Testing Robots using CSP

Ana Cavalcanti[1], James Baxter[1], Robert M. Hierons[2], and Raluca Lefticaru[2]

[1] Department of Computer Science, University of York, York, YO10 5GH, UK
[2] Department of Computer Science, University of Sheffield, Sheffield, S1 4DP, UK

Abstract. This paper presents a technique for automatic generation of tests for robotic systems based on a domain-specific notation called RoboChart. This is a UML-like diagrammatic notation that embeds a component model suitable for robotic systems, and supports the definition of behavioural models using enriched state machines that can feature time properties. The formal semantics of RoboChart is given using tock-CSP, a discrete-time variant of the process algebra CSP. In this paper, we use the example of a simple drone to illustrate an approach to generate tests from RoboChart models using a mutation tool called Wodel. From mutated models, tests are generated using the CSP model checker FDR. The testing theory of CSP justifies the soundness of the tests.

1 Introduction

RoboChart [38] is a domain-specific language for the design of robotic systems. Typically, robotic systems are described in the literature using state machines [50, 44, 45, 39] specified informally, with a notation that does not have even a precise syntax. Recently, a number of domain-specific notations have been proposed to enable tool support in the development of models, and automatic generation of code. RoboChart is distinctive in its support to specify timed properties, and in its formal semantics based on the process algebra CSP [32].

We can think of RoboChart as a profile for UML component and state-machine diagrams. It is, however, enriched with facilities to specify time budgets and deadlines. In RoboChart, a system is specified by a module, whose components identify a robotic platform and one or more controllers. The robotic platform identifies just the sensor and actuator functionality required for the system. These requirements are modelled by variables, operations, and atomic and instantaneous events that can communicate data.

Previous work on RoboChart has concentrated on verification by model checking [37] and theorem proving [21]. RoboTool [3] provides support for modelling and automatic generation of a CSP model of a RoboChart module. Extensive work has also been carried out in the verification of simulations of RoboChart models [17]. Extensions of RoboChart deal with collections [16] involving robots defined as instances of various modules, and with probabilistic properties [20].

A formal semantics also creates the opportunity for automatic generation of sound tests from RoboChart models. In this paper, we present an approach via a

[3] https://www.cs.york.ac.uk/robostar/

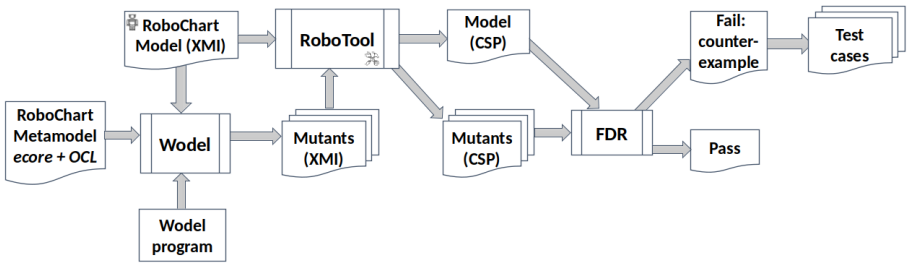

Fig. 1. Overview of our testing approach

simple rescue application that uses a drone. The approach uses mutation testing as supported by the tool Wodel [24, 26]. We describe a few mutation operators for RoboChart and the results of their application to our running example.

A high level overview of the approach is presented in Figure 1. The input to Wodel is an XMI representation of a RoboChart model (developed using RoboTool), the RoboChart metamodel, and Wodel programs that implement RoboChart mutation operators. Wodel applies the RoboChart mutation operators to generate mutants, and eliminates ill-formed mutants, that is, those that do not satisfy the well-formedness rules of RoboChart. For that, we use OCL definitions of the RoboChart rules, which we briefly describe here. The valid mutants can be loaded in RoboTool for analysis.

As already said, RoboTool generates CSP scripts for valid RoboChart models. For each mutant, using the CSP model checker FDR [23], we compare its CSP specification to that of the original model. If the mutant is not a refinement of the original model, FDR generates a trace of interactions common to both models and a continuation that is forbidden by the original model. This is what is needed to define a test for traces refinement as identified in the CSP testing theory [11]. We illustrate this approach for our drone mutants.

The structure of this paper is as follows. Section 2 gives an overview of RoboChart and introduces our running example. Wodel is presented in Section 3, and CSP and FDR in Section 4. Our testing approach is the subject of Section 5. Finally, we discuss related work in Section 6, and future work in Section 7.

2 RoboChart

Our example is a simple rescue application that uses a drone to deliver some relief (water, mask, and so on) to a given target location identified via some feature (a person or a vehicle, for instance) in a particular direction. Figure 2 presents the RoboChart model. The module, called Rescue, includes the definition of a robotic platform Drone and a controller Finder. In general, a module can have several controllers running in parallel; in our example we have just one.

The platform provides two operations: move(lv: nat) and turnBack(), and defines five events: switchOn, takeoff, land, found, and origin. They are abstractions

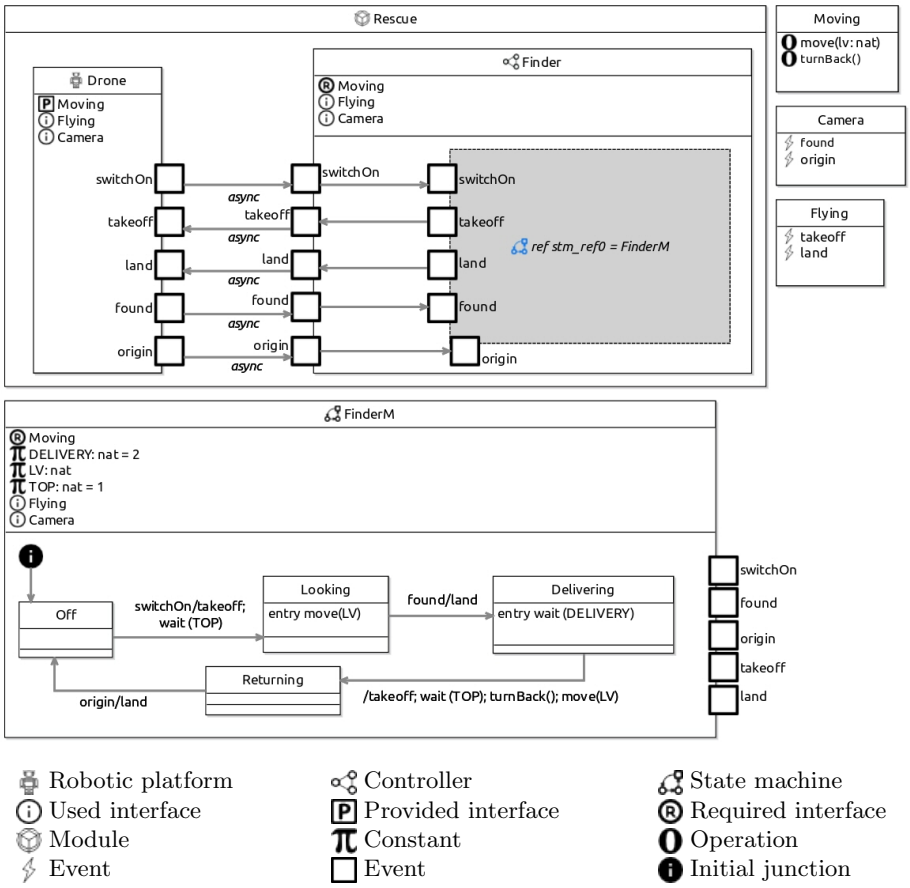

Fig. 2. RoboChart model of a simple rescue drone

for sensors and actuators. We have an on/off button, represented by the event switchOn, and a motor that can be used to raise the drone off the ground, abstracted by the event takeoff. Using the motor, we can also request that the drone lands, moves forward with a particular speed, or turns back to return to its origin: this is abstracted by the events land, and operations move(lv: nat) and turnBack(). Finally, a camera can be used to identify the target of the rescue operation, abstracted by the event found, or the origin.

Operations and events can be declared in interfaces. In the example, we have interfaces Flying, Moving, and Camera also in the controller to define that it requires all of the operations provided by the platform, and that it has the same events. Connections between events define the dataflow. In general, connections can be between events of different names, but of the same type. In our simple example, we use the same event names in every component.

Connections with the platform are asynchronous. Connections between controllers can be either synchronous or asynchronous. Although implementations are typically asynchronous, synchronicity can be defined for abstraction.

The behaviour of a controller is defined by one or more state machines. The use of several state machines can represent different threads of computation or provide modular description of functionality. In our example, we have just one state machine FinderM. The notation is, by and large, standard.

Of note in the state machine notation is the fact that it is a self-contained component that declares all the required variables and operations, and events that it uses. In our example, these are all those provided by the platform (and required by the controller for use in the machine). This means that a state machine can be treated independently in verification, simulation, and testing.

A machine can also declare local variables, constants, and clocks. In our example, we have three constants DELIVERY, LV, and TOP, which define the amount of time that the drone stays on the ground once it finds its target, the speed with which it moves, and the time it takes to take off.

The RoboChart notation to specify entry, during, exit, and transition actions is well defined. It is a language with assignments, operation calls, sequence, conditional, and inputs or outputs via events. Of note is the availability of time primitives. For instance, wait(TOP) is an action that pauses for TOP time units. In the example, it is used in the transition from the state Off to the state Looking.

The behaviour of the Rescue system is as follows. In the initial state Off, indicated by the initial junction, it accepts a request to switchOn, and, as a result, there is a takeoff. After that, the drone waits for TOP time units before it moves to the state Looking. Upon entry in Looking, the operation move(LV) is called and the drone proceeds until the target is reached as signalled by the input event found. When that happens, the controller issues a command to land. The drone moves to the state Delivering, whose entry action forces a pause of DELIVERY time units. Afterwards, the transition to the state Returning is taken, which causes the drone to takeoff, turnBack, and move again. In Returning, the transition back to Off is taken when the origin is found, and then the drone lands.

A full account of RoboChart, including its semantics, can be found in [38]. Several other examples are available at https://www.cs.york.ac.uk/robostar/, where the project files for this example can also be found.

3 Mutation testing and Wodel

The idea behind mutation testing is that we take an entity p, such as a piece of code or a model, and use *mutation operators* to change (mutate) p in order to simulate potential faults. If we have an initial entity p and a set M of mutants, then a test suite T is assessed by determining what proportion of the (non-equivalent) elements of M are distinguished from p by T (are *killed* by T).

The essential concept is that if a test suite T is good at distinguishing p from its mutants, then T is also good at distinguishing p from some unknown correct version of p (if p is faulty). In addition, mutation testing can be used to

drive test-case generation: given a non-equivalent mutant m of p, one might aim to generate a test case that kills m. This is the use of mutation testing that we discuss here, although we use refinement rather than equivalence.

There is potential to automate many parts of mutation testing, such as the generation of mutants and the execution of test cases on the mutants. As a result, the application of mutation testing is typically supported by a tool. However, mutation testing tools are normally language specific, leading to the need to develop a new tool whenever we consider a new language.

This has motivated the development of Wodel, which is a domain-specific language and tool for model-based mutation [24, 26]. Wodel has been used in a range of case studies, for example, automatic generation of exercises for automata training [25] or mutation of security policies [26]. Wodel is metamodel independent, which means that users can define their own mutation operators for arbitrary metamodels – Wodel comes with some predefined examples: finite automata, probabilistic automata, and UML class diagrams.

Wodel is based on Eclipse Modelling Framework (EMF) and is available as an Eclipse plugin. The framework provides an editor to define the mutation operators, a compiler that transforms Wodel programs into Java code, metrics for mutation footprints, which provide information about the static and dynamic coverage of a metamodel and models used, a seed model synthesizer, and an extensibility mechanism that allows pipelining external applications.

Wodel provides high-level mutation primitives, such as, creation, deletion, reference reversal, attribute modification, object retyping, and object cloning, together with strategies for their customization and support for composition of mutation operators. The Wodel IDE provides an easy way for adding extension points, which allows users to register domain-specific post-processors to be executed upon mutant generation, for instance, to identify mutant equivalence.

Section 5 gives example of Wodel statements to implement operators.

4 CSP and FDR

Communicating Sequential Processes (CSP) [46] is a process algebra. Computation is modelled by processes, whose behaviour, in its simplest form, is described in terms of traces, that is, sequences of events. A CSP event is an atomic and instantaneous communication on a channel that may be represented by a simple flag, or carry values of particular types as parameters.

Processes can be defined using the basic processes *Stop*, representing deadlock, *Skip*, representing termination, and *DIV*, representing divergence (that is, livelock). Events can be prefixed to a process P. For example, $c.e \rightarrow P$ is a process that is ready to engage in the event $c.e$ and then behave like P.

CSP events are defined by a channel name and, optionally, parameters. As illustrated, a parameter e can be appended to a channel name c using a dot ($c.e$). This represents a communication of e via c. An exclamation mark can be used ($c!e$) to indicate that e is output on c. Use $c?x$ of a question mark indicates that the parameter is accepted as input and bound to the variable name x.

Processes P and Q can be combined using various operators, such as, internal (nondeterministic) choice ($P \sqcap Q$) for the process, external choice ($P \Box Q$) made by the environment, parallel interleaving ($P \,|||\, Q$), parallel composition with synchronisation on an alphabet of events A ($P \,[\![A]\!]\, Q$), and interrupt ($P \triangle Q$).

A process Q is said to refine another process P, written $P \sqsubseteq Q$, if every behaviour of Q is a possible behaviour of P. This allows for incremental development of a correct program from a specification of how it should behave. In our work, refinement is the conformance relation used in testing. So, a mutant that merely refines the original model is not useful: it does not identify a fault.

CSP has various semantic models that permit reasoning about processes. These models vary in the aspects of behaviour that they can capture, and, therefore, in the processes they can distinguish. The most commonly used semantic models for CSP are the traces model, the stable failures model, and the failures-divergences model. Here, we consider just the traces model, and traces-refinement. We say that P is traces refined by Q, written $P \sqsubseteq_T Q$ if the set of traces of Q is included in that of P. This is our notion of conformance here.

To capture the timed behaviour of RoboChart models, we use a variant of CSP that includes a special event *tock* to mark the passage of time. The testing theory for this version of CSP is ongoing work, based on the testing theory for the refusal-testing semantics of CSP. Here, as already mentioned, we consider just traces refinement. We note, however, that testing in CSP-based theories can consider traces refinement [11, 12] in isolation, and use an additional conformance relation *conf* to deal with refusals. Exhaustive test sets for the richer notions of refinement include the exhaustive test sets for traces refinement and *conf*. We expect that the same approach works for tock-CSP and its notion of refinement.

Using their semantic models, CSP processes can be reasoned about using mathematical proof, but automatic analysis of finite-state CSP processes can be performed using model checking. This is implemented by the tool FDR [23], which checks whether one process refines another, and can produce counterexamples when a refinement does not hold. The semantics for RoboChart models is calculated in RoboTool using the ASCII syntax for CSP (called CSP-M) that enables checking of RoboChart models using FDR. We use checking of RoboChart models in FDR as part of the testing strategy described next.

5 Mutation-driven testing

In this section, we present our approach to test generation using Wodel and FDR. First, we present in Section 5.1 a few mutation operators and their implementation in Wodel. We then discuss how ill-formed mutants are discarded in Section 5.2. Finally, in Section 5.3, we explain how we generate tests.

5.1 Mutation operators

To generate tests from RoboChart models, we have used a number of mutation operators inspired by other works that use Wodel [24–26], and that consider

Mutation	Wodel blocks	#
mStActEnDu	**retype one** EntryAction **as** DuringAction // *modifies a state by changing an entry action into a during action*	2
mStActEnEx	**retype one** EntryAction **as** ExitAction // *modifies entry into exit action*	2
mTransSource	tr = **select one** Transition **where** {^source <> **one** Initial} **modify target** ^source **from** tr **to other** State // *changes the start state of a transition, except the one from the initial junction*	12
mTransTarget	**modify target** ^target **from one** Transition **to other** State // *changes the ending state of a transition*	15
mTransTrigger	interf = **select one** Interface **where** {events <> **null**} ev = **select one** Event **in** interf->events tg = **create** Trigger **with** {event = ev} **modify one** Transition **with** {trigger = tg} // *modifies a transition by replacing its trigger with another event*	6
rSeqStatement	ss = **select one** SeqStatement **remove one** Statement **from** ss->statements [1..5] **remove all** SeqStatement **where** {statements = **null**} // *randomly deletes 1–4 statements from a sequence and all empty sequences*	12
rState	st = **select one** State **remove all** Transition **where** {^source = st} **remove all** Transition **where** {^target = st} **remove** st // *removes a non initial state and all transitions from or to that state*	3
rTran	**remove one** Transition **where** {^source <> **one** Initial} // *deletes one transition, except the one from initial junction*	4
rTranAction	tr = **select one** Transition **where** {action <> **null**} **remove one** Call **from** tr->action **remove one** SendEvent **from** tr->action **remove one** Action **from** tr // *removes the action associated with a transition (call or send event)*	2
Total		**58**

Table 1. Example of mutations used and corresponding Wodel blocks

mutations for UML class diagrams [27], state models [49], or interfaces [19]. We have adapted the operators to the particularities of the RoboChart metamodel. Many of them apply to elements of state machines.

Some operators that we have applied to the example from Figure 2 are given in Table 1. In the first column, we give the name we have assigned to the mutation operator. The second column gives a block of Wodel statements that implements the operator. The third column gives the number of different valid mutants generated using the mutation operator. In total, for the example in Figure 2, we have used 9 mutation operators and generated 58 valid mutants.

In RoboChart, a state can have associated actions identified in the metamodel as `EntryAction`, `DuringAction`, or `ExitAction`, all having as super type `Action`. We can use the Wodel statement **retype** to change one object type with another – the implementations of the mutation operators mStActEnDu and mStActEnEx use **retype** to change the type of a state action.

Wodel provides flexible statements such as **remove**, **create** or **clone**, which make it possible to delete or create new objects. These statements can be used in conjunction with different selection strategies, such as **select one**, or **all**, or as specified by a clause **where** *criteria*. Also, it is possible to change objects using a statement **modify** *object selection strategy* **with** *attribute set*.

We can redirect the **source** or **target** of a reference to another object, as illustrated by the implementation of the operators mTransSource and mTransTarget. A RoboChart `Transition` has the source and target states specified by attributes called `source` and `target` in the RoboChart metamodel. These attribute names, however, are also Wodel keywords. To differentiate between Wodel syntax and RoboChart metamodel elements, the latter are preceded by a caret symbol `^` (a special notation used in the Xtext[4]-based editor for Wodel programs in order to avoid this duplicity). This is illustrated by the implementation of the mutation operators mTransSource, mTransTarget, and rState.

In Wodel, it is possible to compose statements, as shown in the implementation of rSeqStatement, where a **remove** statement is repeated a random number of times between [1..5], to delete an action from a sequence. If all the statements have been deleted, then the empty `SeqStatement` element is further removed. Similarly, elements that would become invalid (having null attributes) are deleted from the model. For instance, the incoming or outgoing transitions from a state that is removed are also removed – see the rState operator.

The output of Wodel consists in XMI files; they have the same structure as the initial model, and describe mutants that conform to the metamodel and well-formedness rules of RoboChart. For Wodel validation of the mutants we have embedded in the metamodel the RoboChart well-formedness rules using the OCLinEcore language. They are further discussed in the next section.

For illustration, we include in Figures 3 and 4 the state machines for two mutants of the model in Figure 2. The module and controller of the mutants are the same, except that their names are changed to make them unique in a

[4] https://www.eclipse.org/Xtext/

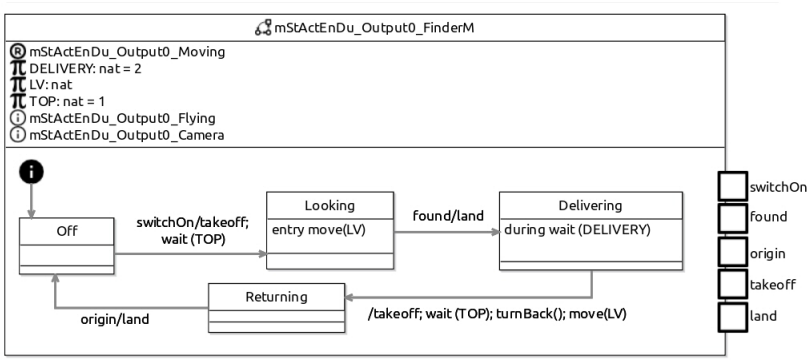

Fig. 3. Mutant for RoboChart model in Figure 2 – the Delivering state has its entry action changed into a during action (mStActEnDu operator)

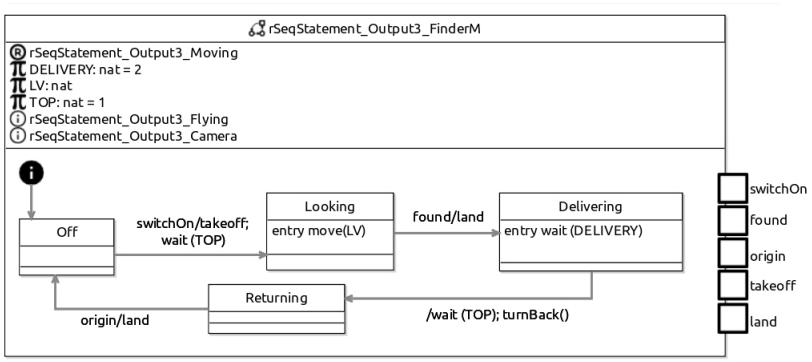

Fig. 4. Mutant for RoboChart model in Figure 2 – two actions are removed from the sequence in the transition from the Delivering state (rSeqStatement operator)

context (like a RoboChart package, for example) that includes both the original model and the mutant. We explain below why this is important.

The machines in the mutants, on the other hand, are obtained by applying two different mutation operators. For the mutant in Figure 3, we have used the operator mStActEnDu that changes the entry action of a state to a during action. We have applied it to the Delivering state in Figure 2. To obtain the mutant in Figure 4, we have used rSeqStatement. We have applied this operator to the action in the transition from Delivering, and it has removed two basic RoboChart actions: the event takeoff and the call move(LV).

5.2 OCL constraints

Application of some operators may lead to an invalid mutant, that is, a mutant that does not satisfy the RoboChart well-formedness conditions. An example

```
context Connection
  inv Cn9: (not self.bidirec
    and ControllerDef.allInstances()
      ->exists(c | c.connections->includes(self))
    and self.from.oclIsKindOf(StateMachine)
  ) implies self.from.oclAsType(StateMachine).stmDef()
          .ncInputEvents()->excludes(self.efrom)
```

Fig. 5. OCL constraint for RoboChart well-formedness condition Cn9

of such an operator is mTransTrigger, which changes the event in a transition trigger. An application of mTransTrigger may give rise to a transition whose trigger event is associated with an outgoing (non-bidirectional) connection from the state machine. In this case, the result is an event that is used as an input, but associated with a connection that indicates it is used as an output. This violates the RoboChart well-formedness condition called Cn9 stating that events connected by such connections must not be used as inputs.

To exclude invalid mutants, we have translated the well-formedness conditions of RoboChart into Object Constraint Language (OCL), a language for specifying constraints on the structure of a metamodel. This allows the conditions to be considered by Wodel along with the RoboChart metamodel. Wodel then eliminates mutants that are not valid according to the constraints.

As an example, we provide in Figure 5 the OCL constraint for the RoboChart well-formedness condition just described. The **context** declaration at the start indicates it is a constraint on a `Connection`, the RoboChart metamodel element representing connections. The constraint `Cn9` is specified as an **inv**ariant for this model element. It is formalised as an implication (**implies**), where the antecedent identifies the connections to which the constraint applies. Specifically, it applies when `self`, the initial connection being considered, is **not** `bidirec`tional, is contained in the `connections` of some `ControllerDef` (that is, it is a connection of a controller rather than a module), and has a source (`self.from`) of type (`oclIsKindOf`) `StateMachine` (that is, it does not connect from the boundary of the containing controller). `ControllerDef.allInstances()` identifies all controllers in the model, and `exists(c | c.connections->includes(self))` requires that there exists such a controller `c` whose `connections` `includes` `self`.

Where these conditions are met, the constraint checks the source, `from`, of `self`, casting it to the `StateMachine` type and applying a function `stmDef()` to obtain the definition of the state machine. The events used as inputs by the state machine are then identified by another function `ncInputEvents()`, and the event that `self` connects from (`efrom`) is required to not be among them.

The full set of OCL constraints can be found in the RoboChart reference manual [52]. These constraints cover most of the well-formedness conditions of RoboChart, but there are three that are not possible to define in OCL without an impractical amount of effort. The first is Cn4, which requires types of the

source and target events of a connection to agree. Checking this would involve an implementation of a large part of the RoboChart type system within OCL.

The second condition not included in our OCL constraints is V1, which requires that the initial value of a variable must agree with those of the declarations of that variable in outer scopes. Since the initial value may be given by an arbitrarily complex expression, this requires an evaluator for RoboChart expressions. Due to the complexity of the RoboChart expression language, this is non-trivial.

The final such well-formedness condition is J2, which states that the guards of the outgoing transitions of a junction must form a cover. Since these guards are arbitrary expressions that need to be checked for all instantiations of the variables within them, this is a condition that can, in general, only be checked by a theorem prover. It is beyond the capacity of OCL to express such a constraint.

The files with the Wodel mutants can be imported back into RoboTool, so that their tock-CSP model is generated. Mutants that are not well formed, because they do not satisfy Cn4 or V1, are identified by RoboTool. For those, no CSP model is generated. For those that do not satisfy J2, a CSP model is generated, but a deadlock check can be used to identify the issue.

So, there is no real problem in importing potentially invalid mutants into RoboTool. Only tests based on valid mutants are generated, as explained next.

5.3 Test generation

With the CSP models for both the original model and for a mutant (automatically generated by RoboTool), we can use the FDR model checker to generate tests (for traces refinement). This is achieved by checking whether the original model is traces-refined by the mutant. If it is, the check passes, and in this case, the mutant does not identify a fault. No test is generated. Of the 58 mutants in our example, five do not identify a fault and so are not useful.

If the mutant does identify a fault, FDR provides a counterexample for the check. This is a trace that is common to both models, except for its last event, which is allowed by the mutant, but not the original model. This last event is, therefore, a forbidden continuation for the preceding trace. For example, for the mutant in Figure 3, the check raises the following counterexample.

```
Rescue_switchOn.in -> Rescue_takeoff.out -> tock ->
moveCall.1 -> moveRet -> Rescue_found.in -> Rescue_land.out ->
Rescue_takeoff.out
```

This indicates that, for the `Rescue` module, if we observe interactions characterised by the events `switchOn` and `takeoff`, and, after one time unit, the time required for taking off, we observe a call and return of the operation move with parameter 1, and then the events `found` and `land`, there should not be an immediate takeoff. This would not allow time for the delivery of relief. Such undesirable behaviour arises in the mutant, because the change of the entry action of Delivering to a during action allows it to be interrupted by takeoff.

For the mutant in Figure 4, the check raises the following counterexample.

```
Rescue_switchOn.in -> Rescue_takeoff.out -> tock ->
moveCall.1 -> moveRet -> Rescue_found.in -> Rescue_land.out ->
tock -> tock -> tock ->
turnBackCall
```

Here, the forbidden continuation is the call of turnBack after the delivery and take off time (three time units), but without the actual takeoff. In this case, we have an attempt to turn the drone on the floor, which may damage it. The mutant captures this fault because its takeoff event has been removed.

These traces and forbidden continuations are exactly the data that we need to construct a test for traces refinement. In the CSP testing theory [11], a test is characterised by a function $T_T(t, e)$ for a trace t and a forbidden continuation. In the first example above, the trace is shown below.

$$\langle\, \mathit{Rescue\_switchOn.in}, \mathit{Rescue\_takeoff.out}, \mathit{tock},\\ \mathit{moveCall}.1, \mathit{moveRet}, \mathit{Rescue\_found.in}, \mathit{Rescue\_land.out}\,\rangle$$

The forbidden continuation is $\mathit{Rescue\_takeoff.out}$. As may be expected, the use of the RoboChart events and operations of the robotic platform of a module are captured in the CSP model as CSP events. The CSP events for RoboChart events are tagged to indicate whether they are *in*puts or *out*puts, and the CSP events for operation calls are tagged with the arguments.

The test corresponding to this trace and forbidden continuation is characterised as a CSP process and it is shown below. It uses special events to indicate whether the verdict of the test execution is *inc*onclusive, *fail* or *pass*.

$$\begin{array}{l}\mathit{inc} \rightarrow \mathit{Rescue\_switchOn.in} \rightarrow \mathit{inc} \rightarrow \mathit{Rescue\_takeoff.out} \rightarrow \mathit{inc} \rightarrow \mathit{tock} \rightarrow\\ \mathit{inc} \rightarrow \mathit{moveCall}.1 \rightarrow \mathit{inc} \rightarrow \mathit{moveRet} \rightarrow\\ \mathit{inc} \rightarrow \mathit{Rescue\_found.in} \rightarrow \mathit{inc} \rightarrow \mathit{Rescue\_land.out} \rightarrow\\ \mathit{pass} \rightarrow\\ \mathit{Rescue\_takeoff.out} \rightarrow \mathit{fail} \rightarrow \mathit{Stop}\end{array}$$

The verdict is given by the last (special) event observed before a deadlock. So, the test first raises the *inc* event, and then offers the input to switchOn the drone. If it is refused, we have a deadlock, and the verdict is *inc*onclusive as it has not been possible to drive the drone through the trace of events of interest for this test. If it is accepted, an additional *inc* event is raised, and the output takeoff is expected. This goes on, until the last event land of the trace is observed, when the test raises the verdict event *pass*. If there is now a deadlock, the test passes. If, however, we observe the forbidden event takeoff, then there is a fault, and the verdict event *fail* is finally raised, before the test deadlocks.

The testing theory of CSP guarantees that this is a sound test. Any fault identified is indeed a fault. The testing theory also guarantees exhaustiveness: if all traces of the original model and all their forbidden continuations are considered, then we can uncover all faults if we can reveal all behaviours under testing. These are, however, too many tests. Mutation helps us to select some.

6 Related work

Mutation has been used in connection with several notations to guide test-generation or assess the quality of a test suite [48, 1, 3]. As said above, it tackles the explosion of test cases by selecting on which errors to concentrate [22, 3, 33].

Budd and Gopal [10] have pioneered work on mutation testing with specifications based on predicate calculus. Mutation has been widely studied, and applied using model checking [8, 29] like here, and in the context of Simulink [9], white-box testing [43], contracts [34], and security properties [54, 48]. Other related works specifically for diagrammatic models, include mutation testing for state and activity models [49], UML class diagrams [27], interfaces [19], or component-based real-time systems described using a graph notation [28].

Aichernig et al. have considered various formalisms [2, 1, 3, 5]. In those works, a test case is an abstraction according to traces refinement of the specification, and so an implementation should refine the test case if it passes the test. Test-case generation is, therefore, a reverse-refinement problem. We adopt a more standard approach and use (failed) traces-refinement checks to identify tests.

For CSP, Clark et al. [48] presents mutant operators and uses mutation testing for checking system security; the goal is to validate the specifications rather than generate tests. For *Circus*, a data-rich extension of CSP, mutation is considered in [7] for test generation. In practical experiments, *Circus* [18] models are translated to CSP for use of model checking via FDR as illustrated here. Our mutation operators, however, are for a language whose semantics is given in CSP, rather than CSP directly. Still, it would be interesting to explore the value of mutating the semantics of the RoboChart models.

A recent overview paper [53] focusses on model-based mutation testing for timed systems. It presents a taxonomy of the mutation operators and discusses their usages on various formalisms, such as timed automata or synchronous languages. For timed systems initial works focused on timeliness (the ability of a system to meet its deadlines) and introduced mutation operators for timed automata with tasks (TAT) [41, 42, 40], or timed automata similar to the UPPAAL format [51, 6]. We plan to use these works as a starting point for developing additional mutation operators for RoboChart time features.

There are several research groups working on mutation testing for timed automata, developing tools such as MoMuT::TA [6, 4], Ecdar [35, 36] and using model-checking or refinement-check approaches to generate test data. The mutation tool μUTA for UPPAAL timed automata is presented in [47] where the authors introduce three new operators compared to previous works [51, 6]. In contrast to this work, our approach is based on a domain-specific language similar to those used by practitioners. In addition, by basing the semantics on CSP we introduce the potential to include richer types of observations such as failures and refusal traces; future work will explore the use of such observations within the mutation-testing framework presented here.

7 Conclusions

This paper presents an approach to apply the well-established technique of mutation testing to a domain-specific notation for robotics: RoboChart. It uses generic tools: Wodel, for generation of mutants, and FDR, for generation of traces and forbidden continuations. The conformance relation is traces refinement.

This experience has raised some interesting issues. First of all, we are definitely interested in testing for stronger conformance relations. More specifically, we will ultimately define a testing theory for tock-CSP and adopt timed refinement as a conformance relation. In that work, we will also take advantage of results on testing with inputs and outputs [14, 15]. Experience shows that testing for traces refinement, as considered here, is an important common step to consider all the stronger conformance relations.

With the automation that we have in place, we will consider additional case studies, completing a comprehensive implementation of mutation operators. We are interested in assessing the performance of the approach in terms of generation of useless mutants. Another aspect of performance to be investigated is the trade off between more complex implementations of mutation operators that do not generate invalid mutants, and simpler implementations that rely on the OCL constraints to eliminate invalid mutants.

More interestingly, even for our simple example, we note the generation of tests that are not feasible or not typical of execution in realistic environments. Of particular note here is the issue of time. In our example tests, because FDR generates the shortest traces that lead to a counterexample, as soon as the drone takes off it finds the target. In typical environments, this is not going to be the case; the target is going to be some distance away from the origin. The RoboChart model, however, simply states that control software is ready to react to a detection at any time, including immediately. It is in our plan for future work to enrich RoboChart with features to model environments. The richer language will discard such unlikely or even infeasible scenarios for tests.

Finally, the work described in this paper has assumed that there is a single robotic systems, and tests apply inputs and observes outputs. Swarm robotic systems, however, are formed by a collection of robots and has many interesting and important applications. Swarms carry out tasks via collaboration between robots, and are resilient to faults. In this case, the system interacts with its environment at a number of physically distributed locations, namely, via a number of robotic platforms. In such situations, there may be independent local testers and this scenario alters how testing proceeds and the conformance relations used [31, 30]. There has been some work within the context of refinement and CSP [13], but there does not yet appear to be research that uses mutation.

Acknowledgements This work is funded by the EPSRC grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering. We have benefited from discussions with Pablo Gómez-Abajo and Mercedes Merayo with regards to Wodel implementation, and Sharar Ahmadi, Alvaro Miyazawa, and Augusto Sampaio with regards to our example and its simulation.

References

1. B. Aichernig and He Jifeng. Mutation testing in UTP. *Formal Aspects of Computing*, 2008.
2. B. K. Aichernig. Mutation testing in the refinement calculus. *Formal Aspects of Computing*, 15(2):280–295, 2003.
3. B. K. Aichernig. Model-based mutation testing of reactive systems. In *Theories of Programming and Formal Methods*, pages 23–36. Springer, 2013.
4. B. K. Aichernig, K. Hörmaier, and F. Lorber. Debugging with timed automata mutations. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *Computer Safety, Reliability, and Security - 33rd International Conference, SAFECOMP 2014, Florence, Italy, September 10-12, 2014. Proceedings*, volume 8666 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014.
5. B. K. Aichernig, E. Jöbstl, and S. TiranStefan. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97(P4):383–404, 2015.
6. B. K. Aichernig, F. Lorber, and D. Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013.
7. A. Alberto, A. L. C. Cavalcanti, M.-C. Gaudel, and A. Simao. Formal mutation testing for *Circus*. *Information and Software Technology*, 81:131–153, 2017.
8. P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *2nd International Conference on Formal Engineering Methods*, pages 46–54. IEEE, 1998.
9. A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rummer, and G. Weissenbacher. Mutation-based test case generation for Simulink models. In *Formal Methods for Components and Objects*, pages 208–227, 2009.
10. T. A. Budd and A. S. Gopal. Program testing by specification mutation. *Computer Language*, 10(1):63–73, 1985.
11. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, 2007.
12. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in *Circus*. *Acta Informatica*, 48(2):97–147, 2011.
13. A. L. C. Cavalcanti, M.-C. Gaudel, and R. M. Hierons. Conformance Relations for Distributed Testing based on CSP. In B. Wolff and F. Zaidi, editors, *IFIP International Conference on Testing Software and Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2011.
14. A. L. C. Cavalcanti and R. M. Hierons. Testing with Inputs and Outputs in CSP. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 359–374, 2013.
15. A. L. C. Cavalcanti, R. M. Hierons, S. Nogueira, and A. C. A. Sampaio. A suspension-trace semantics for CSP. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 3–13, 2016. Invited paper.
16. A. L. C. Cavalcanti, A. Miyazawa, A. C. A. Sampaio, W. Li, P. Ribeiro, and J. Timmis. Modelling and verification for swarm robotics. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods*, pages 1–19. Springer, 2018.

17. A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.
18. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.
19. M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Software Eng.*, 27(3):228–247, 2001.
20. M. S. Conserva Filho, R. Marinho, A. C. Mota, and J. C. P. Woodcock. Analysing robochart with probabilities. In T. Massoni and M. R. Mousavi, editors, *Formal Methods: Foundations and Applications*, pages 198–214. Springer, 2018.
21. S. Foster, J. Baxter, A. L. C. Cavalcanti, A. Miyazawa, and J. C. P. Woodcock. Automating Verification of State Machines with Reactive Designs and Isabelle/UTP. In K. Bae and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, pages 137–155, Cham, 2018. Springer.
22. G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.
23. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.
24. P. Gómez-Abajo, E. Guerra, and J. de Lara. Wodel: a domain-specific language for model mutation. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1968–1973. ACM, 2016.
25. P. Gómez-Abajo, E. Guerra, and J. de Lara. A domain-specific language for model mutation and its application to the automated generation of exercises. *Computer Languages, Systems & Structures*, 49:152–173, 2017.
26. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Sci. Comput. Program.*, 163:85–92, 2018.
27. M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor. Mutation operators for UML class diagrams. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, volume 9694 of *Lecture Notes in Computer Science*, pages 325–341, 2016.
28. J. Guan and J. Offutt. A model-based testing technique for component-based real-time embedded systems. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015.
29. W. Herzner, R. Schlick, H. Brandl, and J. Wiessalla. Towards fault-based generation of test cases for dependable embedded software. *Softwaretechnik-Trends*, 31(3), 2011.
30. R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012.
31. R. M. Hierons and H. Ural. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510, 2008.
32. C. A. R. Hoare. Programming: Sorcery or Science? *IEEE Transactions on Software Engineering*, 4, 1984.
33. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Software Engineering*, 37(5):649–678, 2011.
34. W. Krenn and B. K. Aichernig. Test case generation by contract mutation in Spec#. *Electronics Notes in Theoretical Computer Science*, 253(2):71–86, 2009.

35. K. G. Larsen, F. Lorber, B. Nielsen, and U. Nyman. Mutation-based test-case generation with Ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 319–328. IEEE Computer Society, 2017.
36. F. Lorber, K. G. Larsen, and B. Nielsen. Model-based mutation testing of real-time systems via model checking. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 59–68. IEEE Computer Society, 2018.
37. A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3869–3876, 2017.
38. A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.
39. B. Naylor, M. Read, J. Timmis, and A. Tyrrell. The Relay Chain: A Scalable Dynamic Communication link between an Exploratory Underwater Shoal and a Surface Vehicle. 2014.
40. R. Nilsson and J. Offutt. Automated testing of timeliness: A case study. In Hong Zhu, W. Eric Wong, and Amit M. Paradkar, editors, *Proceedings of the Second International Workshop on Automation of Software Test, AST 2007, Minneapolis, MN, USA, May 26-26, 2007.*, pages 55–61. IEEE Computer Society, 2007.
41. R. Nilsson, J. Offutt, and S. F. Andler. Mutation-based testing criteria for timeliness. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings*, pages 306–311. IEEE Computer Society, 2004.
42. R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. *Electr. Notes Theor. Comput. Sci.*, 164(4):97–114, 2006.
43. M. Papadakis and N. Malevris. Searching and generating test inputs for mutation testing. *SpringerPlus*, 2(1):1–12, 2013.
44. H. W. Park, A. Ramezani, and J. W. Grizzle. A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. *IEEE Transactions on Robotics*, 29(2):331–345, 2013.
45. C. A. Rabbath. A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. *Journal of Intelligent & Robotic Systems*, 70(1):233–253, 2013.
46. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
47. F. Siavashi, D. Truscan, and J. Vain. Vulnerability assessment of web services with model-based mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 301–312, 2018.
48. T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a CSP security example. In *10th Asia-Pacific Software Engineering Conference*, pages 340–350. IEEE Press, 2003.
49. S. K. Swain, D. P. Mohapatra, and R. Mall. Test case generation based on state and activity models. *Journal of Object Technology*, 9(5):1–27, 2010.
50. T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. L. Grixa, F. Ruess, M. Suppa, and D. Burschka. Toward a Fully Autonomous UAV: Research Platform for Indoor and Outdoor Urban Search and Rescue. *IEEE Robotics Automation Magazine*, 19(3):46–56, 2012.

51. M. S. A. Trab, S. Counsell, and R. M. Hierons. Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669, 2012.
52. University of York. *RoboChart Reference Manual*. `www.cs.york.ac.uk/circus/RoboCalc/robotool/`.
53. J. J. O. Vega, G. Perrouin, M. Amrani, and P.-Y. Schobbens. Model-based mutation operators for timed systems: A taxonomy and research agenda. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 325–332, 2018.
54. G. Wimmel and J. Jurjens. Specification-based test generation for security-critical systems using mutations. In *International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Nontes in Computer Science*, pages 471–482. Springer, 2002.