White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/238644/

Version: Accepted Version

Article:

Cavalcanti, A. and Hierons, R.M. (2026) Reactive model-based testing of cyclic systems. ACM Transactions on Computational Logic. ISSN: 1529-3785

https://doi.org/10.1145/3801960

© 2026 The Authors. Except as otherwise noted, this author-accepted version of a journal article published in ACM Transactions on Computational Logic is made available via the University of Sheffield Research Publications and Copyright Policy under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution and reproduction in any medium, provided the original work is properly cited. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS | University of Sheffield | UNIVERSITY of York

Reactive model-based testing of cyclic systems

ANA CAVALCANTI, University of York, UK
ROBERT M. HIERONS, University of Sheffield, UK

There is extensive literature on automated test generation using reactive design models, where control is determined by events. In contrast, the (idealised) simulation paradigm defines control through cycles dictated by the passage of time. Within each cycle, inputs are read and processed, and outputs are provided, all instantaneously, and afterwards time progresses. To exercise a simulation using tests generated from a reactive design model requires changes to the tests to take into account this paradigm shift. This paper focuses on automation of the necessary changes and of the use of the resulting tests in a simulation campaign. Based on a notion of conformance that establishes whether a simulation is correct with respect to a reactive design, we (1) identify the reactive tests that are meaningful; (2) define a process to convert those tests; (3) provide an algorithm to execute those tests; and (4) prove soundness and completeness of our approach. Our work is described in the context of the RoboStar framework for model-based development of control software for robotics applications, and its process algebraic semantics. The testing approach we propose here represents a significant advancement in the current testing practices within the field of robotics, where simulations are widely used.

CCS Concepts: • **Software and its engineering** → **Formal methods**; **Software testing and debugging**; • **Computing methodologies** → *Concurrent computing methodologies.*

Additional Key Words and Phrases: Model-based testing, process algebra, refinement, simulation, robotics

ACM Reference Format:
ANA CAVALCANTI and ROBERT M. HIERONS. 2026. Reactive model-based testing of cyclic systems. *ACM Trans. Comput. Logic* 1, 1 (March 2026), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In model-based testing (MBT), tests are defined in terms of a model, enabling substantial automation and cost reductions. MBT also supports strong assertions about test effectiveness, allowing testers to employ a test suite that establishes correctness under specific conditions. Many modelling languages have been studied as a basis for automatic test generation; examples are [3, 14, 43]. Our work has focussed on the process algebra CSP [35] and its variants [8, 11, 12].

In this paper, we present support for automatic generation of simulation tests based on reactive models described in *tock*-CSP [2], a discrete-time variant of CSP. Design-level models, even those described informally via diagrams, are often reactive. There are numerous examples in the robotics literature [29, 34], for instance, where informal diagrams for state machines refer, for example, to inputs leading to changes of state with no implicit or explicit reference to cycles of execution. On the other hand, simulations and implementations (based, for instance, on a cyclic executive pattern) normally adopt a cyclic paradigm of execution. There are modelling languages that adopt a cyclic control flow [28, 39], and it is possible to define a testing approach based directly on models written in such languages. For

Authors' addresses: ANA CAVALCANTI, University of York, York, UK, ana.cavalcanti@york.ac.uk; ROBERT M. HIERONS, University of Sheffield, Sheffield, UK, r.hierons@sheffield.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Association for Computing Machinery.
Manuscript submitted to ACM

testing of properties of the design, and for traceability, however, it is important to consider test generation based on more abstract reactive design models, and how the tests can be used against simulations or cyclic implementations.

Our results are relevant for any models with a *tock*-CSP semantics. For practical purposes, however, we consider models described using RoboChart [27], a domain-specific notation for modelling and verifying reactive designs of control software for robotic systems. RoboChart utilises state machines enriched with time constructs, and a simple component model to specify timed, platform-independent behavioural models for control software. The RoboChart *tock*-CSP semantics defines processes that characterise reactive behaviour in terms of interactions representing sensor outputs (inputs to the software) and actuator inputs (outputs of the software), influencing the robotic system's environment. Support for modelling, model checking, and test generation based on RoboChart is available via RoboTool[1].

The testing theory for *tock*-CSP [1] defines the construction of tests from forbidden traces of a process, including input and output events and a special *tock*-CSP event, namely, *tock*, representing the passage of time. To illustrate, we consider a model for a ranger robot, with events *obstacle*, *move*, and *stop* representing inputs from sensors and calls to motor operations, that is, outputs. The forbidden trace $\langle\, move.out.lv.0, tock, obstacle.in, move.out.lv.0 \,\rangle$ records an immediate output, namely, a call move(lv,0) to an operation move with arguments lv and 0 defining a linear velocity lv and an angular velocity 0, the passage of one time unit, represented by *tock*, and an input flagging an obstacle. This is followed by a forbidden additional call move(lv,0), since the robot must stop after an obstacle is detected.

Tests derived from such forbidden traces drive the system under test (SUT), here the robotic control software, and attempt the forbidden event. The test is inconclusive if the SUT cannot be driven to just before the forbidden event by providing the indicated inputs and observing the prescribed outputs and passage of time. If the SUT can reach that point, but the forbidden event does not occur, the test passes. Finally, if the forbidden event is observed, the test fails.

The SUT is an implementation proposal for the robotic control software, often exhibiting cyclic behaviour, especially in simulations. While RoboChart models are reactive, simulations follow a cyclic paradigm. In each cycle, the SUT evolves by accepting inputs, processing them, and providing outputs, infinitely fast, before advancing time to the next cycle. Our work with RoboChart involves comparing reactive and cyclic models using a conformance relation defined and formalised in *tock*-CSP. Test generation and execution must adapt to this notion of conformance.

For example, the test described above is not suitable for a simulation. First of all, it starts with an output, $move.out.lv.0$, but a simulation does not provide outputs until all inputs are provided. The lack of inputs before the output indicates that no inputs are available, but the test driver must indicate that explicitly to the SUT, and not simply wait for an output. Another issue with this example test arises if the cycle of the SUT is longer than one time unit, in which case the trace described is not feasible. After the *tock* event, the SUT will always insist on another *tock*. So, the execution of that test is always going to be inconclusive and so useless to find the fault that it describes.

In [10], we have identified some of the conditions that must be satisfied by a RoboChart forbidden trace that can be used to generate a useful test for a cyclic simulation, and given some examples. Here, we go much further: we fully characterise useful forbidden traces, and define a conversion function to transform them into traces of a cyclic model, providing traceability between abstract design and the concrete simulation (or implementation). Additionally, we provide an algorithm to apply the converted tests, and prove the soundness and completeness of our testing approach.

The next section presents related work on testing from reactive models. Section 3 describes background notations and results: RoboChart and its sister notation, RoboSim [13], which can be used to describe simulations, and the *tock*-CSP testing theory. Section 4 describes properties of our conformance notion between reactive and cyclic models. Our

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

testing approach is the subject of Section 5. Soundness and completeness are established in Sections 6 and 7. Finally, we conclude in Section 8, where we summarise indicate ongoing and future work.

2 RELATED WORK

Model-Based Testing (MBT) is a very active and mature area of research. There is a rich literature and collection of tools based on formal models that adopt different modelling paradigms and notions of conformance [22, 24, 43]. To the best of our knowledge, however, the majority of the results are based on a reactive view of systems. They can, therefore, benefit from the complementary ideas and results presented here when applying tests to a cyclic SUT.

A significant body of work focuses on conformance testing for finite state machines (FSM) or labelled transition systems (LTS) representing the operational semantics of a model [14, 23, 42]. Much of the focus is on generating minimal test sequences to verify whether an SUT conforms to its specification in terms of responsiveness and state correctness.

For FSMs, input sequences are typically used as test cases, with the response of the SUT being checked against the specification FSM. As a result, test cases do not contain verdicts. Many test generation algorithms aim to produce a set of test cases T that is m-complete for a given m. This requires that if the SUT behaves like an unknown FSM N with at most m states then the SUT passes all test cases in T if, and only if, the SUT conforms to the specification FSM. Thus, the application of an m-test suite to the SUT determines correctness subject to the assumption (test hypothesis [17]) that the behaviour of the SUT can be represented by an FSM with at most m states. Different test-generation techniques have been devised for deterministic FSMs [14, 20, 41] and non-deterministic FSMs [21, 26, 33, 36].

For LTSs, test cases are normally trees in which each node has an associated verdict [40]. In addition, test generation is typically random and is complete in the limit [40]: if the SUT is faulty then the random test-generation process will eventually produce a test cases that the SUT fails. This approach is much closer to ours.

It is normal to distinguish between input and output events, since they play very different roles in testing, with the tester controlling inputs and the SUT controlling outputs. In addition, it is often assumed that the tester cannot block outputs, and the implementation cannot block inputs [4, 40]. In contrast, here, the tester does not control either inputs or outputs, since the cycles of the SUT determine the point where the inputs are taken and they are not blocked.

For testing systems with time constraints, timed LTSs allow transitions that denote durations, capturing the passing of (typically discrete) time [5, 25, 38]. These events, which represent durations, are not inputs to the system (since they are not controlled by the tester) and also are not outputs (since they are not controlled by the SUT). Similarly, we do not regard the *tock* event as an input or an output, but it works as a marker that separates cycles.

More recent related work addresses the application of MBT in modern industrial contexts, based on UML and SysML Statecharts, for example [30]. Here, we also benefit from a diagrammatic notations, but also from their formal semantics. In particular, we benefit from the body of work on testing from process algebras [7, 8, 31]. All these approaches are based on the idea of forbidden traces, already presented by Hennessy, back in 1988 [19].

In the next section, we present the notations and results we use in this paper.

3 PRELIMINARIES

In this section, we introduce RoboChart and RoboSim (Section 3.1), along with the *tock*-CSP testing theory (Section 3.2).

3.1 Our modelling notations

To give an overview of RoboChart, we present in Figure 1 a model for the simple ranger robot described above. The top boxes labelled MovementI and ObstacleI are interfaces that declare the operations and events representing the

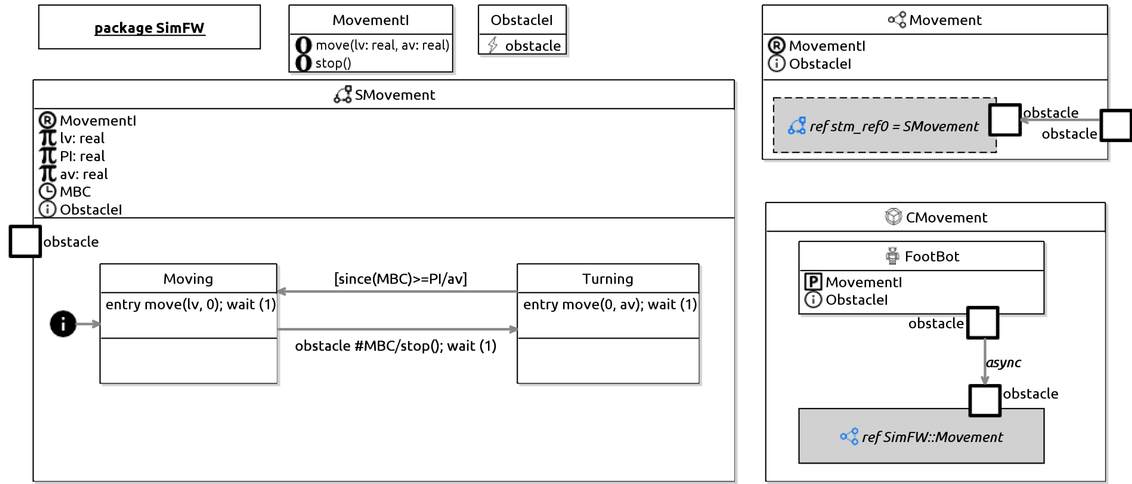

Fig. 1. A RoboChart model for a small ranger robot, defined in a package called SimFW.

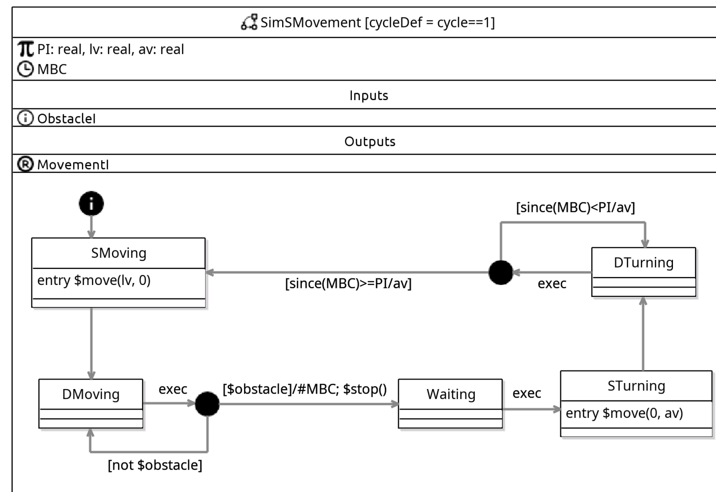

Fig. 2. A RoboSim state machine for the small ranger robot.

services of the robotic platform. Control software is modelled in RoboChart using a component called a module. For our example, the module is depicted by the box CMovement in the lower right-hand corner.

A module includes a robotic-platform block, here named FootBot, declaring MovementI and ObstableI to define the services used to characterise the visible behaviour of the software. A robotic platform can connect to one or more controllers, running in parallel, with behaviour defined by one or more state machines. In this example, we have the controller Movement using the machine SMovement, shown in Figure 2, to specify its behaviour.

RoboChart machines include a context of declarations defining (local) variables, constants, clocks, events, and operations that it can use. RoboChart also has a well-defined action language, to specify data and communication

operations in states and transitions. Additionally, RoboChart machines can specify time budgets and deadlines. Simple restrictions, such as absence of inter-level transitions, enable the definition of a tractable compositional semantics.

In our example, SMovement starts in the Moving state, as indicated by the initial junction: a black circle with an i. In Moving, the entry action triggers an immediate call move(lv,0) and pauses for one time unit using wait(1) to allow the robot to start moving. This is the time budget for the call. This design embeds the assumption that there is nothing right in front of the robot, so that it can move straight ahead for at least one time unit.

SMovement remains in Moving until an event obstacle occurs, leading to an immediate transition to the Turning state. In this transition, the clock MBC is reset (indicated by #MBC), and then a sequence of actions follow: stop is called immediately, and the robot pauses for another time unit. In Turning, an urgent call move(0,av), where av is a constant (declared at the top of the SMovement block) that defines an angular velocity, is followed by a pause for one time unit. Subsequently, SMovement takes the transition back to Moving when its guard, requiring the value of the clock (since(MBC)) to be high enough for the robot to have turned, becomes true.

Although SMovement exhibits a form of cyclic control flow, with transitions from Moving to Turning and back, this is not a simulation cycle. It lacks a fixed period and is triggered by obstacle detection via an occurrence of an event, unlike a simulation cycle that is determined by passage of time. The RoboChart model abstracts from this cyclic polling behaviour used, for instance, to detect the occurrence of events, and from the allocation of actions to these short cycles.

To model cyclic mechanisms, there is a diagrammatic notation that is similar to RoboChart but embeds the cyclic paradigm; it is called RoboSim. Events in RoboSim are captured by Boolean variables, with associated extra variables recording any values communicated by the events. The cycle period and the scheduling for each cycle are explicit in RoboSim models. Like RoboChart, RoboSim also has a *tock*-CSP semantics.

Figure 2 shows the RoboSim machine SimSMovement for our example. The RoboSim module and controller are just like those from the RoboChart model, but have a definition of a period for the cycle via a cycleDef clause like that in SimSMovement (at the top of the box). In a RoboSim state machine, a period is defined, along with a context of variables, constants, and clocks. Inputs and outputs are specified separately. For example, in SimSMovement, the period is 1, and declarations indicate obstacle as an input and the operations (declared in MovementI) as outputs.

A RoboSim machine uses a single event, exec, to mark the end of the processing phase in a cycle. It also uses $ to distinguish Boolean variables representing events (from ordinary RoboChart events), and calls to platform operations from calls to software operations (defined in a controller, and, therefore, not an output of the module). In SimMovement, the control flow starts in the SMoving state, where the entry action is the call move(lv,0). The flow then moves to a DMoving state, where a transition with an exec trigger goes to a junction. With exec, the processing phase of the cycle ends, and time advances to the start of the next cycle. Depending on whether $obstacle is true or not, one of a pair of transitions from the junction is taken. This decision is based on whether the input obstacle has been flagged at the start of the cycle. If not ($not obstacle), the transition leads back to DMoving, where another exec advances the cycle. Otherwise, the clock MBC is reset, and $stop is called in the transition with guard $obstacle.

The *tock*-CSP semantics of RoboChart and RoboSim capture the differences between reactive and cyclic control flows. In the *tock*-CSP process defining the semantics of a RoboChart module, CSP events represent robotic platform services. For instance, $\langle\, move.out.lv.0, tock, obstacle.in \,\rangle$ is a trace of the *tock*-CSP process that defines the semantics of the RoboChart module in Figure 1. In such traces, CSP events corresponding to events and operations of the RoboChart robotic platform distinguish whether they are RoboChart inputs or output via the tags *out* and *in*. In RoboSim's semantics, the *tock*-CSP processes use *read* and *write* events to represent interaction points via registers of sensors and actuators, that is, inputs and outputs of the software. For SimMovement, for instance, a possible trace is

$\langle read.obstacle.false, write.move.lv.0, tock, read.obstacle.true\rangle$, reflecting the fact that an obstacle event has not occurred in the first cycle (indicated by the *read* event with the value *false*), then an output (*write* event) corresponding to a call to the move operation takes place, after which the cycle finishes and the time advances (event *tock*), and finally, the input in the next cycle indicates the presence of an obstacle (value *true* in the *read* event).

Section 4 delves into the challenges of comparing RoboChart and RoboSim models: straightforward comparison via refinement is not suitable. We first, however, describe the testing support available for RoboChart.

3.2 Testing using RoboChart and *tock*-CSP

RoboTool implements a mutation-based technique to automatically generate tests from a RoboChart model [6]. In this approach, using Wodel, a third-party tool to support the development of mutation-testing tools [18], various mutation operators, such as removing or altering transitions, are applied to the RoboChart model resulting in a series of mutated models. RoboTool provides a set of mutation operators and supports the definition of additional ones via Wodel. Each operator can be applied at one or multiple points in the model, such as all its transitions. Model checking is then used to compare each mutated model to the original RoboChart model. If the mutation leads to a fault, a counterexample is produced: it is a forbidden trace of the original RoboChart model.

The *tock*-CSP testing theory defines how a test can be constructed from such a trace, ensuring the soundness of the tests, that is, ensuring they provide the correct verdict. As already said, the *tock*-CSP tests are concerned with the faults represented by the last events e of forbidden traces $t \frown \langle e\rangle$, which are necessarily forbidden continuations after the SUT goes through the trace t, if at all. Here, $\frown$ is the sequence concatenation operator. The test for $t \frown \langle e\rangle$ tries to drive the SUT to the end of t, by providing the inputs and checking passage of time and the outputs in t, in the order defined in t. If the SUT deadlocks before the end of t, the test is inconclusive. If the SUT gets to the end of t, but deadlocks before e is observed, the test passes. Finally, if the SUT engages in e, then the test fails.

The set of all tests defined by all (minimal) forbidden traces can determine the correctness of any SUT under trace refinement in the *tock*-CSP semantics. It is an exhaustive test suite: all faulty implementations, that is, all implementations that do not trace refine the specification (the RoboChart model), fail a test in that set. The *tock*-CSP testing theory accommodates the distinction between inputs and outputs, with RoboTool traces indicating the input and output events through the *in* and *out* tags in events as already exemplified. In addition, it must be noted that the *tock*-CSP theory also caters for timewise refinement, but that is not studied in this paper.

Typically, the complete (sound and exhaustive) test suite defined by the *tock*-CSP testing theory is infinite, so impractical for a test campaign. Instead, a selection of tests is necessary. As said, RoboTool facilitates test selection based on faults represented by mutations. Selection based on trace size and data equivalence classes is also available.

In this paper, we examine soundness and exhaustiveness results when the SUT is a cyclic mechanism. As we elaborate in Section 5, some tests may be disregarded, while others must undergo modifications to account for cycles.

4 CONFORMANCE

With a reactive (RoboChart) model used as a specification, and a cyclic implementation as the SUT, the notion of conformance that is needed to define the test verdicts has to consider the discrepancy in the approaches to control flow. For example, if an event obstacle happens after one time unit, the RoboChart model in Figure 1 specifies that immediate reaction is required (via the transition to the state Turning and consequent call to move). In a simulation, however, there are periods in which no input is accepted. As mentioned before, the control flow of a simulation is defined by a sequence of cycles determined by the passage of a given period of time. At the start of each cycle, inputs,

$$\text{RC}\ \mathit{conf}_p\ \text{RS} \mathrel{\widehat{=}} \mathit{Cyclic}(\text{RC}, p) \sqsubseteq [\![\text{RS}]\!]_{RSM} \text{ where } p \text{ is the cycle of RS}$$

$$\mathit{Cyclic}(\text{RC}, p) = ([\![\text{RC}]\!]_{RCM};\ \mathit{end} \rightarrow \mathbf{Skip} \mathrel{[\![} I_{RC} \cup O_{RC} \cup \{\mathit{end}\} \mathrel{]\!]} \mathit{SimAssump}(I_{RC}, O_{RC}, p)) \setminus I_{RC} \cup O_{RC} \cup \{\mathit{end}\}$$

$$\begin{aligned} \mathit{SimAssump}(I_{RC}, O_{RC}, p) = \ & (\mathit{TA1}(I_{RC}) \mathrel{[\![} I_{RC} \cup \{\!|\mathit{read}, \mathit{end}|\!\} \mid O_{RC} \cup \{\!|\mathit{write}, \mathit{end}|\!\} \mathrel{]\!]} \mathit{TA2}(O_{RC}, p)) \\ & \quad \mathrel{[\![} I_{RC} \cup O_{RC} \cup \{\!|\mathit{read}, \mathit{write}, \mathit{end}|\!\} \mid I_{RC} \cup O_{RC} \cup \{\!|\mathit{read}, \mathit{end}|\!\} \mathrel{]\!]} \\ & \mathit{TA3}(I_{RC}, O_{RC}, p) \end{aligned}$$

$$\begin{aligned} & \mathit{TA1}(I_{RC}) = \mathit{A1}(I_{RC}) \mathbin{\triangle} (\mathit{end} \rightarrow \mathbf{Skip}) \\ & \mathit{A1}(I_{RC}) = \mathop{|||} e : I_{RC} \bullet \mathit{A1Event}(e) \\ & \mathit{A1Event}(e) = \mathit{read}.e?b?x \rightarrow \mathbf{if}\ b\ \mathbf{then}\ ((e.\mathit{in}.x \rightarrow \mathit{A1Event}(e)) \mathbin{\Box} \mathit{A1Event}(e))\ \mathbf{else}\ \mathit{A1Event}(e) \end{aligned}$$

$$\begin{aligned} & \mathit{TA2}(O_{RC}, p) = \mathit{A2}(O_{RC}, p) \mathbin{\triangle} (\mathit{end} \rightarrow \mathbf{Skip}) \\ & \mathit{A2}(O_{RC}, p) = (\Box\, e : O_{RC} \bullet e.\mathit{out}?x \rightarrow (\mathit{write}.e!x \blacktriangleright 0);\ \mathit{A2}(O_{RC} \setminus \{e\}), p) \mathbin{\triangle_1} (\mathbf{Wait}(p-1);\ \mathit{A2}(O_{RC}, p)) \end{aligned}$$

$$\begin{aligned} \mathit{TA3}(I_{RC}, O_{RC}, p) = \ & ((\mathop{|||} e : I_{RC} \bullet \mathit{read}.e?b?x \rightarrow \mathbf{Skip}) \blacktriangleright 0); \\ & (\mathit{RUN}(I_{RC} \cup O_{RC}) \mathbin{\triangle} (\mathit{end} \rightarrow \mathbf{Skip}) \mathbin{\triangle_1} (\mathbf{Wait}(p-1);\ \mathit{TA3}(I_{RC}, O_{RC}, p)) \end{aligned}$$

Fig. 3. Definition of conformance of a RoboSim model RS of a simulation with period p with respect to a RoboChart model RC. I_{RC} and O_{RC} refer to the inputs and outputs of RC, which can be calculated from its definition. RS admits the same inputs and outputs.

processing, and outputs all happen infinitely fast. This is followed by a period of quiescence, in which an amount of time determined by the cycle period advances, but no inputs or outputs happen. So, in a simulation, if obstacle happens during the quiescent period, that input is ignored. So, a naive check can indicate that the simulation is wrong. Instead, however, the assumption is that events do not happen during the quiescent period of the simulation. In this section we review the definition of conformance required (Section 4.1) and then explore some of its properties (Section 4.2).

4.1 Defining conformance

We have previously defined a notion of conformance that compares a RoboChart and a RoboSim model, taking the paradigm changes and the assumptions associated with simulations into account [13]. Here, we can take advantage of this work because we make the usual assumption in formal testing that the behaviour of the SUT can be described using the same formalism as the specification. We assume that the SUT can be described by a RoboSim model, so that both the specification and the SUT have a *tock*-CSP semantics. This does not mean that a RoboSim model needs to be written as a prerequisite to use our testing approach. We merely use its existence to reason about our technique, and study soundness and completeness of our tests. This is standard in the formalisation of testing techniques.

In our notion of conformance, the specification is taken to be not the RoboChart model itself, but that model in the context of the assumptions that capture the cyclic paradigm of implementation. We present the formal definition of the conformance relation in Figure 3. Relevant properties of this relation used in our work are presented in the sequel.

The conformance relation RC conf_p RS between a RoboChart model RC and a RoboSim model RS with a cycle p is defined in terms of the refinement relation $\sqsubseteq$ between *tock*-CSP processes. In a sense, conformance RC conf_p RS is parametrised by the notion of refinement used to compare the processes for RC and RS. In words, full refinement $P \sqsubseteq Q$ between *tock*-CSP processes P and Q ensures that the traces of events of Q are all possible for P, although P may be more nondeterministic and admit additional traces. Refinement also ensures that any deadlocks (refusals) in Q are admitted by P. Finally, with refinement, we ensure that the P requirements regarding time budgets and deadlines are

preserved by Q. In this paper, however, we restrict ourselves to trace refinement in *tock*-CSP; in this case, refinement caters for traces of events and time properties as just described, but not deadlocks.

An obvious alternative is to use the input-output traces semantics of *tock*-CSP and its refinement relation instead. The difference here is that we eliminate traces that record behaviour where an enabled output is delayed for one or more time units. This can be too restrictive as the specification of a simulation, since outputs via the same events, or calls to the same operation, need to be scheduled in different cycles.

In the definition of RC $conf_p$ RS, we require that the process $[[\mathsf{RS}]]_{RSM}$, which captures the semantics of RS, refines not the process $[[\mathsf{RC}]]_{RCM}$, which captures the semantics of RC, but another process $Cyclic(\mathsf{RC}, p)$. In this specification, $[[\mathsf{RC}]]_{RCM}$ is considered in a context in which the assumptions of the cyclic paradigm, such as events do not occur during the quiescence period, are captured by the process $SimAssump(I, O, p)$ in Figure 3, hold. The traces in $[[\mathsf{RC}]]_{RCM}$ are from a set, called here $TTTrace_{RC}$, of *tock*-CSP traces [2] whose events are *tock* or from a set Σ_{RC} of events representing the platform services (as illustrated in Section 3.1). The traces of $[[\mathsf{RS}]]_{RSM}$ and $Cyclic(\mathsf{RC}, p)$ are from the set $TTTrace_{RS}$ of *tock*-CSP traces with *read* and *write* events from a set Σ_{RS}, in addition to *tock*.

Conjunction of requirements captured by CSP processes is specified using parallelism ($|[\ldots]|$). In Figure 3, $[[\mathsf{RC}]]_{RCM}$ is composed in parallel with $SimAssump(I_{RC}, O_{RC}, p)$ to define $Cyclic(\mathsf{RC}, p)$. The set between the parallelism brackets $|[\ldots]|$ defines the events on which the parallel processes need to synchronise (agree on). Here these are all inputs I_{RC}, all outputs O_{RC}, and a new fresh event *end*. Synchronisation on inputs and outputs means that RC can only progress when the assumptions of the cyclic paradigm defined by $SimAssump(I_{RC}, O_{RC}, p)$ are satisfied. Once RC terminates, it uses $end \rightarrow \textbf{Skip}$ to signal to $SimAssump(I_{RC}, O_{RC}, p)$ that it has terminated. This terminates $SimAssump(I_{RC}, O_{RC}, p)$ as well, and so all of $Cyclic(\mathsf{RC}, p)$. Finally, all inputs, outputs, and *end* are hidden (as specified by $\setminus I_{RC} \cup O_{RC} \cup \{end\}$). So, $Cyclic(\mathsf{RC}, p)$ defines visible behaviour in terms of the events *read* and *write* of the simulation.

$SimAssump(I_{RC}, O_{RC}, p)$ is itself defined by a parallelism, that is, conjunction, of three processes $TA1(I_{RC})$, $TA2(O_{RC}, p)$, and $TA3(I_{RC}, O_{RC}, p)$ defining three assumptions of the cyclic paradigm. $TA1(I_{RC})$ is concerned with the inputs in I_{RC}. Its behaviour is given by $A1(I_{RC})$, but can be interrupted (as defined by the CSP operator $\triangle$) by an *end* event, when it then terminates (as defined by the CSP process **Skip**). $A1(I_{RC})$ is the interleaving, that is, a parallel composition with no synchronisation, of processes $A1Event(e)$, one for each event e in I_{RC}. So, $A1(I_{RC})$ captures the conjunction of the restrictions imposed by $A1Event(e)$ processes. The restriction is that the input $read.e?b?x$ must be taken, defining the value of the Boolean b and of the actual input value x for e. If the Boolean is *true*, then the input $e.in.x$ may or may not be used by the RoboChart model, as defined by the choice operator $\Box$, otherwise it cannot. Forbidding $e.in.x$ corresponds to $A1Event(e)$ recursing without allowing engagement in $e.in.x$ by the RoboChart model.

For illustration, we consider a version of the ranger with another input, called, for instance, cliff, which indicates the edge of the floor (rather than an obstacle in the middle of the floor). All input traces below for a ranger simulation, where the event obstacle is indicated to occur, allow the RoboChart model to engage in the event obstacle.

$$\langle read.obstacle.true, read.cliff.true\rangle \quad \langle read.obstacle.true, read.cliff.false\rangle$$
$$\langle read.cliff.true, read.obstacle.true\rangle \quad \langle read.cliff.false, read.obstacle.true\rangle$$

On the other hand, the four input traces that are similar to these, but in which the *obstacle* event is *read.obstacle.false*, cannot be followed by *write* events corresponding to outputs of the RoboChart model that can only occur if obstacle occurs. We emphasise, however, that, the order of the inputs does not matter. Moreover, the four input traces above do not ensure that the RoboChart model does engage in an obstacle event, although such an event would not be blocked. If the RoboChart model is ready to accept an obstacle event, that option is certain to be available. Although the definition

of $A1Event(obstacle)$ allows another $read.obstacle$ event to happen instead (because the process in the **then** branch of the conditional of the definition of $A1Event(obstacle)$ offers $A1Event(obstacle)$ itself as a choice), the process $TA3$, explained in the sequel, ensures that another *read* event is not possible until the next cycle. So, $TA1(I_{RC})$ imposes the necessary restrictions on inputs in the context of the cyclic control flow defined by $TA3(I_{RC}, O_{RC}, p)$.

$TA2(O_{RC}, p)$ is concerned with the outputs O_{RC}. Like $TA1(I_{RC})$, $TA2(O_{RC}, p)$ is defined in terms of another process, here $A2(O_{RC}, p)$, which can be interrupted by an *end* event. Unlike the inputs, which are offered in interleaving in $A1(I_{RC})$, the outputs $e.out?x$ are offered in choice ($\Box$), since the simulation does not need to produce all outputs. If an output $e.out?x$ occurs, then $A2(O_{RC}, p)$ ensures that the corresponding simulation event $write.e.x$, which actually outputs the value x, occurs urgently, that is, with deadline (defined by the *tock*-CSP operator $\blacktriangleright$) 0. Once $write.e.x$ happens, $A2$ recurses, but now allowing all outputs except e (that is, $O_{RC} \setminus \{e\}$). This encodes the assumption that no output occurs twice in a cycle. At each choice, if no output occurs within one time unit, so that the timeout, defined by $\triangle_1$, happens, the wait for an output is interrupted, and $A2$ waits for the rest of the simulation period (as defined by the process $\mathbf{Wait}(p-1)$) before recursing, but now allowing all outputs O_{RC} again as a new cycle starts.

We note that internal events (that is, hidden using the CSP $\setminus$ operator) are always urgent. This follows the principle of maximal progress, which ensures that events that can happen without requiring further synchronisation are not subject to undue, potentially arbitrary, delay. Events that are not hidden, however, require synchronisation to go ahead, and so can be delayed for any amount of time units. A deadline limits that delay.

Finally, $TA3(I_{RC}, O_{RC}, p)$ enforces the pattern of the simulation cycles previously explained, considering the simulation inputs and the reactive input and outputs. It requires all simulation inputs $read.e?b?x$ to happen urgently, then all reactive inputs and outputs to happen in any order (as defined by $RUN(I_{RC} \cup O_{RC})$), until *end* happens, when $TA3(I_{RC}, O_{RC}, p)$ terminates, or until a timeout happens ($\triangle_1$), when $TA3(I_{RC}, O_{RC}, p)$ proceeds to the next cycle via a recursion.

For static verification, we can check if $Cyclic(\mathrm{RC}, p)$ is deadlock free. If so, that means that the RoboChart model RC can be scheduled in a simulation with period p. Otherwise, there can be no feasible cyclic implementation of period p.

Next, we present properties related to $conf_p$ that we use to prove soundness and completeness of our testing approach.

4.2 Properties of conformance

First, we define some relations between traces. Since a cyclic mechanism $Cyclic(\mathrm{RC}, p)$ does not restrict the order of inputs in a cycle, except that all inputs occur before the outputs, we define equivalence relations between traces that differ just in the order of inputs. For the traces of a reactive mechanism RC, $\equiv^R_I$ captures the fact that the relative order of an input and an output within a time unit is not relevant. The set I of inputs is a parameter.

Definition 4.1. $\equiv^R_I$ is the reflexive, symmetric, and transitive closure of the set of pairs of the form

$$(\rho_1 \frown \langle e_1, e_2 \rangle \frown \rho_2, \rho_1 \frown \langle e_2, e_1 \rangle \frown \rho_2)$$

where e_1 is an event in the given set I of inputs and e_2 is different from *tock*.

Example 4.2. We consider the trace $\sigma = \langle e1.in, e3.out, e2.in \rangle$. It is equivalent to the following traces under $\equiv^R_I$, where $I = \{e1.in, e2.in\}$. (We recall that *tock*-CSP events representing inputs of a RoboChart model have the *in* tag.)

$$\langle e1.in, e2.in, e3.out \rangle, \langle e2.in, e1.in, e3.out \rangle,$$
$$\langle e1.in, e3.out, e2.in \rangle, \langle e2.in, e3.out, e1.in \rangle,$$
$$\langle e3.out, e1.in, e2.in \rangle, \langle e3.out, e2.in, e1.in \rangle.$$

In this case, σ is equivalent under $\equiv_I^R$ to all its permutations. This is not the case, however, once we have more than one output or more than one cycle. For example, $\langle e1.in, e3.out, e4.out, e2.in \rangle$ is not equivalent to $\langle e1.in, e4.out, e3.out, e2.in \rangle$ under $\equiv_I^R$ because it inverts the order of the outputs. No trace that makes such an inversion is related to σ.

The equivalence $\equiv_I^S$ for simulation traces is similar, but considers the special nature of the *read* events.

Definition 4.3. $\equiv_I^S$ is the reflexive, symmetric, and transitive closure of the set of pairs of the form

$$(\rho_1 \frown \langle read.e_1.b_1, read.e_2.b_2 \rangle \frown \rho_2, \rho_1 \frown \langle read.e_2.b_2, read.e_1.b_1 \rangle \frown \rho_2)$$

Here, the order of inputs $read.e.b$, where b is either *true* or *false*, can be different, but outputs and *tock* events, which happen after all input events, keep their original positions.

As said above, all traces of $Cyclic(\mathrm{RC}, p)$ have the property that, within a cycle (that is, between tock events), all inputs precede all outputs. We call such traces *io-traces* and we use this term to refer to traces of a design or simulation.

The lemma below establishes that, if we restrict out attention to io-traces, the set of traces of $Cyclic(\mathrm{RC}, p)$ is closed under $\equiv_I^S$, where I is the set of input events of the simulation. These are all *read* events.

Lemma 4.4. *If ρ_1 and ρ_2 are io-traces, ρ_1 is a trace of $Cyclic(RC, p)$ and $\rho_2 \equiv_I^S \rho_1$ then ρ_2 is also a trace of $Cyclic(RC, p)$.*

A cyclic model also allows for the occurrence of inputs that are ignored in the processing phase. The partial orders $\leq_I^R$ and $\leq_I^S$ below capture this relationship for traces of a reactive model and of a simulation. For traces ρ_1 and ρ_2 of the reactive model, $\rho_1 \leq_I^R \rho_2$ when ρ_1 differs from ρ_2 just in that it may have more inputs than ρ_2 in corresponding time units. For simulation traces, $\rho_1 \leq_I^S \rho_2$ when an input $read.e.false$ in ρ_2 might correspond to $read.e.true$ in ρ_1.

Definition 4.5. $\leq_I^R$ is the reflexive and transitive closure of the set of pairs

$$(\rho_1 \frown \langle e_1 \rangle \frown \rho_2, \rho_1 \frown \rho_2)$$

where e_1 is in the set of inputs I. For, $\leq_I^S$, we have the reflexive and transitive closure of the pairs

$$(\rho_1 \frown \langle read.e.true \rangle \frown \rho_2, \rho_1 \frown \langle read.e.false \rangle \frown \rho_2)$$

Example 4.6. We consider the trace $\langle e1.in, e3.out \rangle$. We have that $\langle e1.in, e3.out \rangle \leq_I^R \langle e1.in, e3.out \rangle$ since $\leq_I^R$ is reflexive. In addition, it is possible to add an input: we can add $e2.in$ as captured by $\langle e2.in, e1.in, e3.out \rangle \leq_I^R \langle e1.in, e3.out \rangle$, $\langle e1.in, e2.in, e3.out \rangle \leq_I^R \langle e1.in, e3.out \rangle$, $\langle e1.in, e3.out, e2.in \rangle \leq_I^R \langle e1.in, e3.out \rangle$. We can add repeated inputs, but, as we formalise in the next section, in our testing approach, we consider traces where such repetitions are restricted.

Since a simulation can ignore inputs, the set of traces of $Cyclic(\mathrm{RC}, p)$ is downwardly closed under $\leq_I^S$ if we restrict our attention to traces with no repeated inputs. Precisely, these are traces in which, in between *tock* events, there are no two events $read.e$, for the same e. This is a property of every trace of $Cyclic(\mathrm{RC}, p)$.

Lemma 4.7. *For $\rho_1, \rho_2 \in TTrace_{RS}$, such that ρ_2 is a trace of $Cyclic(RC, p)$ and ρ_1 is an io-trace that has no repeated inputs, then $\rho_1 \leq_I^S \rho_2$ implies that ρ_1 is a trace of $Cyclic(RC, p)$.*

Finally, we define partial orders $\trianglelefteq_{r_I}$ for reactive traces and $\trianglelefteq_{s_I}$ for simulation traces that combine the orders above; they are defined as the transitive closure of the union of those relations. These partial orders are key to our testing approach.

Definition 4.8. $\trianglelefteq_{r_I} = (\equiv_I^R \cup \leq_I^R)^*$ and $\trianglelefteq_{s_I} = (\equiv_I^S \cup \leq_I^S)^*$

Given traces ρ_1 and ρ_2 such that $\rho_1 \trianglelefteq_I \rho_2$, we have that ρ_1 can have inputs not contained in ρ_2 and their order between *tock* events is arbitrary, but the presence and relative order of the outputs and of the *tock* events are maintained. Similar comments apply to $\leq_{S_I}$. The following is immediate from Lemmas 4.4 and 4.7.

LEMMA 4.9. *For $\rho_1, \rho_2 \in TTTrace_{RS}$, such that $\rho_1 \leq_{S_I} \rho_2$, ρ_2 is a trace of $Cyclic(RC, p)$, and ρ_1 is an io-trace that has no repeated inputs, then ρ_1 is a trace of $Cyclic(RC, p)$.*

The above relations are used in Sections 6 and 7, when we reason about test completeness and soundness. However, first we explain how we convert RoboChart tests and how testing proceeds.

5 TESTING APPROACH

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 mentioned. In this section we describe how we can convert the tests (Section 5.1), and an algorithm to apply the converted tests and ensure that we get sound verdicts (Section 5.2).

5.1 Test conversion

To convert a test derived from a RoboChart model for use with an SUT that is posed as a cyclic implementation of that model, we convert the forbidden traces of the RoboChart model that defines the test. With the new converted trace, we use the *tock*-CSP testing theory to define the converted test. We consider only the forbidden traces of a reactive model that satisfy properties defined in Section 5.1.1; other traces are discarded as they do not give rise to useful tests. For the valid traces, the function defined in Section 5.1.2 formalises conversion.

5.1.1 Valid (forbidden) traces. Some forbidden traces of a RoboChart model lead to tests whose verdict is always inconclusive or pass. Such tests are not useful: they cannot reveal faults because they cannot lead to a fail verdict (although they can be used to exercise a simulation). Traces defining useful tests satisfy four properties defined below. Namely, they must be output-constraining, p-compliant, where p is the simulation period of the SUT, input-cyclic, and output-cyclic. We call traces satisfying these conditions fs-valid: they are traces that are valid for a simulation, but forbidden. We also name $[\![RC]\!]_{FSV}$ the set of fs-valid traces of the RoboChart model RC.

Output-constraining. In a simulation, the refusal of an input characterised by a reactive event is not observable; in fact, in a $Cyclic(\mathsf{RC}, p)$ process, the corresponding input events I_{RC} are hidden (see Figure 3). In a reactive model, an input represents an interaction where the environment, in this case, the robotic platform, supplies an input, which is read by the software. The interaction is the software access (which affects its control flow). Simulation inputs are of a different nature. The platform still provides the input data, but the simulation accesses this information in every cycle. It is not directly visible whether the simulation actually uses the input.

For instance, in our example, the only input is obstacle. In the simulation, a sensor register is read in every cycle. This is represented by an event *read.obstacle* that determines via a Boolean value whether the robotic platform has signalled the presence of an obstacle. This register input occurs in every cycle, regardless of whether obstacle has actually happened. Furthermore, even if the robotic platform indicates the presence of an obstacle, it does not mean that there is an immediate response from the software. For instance, when in states such as Waiting, STurning, and DTurning, the simulation depicted in Figure 2 does not consider the value of $obstacle, even though it is read.

Consequently, direct confirmation of an input being forbidden in a simulation is not feasible. What is possible to verify is whether the outputs following such an input align with expectations. So, we only convert traces forbidding

an output or *tock*, that is, those that finish with an output event or *tock*. We name these traces "output-constraining," acknowledging that they specify required input and output values and passage of time leading to a forbidden output or passage of time. For instance, $\langle$ *move.out.lv*.0, *tock*, *obstacle.in*, *move.out.lv*.0 $\rangle$ is output-constraining: it states that after the detection of an obstacle, the output corresponding to a call to move with arguments lv and 0 is forbidden. A forbidden *tock* event captures a deadline. A passing SUT engages in an immediate interaction, before time passes.

p-compliance. In instances where the simulation period is greater than 1, certain quiescence periods are not possible. For instance, during the ranger's wait for the detection of an obstacle (in the state Moving in the machine in Figure 1), an arbitrary amount of time may pass. So, there exist tests that require immediate obstacle detection or detection after any number of time units. If, however, the simulation period is 2, a test requiring detection after one time unit can never reach a conclusion. This is because, after one time unit, the simulation always advances time by one more unit before reading inputs. Such tests always yield inconclusive results and can be discarded.

A trace ρ is p-compliant, if for every maximal subsequence of *tock* events in ρ, its length is a multiple of p or it occurs only as a suffix of ρ. More formally, for every n such that $\rho_1 \frown tock^n \frown \rho_2$ is a prefix of ρ, ρ_1 does not finish in *tock*, and ρ_2 does not start with *tock*, either n is a multiple of p or ρ_2 is empty. Here, $tock^n$ is a sequence of n *tock* events.

The allowance for an arbitrary suffix of *tock* events is crucial. For example, for a suffix of size 1, the forbidden trace signifies that the conclusion of the processing phase at that point is forbidden. This may stem from a design mandating an urgent output, so that the test verifies the simulation's compliance with the deadline. For a suffix of *tock* events of size 2, the penultimate *tock* is potentially required, while the second is necessarily forbidden. This situation may arise from a design featuring a deadline for an input or output that expires after one time unit, so that the test assesses the simulation's responsiveness. Analogous observations apply to any number of trailing *tock* events.

The specific value of a time unit (marked by a *tock* event) in terms of simulation or real time remains undefined in both RoboChart and RoboSim models. In principle, a *tock* event in the RoboChart model could correspond to several *tock* events in the RoboSim model, and vice versa. However, this flexibility is spurious, as the true utility of the time abstraction emerges during code generation. Assuming a one-to-one correspondence between time units in the RoboChart and RoboSim models significantly simplifies the notion of conformance (see Section 4).

Output-cyclic. In a simulation, it is not possible for an output to be provided twice within the same cycle. Consequently, traces defining such behaviour result in tests that are either consistently inconclusive or consistently successful. When both outputs are mandated in the front of the trace, the test is persistently inconclusive since the SUT cannot provide the second output. Conversely, if the second output is forbidden, the SUT consistently passes the test for the same reason. Consequently, traces with these characteristics are deemed irrelevant and excluded.

Input-cyclic. Similarly, a simulation cannot read the same input twice in a cycle. If the design requires two occurrences of the same input (two occurrence of obstacle, for instance) before time passes, that cannot be implemented in a simulation. (Such a design is not schedulable in a simulation. As already said, a static check is available to ensure that a RoboChart model is schedulable [13].) Assuming that we have an output-constraining trace, if there are two inputs without *tock* events between them, both inputs are required, because output-constraining traces do not forbid inputs. The corresponding test is always inconclusive, so we discard such traces.

In the next section, we show how one can convert an fs-valid trace to specify a test for a cyclic SUT. Moreover, in our proofs in later sections, we consider s-valid traces of RC, that is, traces of RC that are valid for a *s*imulation. In contrast, fs-valid traces in $[[RC]]_{FSV}$ are by definition forbidden traces. An s-valid trace has all properties of an fs-valid

Table 1. Examples of traces converted using $T_{RCS}(p, I, ft)$, where ft is the forbidden trace, and 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*	N/A: trace is not output-constraining
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*, *write.move*.0.1, *tock*, *tock*, *tock*

trace, except only that it is not output-constraining, since it is not necessarily a forbidden trace. We use $[[RC]]_{SV}$ to denote the set of all s-valid traces of RC. The front of an fs-valid trace, for example, is an s-validtrace.

5.1.2 Conversion function. Formally, conversion is defined by a function $T_{RCS}(p, I, ft)$, whose arguments are the period p, the set I of inputs, and a forbidden s-valid trace ft. Examples of conversions defined by $T_{RCS}(p, I, ft)$ are given in Table 1. The second column includes forbidden traces of the RoboChart module CMovement in Figure 1. Examples (2), (4), and (6) are not converted because they are not fs-valid. Examples (1), (8), and (9) show that a *tock* is just maintained by the conversion. Examples (3) and (5) illustrate that, before an output can be observed, all inputs, here just one, must be provided, even if with the value *false*. Examples (7)-(9) show that the insertion of inputs is needed for every cycle: after every maximal subsequence of *tock* events. Example (8) shows that, if an input is required in the cycle, then, for that cycle, it must be provided in the converted trace with value *true*. The definition of $T_{RCS}(p, I, ft)$ is below.

Definition 5.1. $T_{RCS}(p, I, ft) = {}^\frown/(smap\ Sconv_I\ slots_p(ft))$

Here $slots_p(ft)$ is the sequence of traces that define the behaviour recorded by ft in each cycle of period p. For instance, for the example (8) in Table 1, we get a sequence with two traces, namely, $\langle \langle move.out.lv.0, tock \rangle, \langle obstacle.in, tock \rangle \rangle$. With $Sconv_I$, we can convert these traces, considering that the inputs are the events in the set I. In our example, this is the set $\{obstacle\}$. In Definition 5.1, $Sconv_I$ is mapped over the elements of the sequence $slots_p(ft)$ using a mapping function *smap*. Finally, distributed concatenation ${}^\frown/$ of the resulting sequence of traces defines the converted trace. (The distributed concatenation of a sequence of sequences S defines a single sequence obtained by concatenating the sequences of S in order. Distributed concatenation is the counterpart of distributed union for sets of sets.)

We define $slots_p(\rho)$ for p-compliant traces ρ as follows.

Definition 5.2. For every p-compliant trace ρ_1, the sequence of sequences $slots_p(\rho_1)$ is defined by ${}^\frown/\ slots_p(\rho_1) = \rho_1$ and, for every i from 1 to the size of $slots_p(\rho_1)$ minus 1, $\exists\, \rho_2 : \mathrm{seq}(I \cup O) \bullet slots_p(\rho_1)\ i = \rho_2 {}^\frown tock^p$.

Here, O is the set of outputs. We note that $slots_p(\rho)$ is not well defined if ρ is not p-compliant.

The function $Sconv_I(\rho)$, which converts traces defining behaviour inside a slot, that is, traces with no *tock* events before the end, is defined next in terms of three other functions.

Definition 5.3. $Sconv_I(\rho) = \quad Iconv(\rho \upharpoonright I) {}^\frown NIconv(events(\!|\ I\ |\!) \setminus events(\mathrm{ran}\, \rho)) {}^\frown Tconv(\rho \upharpoonright (O \cup \{tock\}))$

The converted trace has three parts: first events for all inputs required in the slot, then events for all other inputs, and finally all expected outputs and *tock*. With $\rho \upharpoonright S$, we get the sequence obtained by removing from ρ every element not

in the set S. Above, we first get from ρ just the elements in I, that is, the inputs ($\rho \upharpoonright I$). The conversion of this trace is defined by a function *Iconv* defined below. Afterwards, we consider the set of events that are inputs (in I), but are not in ρ (not in $\operatorname{ran} \rho$). From these events, we get a part of the converted trace as defined by a function *NIconv*. Finally, we get the conversion of the other elements of ρ, the outputs and *tock* ($\rho \upharpoonright (O \cup \{tock\})$), using *Tconv*.

For the example (8) in Table 1, we get for the first slot, that is, for $\langle move.out.lv.0, tock\rangle$, first the trace obtained by applying *Iconv* to the empty trace $\langle\rangle$, since there are no required inputs in this slot, followed by the trace $NIconv(\{obstacle.in\})$, followed by $Tconv(\langle move.out.lv.0, tock\rangle)$. For the second slot, that is, $\langle obstacle.in, tock\rangle$, we get $Iconv(\langle obstacle.in\rangle)$, followed by $NIconv(\emptyset)$, followed by $Tconv(\langle tock\rangle)$. We note that the inputs may occur at any point in the slot, but, in the converted traces, they are all recorded at the start. Conversion of inputs is defined as follows.

Definition 5.4. $Iconv(\langle\rangle) = \langle\rangle$ and $Iconv(\langle e.in.v\rangle \frown \rho) = \langle read.e.true.v\rangle \frown Iconv(\rho)$

The empty trace requires no conversion. An input $e.in.v$, representing a communication via a RoboChart event e of a value v, is translated to an event $read.e.true.v$. We recall that the RoboSim event *read* represents an interaction between the (simulation of the) control software and the platform sensor or API service that implements e. The values communicated via *read* represent e itself, the Boolean *true* to indicate that the input has happened in the current cycle, and the value v input. (According to the RoboSim semantics, in line with the simulation paradigm, the value read is available for use throughout the cycle, but the actual input occurs just once as represented by the $read.e.b.v$ event.) The conversion of $e.in.v$ to $read.e.true.v$ reflects the translation from the view of e as a reactive event in the design model to the view of e as data defining whether the input has occurred or not in the current cycle.

If the RoboChart event e does not communicate any values, like obstacle in our running example, v is not present. For our example (8) from Table 1, for the first slot, we get the empty trace, for the second we get $\langle read.obstacle.true\rangle$.

For the inputs NI not required in the slot, $NIconv(NI)$ defines their contribution to the converted trace as defined below. In Definition 5.3, the function *events* projects out values communicated as recorded in ρ. For instance for a trace $\rho = \langle a.in.0, b.out.1\rangle$, with $events(\operatorname{ran} \rho)$ we get $\{a, b\}$. With $f(\!| S |\!)$, we get the set obtained by applying f to the elements of S. So, in Definition 5.3, we use *events* also to project out communicated values from I. With $events(\!| I |\!) \setminus events(\operatorname{ran} \rho)$ we then have just the names of the events (channels in *tock*-CSP terminology) used to communicate inputs. It is this set that is used by *NIconv* to define the *read* events of a converted trace.

Definition 5.5. $NIconv(NI) = \langle e : NI \bullet read.e.false.input(e)\rangle$

The inputs NI not required in the original trace still need to be read by the simulation. So, the converted trace contains a sequence of events $read.e.false.input(e)$, where e is an input event from the set NI given as argument. The Boolean *false* indicates that the input has not been provided (by the platform). Regardless, for well typedness, a value argument $?v$ needs to be defined if e communicates values. So, we use $input(e)$ as the value; the syntactic function $input(e)$ uses type information about e to decide whether we need to include a $?v$ in the event. The order of the *read* events is arbitrary. In our example (8) from Table 1, for the first slot, we get $\langle read.obstacle.false\rangle$, and for the second, the empty trace.

Finally, the definition of $Tconv(\rho)$ is as follows.

Definition 5.6.

$$Tconv(\langle\,\rangle) = \langle\,\rangle$$
$$Tconv(\langle tock\rangle \frown \rho) = \langle tock\rangle \frown Tconv(\rho)$$
$$Tconv(\langle e.out.v\rangle \frown \rho) = \langle write.e.v\rangle \frown Tconv(\rho)$$

Algorithm 1 Check and conversion of a minimal forbidden trace of a reactive model

1: **procedure** TestGeneration(RC: Module,$p : \mathbb{N}^+, I_{RC}, ft : ETrace$)
2: $possibleTraces \leftarrow$ Implied_by(I, ft)
3: **if** ($possibleTraces \cap ett[[\text{RC}]] = \emptyset$)
4: **return** $T_{RCS}\ (p, I, ft)$
5: **endif**
6: **end procedure**

For the empty trace or *tock*, there is no conversion. An output $e.out.v$ is converted to $write.e.v$ reflecting the translation from the reactive to the data view of e, where e is a parameter of the event *write* representing an interaction between the simulation and the platform. For our example, we get traces $\langle write.stop, tock\rangle$ for the first slot, and just $\langle tock\rangle$ for the second. Overall, we get the converted traces in Table 1 by concatenating the converted slots back to a full trace.

Next, we explain how to use the converted tests.

5.2 Testing campaign

A test defined from a forbidden trace of a RoboChart model is sound: when it is used to drive a reactive SUT (that is, an SUT that can be described by a RoboChart model), it gives the right verdict. With conversion of these tests to deal with a cyclic SUT, however, a first question is whether the converted tests are sound. The answer is no, as illustrated below.

Example 5.7. We consider a reactive model RC1 with two inputs and one output represented by CSP events $e1.in$, $e2.in$, and $e3.out$, and whose behaviour is described by the process $e1.in \rightarrow e2.in \rightarrow e3.out \rightarrow \textbf{Stop} \;\Box\; e2.in \rightarrow e1.in \rightarrow \textbf{Stop}$ defined by a(n external) choice ($\Box$) between two other processes. The trace $\langle e2.in, e1.in, e3.out\rangle$ is an fs-valid forbidden trace of RC1 whose conversion results in $\langle read.e2.true, read.e1.true, write.e3.out\rangle$. This is, however, a trace of $Cyclic(\text{RC}1, p)$, for any p, because $Cyclic(\text{RC}1, p)$ allows for RC1 to use inputs in any order. Since this behaviour is allowed by $Cyclic(\text{RC}1, p)$, it is unsound to flag a simulation that executes the trace $\langle read.e2.true, read.e1.true, write.e3.out\rangle$ as incorrect. This example shows that, if we generate forbidden traces of a reactive model and convert them, the tests that arise from the converted traces may not be sound. So, we need an algorithm to select the converted tests.

The problem illustrated by the above example is that, as explained, the order of inputs is not fixed in a cyclic mechanism. Additional issues come up because inputs can also be disregarded.

Example 5.8. We consider a reactive model RC2 with the same events used in Example 5.7, and whose behaviour is described by the process $e1.in \rightarrow e2.in \rightarrow \textbf{Stop} \;\Box\; e2.in \rightarrow e3.out \rightarrow \textbf{Stop}$. The trace $\langle e1.in, e2.in, e3.out\rangle$ is an fs-valid forbidden trace whose conversion results in $\langle read.e1.true, read.e2.true, write.e3.out\rangle$. This is a trace of $Cyclic(\text{RC}2, p)$, for any p, because $Cyclic(\text{RC}2, p)$ allows for RC2 to ignore inputs. So, a cyclic implementation of RC2 can use $e2$, but not $e1$, even though it is provided ($read.e1.true$), and execute the second process $e2.in \rightarrow e3.out \rightarrow \textbf{Stop}$ in the choice.

These observations motivate the definition of Algorithm 1, which takes as input a RoboChart model RC (defined, as discussed in Section 3.1 by a module), the period p of the simulation, the set I of inputs of RC (although this can be calculated from RC), and an fs-valid forbidden trace ft of RC. The type *ETrace* is the set of all traces of the semantic model of a *tock*-CSP process. They are sequences of events that include the inputs and outputs of RC and *tock*.

Algorithm 1 determines whether conversion of ft leads to a sound test. If it does, then it returns the converted test, otherwise, ft is basically rejected. A testing campaign based on a test suite *TS* automatically generated from RC should use Algorithm 1 to filter and convert the forbidden traces of *TS* before using them to generate a test.

Algorithm 2 Calculation of the traces implied by a trace of a reactive model

```
1:  procedure IMPLIED_BY(I, ρ : ETrace)
2:      implied ← ⟨⟩
3:      for i ← 1 . . #slots(ρ) do
4:          slot ← slots(ρ) i ;   Islot ← ran slot ∩ I
5:          impliedset ←SHUFFLE(Islot, slot)
6:          for inp ← Islot do
7:              impliedset ← impliedset ∪ remove(inp, _)⦇ impliedset ⦈
8:          end for
9:          implied ← implied ⌢ ⟨impliedset⟩
10:     end for
11:     return crossproduct(implied)
12: end procedure
```

Algorithm 1 uses Algorithm 2 (line 2) to calculate the set *possibleTraces* of all traces greater than or equal to *ft* according to $\trianglelefteq_{T_I}$ (see Definition 4.8). These are the traces allowed by the cyclic paradigm. The conditional (lines 3–5) then checks whether any trace in that set is a trace of RC. The set $ett[\![RC]\!]$ contains the traces obtained from the set of *t*imed *t*races of the process that defines the *tock*-CSP semantics of RC by keeping just the *e*vents of those traces. (Timed traces in that semantics also record refusals before each *tock* even, byt they are not considered here as they are used to define timewise refinement.) If any of the traces in *possibleTraces* is in $ett[\![RC]\!]$, then *ft* is not actually forbidden for a simulation of RC. So, *ft* is discarded. Otherwise, we use T_{RCS} (see Definition 5.1) to convert *ft* (line 4).

Example 5.9. For $\langle e2.in, e1.in, e3.out\rangle$ from Example 5.7, the set of possible traces includes, for example, $\langle e3.out\rangle$, $\langle e2.in, e3.out\rangle$, $\langle e3.out, e2.in\rangle$, and $\langle e1.in, e2.in, e3.out\rangle$, among others. Here, $\langle e1.in, e2.in, e3.out\rangle$ is a trace of RC1, and so we can conclude that $\langle e2.in, e1.in, e3.out\rangle$ is a possible behaviour of a cyclic implementation. For RC2 in Example 5.8, the set of possible traces defined by $\langle e1.in, e2.in, e3.out\rangle$ is the same, and $\langle e2.in, e3.out\rangle$ is a trace of RC2.

Algorithm 2, used to define *possibleTraces*, takes as parameters the set I of inputs and a (forbidden) trace ρ. The result of Algorithm 2 is obtained by concatenating the sequences in a local variable *implied* in all possible ways (line 11). The value of *implied* is a sequence of sets of traces, recording, in each position i, the traces above the ith slot of ρ according to $\trianglelefteq_{T_I}$. Combination of the traces for each slot uses the crossproduct(S) operator, which defines the set of all sequences obtained by concatenating one sequence from each position of the sequence of sets S.

Example 5.10. We consider the application of Algorithm 2 to the trace ρ given by $\langle e1.in, e2.in, e3.out, tock, e2.in\rangle$. There are two slots in ρ, so the final value of the sequence *implied* has two elements. The first set is

$$\{\ \langle e3.out, tock\rangle, \langle e1.in, e3.out, tock\rangle, \langle e3.out, e1.in, tock\rangle, \langle e2.in, e3.out, tock\rangle, \langle e3.out, e2.in, tock\rangle,$$
$$\langle e1.in, e2.in, e3.out, tock\rangle, \langle e1.in, e3.out, e2.in, tock\rangle, \langle e3.out, e1.in, e2.in, tock\rangle, \langle e3.out, e2.in, e1.in, tock\rangle$$
$$\langle e2.in, e1.in, e3.out, tock\rangle, \langle e2.in, e3.out, e1.in, tock\rangle\ \}$$

These are all the traces related to $\langle e1.in, e2.in, e3.out, tock\rangle$. In the second position of *implied*, we have just $\{\langle\rangle, \langle e2.in\rangle\}$ corresponding to the slot $\langle e2.in\rangle$. In the end (line 11), Algorithm 2 returns the set containing a trace from the first set above, possibly extended with $e2.in$, depending on whether we take $\langle\rangle$ or $\langle e2.in\rangle$ from the second element of *implied*.

Algorithm 3 Calculation of traces obtained by shuffling inputs of a slot

1: **procedure** SHUFFLE($I, \rho_1 : ETrace$)
2: **if** ($\rho_1 = \langle\rangle$)
3: **return** $\{ \langle\rangle \}$
4: **else if** (head $\rho_1 \in I$)
5: **return** $\bigcup\{ \rho_2$:SHUFFLE(I, tail(ρ_1)) $\bullet$ insertI(head ρ_1, ρ_2)$\}$
6: **else**
7: **return** $\bigcup\{ \rho_2$:SHUFFLE(I, tail(ρ_1)) $\bullet$ insertO$_I$(head ρ_1, ρ_2)$\}$
8: **end procedure**

To construct *implied*, Algorithm 2 initialises it with the empty sequence (line 2) and then loops over the slots of ρ (lines 3–10). For each such slot $slots(\rho)\ i$, the set of traces above it according to $\trianglelefteq_I$ is calculated and recorded in another local variable *impliedset*. At the end of each iteration, *impliedset* is added to the end of *implied* (line 9).

The current slot is recorded in a variable *slot*, and the set of inputs that it contains (that is, the elements in the range of *slot* (ran *slot*) that belong to the set I) is recorded in another variable *Islot*. Afterwards, *impliedset* is initialised with the set of traces obtained from *slot* by shuffling its inputs. This is calculated by another Algorithm 3 explained below. We recall, however, that not only the order of the inputs can change, but inputs can be ignored. So, Algorithm 2 iterates over the inputs *inp* (lines 6–8), adding to *impliedset* the set of traces obtained by removing *inp* from each trace already in *impliedset*. The operator remove(e, ρ) defines the sequence obtained by removing occurrences of e from ρ.

Example 5.11. For the first slot $\langle e1.in, e2.in, e3.out, tock\rangle$ in Example 5.10, Algorithm 3 initialises *impliedset* as

$$\{ \langle e1.in, e2.in, e3.out, tock\rangle, \langle e1.in, e3.out, e2.in, tock\rangle,$$
$$\langle e3.out, e1.in, e2.in, tock\rangle, \langle e3.out, e2.in, e1.in, tock\rangle,$$
$$\langle e2.in, e1.in, e3.out, tock\rangle, \langle e2.in, e3.out, e1.in, tock\rangle \}$$

With remove$(e1.in, _)$, we add $\langle e2.in, e3.out, tock\rangle$ and $\langle e3.out, e2.in, tock\rangle$. With remove$(e2.in, _)$ applied to the enriched set, we get additionally $\langle e1.in, e3.out, tock\rangle$, $\langle e3.out, e1.in, tock\rangle$, and $\langle e3.out, tock\rangle$. This is the set in Example 5.10.

The recursive Algorithm 3 takes the set I of inputs and a trace ρ_1 as arguments. There are three cases that it considers. If ρ_1 is empty, then there is no shuffling to be done; the result is the set that contains just the empty trace itself (line 3). If ρ_1 starts with an input, the set calculated (line 5) is of traces from insertI$(\langle \text{head}\ \rho_1\rangle, \rho_2)$, where ρ_2 is a trace obtained by shuffling the rest of ρ_1 (tail ρ_1). The insertI(e, ρ) operator defined below specifies the set of sequences obtained by adding to ρ the (input) event e in every possible position before the first *tock* event in ρ, if any.

Definition 5.12.

$$\text{insertI}(e, \langle\rangle) = \{ \langle e\rangle \}$$
$$\text{insertI}(e, \langle tock\rangle \frown \rho) = \{ \langle e, tock\rangle \frown \rho \}$$
$$\text{insertI}(e_1, \langle e_2\rangle \frown \rho_1) = \{ \langle e_1, e_2\rangle \frown \rho_1, \langle e_2, e_1\rangle \frown \rho_1 \} \cup \{\rho_2 : \text{insertI}(e_1, \rho_1) \bullet \langle e_2\rangle \frown \rho_2 \} \quad \text{provided } e_2 \neq tock$$

There is only one position to insert an event e in an empty sequence: the result is the singleton set including $\langle e\rangle$. Equally, for a sequence starting with *tock*, the event can only go just before the *tock*. If, however, the sequence $\langle e_2\rangle \frown \rho_1$ starts with an event e_2 that is not *tock*, we can insert an event e_1, just before or after e_2, or in a later position in ρ_1. So, we consider the sequences ρ_2 resulting from inserting e_1 in ρ_1, and the set contains the result of adding e_2 to their start.

The last case in Algorithm 3 is for a trace ρ_1 that starts with an output event or *tock*. The set calculated is similar, but uses the insertion function insertO (line 7), which takes the set of inputs as argument and is defined as follows. The other arguments are an output, instead of an input like insertI, and the trace where the output is to be inserted.

Definition 5.13.

$$\text{insertO}_I(e, \langle\rangle) = \{ \langle e \rangle \}$$
$$\text{insertO}_I(e_1, \langle e_2 \rangle \frown \rho) = \{ \langle e_1, e_2 \rangle \frown \rho \} \quad \text{provided } e_2 \notin I$$
$$\text{insertO}_I(e_1, \langle e_2 \rangle \frown \rho_1) = \{ \langle e_1, e_2 \rangle \frown \rho_1, \langle e_2, e_1 \rangle \frown \rho_1 \} \cup \{ \rho_2 : \text{insertO}_I(e_1, \rho_1) \bullet \langle e_2 \rangle \frown \rho_2 \} \quad \text{provided } e_2 \in I$$

There is only one position to insert an output event e in an empty sequence: the result is the singleton set including $\langle e \rangle$. Since the order of outputs cannot change, for a trace starting with an output or *tock*, the insertion can happen only at the start. For a sequence $\langle e_2 \rangle \frown \rho_1$ starting with an input e_2, the definition of insertO is similar to that of insertI.

Optimisation. Algorithm 1 is very expensive. The verdict of our tests can be traced back to the design model, but it may be a heavy price to pay. A possible optimisation is to run all the tests generated from the reactive model. In the case of simulation experiments, they are useful to exercise the model and gather data anyway. If a tests passes, we can be confident of that verdict. If, however, a test fails, then we must consider whether the test is sound. Only then, we need to run Algorithm 1 to check whether we indeed have identified a problem with that particular test.

6 SOUNDNESS

We recall that Algorithm 1 takes as input a forbidden trace $ft \in TTTrace_{RC}$ of a (RoboChart) reactive model RC and finds the set *possibleTraces* of traces in $TTTrace_{RC}$ that are above ft under $\lesssim_{I_{RC}}$ (these are generated by Algorithm 2). Afterwards, ft is converted (for later use to generate a test case) if, and only if, none of the traces in *possibleTraces* is also a trace of RC (in the set $ett[\![\text{RC}]\!]$). In this section we prove that this proposed approach is sound: if a forbidden trace ft is converted, then the resulting trace $T_{RCS}(p, I, ft)$ is not a trace of $Cyclic(\text{RC}, p)$.

We start, in Section 6.1, by proving results regarding the equivalence relation $\equiv^R_{I_{RC}}$ and therefore $\lesssim_{I_{RC}}$ (see Definition 4.5). Specifically, we show that if an s-valid trace $\rho \in TTTrace_{RC}$ is equivalent to a trace of RC under $\equiv^R_{I_{RC}}$ then $T_{RCS}(p, I, \rho)$ is a trace of $Cyclic(\text{RC}, p)$. This is established in Theorem 6.4.

In Section 6.2, we use the notion of input-complete io-traces. An io-trace is defined to be input-complete if, and only if, it does not have a maximal non-empty sequence of *read* events that does not have a read event for every input in $events(\!| I_{RC} |\!)$. We are particularly interested in input-complete traces because, as we explained before, failures of the SUT are all input-complete io-traces. (This follows from the assumption that the SUT is a cyclic mechanism, and, therefore, accesses the inputs at the start of every cycle as required.) We prove that for every input-complete io-trace ρ_1 of $Cyclic(\text{RC}, p)$ there is an input-complete io-trace ρ_2 of $Cyclic(\text{RC}, p)$ such that $\rho_1 \lesssim_{S I_{RS}} \rho_2$ and ρ_2 can be formed by converting an s-valid trace of RC. This is established in Theorem 6.10. As a result, the set of input-complete traces of $Cyclic(\text{RC}, p)$ can be formed by first converting the s-valid traces of RC and then producing all traces that are either equivalent under $\equiv^S_{I_{RS}}$ (and so related by $\lesssim_{S I_{RS}}$ according to Definition 4.5) or below one of these under $\lesssim_{S I_{RS}}$.

Finally, in Section 6.3, we bring Theorems 6.4 and 6.10 together to prove soundness (Theorem 6.12).

In this section and the next we use the metavariables *inp* and *out* to represent events in the sets I_{RC} and O_{RC}, adding subscripts when extra metavariables are needed. We also use these metavariables to represent inputs (in I_{RS}) and outputs (in O_{RS}) of the SUT, that is, *inp*, inp_1, inp_2, and so on, for *read* events, and *out*, out_1, out_2, and so on, for *write* events. The context determines whether the events in question are from the reactive design or the cyclic SUT.

6.1 The order of inputs within a cycle

In this section we explore properties of $\equiv^R_{I_{RC}}$. We start by proving that conversion preserves equivalence (Lemmas 6.1 and 6.2). We then prove that if ρ is an s-valid trace of RC then $T_{RCS}(p, I, \rho)$ is a trace of $Cyclic(\mathrm{RC}, p)$ (Lemma 6.3). Finally, using these lemmas, we prove the main result of this section in Theorem 6.4. As said, we establish that if an s-valid trace $\rho \in TTrace_{RC}$ is equivalent to a trace of RC under $\equiv^R_{I_{RC}}$ then $T_{RCS}(p, I, \rho)$ is a trace of $Cyclic(\mathrm{RC}, p)$.

To start with, we prove that conversion preserves equivalence (in both directions), a result that we first establish in Lemma 6.1 below for traces that contain only one slot. For that, we observe that the conversion function $T_{RCS}(p, I, \rho)$ is well-defined for any p-compliant trace ρ, not only those that are fs-valid (or s-valid).

Lemma 6.1. *Given s-valid traces $\rho_1, \rho_2 \in TTrace_{RC}$ that have one slot, we have that: $\rho_1 \equiv^R_{I_{RC}} \rho_2$ if, and only if, $T_{RCS}(p, I_{RC}, \rho_1) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$.*

Proof. Since ρ_1 and ρ_2 contain only one slot, we have that

$$T_{RCS}(p, I_{RC}, \rho_1) = Iconv(\rho_1 \upharpoonright I_{RC}) \frown NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_1)) \frown Tconv(\rho_1 \upharpoonright (O \cup \{tock\}))$$

$$T_{RCS}(p, I_{RC}, \rho_2) = Iconv(\rho_2 \upharpoonright I_{RC}) \frown NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_2)) \frown Tconv(\rho_2 \upharpoonright (O \cup \{tock\}))$$

We therefore have the following.

$$T_{RCS}(p, I_{RC}, \rho_1) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$$

$$\Leftrightarrow \quad Iconv(\rho_1 \upharpoonright I_{RC}) \frown NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_1)) \frown Tconv(\rho_1 \upharpoonright (O \cup \{tock\})) \quad \text{[property above]}$$
$$\equiv^S_{I_{RS}}$$
$$Iconv(\rho_2 \upharpoonright I_{RC}) \frown NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_2)) \frown Tconv(\rho_2 \upharpoonright (O \cup \{tock\}))$$

$$\Leftrightarrow \quad Iconv(\rho_1 \upharpoonright I_{RC}) \equiv^S_{I_{RS}} Iconv(\rho_2 \upharpoonright I_{RC}) \wedge$$
$$NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_1)) = NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_2)) \wedge$$
$$Tconv(\rho_1 \upharpoonright (O \cup \{tock\})) = Tconv(\rho_2 \upharpoonright (O \cup \{tock\}))$$
$$\text{[definition of } \equiv^S_{I_{RS}}\text{]}$$

$$\Leftrightarrow \quad \rho_1 \upharpoonright I_{RC} \equiv^R_{I_{RC}} \rho_2 \upharpoonright I_{RC} \wedge NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_1)) = NIconv(events(\!|\, I_{RC} \,|\!) \setminus events(\operatorname{ran} \rho_2)) \wedge$$
$$Tconv(\rho_1 \upharpoonright (O \cup \{tock\})) = Tconv(\rho_2 \upharpoonright (O \cup \{tock\}))$$
$$\text{[definition of } Iconv \text{ and } \equiv^R_{I_{RC}}\text{]}$$

$$\Leftrightarrow \rho_1 \upharpoonright I_{RC} \equiv^R_{I_{RC}} \rho_2 \upharpoonright I_{RC} \wedge Tconv(\rho_1 \upharpoonright (O \cup \{tock\}) = Tconv(\rho_2 \upharpoonright (O \cup \{tock\})$$
$$[\rho_1 \upharpoonright I_{RC} \equiv^R_{I_{RC}} \rho_2 \upharpoonright I_{RC} \Longrightarrow I_{RC} \setminus events(\operatorname{ran} \rho_1) = I_{RC} \setminus events(\operatorname{ran} \rho_2)]$$

$$\Leftrightarrow \rho_1 \upharpoonright I_{RC} \equiv^R_{I_{RC}} \rho_2 \upharpoonright I_{RC} \wedge \rho_1 \upharpoonright (O \cup \{tock\}) = \rho_2 \upharpoonright (O \cup \{tock\}) \quad \text{[definition of } Tconv\text{]}$$

$$\Leftrightarrow \rho_1 \equiv^R_{I_{RC}} \rho_2 \quad \text{[definition of } \equiv^R_{I_{RC}} \text{ and } \rho_1, \rho_2 \text{ have one slot]}$$

□

We now extend this to traces with any number of slots.

Lemma 6.2. *Given s-valid traces $\rho_1, \rho_2 \in TTTrace_{RC}$, we have that $\rho_1 \equiv^R_{I_{RC}} \rho_2$ if, and only if,*

$$T_{RCS}(p, I_{RC}, \rho_1) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$$

Proof. By induction on the number of slots of ρ_2.

Base case. The result is immediate for the base case in which ρ_2 has zero slots and so is empty.

Inductive step. We assume that the result holds for every ρ_2 that has $k \geq 0$ or fewer slots. Let us suppose that ρ_2 has $k+1$ slots. Thus, $\rho_2 = \rho_2' \frown \rho_2''$ for the longest proper prefix ρ_2' of ρ_2 that has k slots. We start with $\rho_1 \equiv^R_{I_{RC}} \rho_2$ and prove that this holds if, and only if, $T_{RCS}(p, I_{RC}, \rho_1) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$. Since $\rho_1 \equiv^R_{I_{RC}} \rho_2$, we have that ρ_2 and ρ_1 have the same number of slots. Thus, we can write $\rho_1 = \rho_1' \frown \rho_1''$ for some longest proper prefix ρ_1' of ρ_1 that has k slots.

$$\begin{aligned}
&\rho_1 \equiv^R_{I_{RC}} \rho_2 \\
\Leftrightarrow\ &\rho_1' \equiv^R_{I_{RC}} \rho_2' \wedge \rho_1'' \equiv^R_{I_{RC}} \rho_2'' && \text{[definitions of s-valid, } \rho_2'\text{, and } \rho_1' \text{ and } \equiv^R_{I_{RC}}] \\
\Leftrightarrow\ &T_{RCS}(p, I_{RC}, \rho_1') \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2') \wedge \rho_1'' \equiv^R_{I_{RC}} \rho_2'' && \text{[inductive hypothesis]} \\
\Leftrightarrow\ &T_{RCS}(p, I_{RC}, \rho_1') \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2') \wedge T_{RCS}(p, I_{RC}, \rho_1'') \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2'') && \text{[Lemma 6.1]} \\
\Leftrightarrow\ &T_{RCS}(p, I_{RC}, \rho_1') \frown T_{RCS}(p, I_{RC}, \rho_1'') \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2') \frown T_{RCS}(p, I_{RC}, \rho_2'') && \text{[definition of } \equiv^S_{I_{RS}}] \\
\Leftrightarrow\ &T_{RCS}(p, I_{RC}, \rho_1' \frown \rho_1'') \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2' \frown \rho_2'') && \text{[definition of } T_{RCS}] \\
\Leftrightarrow\ &T_{RCS}(p, I_{RC}, \rho_1) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2) && \text{[definitions of } \rho_2', \rho_2'', \rho_1'\text{, and } \rho_1'']
\end{aligned}$$

□

We now prove that the s-valid traces of RC get mapped to traces of $Cyclic(\mathrm{RC}, p)$. In this proof, we use the standard 'after' operator: for a process P and trace ρ of P, we use 'P after ρ' to denote the process whose traces are exactly those that can follow ρ in the set of traces of P. Thus, ρ_2 is a trace of P after ρ_1 if, and only if, $\rho_1 \frown \rho_2$ is a trace of P.

Lemma 6.3. *If ρ is a trace of $[\![RC]\!]_{SV}$, then $T_{RCS}(p, I_{RC}, \rho)$ is a trace of $Cyclic(RC, p)$.*

Proof. By induction on the length of $slots_p(\rho)$.

Base case. Every process has the empty trace.

Inductive step. We assume that the result holds if the length of $slots_p(\rho)$ is at most $k \geq 0$, and we let ρ be an s-valid trace of RC such that $slots_p(\rho)$ has length $k+1$. Thus, $\rho = \rho_p \frown \rho_f$ for some maximal prefix ρ_p of ρ such that $slots_p(\rho_p)$ has length k. Both ρ_p and ρ_f are s-valid, and, by definition, $T_{RCS}(p, I_{RC}, \rho) = T_{RCS}(p, I_{RC}, \rho_p) \frown T_{RCS}(p, I_{RC}, \rho_f)$.

By the inductive hypothesis, $\rho_p^s = T_{RCS}(p, I_{RC}, \rho_p)$ is a trace of $Cyclic(\mathrm{RC}, p)$. By the definition of $Cyclic(\mathrm{RC}, p)$, once ρ_p^s has occurred, the process $Cyclic(\mathrm{RC}, p)$ – that is, $Cyclic(\mathrm{RC}, p)$ after ρ_p^s – is in the state in which $[\![\mathrm{RC}]\!]_{RCM}$ is in state ($[\![\mathrm{RC}]\!]_{RCM}$ after ρ_p) and the parallel process $SimAssump(I_{RC}, O_{RC}, p)$ is back in its initial state.

Further, by definition ρ_f is an s-valid trace of ($[\![\mathrm{RC}]\!]_{RCM}$ after ρ_p). We observe that $\rho_f \equiv^R_{I_{RC}} \rho_1 \frown \rho_2$ for some trace ρ_1 that contains only inputs and trace ρ_2 that contains only outputs and *tock* events. By the definition of $T_{RCS}(p, I_{RC}, \rho)$, we have that $T_{RCS}(p, I_{RC}, \rho_f) = \rho_3 \frown \rho_4 \frown \rho_5$ for $\rho_3 = Iconv(\rho_1)$, $\rho_4 = NIconv(events(\!| I_{RC} |\!) \setminus events(\mathrm{ran}\, \rho_1))$ and $\rho_5 = Tconv(\rho_2)$. In addition, by the definition of $Cyclic(\mathrm{RC}, p)$, we have that $\rho_3 \frown \rho_4 \frown \rho_5$ is a trace of $Cyclic(\mathrm{RC}, p)$ after ρ_p^s, since it satisfies the restrictions in $SimAssump(I_{RC}, O_{RC}, p)$. (Precisely, it is a trace of $SimAssump(I_{RC}, O_{RC}, p)$, after

hiding the events of $[[\mathrm{RC}]]_{RCM}$ and *end*, that is kept by the parallelism with $[[\mathrm{RC}]]_{RCM}$). With *Iconv* and *NIconv*, the restrictions of $TA1(I_{RC})$ are satisfied. With *Tconv*, the restrictions of $TA2(O_{RC}, p)$ are satisfied. The restrictions of $TA3(I_{RC}, O, p)$ are enforced by the fact that ρ_f is s-valid; here we observe that the set of traces of a process is prefix closed. So, even if the last slot of ρ_f does not cover all events of a cycle, we still get a trace of $Cyclic(\mathrm{RC}, p)$ after ρ_p^s. We thus have that $\rho_p^s \frown T_{RCS}(p, I_{RC}, \rho_f) = T_{RCS}(p, I_{RC}, \rho)$ is a trace of $Cyclic(\mathrm{RC}, p)$ as required. □

We now use the above lemmas and some in Section 4.2 to prove the main result from this section: if we convert a trace ρ that is equivalent to an s-valid trace of RC then we obtain a trace of $Cyclic(\mathrm{RC}, p)$.

Theorem 6.4. *Given an s-valid trace* $\rho \in TTTrace_{RC}$*, if* ρ *is equivalent to a trace of RC under* $\equiv^R_{I_{RC}}$*, then* $T_{RCS}(p, I_{RC}, \rho)$ *is a trace of* $Cyclic(RC, p)$.

Proof. Let ρ_1 be an s-valid trace of RC that is equivalent to ρ under $\equiv^R_{I_{RC}}$. By Lemma 6.3 we have that $T_{RCS}(p, I_{RC}, \rho_1)$ is a trace of $Cyclic(\mathrm{RC}, p)$. Since $\rho \equiv^R_{I_{RC}} \rho_1$, by Lemma 6.2 we have that $T_{RCS}(p, I_{RC}, \rho) \equiv^S_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_1)$. Thus, since the set of io-traces of $Cyclic(\mathrm{RC}, p)$ is closed under $\equiv^S_{I_{RS}}$ (Lemma 4.4), we have that $T_{RCS}(p, I_{RC}, \rho)$ is a trace of $Cyclic(\mathrm{RC}, p)$, because $T_{RCS}(p, I_{RC}, \rho_1)$ is an io-trace by definition. □

Because a simulation can ignore inputs, the above result does not hold in the opposite direction: $T_{RCS}(p, I_{RC}, \rho)$ being a trace of $Cyclic(\mathrm{RC}, p)$ does not imply that ρ is equivalent to a trace of RC under $\equiv^R_{I_{RC}}$.

Example 6.5. For instance, $\langle read.e.true \rangle$ is a trace of the simulation $Cyclic(\mathrm{RC}, p)$, if $e.in$ is an input event. If, however, RC does not have a transition from the initial state to accept e, then $Cyclic(\mathrm{RC}, p)$ ignores that input. In this case $\langle e.in \rangle$ is not a trace of RC, and that is the only trace ρ such that $T_{RCS}(p, \{e.in\}, \rho) = \langle read.e.true \rangle$.

We address this issue in the next section.

6.2 Software ignoring inputs within a cycle

In the previous section, we have explored the consequences of the relative order of inputs not mattering within a simulation. As already said, we have shown that if ρ is an s-valid trace of RC, then all traces equivalent to ρ under $\equiv_{I_{RC}}$ get mapped to traces of $Cyclic(\mathrm{RC}, p)$ by T_{RCS} (Theorem 6.4). In this section, we consider the consequences of the fact that a simulation can ignore inputs. The main result, captured by Theorem 6.10, concerns input-complete traces of $Cyclic(\mathrm{RC}, p)$. Theorem 6.10 tells us that an input-complete trace of $Cyclic(\mathrm{RC}, p)$ is always (equivalent to or) below, under $\leq s_{I_{RS}}$, another one that can be obtained by conversion. So, an input-complete trace of $Cyclic(\mathrm{RC}, p)$ is either in the image of T_{RCS} (when applied to s-valid traces of RC) or is in the downward closure of this image.

We start by proving a result similar to Lemma 6.2: if $\rho_1, \rho_2 \in TTTrace_{RC}$ are s-valid traces then $\rho_1 \trianglelefteq r_{I_{RC}} \rho_2$ if, and only if, $T_{RCS}(p, I_{RC}, \rho_1) \leq s_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$. We first prove this result for traces that have only one slot (Lemma 6.7). The proof uses the lemma below for traces of input events without repetitions. Here, we are concerned with repetitions regarding $events(\!| I_{RC} |\!)$. So, we require that there are no events $a.x$ and $a.y$ using the same input (channel) a.

Lemma 6.6. *Given traces* $\rho_1, \rho_2 \in TTTrace_{RC}$ *that have just inputs events, but no repetitions,*

$$(\rho_1 \trianglelefteq r_{I_{RC}} \rho_2)$$
$$\Leftrightarrow$$
$$Iconv(\rho_1) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\mathrm{ran}\,\rho_1)) \leq s_{I_{RS}} Iconv(\rho_2) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\mathrm{ran}\,\rho_2))$$

Proof. In the following, given an input event *inp* standing for *read.e.true*, for some *e*, we let $\bar{inp}$ denote *read.e.false* and we extend this notation to traces of inputs in the natural way.

$$\rho_1 \trianglelefteq r_{I_{RC}} \rho_2$$

$$\Leftrightarrow events(\operatorname{ran} \rho_2) \subseteq events(\operatorname{ran} \rho_1)$$

[definition of $\trianglelefteq r_{I_{RC}}$ ($\Rightarrow$), ρ_1 and ρ_2 are traces of input events without repetition ($\Leftarrow$)]

$$\Leftrightarrow events(\operatorname{ran} \rho_2) \subseteq events(\operatorname{ran} \rho_1) \land (events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \subseteq (events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2))$$

[property of sets]

$$\begin{aligned}\Leftrightarrow\ & events(\operatorname{ran} \rho_2) \subseteq events(\operatorname{ran} \rho_1) \land \\ & \exists \rho_3 : TTTrace_{RS} \bullet \\ & \quad events(\operatorname{ran} \rho_3) = Iconv(events(\operatorname{ran} \rho_1) \setminus events(\operatorname{ran} \rho_2)) \land Iconv(\rho_1) \equiv_{I_{RS}} Iconv(\rho_2) \frown \rho_3 \land \\ & \quad NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2)) \equiv_{I_{RS}} \bar{\rho_3} \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1))\end{aligned}$$

[ρ_3 is a trace of input events in ρ_1 that are not in ρ_2, and definitions of *Iconv*, *NIconv*, and $\equiv_{I_{RS}}$]

$$\begin{aligned}\Leftrightarrow\ & events(\operatorname{ran} \rho_2) \subseteq events(\operatorname{ran} \rho_1) \land \\ & \exists \rho_3 : TTTrace_{RS} \bullet events(\operatorname{ran} \rho_3) = Iconv(events(\operatorname{ran} \rho_1) \setminus events(\operatorname{ran} \rho_2)) \land \\ & \quad Iconv(\rho_1) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \equiv_{I_{RS}} \\ & \qquad Iconv(\rho_2) \frown \rho_3 \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \land \\ & \quad Iconv(\rho_2) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2)) \equiv_{I_{RS}} \\ & \qquad Iconv(\rho_2) \frown \bar{\rho_3} \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \land\end{aligned}$$

[the images of *Iconv* and *NIconv* are disjoint]

$$\begin{aligned}\Leftrightarrow\ & Iconv(\rho_1) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \lesssim s_{I_{RS}} \\ & \quad Iconv(\rho_2) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2))\end{aligned}$$

[definition of $\lesssim s_{I_{RS}}$ (which includes $\equiv_{I_{RS}}$)]

□

Lemma 6.7. *Given s-valid traces $\rho_1, \rho_2 \in TTTrace_{RC}$ that have one slot, $\rho_1 \trianglelefteq r_{I_{RC}} \rho_2$ if, and only if,*

$$T_{RCS}(p, I_{RC}, \rho_1) \lesssim s_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$$

Proof.

$$T_{RCS}(p, I_{RC}, \rho_1) \lesssim s_{I_{RS}} T_{RCS}(p, I_{RC}, \rho_2)$$

$$\begin{aligned}\Leftrightarrow\ & Iconv(\rho_1 \upharpoonright I_{RC}) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \frown Tconv(\rho_1 \upharpoonright (O \cup \{tock\}) \\ & \lesssim s_{I_{RS}} \\ & Iconv(\rho_2 \upharpoonright I_{RC}) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2)) \frown Tconv(\rho_2 \upharpoonright (O \cup \{tock\})\end{aligned}$$

[definition of $T_{RCS}(p, I_{RC}, \rho)$, and ρ_1 and ρ_2 have one slot]

$$\begin{aligned}\Leftrightarrow\ & Iconv(\rho_1 \upharpoonright I_{RC}) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_1)) \lesssim s_{I_{RS}} \\ & \quad Iconv(\rho_2 \upharpoonright I_{RC}) \frown NIconv(events(\!| I_{RC} |\!) \setminus events(\operatorname{ran} \rho_2)) \land \\ & Tconv(\rho_1 \upharpoonright (O \cup \{tock\}) = Tconv(\rho_2 \upharpoonright (O \cup \{tock\})\end{aligned}$$

[definition of $\lesssim s_{I_{RS}}$ and ρ_1 and ρ_2 have one slot]

$$\Leftrightarrow (\rho_1 \upharpoonright I_{RC}) \trianglelefteq r_{I_{RC}} (\rho_2 \upharpoonright I_{RC}) \land Tconv(\rho_1 \upharpoonright (O \cup \{tock\}) = Tconv(\rho_2 \upharpoonright (O \cup \{tock\})$$

[Lemma 6.6 with $\rho_1 \upharpoonright I_{RC}$ and $\rho_2 \upharpoonright I_{RC}$]

$\Leftrightarrow \rho_1 \upharpoonright I \trianglelefteq r_{I_{RC}}\ \rho_2 \upharpoonright I \wedge \rho_1 \upharpoonright (O \cup \{tock\}) = \rho_2 \upharpoonright (O \cup \{tock\})$ [definition of $Tconv$]

$\Leftrightarrow \rho_1 \trianglelefteq r_{I_{RC}}\ \rho_2$ [definition of $\trianglelefteq r_{I_{RC}}$ and ρ_1, ρ_2 having one slot]

□

We now extend this to all s-valid traces.

Lemma 6.8. *Given s-valid traces $\rho_2, \rho_1 \in TTTrace_{RC}$, we have that $\rho_1 \trianglelefteq r_{I_{RC}}\ \rho_2$ if, and only if,*

$T_{RCS}(p, I_{RC}, \rho_1) \lesssim s_{I_{RS}}\ T_{RCS}(p, I, \rho_2)$

Proof. By induction on the number of slots that ρ_2 contains.

Base case. The result immediately holds for the base case in which ρ_2 has zero slots and so is empty.

Inductive step. We assume the result holds for every ρ_2 that has $k \geq 0$ or fewer slots. Let us suppose that ρ_2 has $k+1$ slots. Thus, $\rho_2 = \rho_2' \frown \rho_2''$ for some longest proper prefix ρ_2' of ρ_2 that has k slots. We assume that $\rho_1 \trianglelefteq r_{I_{RC}}\ \rho_2$ and prove that $T_{RCS}(p, I, \rho_1) \lesssim s_{I_{RS}}\ T_{RCS}(p, I, \rho_2)$. Since $\rho_1 \trianglelefteq r_{I_{RC}}\ \rho_2$, we have that ρ_2 and ρ_1 have the same number of slots. Thus, we can write $\rho_1 = \rho_1' \frown \rho_1''$ for some longest proper prefix ρ_1' of ρ_1 that has k slots.

$\rho_1 \trianglelefteq r_{I_{RC}}\ \rho_2$

$\Leftrightarrow \rho_1' \trianglelefteq r_{I_{RC}}\ \rho_2' \wedge \rho_1'' \trianglelefteq r_{I_{RC}}\ \rho_2''$ [definitions of ρ_1', ρ_2', ρ_1'', ρ_2'', and $\trianglelefteq r_{I_{RC}}$]

$\Leftrightarrow T_{RCS}(p, I_{RC}, \rho_1') \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2') \wedge \rho_1'' \trianglelefteq r_{I_{RC}}\ \rho_2''$ [inductive hypothesis]

$\Leftrightarrow T_{RCS}(p, I_{RC}, \rho_1') \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2') \wedge T_{RCS}(p, I_{RC}, \rho_1'') \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2'')$ [Lemma 6.7]

$\Leftrightarrow T_{RCS}(p, I_{RC}, \rho_1') \frown T_{RCS}(p, I_{RC}, \rho_1'') \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2') \frown T_{RCS}(p, I_{RC}, \rho_2'')$ [definition of $\lesssim s_{I_{RS}}$]

$\Leftrightarrow T_{RCS}(p, I_{RC}, \rho_1' \frown \rho_1'') \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2' \frown \rho_2'')$ [definition of T_{RCS}]

$\Leftrightarrow T_{RCS}(p, I_{RC}, \rho_1) \lesssim s_{I_{RS}}\ T_{RCS}(p, I_{RC}, \rho_2)$

□

As said, traces that characterise failures of a cyclic SUT are input-complete. In the next lemma, used to prove our main result, we consider how such traces that are maximal under $\lesssim s_{I_{RS}}$ are related to the result of converting an s-valid trace.

Lemma 6.9. *If ρ_2 is an input-complete trace of $Cyclic(RC, p)$ such that there is no trace ρ_1 of $Cyclic(RC, p)$ with $\rho_1 \not\equiv^S_{I_{RS}}\ \rho_2$ and $\rho_2 \lesssim s_{I_{RS}}\ \rho_1$, then there is a trace $\rho \in [[RC]]_{SV}$ such that $T_{RCS}(p, I_{RC}, \rho) \equiv^S_{I_{RS}}\ \rho_2$.*

Proof. We consider how $Cyclic(\mathrm{RC}, p)$ can produce an input-complete trace ρ_2. If $Cyclic(\mathrm{RC}, p)$ were able to do so while ignoring some of the inputs received then we could remove these inputs that are ignored to produce a trace ρ_1 of $Cyclic(\mathrm{RC}, p)$ with $\rho_1 \not\equiv_{I_{RS}}\ \rho_2$ and $\rho_2 \lesssim s_{I_{RS}}\ \rho_1$. Thus, to produce the trace ρ_2, $Cyclic(\mathrm{RC}, p)$ cannot ignore any inputs. Now, let us suppose that $Cyclic(\mathrm{RC}, p)$ produces the trace ρ_2 and, in doing so, RC follows a trace ρ. By way of contradiction, we assume that $T_{RCS}(p, I, \rho) \equiv^S_{I_{RS}}\ \rho_2$ does not hold. In this case, there are four possibilities: (1) there is an input $read.e$ of ρ_2 that is not in $T_{RCS}(p, I, \rho)$; (2) there is an input $read.e$ of $T_{RCS}(p, I, \rho)$ that is not in ρ_2; (3) $T_{RCS}(p, I, \rho)$ and ρ_2 have exactly the same $read.e$ events in each of their corresponding subsequence of $read$ events, but there is at least one such event $read.e$ for which $read.e.false$ is in $T_{RCS}(p, I, \rho)$, and $read.e.true$ is in ρ_2; or (4) vice-versa, $read.e.true$

is in $T_{RCS}(p, I, \rho)$, and $read.e.false$ is in ρ_2. Scenario (1) is not possible, because the traces in the range of $T_{RCS}(p, I, \rho)$ are input-complete. Scenario (2) is not possible, because ρ_2 is input-complete by assumption. Scenario (3) is not possible because, since ρ needs all inputs in ρ_2, then $TA1(I_{RC})$ ensures that ρ has the event $e.in$ enabled by $read.e.true$. So, by definition of T_{RCS}, and of *Iconv* in particular, $read.e.true$ is in $T_{RCS}(p, I, \rho)$. Finally, scenario (4) is not possible because, if $read.e.false$ is in ρ_2, then $TA1(I_{RC})$ ensures that $e.in$ is not in ρ. So, by definition of T_{RCS}, and of *NIconv* in particular, $read.e.false$ is in $T_{RCS}(p, I, \rho)$. The result therefore follows. □

We finally give the main result in this section, which, as already said, shows how traces in $Cyclic(\mathrm{RC}, p)$ relate to the set of traces produced by converting s-valid traces of RC, that is, traces in $[\![RC]\!]_{SV}$.

Theorem 6.10. *If ρ_1 is an input-complete trace of $Cyclic(RC, p)$, then there is an input-complete trace ρ_2 of $Cyclic(RC, p)$ and an s-valid trace $\rho \in TTTrace_{RC}$ of RC (that is, ρ is in $[\![RC]\!]_{SV}$) such that $\rho_1 \lesssim s_{I_{RS}} \rho_2$ and $T_{RCS}(p, I, \rho) \equiv_{I_{RS}} \rho_2$.*

Proof. An input-complete trace ρ_2 of $Cyclic(\mathrm{RC}, p)$ is maximal under $\lesssim s_{I_{RS}}$ if there is no input-complete trace ρ_1 of $Cyclic(\mathrm{RC}, p)$ with $\rho_1 \not\equiv^S_{I_{RS}} \rho_2$ and $\rho_2 \lesssim s_{I_{RS}} \rho_1$. Since there are only finitely many channels, by the definition of $\lesssim s_{I_{RS}}$, there is an input-complete trace ρ_2 of $Cyclic(\mathrm{RC}, p)$, with $\rho_1 \lesssim s_{I_{RS}} \rho_2$, such that ρ_2 is maximal under $\lesssim s_{I_{RS}}$. From Lemma 6.9, we know that there is an s-valid trace $\rho \in TTTrace_{RC}$ of RC such that $T_{RCS}(p, I_{RC}, \rho) \equiv^S_{I_{RS}} \rho_2$. The result thus follows. □

Next, we use this result to reason about which forbidden traces of RC get mapped to forbidden traces of $Cyclic(\mathrm{RC}, p)$.

6.3 Sound test cases

Given a RoboChart model RC, we now define what it means for a trace to be implied by another trace. (These are the traces identified by Algorithm 2.) We also define, more generally, what it means for a trace to be implied by RC.

Definition 6.11. For an s-valid trace $\rho_2 \in TTTrace_{RC}$, we say that ρ_2 is *implied* by a trace ρ_1 if $\rho_2 \lesssim r_{I_{RC}} \rho_1$. Furthermore, for a RoboChart model RC, ρ_2 is *implied* by RC if there exists a trace $\rho_1 \in ett[\![RC]\!]$ of RC such that ρ_2 is implied by ρ_1.

The idea here is that this notion of *implied* mirrors the fact that the set of traces of $Cyclic(\mathrm{RC}, p)$ is downwardly closed under $\lesssim s_{I_{RS}}$. We recall that, given a trace ρ, Algorithm 2 returns the set of traces ρ_1 such that ρ is implied by ρ_1 (according to Definition 6.11). Algorithm 1 then assigns this set to the set *possibleTraces* and checks whether any of the traces in *possibleTraces* are traces of RC (are in $ett[\![RC]\!]$). Algorithm 1 is therefore checking whether ρ is implied by RC and only converts ρ to form a test case if this is not the case (that is, ρ is not implied by RC).

We now show that the required property holds: ρ is implied by RC if, and only if, $T_{RCS}(p, I, \rho)$ is a trace of $Cyclic(\mathrm{RC}, p)$.

Theorem 6.12. *Given an s-valid trace $\rho \in TTTrace_{RC}$, we have that ρ is implied by RC if, and only if,*

$$T_{RCS}(p, I, \rho) \in ett[\![Cyclic(RC, p)]\!]$$

Proof. We separately prove the two directions of the implication.

Case 1: ($\Rightarrow$)

ρ is implied by RC

$\Rightarrow \exists \rho_1 : [\![RC]\!]_{SV} \bullet \rho \lesssim r_{I_{RC}} \rho_1$ [definition of implied]

$\Rightarrow \exists \rho_1 : [\![RC]\!]_{SV} \bullet \rho \lesssim r_{I_{RC}} \rho_1 \wedge T_{RCS}(p, I, \rho_1) \in ett[\![Cyclic(\mathrm{RC}, p)]\!]$ [Lemma 6.3]

$\Rightarrow \exists \rho_1 : [[RC]]_{SV} \bullet \rho \precsim r_{I_{RC}} \rho_1 \wedge T_{RCS}(p, I, \rho_1) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge T_{RCS}(p, I, \rho) \precsim s_{I_{RS}} T_{RCS}(p, I, \rho_1)$

[Lemma 6.8]

$\Rightarrow T_{RCS}(p, I, \rho_1) \in ett[[Cyclic(\mathrm{RC}, p)]]$ [Lemma 4.9 and ρ is s-valid]

Case 2: ($\Leftarrow$) We name $\rho_1 = T_{RCS}(p, I, \rho)$.

$T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]]$

$\Rightarrow T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge \rho_1$ is input-complete [ρ_1 is in the image of T_{RCS}]

$\Rightarrow T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge$ [Theorem 6.10]
$\exists \rho_2 : ett[[Cyclic(\mathrm{RC}, p)]];\ \rho_3 \in [[\mathrm{RC}]]_{SV} \bullet \rho_2$ is input-complete $\wedge\ \rho_1 \precsim s_{I_{RS}} \rho_2 \wedge \rho_2 \equiv_{I_{RS}} T_{RCS}(p, I, \rho_3)$

$\Rightarrow T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge$ [definition of $\precsim s_{I_{RS}}$, $\rho_2 \equiv_{I_{RS}} T_{RCS}(p, I, \rho_3)$ and $\rho_1 \precsim s_{I_{RS}} \rho_2$]
$\exists \rho_2 : ett[[Cyclic(\mathrm{RC}, p)]];\ \rho_3 \in [[\mathrm{RC}]]_{SV} \bullet$
ρ_2 is input-complete $\wedge\ \rho_1 \precsim s_{I_{RS}} \rho_2 \wedge \rho_2 \equiv_{I_{RS}} T_{RCS}(p, I, \rho_3) \wedge \rho_1 \precsim s_{I_{RS}} T_{RCS}(p, I, \rho_3)$

$\Rightarrow T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge \exists \rho_3 : [[\mathrm{RC}]]_{SV} \bullet \rho_1 \precsim s_{I_{RS}} T_{RCS}(p, I, \rho_3)$ [predicate calculus]

$\Leftrightarrow T_{RCS}(p, I, \rho) \in ett[[Cyclic(\mathrm{RC}, p)]] \wedge \exists \rho_3 : [[\mathrm{RC}]]_{SV} \bullet T_{RCS}(p, I, \rho) \precsim s_{I_{RS}} T_{RCS}(p, I, \rho_3)$ [substitution]

$\Rightarrow \exists \rho_3 : [[\mathrm{RC}]]_{SV} \bullet \rho \precsim r_{I_{RC}} \rho_3$ [Lemma 6.8]

$\Rightarrow \rho$ is implied by RC [definition of implied]

□

This tells that if we take an s-valid trace that is not implied by RC and convert it, then the test case for the converted trace is sound because it is a trace that the simulation should not have.

We next consider completeness.

7 COMPLETENESS

In this section we show how a complete test suite can be produced from a specification given by a RoboChart model RC, or more precisely, its *tock*-CSP semantics. We recall that a test suite is complete if it is sound and exhaustive, and a test set is exhaustive if all possibly faulty implementations fail the test suite, that is, fail at least one test in the suite. We recall also that the test cases used are negative: they are for traces that the SUT should not have.

Many testing theories based on CSP define exhaustive test suites including just minimal forbidden traces: formed by a trace of the specification followed by a single forbidden event. The next example shows that this is not enough here.

Example 7.1. We consider the process RC2 from Example 5.8, and specification $S = \mathrm{RC2} \blacktriangleright 0$, which imposes a deadline 0 for termination of RC2. This removes *tock* from the traces of RC2 and simplifies the example, but is not key to the point here. Finally, we extend the output set to also contain $e4.out$. We recall that RC2 is defined by $e1.in \rightarrow e2.in \rightarrow$ **Stop** $\Box\ e2.in \rightarrow e3.out \rightarrow$ **Stop**. The following is the set of minimal (fs-valid) forbidden traces of RC2.

$$\{ \langle e3.out \rangle, \langle e4.out \rangle, \langle tock \rangle, \langle e1.in, e3.out \rangle, \langle e1.in, e4.out \rangle, \langle e1.in, tock \rangle, \langle e1.in, e2.in, e3.out \rangle, \langle e1.in, e2.in, e4.out \rangle, \langle e1.in, e2.in, tock \rangle, \langle e2.in, e4.out \rangle, \langle e2.in, tock \rangle, \langle e2.in, e3.out, e4.out \rangle, \langle e2.in, e3.out, tock \rangle \}$$

We consider an SUT defined by $Imp \blacktriangleright 0$, where Imp is the process defined as follows.

$$\begin{array}{l}
read.e1.true \rightarrow read.e2.true \rightarrow write.e3 \rightarrow write.e4 \rightarrow \mathbf{Stop} \\
\Box \\
read.e2.true \rightarrow read.e1.true \rightarrow write.e3 \rightarrow write.e4 \rightarrow \mathbf{Stop} \\
\Box \\
read.e1.true \rightarrow read.e2.false \rightarrow \mathbf{Stop} \\
\Box \\
read.e2.false \rightarrow read.e1.true \rightarrow \mathbf{Stop} \\
\Box \\
read.e1.false \rightarrow read.e2.true \rightarrow write.e3 \rightarrow \mathbf{Stop} \\
\Box \\
read.e2.true \rightarrow read.e1.false \rightarrow write.e3 \rightarrow \mathbf{Stop} \\
\Box \\
read.e1.false \rightarrow read.e2.false \rightarrow \mathbf{Stop} \\
\Box \\
read.e2.false \rightarrow read.e1.false \rightarrow \mathbf{Stop}
\end{array}$$

Imp has just one cycle and accepts the inputs in any order with any values as expected in a cyclic implementation. When both $e1$ and $e2$ happen, Imp outputs $e3$ and $e4$. When $e2$ happens, but not $e1$, Imp outputs $e3$. In all other cases, Imp produces no outputs. Imp is incorrect since it has, for example, the trace $\langle read.e1.true, read.e2.true, write.e3, write.e4\rangle$, and RC2 does not have any trace that includes $e4.out$. The only minimal forbidden trace of S that is converted to a trace of $Imp \blacktriangleright 0$ is $\langle e1.in, e2.in, e3.out\rangle$. All the other minimal forbidden traces are converted to traces that $Imp \blacktriangleright 0$ cannot perform, and so it would not fail the test. We note, however, that $\langle e1.in, e2.in, e3.out\rangle \trianglelefteq_{I_{RC}} \langle e2.in, e3.out\rangle$ and $\langle e2.in, e3.out\rangle$ is a trace of RC2, and so the presence of $\langle e1.in, e2.in, e3.out\rangle$ as a forbidden trace is not enough to generate a test to identify a failure. In conclusion, tests based on the minimal forbidden traces can let an incorrect SUT pass.

On the other hand, we can restrict ourselves to the set of s-valid forbidden traces defined below.

Definition 7.2. $TS_S(\text{RC}) = \{\rho : ETrace \mid \rho \notin [[\text{RC}]]_{SV} \wedge \rho \text{ is s-valid}\}$

We can take the set of traces in $TS_S(\text{RC})$, remove those implied by RC (using Algorithm 1) and use the resultant traces to construct test cases as already defined in the *tock*-CSP testing theory. We know that this approach is sound (Theorem 6.12) and we now show that the resultant test suite is exhaustive, and therefore complete.

Theorem 7.3. *The suite of tests obtained from traces of $TS_S(RC)$, after removing the traces implied by RC, is exhaustive.*

Proof. We assume that the SUT is a simulation RS with period p that is not a correct implementation of RC under $conf_p$. So, there must exist traces of RS that are not traces of $Cyclic(\text{RC}, p)$. Let ρ_1 be such a trace that is minimal, that is, there are no prefixes of ρ_1 that is not a trace of $Cyclic(\text{RC}, p)$. Since simulations are cyclic, we must have that ρ_1 is input-complete. By the definition of T_{RCS}, since ρ_1 is input-complete we have that there is an s-valid trace $\rho \in TTrace_{RC}$ such that $T_{RCS}(p, I, \rho) \equiv_{I_{RS}} \rho_1$. Since the set of traces of $Cyclic(\text{RC}, p)$ is closed under $\equiv_I^S$ (Lemma 4.4), then $T_{RCS}(p, I, \rho)$ is not a trace of $Cyclic(\text{RC}, p)$. So, by Theorem 6.12 we know that ρ is not implied by RC. This means that ρ is in the test suite produced by removing the traces implied by RC from $TS_S(\text{RC})$. The result therefore follows. □

Of course, the test suite $TS_S(\mathrm{RC})$ need not be finite and completeness is then in the limit: if the SUT is faulty and we apply the test cases then eventually the SUT will fail. In practice we might, for example, bound trace length.

8 CONCLUSIONS

There has been a sustained and substantial interest in model-based testing. When the model possesses a formal semantics, the potential arises for automatically generating sets of test cases that offer guarantees concerning effectiveness and soundness. This paper covers a scenario wherein testing is conducted from a reactive design (perhaps articulated in RoboChart) with the aim of testing a cyclic implementation (perhaps a simulation). The focus lies in generating tests from the abstract reactive model to interpret test results within that higher-level design context.

The proposed approach initiates with the identification of forbidden traces of a RoboChart model. For a reactive SUT, these traces can serve as the foundation for testing using the existing testing theory for *tock*-CSP. On the other hand, using the traces to define tests for a cyclic SUT needs to consider assumptions and restrictions inherent in the cyclic paradigm. We have identified the set of fs-valid forbidden traces that can be made useful when inputs and outputs correspond to data read and written during each period in a cyclic implementation.

We have introduced a technique that transforms an fs-valid trace from a reactive (RoboChart) model to consider the cyclic paradigm. In this process, input and output events are converted into read and write interactions of a simulation with platform services. Three algorithms specify testing campaigns based on the converted traces and their associated tests as defined by the *tock*-CSP testing theory. Finally, we have shown soundness and completeness of our approach.

Although our theory is concerned with RoboChart and RoboSim, the results 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, since the simulation of the platform and of the elements of the environment is kept separate.

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 implement a wrapper that captures the reactive view of a simulation, abstracting away the reading of input sensors and writing to actuators. There is currently ongoing work on automated instrumentation of simulations written using ROS, a widely used middleware for robotics, to allow direct use of tests derived of RoboChart models. That work also covers generation of ROS code for the tests themselves. For code used in deployment, however, instrumentation is not likely to be appropriate; for example, it interferes with time performance.

In the approach here, we discard traces of the reactive model that are not fs-valid (because they lead to useless tests for a cyclic SUT since they cannot fail). A practical approach needs to take the properties of fs-valid traces into account to generate and select enough tests using the RoboChart model to lead to converted tests providing good coverage of the SUT. Testing coverage of a cyclic SUT (or of its RoboSim model) is a topic for future work.

Another line of future work is the use of timewise refinement for conformance; in this case, we need to consider how refusal sets can be effectively converted. Moreover, there is a recognised need for substantial case studies. Leveraging RoboStar technology, which supports the automatic generation of simulations and integrates with testing from RT-Tester [30, 32], we plan to implement the technique outlined in this paper within that framework.

We also note that a RoboSim model is single rate. Although we can leave the definition of the cycle of a component underspecified (via use of loose constants or predicates), ultimately all state machines, controllers, and the module need to have the same cycle. A potential line of future work is to cater for multi-rate systems. For that, we need, first of all, to extend the semantics of RoboSim. On the other hand, even if a RoboSim module has controllers and machines at different rates, the overall simulation will have a single simulation rate: that of the module. The internal communications

affected by the different cycles are not visible. So, the definition of conformance and testing approach presented here are still relevant, and the issue is of compositionality. Different periods would need to be used to test different components. If, on the other hand, we want the internal connections to be visible, we need a different notion of conformance.

Given the compositional nature of RoboSim, it is also possible to think of an SUT where components are distributed. This raises issues of controllability and observability. As discussed in existing work [9, 15, 16, 37], for controllability, we would need to restrict attention to traces (or rather, test cases for traces) that are controllable, in the sense that each distributed test can determine when to apply its inputs. Conformance would need to take into account the inability of local tests to observe global behaviour – instead they observe their local projections.

ACKNOWLEDGEMENT

The authors would like to thank the RoboStar team for useful discussions on RoboChart and RoboSim, and are also grateful for very insightful comments from anonymous reviewers. Ana Cavalcanti is funded by the UK Royal Academy of Engineering under Grant No CiET1718/45, the UKRI (UK Research and Innovation Council) under Grants No EP/R025479/1 and EP/V026801/1, and by EU Horizon project RoboSapiens under agreement number 101133807. Robert M. Hierons is funded by the UKRI, under Grant No EP/R025134/1. For the purpose of open access, the authors have applied a creative commons attribution (CC BY) licence to any author accepted manuscript version arising.

REFERENCES

[1] J. Baxter, A. L. C. Cavalcanti, M. Gazda, and R. M. Hierons. 2023. Testing Using CSP Models: Time, Inputs, and Outputs. *ACM Transactions in Computational Logic* 24, 2 (2023). https://doi.org/10.1145/3572837

[2] J. Baxter, P. Ribeiro, and A. L. C. Cavalcanti. 2022. Sound reasoning in tock-CSP. *Acta Informatica* 59 (2022), 125–162. https://doi.org/10.1007/s00236-020-00394-3

[3] E. Brinksma and J. Tretmans. 2001. Testing Transition Systems: An Annotated Bibliography. In *MOVEP 2000 Summer School (Lecture Notes in Computer Science, Vol. 2067)*. Springer-Verlag, 187–195.

[4] L. B. Briones and E. Brinksma. 2005. Testing Real-Time Multi Input-Output Systems. In *7th International Conference on Formal Engineering Methods (Lecture Notes in Computer Science, Vol. 3785)*. Springer, 264–279.

[5] Laura Brandán Briones, Marcus Gerhold, Petra van den Bos, and Mariëlle Stoelinga. 2025. Time for Quiescence: Modelling Quiescent Behaviour in Testing via Time-Outs in Timed Automata. In *Testing Software and Systems - 37th IFIP WG 6.1 International Conference, ICTSS 2025, Limassol, Cyprus, September 17-19, 2025, Proceedings (Lecture Notes in Computer Science, Vol. 16107)*, Silvia Bonfanti and George Angelos Papadopoulos (Eds.). Springer, 35–52. https://doi.org/10.1007/978-3-032-05188-2_3

[6] A. L. C. Cavalcanti, J. Baxter, R. M. Hierons, and R. Lefticaru. 2019. Testing Robots using CSP. In *Tests and Proofs*, D. Beyer and C. Keller (Eds.). Springer, 21–38. https://doi.org/doi.org/10.1007/978-3-030-31157-5_2

[7] A. L. C. Cavalcanti and M.-C. Gaudel. 2007. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods (Lecture Notes in Computer Science, Vol. 4789)*. Springer-Verlag, 151–170. https://doi.org/10.1007/978-3-540-76650-6_10

[8] A. L. C. Cavalcanti and M.-C. Gaudel. 2011. Testing for Refinement in *Circus*. *Acta Informatica* 48, 2 (2011), 97–147. https://doi.org/10.1007/s00236-011-0133-z

[9] A. L. C. Cavalcanti, M.-C. Gaudel, and R. M. Hierons. 2011. Conformance Relations for Distributed Testing based on CSP. In *IFIP International Conference on Testing Software and Systems (Lecture Notes in Computer Science)*, B. Wolff and F. Zaidi (Eds.). Springer-Verlag. https://doi.org/10.1007/978-3-642-24580-0_5

[10] A. L. C. Cavalcanti and R. Hierons. 2023. Challenges in testing cyclic systems. In *27th International Conference on Engineering of Complex Computer Systems*. IEEE.

[11] A. L. C. Cavalcanti, R. Hierons, and S. Nogueira. 2020. Inputs and outputs in CSP: a model and a testing theory. *ACM Transactions on Computational Logic* (2020). https://doi.org/10.1145/3379508

[12] A. L. C. Cavalcanti and R. M. Hierons. 2013. Testing with Inputs and Outputs in CSP. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science, Vol. 7793)*. 359–374. https://doi.org/10.1007/978-3-642-37057-1_26

[13] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis. 2019. Verified simulation for robotics. *Science of Computer Programming* 174 (2019), 1–37. https://doi.org/doi.org/10.1016/j.scico.2019.01.004

[14] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* SE-4, 3 (1978), 178–187.

[15] Rachida Dssouli and Gregor von Bochmann. 1985. Error detection with multiple observers. In *Protocol Specification, Testing and Verification V*. Elsevier Science (North Holland), 483–494.

[16] Rachida Dssouli and Gregor von Bochmann. 1986. Conformance testing with multiple observers. In *Protocol Specification, Testing and Verification VI*. Elsevier Science (North Holland), 217–229.

[17] M.-C. Gaudel. 1995. Testing can be formal, too. In *International Joint Conference, Theory And Practice of Software Development (Lecture Notes in Computer Science, Vol. 915)*. Springer-Verlag, 82–96.

[18] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. 2018. A tool for domain-independent model mutation. *Science of Computer Programming* 163 (2018), 85–92.

[19] M. C. B. Hennessy. 1988. *Algebraic Theory of Processes*. MIT Press.

[20] F. C. Hennie. 1964. Fault-detecting experiments for sequential circuits. In *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*. Princeton, New Jersey, 95–110.

[21] R. M. Hierons. 2004. Testing from a Non-Deterministic Finite State Machine Using Adaptive State Counting. *IEEE Trans. Comput.* 53, 10 (2004), 1330–1342.

[22] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul J. Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. *ACM Comput. Surv.* 41, 2 (2009), 9:1–9:76. https://doi.org/10.1145/1459352.1459354

[23] Wen-ling Huang, Niklas Krafczyk, and Jan Peleska. 2024. Exhaustive property oriented model-based testing with symbolic finite state machines. *Sci. Comput. Program.* 231 (2024), 103005. https://doi.org/10.1016/J.SCICO.2023.103005

[24] W.-l. Huang and J. Peleska. 2017. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing* 29, 2 (2017), 335–364.

[25] Moez Krichen and Stavros Tripakis. 2004. Black-Box Conformance Testing for Real-Time Systems. In *11th International SPIN Workshop on Model Checking Software (Lecture Notes in Computer Science, Vol. 2989)*, Susanne Graf Laurent Mounier (Ed.). Springer, 109–126.

[26] G. L. Luo, G. v. Bochmann, and A. Petrenko. 1994. Test Selection Based on Communicating Nondeterministic Finite-State machines Using a Generalized Wp-Method. *IEEE Transactions on Software Engineering* 20, 2 (1994), 149–161.

[27] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. 2019. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling* 18, 5 (2019), 3097–3149. https://doi.org/doi.org/10.1007/s10270-018-00710-z

[28] OMG. 2017. OMG Systems Modeling Language (OMG SysML), Version 2.0. www.omg.org/spec/SysML/

[29] H. W. Park, A. Ramezani, and J. W. Grizzle. 2013. A Finite-State Machine for Accommodating Unexpected Large Ground-Height Variations in Bipedal Robot Walking. *IEEE Transactions on Robotics* 29, 2 (2013), 331–345.

[30] J. Peleska and W. Huang. 2016. 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, 3–22.

[31] J. Peleska and M. Siegel. 1996. Test automation of safety-critical reactive systems. In *Formal Methods Europe, Industrial Benefits and Advances in Formal Methods (Lecture Notes in Computer Science, Vol. 1051)*.

[32] J. Peleska, E. Vorobev, and F. Lapschies. 2011. 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, 298–312.

[33] Alexandre Petrenko, Adenilso da Silva Simão, and Nina Yevtushenko. 2012. Generating Checking Sequences for Nondeterministic Finite State Machines. In *Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*. 310–319.

[34] C. A. Rabbath. 2013. A Finite-State Machine for Collaborative Airlift with a Formation of Unmanned Air Vehicles. *Journal of Intelligent & Robotic Systems* 70, 1 (2013), 233–253.

[35] A. W. Roscoe. 2011. *Understanding Concurrent Systems*. Springer.

[36] Robert Sachtleben and Jan Peleska. 2022. Effective grey-box testing with partial FSM models. *Softw. Test. Verification Reliab.* 32, 2 (2022). https://doi.org/10.1002/STVR.1806

[37] Behcet Sarikaya and Gregor von Bochmann. 1984. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications* 32 (April 1984), 389–395.

[38] J. Schmaltz and J. Tretmans. 2008. On Conformance Testing for Timed Systems. In *6th International Conference on Formal Modeling and Analysis of Timed Systems (Lecture Notes in Computer Science, Vol. 5215)*. Springer, 250–264.

[39] The MathWorks,Inc. [n.d.]. *Stateflow and Stateflow Coder 7 User's Guide*. The MathWorks,Inc. www.mathworks.com/products.

[40] J. Tretmans. 2008. Formal Methods and Testing. Springer-Verlag, Chapter Model Based Testing with Labelled Transition Systems, 1–38.

[41] Frits W. Vaandrager, Paul Fiterau-Brostean, and Ivo Melse. 2024. Completeness of FSM Test Suites Reconsidered. *CoRR* abs/2410.19405 (2024). https://doi.org/10.48550/ARXIV.2410.19405 arXiv:2410.19405

[42] Gijs van Cuyck, Lars van Arragon, and Jan Tretmans. 2024. Testing Compositionality. In *Formal Aspects of Component Software - 20th International Conference, FACS 2024, Milan, Italy, September 9-10, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 15189)*, Diego Marmsoler and Meng Sun (Eds.). Springer, 39–56. https://doi.org/10.1007/978-3-031-71261-6_3

[43] M. van der Bijl, A. Rensink, and J. Tretmans. 2004. Compositional Testing with ioco. In *Formal Approaches to Software Testing*, A. Petrenko and A. UlrichAndreas (Eds.). Lecture Notes in Computer Science, Vol. 2931. Springer, 86–100.