otherwise, the clock MBC is reset and $stop() is called in the actions of the transition with guard $obstacle. There is no point in checking an input twice in the

same cycle: a new value is read once per cycle, as mentioned. Also, there is no support for actions such as wait(1), since passage of time is dictated by the cycle evolution.

The *tock*-CSP semantics of RoboChart and RoboSim reflect the differences between the reactive and cyclic control flows. A *tock*-CSP model that defines the semantics of a RoboChart model uses CSP events to represent the services of the robotic platform. For example, $\langle\, move.out.lv.0, tock, obstacle.in \,\rangle$ is a trace of the RoboChart model of our ranger in Fig. 1. Inputs and outputs are distinguished in the semantics via tag values *in* and *out* for the CSP events as previously indicated. In contrast, in the semantics of RoboSim, the *tock*-CSP process uses events *read* and *write* to represent the interaction points via registers of sensors and actuators in the processing phase. For instance, for SimMovement, we have a trace $\langle read.obstacle.false, write.move.lv.0, tock, read.obstacle.true\rangle$, recording that in the first cycle the event obstacle has not happened, then the call to move is output, before time passes and we move to the next cycle and read the input again.

In the next sections, we discuss the challenges of comparing a RoboChart and a RoboSim models via testing.

III. Test conversion

RoboTool implements a mutation-based technique for automatic generation of tests from a RoboChart model [16]. In this approach, mutation operators (to remove or change transitions, for instance) are applied to the RoboChart model, generating a series of mutated models. There are several mutation operators available in RoboTool, and support for the definition of more. For each operator, several mutated models can be generated, by applying that operator at several points of the model: all its transitions, for example. Based on a mutated model, model checking can be used to generate a trace of the original RoboChart model, followed by an event that is not allowed by that model. This trace, including the last forbidden event, is a minimal forbidden trace of the RoboChart original model that reveals the visible error that arises from the mutation.

The *tock*-CSP testing theory defines how to construct a test from such a trace. It also guarantees that such tests are sound, that is, they give the correct verdict, and if we consider all possible minimal forbidden traces, the resulting test set can determine whether any SUT is correct or not, with respect to traces refinement (in the *tock*-CSP semantics). The testing theory caters for the distinction between inputs and outputs, and, accordingly, as said, RoboTool traces indicate implicitly, via the *in* and *out* tags, the input and output events.

To use a test derived from a RoboChart model for experimenting with an SUT that is posed as a cyclic implementation of that model, we need to convert the test, as already illustrated. For that, we propose conversion of the minimal forbidden traces of the RoboChart model that defines the test. With the new converted trace, we can then use the *tock*-CSP testing theory to define the converted test.

To explain the issues involved in conversion, we consider that the SUT can be described by an unknown RoboSim model. This kind of assumption is standard in testing theories, and does not mean that we require a RoboSim model to carry out testing. Our proposed approach does not require even use the RoboStar framework to generate simulation code (via translation from RoboChart to RoboSim, and then to code) or not. With the assumption about an unknown RoboSim model, however, we can study test conversion in the context of *tock*-CSP, because, since RoboSim has a *tock*-CSP semantics, the converted traces are also *tock*-CSP traces.

Examples of conversions are given in Table I. The second column includes forbidden traces of the RoboChart module CMovement in Fig. 1. Examples 2 and 6 are not converted because they do not satisfy properties that ensure the resulting tests are useful. Below, we describe and motivate these restrictions: we should consider only the minimal forbidden traces of a reactive model that are output constraining, p-compliant, where p is the period of the SUT, and output cyclic.

a) Output constraining: In a simulation, we cannot observe refusal of an input, as characterised by a reactive event. In a reactive model, an input captures an interaction, synchronous or asynchronous, where the environment, here, the robotic platform, provides an input and it is read by the software. In a simulation, the events are of a different nature. The platform provides, via registers, values for all inputs, indicating whether input data is available or not. The simulation accesses that information in every cycle, but may or not use it, and this is not directly observable.

In our example, we have just one input obstacle. We recall that, in the simulation, in every cycle, a register (represented in the semantics by *read.obstacle*) is read to identify whether the robotic platform has indicated the presence of an obstacle (via a Boolean value). The interaction with the register, that is, the input from the register, always happens, regardless of whether obstacle has happened or not. Moreover, if the robotic platform indicates the presence of the obstacle, it does not mean that the software responds to that. For instance, when in the states Waiting, STurning, and DTurning, the simulation in Fig. 2 does not consider the value of $obstacle. Finally, if there were more inputs, all the registers corresponding to them would be read, in every cycle.

For this reason, we cannot confirm directly whether an input is forbidden in a simulation. We can, however, check whether the outputs that arise after such an input are as expected. So, we need to convert only tests for traces that define forbidden outputs, that is, traces whose final event is an output. We call such traces *output constraining*, although, of course, they do define required values for inputs leading to the forbidden output. For example, $\langle\, move.out.lv.0, tock, obstacle.in, move.out.lv.0 \,\rangle$ is output constraining, but requires that an obstacle is detected before the forbidden output is observed, so that input is needed.

b) p-compliance: A simulation with a period greater than 1 may eliminate the possibility of certain quiescence periods. For example, when the ranger starts waiting for the detection of an obstacle, an arbitrary amount of time may pass before detection. So, there are tests that require an obstacle detection immediately or after any amount of time units (*tock* events).

TABLE I
EXAMPLES OF TRACES CONVERTED USING $T_{RCS}(p, I, ft)$, WHERE ft IS THE FORBIDDEN TRACE,A ND p IS 1.

	Forbidden trace	**Converted trace**
1	*tock*	*tock*
2	*obstacle.in*	N/A: trace is not output constraining
3	*stop.out*	*read.obstacle.false*, *write.stop*
4	*move.out.lv*.0, *obstacle.in*	*read.obstacle.false*, *write.move.lv*.0, *tock*, *read.obstacle.true*
5	*move.out.lv*.0, *stop.out*	*read.obstacle.false*, *write.move.lv*.0, *write.stop*
6	*move.out.lv*.0, *move.out.lv*.0	N/A: trace is not output cyclic
7	*move.out.lv*.0, *tock*, *stop.out*	*read.obstacle.false*, *write.move.lv*.0, *tock*, *read.obstacle.false*, *write.stop*
8	*move.out.lv*.0, *tock*, *obstacle.in*, *tock*	*read.obstacle.false*, *write.move.lv*.0, *tock*, *read.obstacle.true*, *tock*
9	*move.out.lv*.0, *tock*, *obstacle.in*, *stop.out*, *tock*, *move.out*.0.1, *tock*, *tock*, *tock*	*read.obstacle.false*, *write.move.lv*.0, *tock*, *read.obstacle.true*, *write.stop*, *tock*, *read.obstacle.false*, *write.move*.0.1, *tock*, *read.obstacle.false*, *tock*, *read.obstacle.false*, *tock*

If, however, the period of the simulation were 2, then a test that requires detection after one time unit can never conclude, because after one time unit, the simulation will always advance the time one more time unit before reading the inputs. So, the test would never succeed in driving the simulation through the trace. Such tests are useless and we should not convert them.

So, we define a trace t to be *p-compliant* if, for every subsequence of *tock* events in t, either its length is a multiple of p or it is a suffix of t. To explain the point of allowing an arbitrary suffix of *tock* events, we consider first a suffix of size 1. In this case, the forbidden trace indicates that conclusion of the processing phase at that stage is forbidden. This may arise from a design model that requires an urgent output; the test checks that the deadline is met by the simulation. For a suffix of *tock* events of size 2, the penultimate *tock* event is required by the test, but the second is forbidden. This may similarly arise from a design that provides a deadline for an input or output that runs out after one time unit. In this case, the test checks that the simulation is responsive enough. Similar observations arise for any number of trailing *tock* events.

The actual value of a time unit (whose passage is recorded by a *tock*), in terms of simulation or real time, is left undefined, in both the RoboChart and the RoboSim models. So, in principle, we could have a *tock* event of the RoboChart model corresponding to several *tock* events of the RoboSim model, and vice-versa. This flexibility, however, is spurious, since the point where flexibility is useful is when code is generated. With the assumption of a one-to-one correspondence between time units in the RoboChart and RoboSim models, we significantly simplify the notion of conformance.

c) Output cyclic: A simulation cannot provide the same output twice in the same cycle. So, traces of a RoboChart model that record such behaviour lead to tests that are always inconclusive or always pass. If both outputs are required in the trace, the test is always inconclusive, because the SUT cannot be observed to provide the second input. If the second output is forbidden, the SUT always passes the test for the same reason. Such traces, therefore, are discarded.

For a simulation, we can instrument the code to address the issues related to the visibility of whether the software has accepted an input or not. We can, in fact, implement a wrapper that captures the reactive view of a simulation, abstracting away the reading of input sensors and writing to actuators. For code used in deployment, however, instrumentation is not likely to be appropriate; for example, it interferes with time performance. Moreover, instrumentation is application dependent and cannot take advantage of the identification, and consequent discarding, of useless tests suggested here.

Examples in Table I show that, for traces that satisfy the above restrictions, conversion needs to: (a) identify the cycles; (b) identify the required inputs in each cycle and record those inputs in the converted trace; (c) record the absence of the remaining inputs in the converted trace; and (d) record the outputs and passage of time in the recorded trace. For instance, in Example 9, we have a trace recording five cycles (with period 1). In the last two cycles, there are no inputs or outputs, but the converted trace still includes the *read* event for obstacle. Extra *read* events are added to the record of all cycles, but the input *read.obstacle.true* occurs only in the cycle where *obstacle.in* is recorded in the original trace.

IV. CONCLUSIONS

There has been significant long-running interest in the generation of test cases from a model or specification. Where the model has a formal semantics, there is the potential to automatically generate sets of test cases that provide guarantees regarding effectiveness and soundness.

This paper has explored the scenario in which we are testing from a reactive design, written in RoboChart, and wish to test a cyclic implementation. We are interested in generating tests from the abstract reactive model since test results can then be understood in terms of the context of the design. The proposed approach is to start with minimal forbidden traces of the RoboChart model. If testing a reactive SUT then we can simply use these forbidden traces as the basis of testing. We cannot, however, directly consider such forbidden traces to define tests for a cyclic SUT since a number of assumptions or restrictions are encoded into the cyclic paradigm, and the inputs and outputs in a reactive model correspond to data read and written every period of a cyclic model.

We suggest the use of a technique that takes a minimal forbidden trace of a reactive (RoboChart) model and converts

it into a corresponding trace for a cyclic SUT, discarding traces of the reactive model that lead to useless tests for a cyclic SUT because they cannot fail. Although our observations are concerned with RoboChart and RoboSim, the challenges mentioned are relevant for timed reactive and cyclic models in general. Although RoboSim is a simulation notation, a RoboSim model describes a cyclic mechanism that may be used also in deployment. In fact, the modular approach to simulation adopted in RoboStar encourages such reuse of code.

There are several lines of future work. We need to formalise the conversion and establish completeness: we need to have enough sound tests to uncover any mistaken SUT. A practical approach to generate the converted tests is also crucial. From a theoretical point of view, there is also the question of whether one should include the observation of the refusal of output within a trace and, if one does, how this can be converted. There is also the need for more substantial case studies. RoboStar technology provides support for automatic generation of simulations, with support for testing from RT-Tester [17], [18]. We will implement the technique described here in that setting, using RT-Tester to obtain code for the tests. In this setting, we can consider significant case studies.

ACKNOWLEDGMENT

We are grateful to members of the RoboStar team for their support of the work here and its overall agenda, in particular, Pedro Ribeiro and Madiel Conserva Filho for their work on the RoboSim semantics and verification technique.

REFERENCES

[1] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *13th IEEE International Conference on Software Testing, Validation and Verification*, 2020, pp. 96–107.

[2] A. Ortega, N. Hochgeschwender, and T. Berger, "Testing service robots in the field: An experience report," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2022, pp. 165–172.

[3] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, M. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, 2009.

[4] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55–71, 2011.

[5] M.-C. Gaudel, "Testing can be formal, too," in *International Joint Conference, Theory And Practice of Software Development*, ser. Lecture Notes in Computer Science, vol. 915. Springer-Verlag, 1995, pp. 82–96.

[6] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock, "RoboChart: modelling and verification of the functional behaviour of robotic applications," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, 2019. [Online]. Available: rdcu.be/bh7dI

[7] J. Baxter, P. Ribeiro, and A. L. C. Cavalcanti, "Sound reasoning in tock-CSP," *Acta Informatica*, vol. 59, pp. 125–162, 2022.

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

[9] J. Baxter, A. L. C. Cavalcanti, M. Gazda, and R. M. Hierons, "Testing Using CSP Models: Time, Inputs, and Outputs," *ACM Transactions in Computational Logic*, vol. 24, no. 2, 2023.

[10] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, *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.

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

[12] S. G. Brunner, F. Steinmetz, R. Belder, and A. Domel, "Rafcon: A graphical tool for engineering complex, robotic tasks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016, pp. 3283–3290.

[13] M. Wachter, S. Ottenhaus, M. Krohnert, , N. Vahrenkamp, and T. Asfour, "The ArmarX Statechart Concept: Graphical Programing of Robot Behavior," *Frontiers in Robotics and AI*, vol. 3, p. 33, 2016.

[14] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. C. Filho, A. Didier, W. Li, and J. Timmis, "Verified simulation for robotics," *Science of Computer Programming*, vol. 174, pp. 1–37, 2019. [Online]. Available: papers/CSMRCD19.pdf

[15] A. L. C. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, and A. C. A. Sampaio, *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. Springer International Publishing, 2021, pp. 249–293. [Online]. Available: papers/CBBCFMRS21.pdf

[16] A. L. C. Cavalcanti, J. Baxter, R. M. Hierons, and R. Lefticaru, "Testing Robots using CSP," in *Tests and Proofs*, D. Beyer and C. Keller, Eds. Springer, 2019, pp. 21–38. [Online]. Available: papers/CBHL19.pdf

[17] J. Peleska, E. Vorobev, and F. Lapschies, "Automated test case generation with smt-solving and abstract interpretation," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzman, and R. Joshi, Eds. Springer, 2011, pp. 298–312.

[18] J. Peleska and W. Huang, "Industrial-strength model-based testing of safety-critical systems," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds. Springer, 2016, pp. 3–22.