EXHAUSTIVE TESTING IN HOL-TestGen/CirTA A CASE STUDY

FELIACHI A / GAUDEL M C / WOLF B

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

07/2013

Rapport de Recherche N° 1562

Exhaustive Testing in HOL-TestGen/CirTA a case study

Abderrahmane Feliachi, Marie-Claude Gaudel and Burkhart Wolff

LRI - Univ. Paris-Sud & CNRS, Orsay, F-91405, France
{feliachi, gaudel, wolff}@lri.fr

Résumé HOL-TESTGEN/*CirTA* est un environnement pour la génération de tests à partir de spécifications écrites dans *Circus*. *Circus* est un langage de spécification formelle qui combine les notions d'état et de types de données complexes dans le style de Z avec une algèbre de processus dans la tradition de CSP. L'originalité de HOL-TESTGEN/*CirTA* est qu'il est basé sur une implémentation vérifiée dans Isabelle/HOL de la sémantique de ce langage. Cela constitue le fondement des règles de génération de tests ainsi que pour leur exécution sous forme de calcul symbolique. Le résultat est une chaîne d'outils intégrée qui transforme, par des règles issues de la définition sémantique, la spécification d'un système en un ensemble de cas de tests symboliques, les instancie et les exécute.
Ce rapport présente une première étude de cas réalisée sur un composant d'un monde réel d'un système de suivi médical écrit en Java. Dans cette étude de cas, les cas de test générés sont compilés dans un ensemble de testeurs JUnit et exécutes sur l'implémentation du système sous test.
Diverses expériences ont été réalisées afin d'analyser les performances et l'efficacité du processus de génération de test. Nous montrons qu'une tactique de génération de test spécifique est plus efficace que la procédure générique. Une analyse de mutation basique a été expérimentée pour donner une évaluation de nos cas de test générés.

Abstract HOL-TESTGEN/*CirTA* is a theorem-prover based environment for test generation from specifications written in *Circus*. *Circus* is a formal specification language which combines the notions of states and complex data types in a Z-like style with a process-algebra in the tradition of CSP. The originality of HOL-TESTGEN/*CirTA* is that it is based on a machine-checked embedding in Isabelle/HOL of the semantics of this language. This provides the foundation of the test-generation rules and for their execution in form of symbolic computation. The result is an integrated tool chain that transforms, via rules derived from the semantic definition, the *Circus* specification of a system into a set of symbolic test cases, instantiates, compiles and submits them.
This report presents a first case study performed on a component of a real-world medical monitoring system written in Java. In this case study, the generated test cases are compiled into a set of JUnit-testers and run against the implementation of the system under test.
Various experiments were performed in order to analyze the performances and the efficiency of the test generation process. We show that a specific test generation tactic is more efficient than the generic procedure. A basic mutation analysis was experimented to give some assessment of our generated test cases.

Keywords: symbolic test-case generations, black box testing, theorem proving, model-based testing, JUnit

1 Introduction

The use of formal specifications has been recognised for a while as providing strong bases for specification-based testing [7]. More recently, it has been shown in [1] how coupling the use of formal specifications and theorem-provers can lead to some powerful environment for test generation and execution, exploiting in a complementary way proving and testing techniques. This paper presents a case study of the use of such an environment, HOL-TestGen/*CirTA*, for testing a component of a real-world safety-critical medical monitoring system, which was specified in *Circus* and written in Java.

Circus [10] is a formal specification language that combines notions of states and complex data types in a Z-like style with a process-algebra in the tradition of CSP and comes with a notion of refinement.

In HOL-TestGen/*CirTA* [5], the rules for test generation are derived from an embedding of the semantics of *Circus* in Isabelle/HOL, and their execution is controlled by proof tactics. By adding suitable front-end and back-ends, we have developed an integrated tool chain that covers the statement of *Circus* specifications, symbolic test generation and test submission and execution, following the test-theory of [2] for testing against refinement in *Circus* .

Technically, *CirTA* extends the HOL-TestGen framework [1] by adding to HOL-TestGen's data-type oriented test-case generation some support for tests over the (abstract) type `action` embedding *Circus* processes. Since HOL-TestGen is implemented in Isabelle/HOL, *CirTA* is just another plugin into the powerful generic Isabelle framework, which provides libraries, proof-tools as well as documentation and code-generation facilities, and makes it possible to obtain concrete JUnit-Testers from *Circus* specifications. All this environment is completely formal and the test generation is derived from the semantics. It guarantees the soundness of the generated tests and their exhaustivity.

Our test environment presented here is a newcomer to the world of specification-based test generation tools. It presents an alternative to the existing tools for symbolic test generation from specifications involving processes and data. For instance, [3] and [6] present tools based on symbolic IOTS and extensions of the *ioco* conformance relation. In this work we go one step further in the guarantee that the test generation is consistent with the semantics of the specification and the conformance relation, thanks to the embedding of the semantics in the theorem prover and the use of proof tactics. The underlying prover powerful symbolic computation machinery is very beneficial to our environment.

There is an infinite number of tests. In the current state of *CirTA*, there is only a rather primitive selection strategy of a finite subset, namely bounded exhaustive testing. Other selection strategies are currently under study, based essentially on some structural constraints over the generated tests.

This paper is organised as follows: Section 2 briefly presents the *Circus* language (2.1), then the *Circus* testing theory associated to the notion of *Circus* refinement (2.2) and then the test generation engine (2.3); Section 3 describes the system under test, which is a part of a remote monitoring system used in a health-care network; Section 4 reports on the use of the *CirTA* system for this case study, first presenting the test specification, which is based on the *Circus* specification of the system and the test goal (here bounded exhaustive testing), then the test generation experiments (4.2), and finally test executions (4.3). Section 5 presents some outcomes of the case study, and Section 6 some conclusions.

2 The CirTA system architecture

The Isabelle framework offers a plug-in infrastructure: on top of the – in many ways – generic system kernel, the HOL instance was integrated, which in itself formed the theoretic and technical environment for HOL-TestGen, into which *CirTA* has been integrated. This way, local plug-ins profit from the generic system functionality providing document or code generation as well as libraries, proofs, and proof-tools.

CirTA (*Circus* Testing Automation) is a test-generation environment for *Circus*. The *CirTA* system is composed of three main components, organized in three different layers. The first layer defines the *Circus* language using its denotational semantics and its UTP basis. This layer is encoded in the Isabelle/*Circus* framework. On top of this framework, the second layer is defined using the operational semantics and the testing theories of *Circus*. In the top-most layer, the test-generation engine is defined by introducing different test-generation tactics.

2.1 The Circus language and its semantics

Circus is a formal specification language which combines the notions of states and complex data types in a Z-like style with a process-algebra in the tradition of CSP. The language comes with a formal notion of refinement allowing a formal development ranging from abstract specifications and to executable models and programs. *Circus* has a denotational semantics [9] sketched in terms of the UTP [8], and a corresponding operational semantics [2]. UTP is essential for providing a seamless semantic framework for states and processes.

We introduce in the following a *Circus* specification of the queue module studied in this report.

$$\textbf{channel}\ get, put, finish : \mathbb{N} \times \mathbb{N}$$

$$\textbf{process}\ Abstract\_Queue \mathrel{\widehat=} \textbf{begin}$$

$$\textbf{state}\ QueueState\ == [\, new : \mathrm{seq}(\mathbb{N} \times \mathbb{N});\ active : \mathbb{P}(\mathbb{N} \times \mathbb{N}) \,]$$

$$RemoveFirst : \mathrm{seq}(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \rightarrow \mathrm{seq}(\mathbb{N} \times \mathbb{N})$$

$$\begin{array}{l} RemoveFirst(\langle\,\rangle, n) = \langle\,\rangle \\ \forall\, S : \mathrm{seq}_1(\mathbb{N} \times \mathbb{N}), e : (\mathbb{N} \times \mathbb{N})\ , n : (\mathbb{N} \times \mathbb{N}) \bullet \\ \quad RemoveFirst(\langle e \rangle \frown S, n) = \\ \quad\quad \textbf{if}_Z\ e = n\ \textbf{then}\ S\ \textbf{else}\ \langle e \rangle \frown RemoveFirst(S, n) \end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InitQueue}} \\ \hline
\mathit{QueueState}' \\ \hline
\mathit{new}' = \langle\,\rangle \land \mathit{active}' = \emptyset \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddNew}} \\ \hline
\Delta \mathit{QueueState} \\
x? : \mathbb{N} \times \mathbb{N} \\ \hline
\mathit{new}' = \mathit{new} \frown \langle x? \rangle \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Choose}} \\ \hline
\mathit{QueueState} \\
y! : \mathbb{N} \times \mathbb{N} \\ \hline
\operatorname{dom} \mathit{new} \setminus \operatorname{dom} \mathit{active} \neq \emptyset \\
y! = \mathit{head}\,(\mathit{new} \upharpoonright \{a, b : \mathbb{N} \mid a \notin \operatorname{dom} \mathit{active} \bullet (a, b)\}) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Activate}} \\ \hline
\Delta \mathit{QueueState} \\
y? : \mathbb{N} \times \mathbb{N} \\ \hline
\mathit{new}' = \mathit{RemoveFirst}(\mathit{new}, y?) \\
\mathit{active}' = \mathit{active} \cup \{y?\} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Remove}} \\ \hline
\Delta \mathit{QueueState} \\
x? : \mathbb{N} \times \mathbb{N} \\ \hline
\mathit{active}' = \mathit{active} \setminus \{x?\} \\ \hline
\end{array}$$

$$\begin{array}{lcl}
\mathit{Put} & \widehat{=} & \mathit{put}\,?x \rightarrow \mathit{AddNew} \\
\mathit{Get} & \widehat{=} & (\operatorname{dom} \mathit{new} \setminus \operatorname{dom} \mathit{active} \neq \emptyset)\ \&\ (\mathit{Choose};\ \mathit{get}!y \rightarrow \mathit{Activate}) \\
\mathit{Finish} & \widehat{=} & \mathit{finish}?x \in \mathit{active} \rightarrow \mathit{Remove}
\end{array}$$

$\bullet$ **var** $x, y : \mathbb{N} \times \mathbb{N} \bullet \mathit{InitQueue}\ ;\ (\mu\, X \bullet ((\,\mathit{Put} \,\Box\, \mathit{Get}\,) \,\Box\, \mathit{Finish}\,)\ ;\ X\,)$
end

2.2 The Circus testing theory

In [2] the foundations of testing based on *Circus* specifications are stated for two conformance relations: *traces inclusion* and *deadlocks reduction* (usually called *conf* in the area of test derivation from transition systems). The conjunction of these relations corresponds to the *Circus*notion of refinement, in the case of non divergent specifications.

The basis of this work is an operational semantics that expresses in a symbolic way the evolution of systems specified in *Circus*. Using this operational semantics, symbolic characterizations of traces, initials, and acceptance sets have been stated and used to define relevant notions of tests. Two symbolic exhaustive test sets have been defined respectively for traces refinement and *deadlocks reduction*: proofs of exhaustivity guarantee that, under some basic testability hypotheses, a system under test (SUT) that would pass all the concrete tests obtained by instantiation of the symbolic tests of the symbolic exhaustive test set satisfies the corresponding conformance relation.
The tests are defined using the following notions:

- *cstraces* : a constrained symbolic trace is a pair composed of a symbolic trace st and a constraint c on the symbolic variables of st.
- *csinitials*: the set *csinitials* associated with a cstrace (st, c) of a *Circus* process P contains the constrained symbolic events that represent valid continuations of (st, c) in P, i.e. events that are initials of P after (st, c).

- $\overline{\mathit{csinitials}}$: given a process P and one of its cstraces (st, c), the set $\overline{\mathit{csinitials}}$ contains the constrained symbolic events that represent the events that are not initials of P for any of the instances of (st, c).
- *csacceptances*: a *csacceptances* set associated with a cstrace (st, c) of a *Circus* process P is a set of sets SX of symbolic acceptances. An acceptance is a set of events in which at least one event must be accepted after (st, c).

An example, in the *Queue* process, of a constrained symbolic trace and a constrained symbolic event after this trace is given by:

$$([put.a, put.b, get.c], a = c) \qquad (\mathit{finish}.d, d = a)$$

Symbolic tests for traces inclusion. *traces inclusion* refers to inclusion of trace sets: process P_2 is a *traces inclusion* of process P_1 if and only if the set of traces of P_2 is included in that of P_1. Symbolic tests for *traces inclusion* are based on some cstrace *cst* of the *Circus* process P used to build the tests, followed by a forbidden symbolic continuation, namely a constrained symbolic event *cse* belonging to the set $\overline{\mathit{csinitials}}$ associated with *cst* in P. Such a test passes if its parallel execution with the SUT blocks before the last event, and fails if it is completed. An example of a symbolic test for *traces inclusion* is given by:

$$([put.a, put.b, get.c], c \neq a)$$

Symbolic tests for *deadlocks reduction*. *deadlocks reduction* (also called *conf*) requires that deadlocks of process P_2 are deadlocks of process P_1. The definition of symbolic tests for *deadlocks reduction* is based on a cstrace *cst* followed by a choice over a set SX, which is a symbolic acceptance of *cst*. Such a test passes if its parallel execution with the SUT is completed and fails if it blocks before the last choice of events. An example of a test for *deadlocks reduction* in *Fibonacci* is given by:

$$([put.a, put.b, get.c], c = a)\ \{put.d, get.b, \mathit{finish}.c\}$$

In [5], we presented a formalization of all these notions in Isabelle/*Circus*. This formalization forms the second layer of the *CirTA* system. This formalization is the basis of the test generation tactics defined in the top-most layer.

2.3 The test-generation engine

Starting from a *Circus* specification, the role of the test-generation engine is to derive traces and tests for each conformance relation. It defines some general tactics for generating, *cstraces* and test-cases for the two conformance relations introduced earlier.

Trace generation Test definitions are introduced as test specifications that will be used for test-generation. For trace generation a proof goal is stated to define the traces a given system may perform. This statement is given by the following rule, for a given process P:

$$\frac{\mathit{length}(\mathit{tr}) \leq k \quad \mathit{tr} \in \mathit{cstraces}(P)}{\mathit{Prog}(\mathit{tr})} \tag{1}$$

where k is a constant used to bound the length of the generated traces.

While in a conventional automated proof, a tactic is used to refine an intermediate step (a "subgoal") to more elementary ones until they eventually get "true", in prover-based testing this process is stopped when the subgoal reach a certain normal form of clauses, in our case, when we reach logical formulas of the form: `C` $\Longrightarrow$ `Prog (tr)`, where `C` is a constraint on the generated trace. Note that different simplification rules are applied on the premises until no further simplification is possible. The final step of the generation produces a list of propositions, describing the generated traces stored by the free variable *Prog*. The test specification 1 is introduced as a proof goal in the proof configuration. The premise of this proof goal is first simplified using the definition of *cstraces*. The application of the trace generation tactic on this proof goal generates the possible continuations in different subgoals. The elimination rules of the operational semantics are applied to these subgoals in order to instantiate the trace elements. Infeasible traces correspond to subgoals whose premises are *false*. In this case, the system is able to close these subgoals automatically.

Specifications may describe unbounded recursive behavior and thus yield an unbounded number of symbolic traces. The generation is then limited by a given trace length k, defined as a parameter for the whole generation process. The list of subgoals corresponds to all possible traces with length smaller than this limit.

The trace generation process is implemented in Isabelle as a tactic. The trace generation tactic can be seen as an *inference engine* that operates with the derived rules of the operational semantics and the trace composition relation.

Test generation for *traces inclusion.* The generation of *csinitials* is done using a similar tactic as for *cstraces*. In order to capture the set of all possible *csinitials*, the test theorem is defined in this case as follows:

$$\frac{S = \mathit{csinitials}(P, \mathit{tr})}{\mathit{Prog}\ S} \tag{2}$$

the free variable *Prog* records the set S of all *csinitials* of P after the trace tr.

The generation of tests for *traces inclusion* is done in two stages. First, the trace generation tactic is invoked to generate the symbolic traces. For each generated trace, the set of the possible *csinitials* after this trace is generated using the corresponding generation tactic. Using this set, the feasible $\overline{\mathit{csinitials}}$ are generated and added as a subgoal in the final generation state.

Test generation for *deadlocks reduction.* test-generation in this case is based on the generation of the $\mathit{csacceptances}_{\mathit{min}}$ set. For a given symbolic trace generated from the specification, the generation of the sets of $csacceptances_{min}$ is performed in three steps. First, all possible stable configurations that can be reached by following the given trace are generated. In the second step, all possible *IOcsinitials* are generated for each configuration obtained in the first step. Finally, the $csacceptances_{min}$ set is computed from all resulting *IOcsinitials*. The different generation tactics are explained in detail in [5].

3 The Application: A Message-Multiplexer in a medical Home-Monitoring System

Our case study addresses a part of a remote monitoring system used in a world-wide health-care network. The network connects a variety of devices that can be for instance pacemaker controllers. The automatic monitoring system keeps track of the status of all connected devices that regularly send diagnostic, therapeutic, and technical data on the current clinical status of the patients.

The monitoring system collects a huge number of messages then routes them to their corresponding processing services in order to be processed. The routing policy is particular and depends on the nature of the message and the type of the device. It is very critical and must be correctly implemented in the monitoring system. A wrong message routing may lead to information misinterpretation that can be risky for the patients health.

An overview of the remote monitoring system is given in Figure 1.

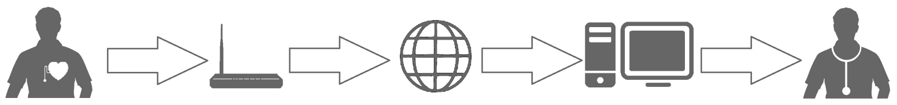

Figure 1. Remote monitoring system overview

The remote monitoring system is composed essentially of a queue module and a set of processing services. The different message manipulations and routing operations are carried out by the queue module. Each message is characterized with a device identifier and its actual content. The queue receives, stores then assigns messages to the corresponding processing services. The main operations that can be performed by the queue are:

- *PUT:* The queue receives the messages using the PUT operation and stores them with the *new* status.
- *GET:* The processing services can retrieve *new* messages from the queue and mark them as *active*.
- *FINISH:* When a processing service successfully completes a message processing, this *active* message is completely removed from the queue.

The device identifiers are associated to the message to indicate their sources. Two messages sent from the same device should not be processed simultaneously, the processing order follows the sending order. This order is important since the processing of the later message may depend on the results of the processing of the first one. Messages that comes from different sources are processed following the arrival order following a FIFO discipline.

4 Testing in CirTA

The testing procedure is very similar to the standard HOL-TestGen procedure described in [1]. First, an HOL-based test specification is introduced describing the test goal. Inference rules are then used to derive test cases from this specification following some predefined tactics. Finally, Testers are generated from the resulting test cases and executed against the SUT.

4.1 Test-Specification

The test specifications are defined using a specification of the SUT. For this, we provide an abstract *Circus* specification of the queue module. For the sake of simplicity, we consider message identifiers and contents as natural numbers. Using the syntax of Isabelle/*Circus*, we introduce in the *CirTA* system a formalization of the `Abstract_Queue` process:

```
circusprocess Abstract_Queue =
  alphabet = [x::nat×nat, y::nat×nat]
  state = [new::(nat×nat) list, active::(nat×nat) set]
  channel = [get nat×nat, put nat×nat, finish nat×nat]
  schema InitQueue = new' = [] ∧active' = {}
  schema AddNew = new := new@[x]
  schema Choose =     (∃a∈set new. fst a ∉fst ‘ active) ∧
           y := hd (filter (λb.∀a∈active. fst a ≠fst b) new)
  schema Activate = new' = RemoveFirst new y ∧active' = active ∪{y}
  schema Remove = active := active - {x}
  action Put = put?x →(Schema AddNew)
  action Get = (λ A. (fst‘(set (new A))) - (fst‘(active A)) ≠{}) &
               ((Schema Choose); get!y →(Schema Activate))
  action Finish = finish?x∈active →(Schema Remove)
  where (Schema InitQueue); μX ·(((Put □ Get) □  Finish); X)
```

For trace generation, a test specification is stated as a proof goal describing the traces to generate. This test specification, in our case, is given by the following formula:

```
 tr ∈cstraces Abstract_Queue ⟹  prog tr
```

where `cstraces` defines the set of traces and `prog` is a free variable used to store the generated traces.

For each generated trace, different test cases are generated to test the trace inclusion relation. A test specification is stated as a proof goal in order to start the generation. The complete test specification is given as follows:

```
tr ∈cstraces Abstract_Queue ∧
  e ∈csinitialsb Abstract_Queue tr  ⟹  prog tr@[e]
```

where `csinitialsb` defines the set of $\overline{csinitials}$.

Similarly, each generated trace is used to generate the corresponding tests *w.r.t.* the deadlock reduction conformance relation. The test specification corresponding to all the possible traces is given as follows:

```
tr ∈cstraces Abstract_Queue ∧
  e ∈csacceptances Abstract_Queue tr ⟹  prog tr e
```

where `csacceptances` is the set of acceptances of `Abstract_Queue` after `tr`.

4.2 Test-Generation Experiments

The test generation for the Queue process is done in two steps. First, all possible traces (up to a given length) are generated. Then, for each generated trace, two test sets are generated, one for trace inclusion and one for deadlock reduction. The symbolic generated tests are then transformed into instantiated tests via some HOL-TestGen's method called `gen_test_data`, and then into executable tests that will be exercised against the system (see subsection 4.3).

Trace generation The generic trace generation tactic is defined using the operational semantics rules applied along with the system simplifier. The constraints associated to the generated tests define the domain of the symbolic tests. These constraints, in our case, can become very complex due to the non-determinism at the level of the retrieved message. A constraint defined by a disjunction of two constraints defines a union of two subdomains. Some DNF decomposition [4] can be used to split this kind of domains into two distinct domains. This significantly increases the number of the generated traces, however it reduces drastically the complexity of the constraints.

The trace generation tactic is invoked using different trace lengths. A first (expected) drawback of the generic trace generation tactic is the lack of efficiency: the generation time grows exponentially *w.r.t.* the trace length. For a length of 4, the generation takes more than 20 seconds against a maximum of 5 seconds for shorter traces. For traces of length 5, the generation time is around 5 minutes.

This is due to the heavy machinery used for trace generation and also to the multiple silent transitions of the operational semantics. A characteristic of our specification is that, after the initialization, it behaves in a recursive way. We can take advantage of this characteristic to improve the trace generation efficiency by factorizing the generations steps. During one recursion, different silent transitions are performed, in addition to one communication transition. All these transitions can be factorized in a one-step transition that covers the silent and the communication transitions. A specific rule for this transition called `OneStep` was proved from the operational semantics.

Using the `OneStep` rule, the overall generation time is reduced. For a length of 5 for example, the trace generation takes less than 2.5 seconds and for 6 less than 8 seconds. A list of all the performed experiments is given in section 5.

We were led to limit the length of generated traces to 7 and thus the generated tests will have a maximum length of 8. This limit is chosen only for practical reasons, due to the current number of cases: the number of generated traces using this limit is around 300. Thus the whole test generation tactics takes an important execution time. This number is important due to the fact that we generate exhaustively. In the near future, we plan to consider more restrictive selection hypothesis in order to focus on interesting and longer selected tests.

Examples of generated traces are the following:

```
⋀a b c d e f. prog [put (a, b), put (c, d), put (e, f), get (a, b)]
⋀a b c d. prog [put (a, b), put (c, d), get (a, b), finish (a, b)]
```

As said above, for practical reasons the length limit considered for the moment is 7. A regularity hypothesis is stated on the length of traces. This regularity hypothesis is given as follows:

```
THYP ((length tr ≤ 7  ⟶ prog tr) ⟶ (∀tr. prog tr))
```

Test generation for trace inclusion The trace generation tactic instantiates the variable `tr` of this test specification to all the possible traces. This results into different test specifications each associated to a different trace. The initials generation tactic is used to generate the corresponding initials for each test specification. This generation is done in parallel for all test specifications resulting from the trace generation step.

The tests are then retrieved by unfolding the definition of `csinitialsb` in the test specification, then simplifying the resulting proof goal. Like for traces, the generation of initials is also slow when using the operational rules directly. A factorized version of the initials generation rules was also derived and proved.

As an example, let us consider the trace `[put (a, b), put (aa, ba)]`. This trace is used to illustrate the test generation tactic and its results. The test specification corresponding to this trace is given in the following:

```
⋀a b c d. e∈csinitialsb Abstract_Queue [put (a, b), put (c, d)]
                ⟹ prog [put (a, b), put (c, d), e]
```

The result of the test generation tactic is the list of the possible tests, defined by the trace and a non initial element. In this case, three different tests are generated, each one associated to a constraint (`af ≠ a` and `bf ≠ b`).

```
⋀a b c d e f. e ≠a ⟹ prog [put (a, b), put (c, d), get (e, f)]
⋀a b c d e f. f ≠b ⟹ prog [put (a, b), put (c, d), get (e, f)]
⋀a b c d e f. prog [put (a, b), put (c, d), finish (e, f)]
```

These tests are represented in a symbolic way, using symbolic HOL variables (e. g. `a, b, ba ...`). To obtain concrete finite test cases, some selection hypotheses must be stated on the symbolic tests. We reused in this step the `gen_test_cases` method of the HOL-TestGen system. This method makes more simplifications on the current symbolic tests. It applies also a uniformity hypothesis on the simplified symbolic tests and returns schematic test cases. Schematic values are represented by schematic variables (e. g.`?X32X18`) which are also constrained. These schematic variables can be instantiated by any values satisfying the constraints. The resulting *schematic* test cases are presented as follows:

```
?X32X18 ≠ ?X44X30 ⟹ prog [put (?X44X30, ?X43X29),
                   put (?X42X28, ?X41X27), get (?X32X18, ?X31X17)]
?X16X17 ≠ ?X28X29 ⟹ prog [put (?X29X30, ?X28X29),
                   put (?X27X28, ?X26X27), get (?X17X18, ?X16X17)]
prog [put (?X14X29, ?X13X28), put (?X12X27, ?X11X26),
                      finish (?X2X17, ?X1X16)]
```

In addition to the *schematic* test cases, a uniformity hypothesis, is stated for each test case. The uniformity covers all the symbolic variables of the symbolic test, so only one hypothesis is obtained by symbolic test. An example of a uniformity hypothesis for the first case is:

```
THYP ((∃x xa xb xc xd xe. xa ≠xe ∧
       prog [put (xe, xd), put (xc, xb), get (xa, x)]) ⟶
      (∀x xa xb xc xd xe. xa ≠xe ⟶
       prog [put (xe, xd), put (xc, xb), get (xa, x)]))
```

In order to be executed, the *schematic* test cases must be instantiated with concrete values. For this, a HOL-TestGen's method called `gen_test_data` is directly used. This method uses smt solvers (e. g. Z3) to instantiate concrete values for the schematic variables. The resulting test cases of our example are:

```
prog [put (3, 1), put (0, 0), get (0, 1)]
prog [put (0, 0), put (3, 1), get (3, 2)]
prog [put (1, 2), put (1, 6), finish (0, 1)]
```

Test generation for deadlock reduction The trace generation tactic is used to generate all the possible traces of a given length. The acceptances generation tactic is then applied automatically to all the generated traces. In order to increase the efficiency of the test generation, we introduce some parallelization: each test specification, associated to one possible trace, is treated separately. The test-generation is then performed in parallel to all these test specifications.

If we consider for example the trace `[put (a, b), put (aa, ba)]`, then the corresponding test specification is given by the following:

```
⋀a b aa ba.
e ∈csacceptances Abstract_Queue [put (a, b), put (aa, ba)] ⟹
       prog [put (a, b), put (aa, ba)] e
```

The acceptances generation tactic is used to retrieve the corresponding acceptance sets. The resulting proof goal contains in each subgoal the variable `prog` associated to a trace and an acceptance set. As for the trace refinement, some factorized rules were derived and used in the generation in order to reduce its time. The previous example produces one test case presented as follows:

```
⋀a b aa ba af bf.
  prog [put (a, b), put (aa, ba)] {get (a, b), put (af, bf)}
```

The generated symbolic acceptance set is finite in this case, due to the structure of the specification. This is the case for all the generated tests. As a consequence, test selection, instantiation and translation are not very complicated.

Following the same strategy as for trace inclusion, HOL-TesGen is reused for test selection and instantiation. First, the `gen_test_cases` method is applied on the proof goal. This method applies some simplifications and then states a uniformity hypothesis on the symbolic variables of the tests. This results, for the previous example, on the following *schematic* test case:

```
prog [put (?X14X45, ?X13X44), put (?X12X43, ?X11X42)]
              {get (?X14X45, ?X13X44), put (?X2X33, ?X1X32)}
```

The following uniformity hypothesis is associated to this test case:

```
THYP ((∃x xa xb xc xd xe.
prog [put (xe, xd), put (xc, xb)] {get (xm, xl), put (xa, x)}) ⟶
      (∀x xa xb xc xd xe.
prog [put (xe, xd), put (xc, xb)] {get (xm, xl), put (xa, x)}))
```

Finally, and in order to make the tests executable, the *schematic* test cases are instantiated to concrete tests. This is done using the `gen_test_data` method provided by HOL-TestGen. For our example, the resulting concrete test obtained by this method is given by:

```
prog [put (1, 1), put (10, 5)] {get (1, 1), put (0, 2)}
```

4.3 Testers and Test-Code

The queue is implemented in Java and integrated to the whole remote monitoring system. In order to test this implementation, JUnit testing facilities are used for test execution. Starting from the queue specification, tests are generated then translated into JUnit test cases. The resulting test cases are then directly executed on the given implementation.

In order to execute the concrete tests against the provided Java implementation of the queue, these test cases must be expressed in terms of JUnit test methods. Each event of the trace is translated to a call to the corresponding method in the implementation. The execution is then done directly in the Eclipse platform using JUnit testing facilities.

Trace inclusion For the first conformance relation, the concrete tests generated previously are automatically translated into Java methods. This translation is done using a new method called `export_test_file` that we developed for this purpose. The translation (ML) method implements some translation rules for each event of the concrete tests.

For the trace events, the translation is straightforward, `put` and `finish` events are translated directly to the corresponding methods. The `get` event is translated into a call to the corresponding method, followed by a check of the resulting value. The call of these methods may fail. This is detected by an exception or by a wrong result returned by `get`. If the call fails at this stage, the test is considered inconclusive.

The last event is treated differently, because it is supposed to fail. The `put` and `finish` should throw an exception. The `get` event is translated as in the case of trace events, but the check of the resulting value is inverted. A test succeeds if one of the methods throws an exception or if the result of the *get* method corresponds to the incorrect value described in the test.
The first test case presented previously, produce the following JUnit method:

```java
public void testqueue1_1() throws Exception {
    AbstractQueueableObject o_3_1 = new NamedEntry("topic", 3, 1);
    AbstractQueueableObject o_0_0 = new NamedEntry("topic", 0, 0);
    AbstractQueueableObject o_0_1 = new NamedEntry("topic", 0, 1);
    AbstractQueueableObject o = null;
    tm.begin();

    try { queueManager.put(o_3_1); tm.commit(); }
    catch (Exception e) { System.out.println("inconclusive"); return;}

    try { queueManager.put(o_0_0); tm.commit(); }
    catch (Exception e) { System.out.println("inconclusive"); return;}

    o = queueManager.get(NamedEntry.class, "topic");
    assertFalse(equals(o_0_1, o));
}
```

Deadlock reduction The generated tests for the second conformance relation are also automatically translated into Java methods. This is done using a method called `export_test_file2`, which is slightly different from the first one. The translation rules are the same for the (sub)traces, by transforming each event into the corresponding method. For the acceptances set, the translation is more tricky. Since our acceptances sets are finite, the concrete acceptances can be enumerated and translated to produce the following behavior: First, the queue state is saved using a *commit* operation. Then for the first acceptance event, the corresponding method is called. If the call fails, the queue state is retrieved using the *rollback* operation and the execution continues with the remaining acceptances. As soon as a call is successfully performed, the test passes. If all the acceptances fail then the test fails as well.

In the special case of infinite acceptances sets, the translation will be slightly different. The instantiation is not possible at the generation step, an on-line testing scenario is more convenient. The symbolic test must be translated directly to the corresponding method call. The obtained input is used to check if the constraint associated to this test is satisfied.
The concrete test case generated previously produces the following test method:

```java
public void testqueue1_1() throws Exception {
    AbstractQueueableObject o_1_1 = new NamedEntry("topic", 1, 1);
    AbstractQueueableObject o_10_5 = new NamedEntry("topic", 10, 5);
    AbstractQueueableObject o_0_2 = new NamedEntry("topic", 0, 2);
    AbstractQueueableObject o = null;
    tm.begin();

    try { queueManager.put(o_1_1); tm.commit(); }
    catch (Exception e) { System.out.println("inconclusive"); return;}

    try { queueManager.put(o_10_5); tm.commit(); }
    catch (Exception e) { System.out.println("inconclusive"); return;}

    try
    { o = queueManager.get(NamedEntry.class, "topic"); tm.commit(); }
    catch (Exception e)
        { tm.rollback(); queueManager.put(o_0_2); tm.commit(); }

    if (o == null || !equals(o_1_1, o)) {
        tm.rollback(); queueManager.put(o_0_2); tm.commit();
    }
}
```

5 Test evaluation

The experiments are done using Isabelle2013 running on a computer working on Windows 7. The computer has an 8-core processor (Intel i7 2600) and 6 GB of RAM. Different experiments are realized by varying the trace length of the tests from 0 to 8. The number of tests and the generation time corresponding to each length for the trace inclusion relation are summarized in table 1.

Trace length	Traces time / number	Symbolic Tests time / number	Schematic Tests time	Instantiation time
0	0 / 1	0.02 / 2	0.02	0.01
1	0.2 / 2	0.02 / 5	0.02	0.01
2	0.2 / 4	0.03 / 11	0.05	0.01
3	0.4 / 8	0.07 / 30	0.1	0.03
4	0.9 / 17	0.19 / 83	0.5	0.2
5	2.5 / 41	1 / 262	2.2	2.6
6	8 / 106	19 / 1039	15	65
7	42 / 297	134 / 4396	351	3000
8	600 / 904	10^4 / 22647	-	-

Table 1. Statistics of test generation for trace inclusion

The generation time, as well as the number of generated test cases grow exponentially *w.r.t.* the trace length. For traces of length 8, the system limits are reached, due to the important number of symbolic test cases. In order to generate longer test cases, one can introduce more specific selection hypotheses. This will produce less but more focused test cases.

The exhaustive test generation for length at most 8 produced a total of 4396 test cases using the 297 traces of length smaller or equal to 7. All the generated test cases are concrete, where all communicated values are instantiated. As explained before, all these test cases are compiled into Java test methods that are executed using JUnit. The test execution time is negligible *w.r.t.* the test generation time (less than 10 seconds).

The same experiments are done for the deadlock reduction conformance relation. The statistics are summarized in table 2.

Trace length	Traces time / number	Symbolic Tests time / number	Schematic Tests time	Instantiation time
0	0 / 1	0.03 / 1	0.02	0.01
1	0.2 / 2	0.03 / 2	0.02	0.01
2	0.2 / 4	0.04 / 4	0.02	0.03
3	0.4 / 8	0.06 / 10	0.04	0.05
4	0.9 / 17	0.5 / 30	0.26	0.1
5	2.5 / 41	2/112	0.9	0.6
6	8 / 106	5 / 496	5.8	15
7	42 / 297	97 / 2473	194	670
8	600 / 904	2100 /13918	-	-

Table 2. Statistics of test generation for deadlocks reduction

The exhaustive test generation produces not less than 2473 test methods from all the traces of length smaller than 7. As for trace inclusion, all resulting test methods are collected in a Java test file and executed against the implementation. Comparing to the first conformance relation, the generation produces less test cases in less time.

Test execution. For the trace-inclusion conformance relation, the execution of the 4396 test methods ended without finding errors, but with 1024 inconclusive tests. In the second case of deadlock reduction relation, no errors and 494 inconclusive tests resulted from executing the 2473 test cases. The component under test was intensively tested during the development of the system; thus it is not a surprise that no errors were detected by our generated tests.

A significant number of test cases ended with an inconclusive verdict (1518 from 6869), which reduces the efficiency of our test. In order to reduce the number of inconclusive tests, one possible solution is to combine the tests of the two conformance relations. The structure of the tests will be more complex (tree-shaped) but the number of resulting tests would be smaller.

In order to make some preliminary evaluation of our generated test cases, some basic mutation testing experiments were performed. One important mutant of the queue is the one that inverts the order of insertion of the elements. This mutant was detected only by 1 test case, but more than 1840 test cases were inconclusive. Different mutations were also applied, mainly by inverting some conditions. All these mutants were killed by some test cases of the trace inclusion conformance relation. Due to the nature of our mutations, no errors were detected by the test cases for deadlocks reduction. However, the number of inconclusive tests increased significantly. All these mutation experiments were performed manually, due to the lack of fully integrated and maintained mutation testing tools for Java and Junit.

6 Conclusion

This paper presents a case study of the HOL-TestGen-*CirTA* test generation environment. This first case study covers a black-box testing experience on a real safety-critical health-care system. The system under test is a queue module, implemented in Java and provided with a JUnit infrastructure. Starting from an abstract specification of the system in *Circus*, our environment is used to generates traces and then tests for two conformance relations. JUnit testers are then extracted form the generated tests and executed against the real system. Different experiments were performed to measure the efficiency of the test generation procedures. Some basic mutation analysis was also performed to evaluate the usefulness of the generated tests.

The test generation environment is a generic *push button* solution for any *Circus* specification. However, from our experiments, it turns that some customised generation tactics should be used.Such specific, factorised, tactics are not always easy to figure out and essentially depend on the studied specification. A fair compromise would be to define some factorized rules, that describe complex but yet generic behaviors.

The test experiments revealed no errors, which is not a big surprise given that the system under test is already in use. However, this case study presents a proof of technology of how our environment can be used for a real system. On the basis of this environment, it remains to introduce more realistic testing strategies than exhaustivity. It will be done by introducing stronger test hypotheses, thus some guidance of the test-generation tactics. The formal basis of our development should make it easy to define such hypotheses in forms of test purposes or coverage criteria. This can be done either on the specification level (by composing the process with a test purpose, similarly to [3]) or at the test generation level (by introducing the test purposes as structural constraints on the resulting tests).

An important drawback of the current approach is the number of resulting inconclusive test cases. We are currently working on some alternative test definitions, that will allow us to reduce the number of these cases. An other hot topic is the implementation of some on-line testing facilities. This would improve drastically the efficiency of the testing process and reduce by the way the number of inconclusive tests.

References

[1] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. *Formal Aspects of Computing (FAOC)*, 2012.

[2] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in Circus. *Acta Inf.*, 48(2):97–147, April 2011. ISSN 0001-5903.

[3] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *TACAS*, pages 470–475, 2002.

[4] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME*, pages 268–284, 1993.

[5] Abderrahmane Feliachi. *Semantics-Based Testing for Circus*. PhD thesis, Université Paris-Sud 11, 2012.

[6] L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 40–54. 2006. ISBN 978-3-540-49699-1.

[7] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, 2008. Springer.

[8] C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.

[9] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electron. Notes Theor. Comput. Sci.*, 187:107–123, 2007.

[10] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB '02*, pages 184–203, London, UK, 2002. Springer-Verlag.