Int J Software Informatics, Volume 9, Issue 2 (2015), pp. 177–203
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Symbolic Test-generation in HOL-TestGen/CirTA A Case Study

Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff

(Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France)
Email: {feliachi,gaudel,wolff}@lri.fr

Abstract HOL-TestGen/*CirTA* is a theorem-prover based test generation environment for specifications written in *Circus*, a process-algebraic specification language in the tradition of CSP. HOL-TestGen/*CirTA* is based on a formal embedding of its semantics in Isabelle/HOL, allowing to *derive* rules over specification constructs in a logically safe way. Beyond the derivation of algebraic laws and calculi for process refinement, the originality of HOL-TestGen/ *CirTA* consists in an entire derived theory for the generation of symbolic test-traces, including optimized rules for test-generation as well as rules for symbolic execution. The deduction process is automated by Isabelle tactics, allowing to protract the state-space explosion resulting from blind enumeration of data.

The implementation of test-generation procedures in *CirTA* is completed by an integrated tool chain that transforms the initial *Circus* specification of a system into a set of equivalence classes (or "symbolic tests"), which were compiled to conventional JUnit test-drivers.

This paper describes the novel tool-chain based on prior theoretical work on semantics and test-theory and attempts an evaluation via a medium-sized case study performed on a component of a real-world safety-critical medical monitoring system written in Java. We provide experimental measurements of the kill-capacity of implementation mutants.
Key words: symbolic test-case generations; black box testing; theorem proving; model-based testing; JUnit

Feliachi A, Gaudel MC, Wolff B. Symbolic test-generation in HOL-TestGen/CirTA a case study. *Int J Software Informatics*, **Vol.9, No.2 (2015): 177–203. http://www.ijsi.org/1673-7288/9/i214.htm**

1 Introduction

In this paper, we present a combination of test and proof techniques for the test of system behaviour, in particular for tests of systems involving both complex data (i. e. infinite or just large states) and concurrent, non-deterministic behaviour. We apply proof techniques for the *foundation* of the rules – derived from a formal semantics – for test-generation and execution in form of an optimized symbolic computation controlled by proof-tactics. By engineering suitable front- and back-ends, we build a novel integrated tool-chain, that computes, from a behavioral specification, concrete JUnit-Testers used for the process-oriented verification of a (black-box) application.

Corresponding author: Burkhart Wolff, Email: wolff@lri.fr
Received 2014-12-08; Revised 2015-04-01; Accepted 2015-04-30.

We chose as starting point a process algebra in the tradition of CSP, namely *Circus*[26]. One advantage of process algebras is that they come with a rich abstract meta-theory, comprising both denotational and operational semantics. The latter produces a natural symbolic foundation for labelled transition systems, where the nodes were labelled with process-expressions representing classes of process states. A particular advantage of process-algebras over other automata-theoretic approaches and related temporal or modal logics consists in the fact that they distinguish internal and external choice, which comes in handy in a test-theory (such as Ref. [3]) where the non-deterministic choices of the tester must be distinguished from the non-deterministic choices of the *system under test* (SUT). A further advantage of the choice for *Circus* is that its processes have first-class state, i. e. in contrast to CSP, thread-local variables over arbitrary, possibly infinite types can be expressed directly.

The present work resides on the shoulders of giants: Besides the theoretic foundations of process-algebras[14,21], there are a number of related test theories (Refs. [3,8], just to name a few). With respect to the former, there had been some works to represent their denotational semantics in a HOL theorem prover (see Refs. [2,15,23]). This also holds for the denotational semantics for *Circus* that been described in Ref. [11], where a basic version of *CirTA* has been *applied* to interactive refinement proofs. Test-theories based on symbolic execution such as Ref. [3] have, to the best of our knowledge, never been formalized in HOL before[12], which represents a first and still partial step towards this now completed goal.

We claim the following contributions of this paper over prior work:

1. We extend the prior denotational semantics to a formal testing theory (partly published in Ref. [12]); this requires a reconstruction of foundamental notions,*e.g.* "symbolic variables" (just constants in Ref. [3]) and symbolic execution,
2. This symbolic execution calculus has been extended by derived rules for *optimized* test-generation and specific tactic support. The resulting integrated environment is called *CirTA*.
3. *CirTA* has been completed by an improved front-end and a novel backend for the generation of JUnit-testdrivers.
4. *CirTA* has been applied to a medium-size case study of a safety-critical medical system, allowing for an experimental evaluation.

In particular, we would like to mention that *CirTA* profits from the underlying Isabelle/HOL environment[18] and its HOL-TestGen [1] extension which gives access to powerful, up-to-date constraint-solvers such as Z3[9]. Note, however, that the core of *CirTA*'s test-case generation engine works very differently from HOL-TestGen; while the latter is based on *data-driven* case-splitting strategies (i. e. splitting over variables of data-types such as lists, trees, *etc*), the former works on constants of the abstract type α process for which the *CirTA* environment generates constant definitions from *Circus* input syntax.

This paper is organised as follows: After a presentation of the *Circus* language, the basis of its formalization in Isabelle/HOL, we develop its formalized and machine-supported testing theory resulting from refinement notions (3.4); the resulting test

generation engine is described (3.5). This is a presentation of essentials of earlier publications[11,12] meant to make this paper self-contained. Section 4 describes the target of our case-study, which is a demultiplexer of a monitoring system, a highly concurrent Java program used in a health-care application. Section 5 presents the test theory of the system and the test specification. Finally, we present the results of test generation experiments (5.2), and their test executions (5.3). The latter three sections consist of original work.

2 Circus in Isabelle/HOL

Circus is a formal specification language[20] which integrates the notions of states and complex data types (in a Z-like style) with communicating parallel processes inspired from CSP. From Z, the language inherits the notion of a schema used to model sets of (ground) states as well as syntactic machinery to describe pre-states and post-states; from CSP, the language inherits the concept of *communication events* and typed communication channels, the concepts of deterministic and non-deterministic choice (reflected by the process combinators $P \,\Box\, P'$ and $P \sqcap P'$), the concept of concealment (hiding) $P \backslash A$ of events in A occurring in the evolution of process P. Due to the presence of state variables, the *Circus* synchronous communication operator syntax is slightly different frome CSP: $P \;[\![\; n \mid c \mid n' \;]\!] P'$ means that P and P' communicate via the channels mentioned in c; moreover, P may modify the variables mentioned in n only, and P' in n' only, n and n' are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics[21], (but coined in terms of the Unifying Theories of Programming UTP[20]) providing a notion of execution trace `tr`, refusals `ref`, and divergences. It is expressed in terms of the UTP[13] which makes it amenable to other refinement-notions in UTP. Figure 1 presents a simple *Circus* specification, `FIG`, the fresh identifiers generator.

$[ID]$
channel req
channel $ret, out : ID$
process $FIG \;\widehat{=}\; \textbf{begin}$
state $S \;==\; [\, idS : \mathbb{P}\; ID \,]$
$Init \;\widehat{=}\; idS := \emptyset$

Out

ΔS
$v! : ID$

$v! \notin idS$
$idS' = idS \cup \{v!\}$

Remove

ΔS
$x? : ID$

$idS' = idS \setminus \{x?\}$

• $Init$; **var** $v : ID$ •
$(\mu\; X \bullet (reqOut\,;\; out!vSkip \;\Box\; ret?xRemove)\,;\; X)$
end

Figure 1. The Fresh Identifiers Generator in (Textbook) *Circus*

Isabelle and Isabelle/HOL.

Isabelle[18] is a generic theorem prover implemented in SML. It is based on the so-called "LCF-style architecture", which makes it possible to extend a small trusted

logical kernel by user-programmed procedures in a logically safe way. New object logics can be introduced to Isabelle by specifying their syntax and semantics, by deriving its inference rules from there and program specific tactic support for the object logic. Isabelle is based on a typed λ-calculus including a Haskell-style type-system with type-classes (e.g. in α :: order, the type-variable ranges over all types that posses a partial ordering.). Higher-order logic (HOL)[4] is a classical logic. The Isabelle instance Isabelle/HOL provides the usual logical connectives like $\_ \wedge \_$, $\_ \Longrightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall\, x \bullet P x$ and $\exists\, x \bullet P x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow$ bool. HOL is more expressive than first-order logic since, *e.g.* , induction *schemes* can be expressed inside HOL as an ordinary formula containing second-order variables. Offering support for data-types, records and recursive-function definitions including pattern-matching, Isabelle/HOL has a "look-and-feel" of a programming language like SML or Haskell on the one hand and a specification language roughly similar to Z providing logical quantifiers ranging over elementary and function types. We will use HOL as semantic meta-language of *Circus* and Isabelle/HOL as implementation framework and symbolic engine to perform symbolic test generation of *Circus* specifications.

Extensible records in Isabelle/HOL.

Isabelle/HOL's support for *extensible records* is of particular importance for our work. Record types are denoted, for example, by:

$$\texttt{record T = a::T}_1 \quad \texttt{b::T}_2$$

which implicitly introduces the record constructor $(\!|\texttt{a:=e}_1\texttt{,b:=e}_2|\!)$ and the update of record r in field a, written as $\texttt{r}(\!|\texttt{a:= x}|\!)$, as well as the selectors `x r`. Extensible records are represented internally by cartesian products with an implicit free component δ, i.e. in this case by a triple of the type $\texttt{T}_1 \times \texttt{T}_2 \times \delta$. Isabelle/HOL provides an alternative syntax for the type of these triples: $(\!|\texttt{a}\mapsto\texttt{T}_1\texttt{,b}\mapsto\texttt{T}_2\texttt{,...}|\!)$. The third component can be referenced by a *special selector* `more` available on extensible records. Thus, the record `T` can be extended later on using the syntax:

$$\texttt{record ET = T + c::T}_3$$

The key point is that theorems can be established, once and for all, on `T` types, even if future parts of the record are not yet known, and reused in the later definition and proofs over `ET`-values. Using this feature, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining the full expressivity of HOL wrt. the types of $\texttt{T}_1$, $\texttt{T}_2$ and $\texttt{T}_3$.

2.1 Denotational semantics in CirTA

Embedding UTP Predicates and Relations in HOL.

The *Circus* denotational semantics[20] has been originally coined in terms of a general framework for denotational semantics theories called the *Unifying Theory of Programs* (UTP) proposed in Ref. [13]. Traditional UTP is based on an *alphabetized relational calculus*, combining *alphabetized predicates* which are pairs (*alphabet*,

predicate) where the free variables appearing in the predicate are all in the alphabet, e.g. $(\{x, y\}, x > y)$. As such, it is very similar to the concept of a *schema* in *Z*.

We will not require a deep knowledge of UTP here; it suffices to understand that the concept of *alphabetized predicate* can be reasonably truthfully represented in Isabelle/HOL as *sets of records*, i. e. the alphabetized predicate above becomes in HOL just the function:

`λ A::(|x↦T,y↦T'|). x A > y A`

where `(|x↦T,y↦T'|)` is the record type of `A` and `(|x↦T,y↦T'|)⇒ bool` is the type of this predicate. Note that our UTP version allows for fully typed alphabets.

As a consequence, an *alphabetized relation* is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this case the predicate describes a relation between input and output variables, for example $(\{x, x', y, y'\}, x' = x + y)$ which is a notation for: `{(A,A').x A' = x A + y A}`, which is a set of pairs, thus a relation; note that typed sets and boolean functions are isomorph in HOL.

Standard predicate calculus operators are used to combine alphabetized predicates. The definition of these operators is very similar to the standard one, with some additional — but in our context irrelevant — constraints on the alphabets.

Embedding UTP Designs and processes in HOL.

In UTP, in order to explicitly record the termination of a program, alphabetized relations of a particular form were introduced. These relations are called *designs* and their alphabet should contain the special boolean variable `ok` (i. e., `ok` of type bool is element of the alphabets we are talking from now on). `ok` is used to record the start and termination of a program. A UTP design is formally defined as follows:

`(P ⊢ Q) ≡ λ (A,A'). (ok A ∧ P (A,A')) ⟶ (ok A' ∧ Q (A,A'))`

where `P` and `Q` describes the pre and post conditions of the design.

By extending the alphabet (and specializing the corresponding record type), the concept of a *reactive process* is introduced in UTP. Three observational variables are defined for this subset of alphabetized relations: `wait`, `tr` and `ref`. The boolean variable `wait` records if the process is waiting for an interaction or has terminated. `tr` records the list (trace) of interactions the process has performed so far. The variable `ref` contains the set of interactions (events) the process may refuse to perform. These observational variables defines the basic alphabet of all reactive processes called "`alpha_rp`". The reader familiar with the denotational semantics of CSP[21] might recognize the classical trace/refusal sets as denotations for the so-called failure-semantics model.

As in standard CSP, not all combinations of traces and failures constitute a legal process; for this reason, some *healthiness conditions* are defined over `wait`, `tr` and `ref` to ensure that a reactive process satisfies some properties[6]. Alltogether, these conditions are grouped to a predicate `is_CSP_process`; A process that satisfies these conditions is said to be CSP healthy.

Embedding Circus Denotational Semantics in HOL/UTP.

Based on this HOL/UTP foundation, it is now straightforward to convert the *Circus* textbook operator semantics one by one to formalized Isabelle/HOL

denotational definitions. The denotational theory of these operator definitions serve as semantic "gold-standard", assuring logical consistency of the derived algebraic equivalences of processes as well as derived rules for operational semantics and our testing theory.

The set of *Circus* healthy reactive process expressions is captured by the HOL type *actions* in *CirTA*. It is based on the type `(α,σ)relation_rp` which is an instance of the reactive process type `alpha_rp` extended by `channels` of type α; the further possible record extensions (the more-field of the extensible record) were summarized under σ and are used for thread-local and global state variables.

Wrapping up, we can encode the concept of a process for a family of possible state instances by the type definition for `action`:

```
typedef(Action)
 (α::chan_eq,σ) action = {p::(α,σ)relation_rp. is_CSP_process p}
proof - {...}
qed
```

As mentioned before, a type-definition introduces a new type by stating a set. In our case it is the set of reactive processes that satisfy the healthiness-conditions for CSP-processes, isomorphic to the new type. Isabelle enforces in a type-definition a proof assuring the non-emptyness of the type; this proof is ommitted here. Technically, this Isabelle specification construct introduces two constants `Abs_Action` and `Rep_Action` respectively of type `(α,σ) relation_rp ⇒(α,σ) action` and `(α,σ) action ⇒(α,σ) relation_rp`. Moreover, it generates the usual two axioms expressing the bijection between the type and its corresponding set: `Abs_Action(Rep_Action(X))= X` and `is_CSP_process p ⟹ Rep_Action(Abs_Action(p))= p` (recall that the predicate `is_CSP_process` captures the healthiness conditions).

Every *Circus* action is an abstraction of an alphabetized predicate. In Ref. [10], we introduce the definitions of all the actions and operators using their denotational semantics. The *CirTA* semantics library contains, for each action, the proof that this predicate is CSP healthy.

We refrain from a full-blown presentation of all operators here (a complete treatment is contained in Refs. [11,10]) and present just two: the sequential composition of processes and the communication operator for *prefixed actions*.

The former is particularly easy to formulate in *Circus* taking advantage of the UTP framework:

```
definition Seq::('α,'σ) action ⇒('α,'σ) action ⇒('α,'σ) action
                (infixl "';'" 24)
where      "P ';' Q ≡Abs_Action (Rep_Action P ;; Rep_Action Q)"
```

In other words, actions `P` and `Q` were represented as transitions, i. e. relations between `action_rp`'s, which were composed by the HOL relational composition `_ ;; _` and abstracted to an action again.

The semantics of the prefix action is given by the following definition:

```
definition Prefix c x P S ≡Abs_Action(R (true ⊢ Y)) ';' S
where     Y =  do_I c x P ∧ (λ (A, A'). more A' = more A)
```

where `c` is a channel constructor, `x` is a variable (of type var), `P` is a predicate and `S` is an action. This definition states that the prefixed action semantics is given by the interaction semantics operator (`do_I`) which is ommitted here (the interested reader is referred to Ref. [10]).

Instead of `Prefix c x true P` we configure the Isabelle syntax engine that it parses the common notation `c?x→P(x)` equivalently.

2.2 'Algebraic' semantics

Based on the denotational definitions of all operators, it is possible (and, due to the UTP representation in terms of relational algebra, actually fairly easy) to *derive* a number of equations on *Circus* actions. These equations (called 'axiomatic semantics in the CSP literature, although in our context they are not axioms but formally proven theorems) allow for the normalization of CSP processes at various places and are a crucial pre-requisite to symbolic evaluation and refinement proofs.

There are about one hundred equations in the denotational theories of *Circus* for space reasons, we mention only a few of them:

```
comp-ndet-r-distr:  (P ;; (Q ⊓ R)) = ((P ;; Q) ⊓ (P ;; R))
comp-ndet-l-distr:  ((P ⊓ Q) ;; R) = ((P ;; R) ⊓(Q ;; R))
cond-idem:          (P ◁ b ▷ P) = P
comp_assoc:         (P ;; (Q ;; R)) = ((P ;; Q) ;; R)
ndet_symm:          (P::'a relation) ⊓ Q = Q ⊓ P
ndet_assoc:          P ⊓ (Q ⊓ R) = (P ⊓ Q) ⊓ R
```

2.3 Operational semantics

The *configurations* of the transition system for the operational semantics of *Circus* are triples $(c \mid s \models A)$ where c is a constraint over the symbolic variables in use, s a symbolic state, and A a *Circus* action. The transition rules over configurations have the form: $(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)$, where the label e represents the performed symbolic event or ε.

The transition relation is also defined in terms of UTP and *Circus* actions. The formalization of the operational semantics is realized on top of Isabelle/*Circus*. In order to introduce the transition relation for all *Circus* actions, configurations must be defined first. Following the shallow symbolic representation, we introduce the following definitions in Isabelle/HOL.

Constraints.

In the *Circus* testing theory[3], the transition relation of the operational semantics is defined symbolically. The symbolic execution system is based on UTP constructs. Symbolic variables (values) are represented by UTP variables with fresh names generated on the fly. The (semantics of the) constraint is represented by a UTP predicate over the values of these symbolic variables.

In our shallow symbolic representation, symbolic values are given by HOL variables, that can be constrained in proof terms, by expressing predicates over them in the premises. This makes the symbolic configuration defined on free HOL variables that are globally constrained in the context. Thus, the explicit

representation of the constraint in the configuration is not needed. It will be represented by a (globally constrained) symbolic state and an action.

Actions.

The action component of the operational semantics as defined in Ref. [3] is a syntactic characterization of some *Circus* action. This corresponds to the syntax of actions defined in the denotational semantics. In our representation of the operational semantics, this component is a semantic characterization of *Circus* actions. The *Circus* action type is given by `(Θ, σ) action` where Θ and σ are polymorphic type parameters for *channels* and *alphabet*; these parameters are instantiated for concrete processes.

Labels and Channels.

All the transitions over configurations are decorated with labels to keep a trace of the events that the system may perform. A label may refer to a communication with a symbolic input or output value, a synchronization (without communication) or an internal (silent) transition ε. In our representation, channels are represented by constructor functions of a data-type specific for a *Circus* process specification. For our symbolic trace example in subsection 2.5, we will have

```
datatype Demux_channels = get thread-id×data
                        | put thread-id×data
                        | finish thread-id×data,
```

which is generated from the syntax for channel declaration. This type `Demux_channels` is the concrete instance of the channel alphabet Θ, and `get`, `put`, and `finish` the only typed channel constructors of the `Demux`-process. A symbolic event is obtained by applying the corresponding channel constructor to a HOL term, thus `get(tid,d)` or `put(tid,d)`. Labels are then defined either by one symbolic event or by ε.

States.

In the *Circus* testing theory[3], the state is represented by an assignment of symbolic values to all *Circus* variables in scope. Scoping is handled by variable introduction and removal and nested scopes are avoided using variable renaming.

Note that we will represent the "symbolic" variables (technically: constants for which an own substitution theory has to be provided for a mechanization) in "shallow embedding" style, i. e. directly by the free and bound variables of Isabelle/HOL, i. e. by its underlying typed λ-calculus. Consequently, all symbolic notions defined in Ref. [3] are mapped to shallow computations from Isabelle's point of view, which opens the way for fast type-checking and fast (built-in) instantiation of rules.

Consequently, the symbolic state can be represented as a symbolic binding (variable x $\mapsto$ HOL term E). Following the representation of bindings by extensible records, the symbolic state corresponds to a record that maps field names to values of an arbitrary HOL type. In order to keep track of nested statements, each *Circus* variable in the state binds to a *stack* of values.

Operational semantics rules revisited.

The operational semantics is defined by a set of inductive inference rules over the transition relation of the form:

$$\frac{C}{(s_0 \models A_0) \xrightarrow{e} (s_1 \models A_1)}$$

where $(s_0 \models A_0)$ and $(s_1 \models A_1)$ are configurations, e is a label and C is the applicability condition of the rule. Note that the revised configurations are *pairs* where s_1 and s_2 are symbolic states in the sense above, and the constraints are no longer kept inside the configuration, but in a side-condition C of the entire operational rule. This way, we can constrain on the HOL-side these symbolic states. A lot of explicit symbolic manipulations (*e.g.* fresh symbolic variable introduction) are built-in quantifiers managed directly by prover primitives. Thus, the shallow representation reduces drastically the complexity of the rules[10].

The entire operational relation is defined inductively in Isabelle covering all *Circus* constructs. Isabelle/HOL uses this specification to define the relation as least fixed-point on the lattice of powersets (according to Knaster-Tarski). From this definition the prover derives three kinds of rules:

- the introduction rules of the operational semantics used in the inductive definition of the transition relation,
- the inversion of the introduction rules expressed as a huge case-splitting rule covering all the cases, and
- an induction principle over the inductive definition of the transition relation.

2.4 Wrapping up: The CirTA system architecture

Isabelle/HOL itself forms the theoretic and technical environment for *CirTA* (similar to HOL-TestGen [1]), a proof and symbolic execution environment for *Circus* specifications. We therefore profit from the generic Isabelle system facilities such as document or code generation as well as libraries, proofs, and proof-procedures.

In particular, we profit from the possibility to reuse the powerful underlying symbolic computing API's of Isabelle to implement our test generation procedures in a logically safe way.

The *CirTA* system is composed of three main components, organized in four different layers.

1. The Isabelle/*Circus* framework, an Isabelle/HOL library that defines the UTP basis and the *Circus* operators in terms of a denotational semantics. This also comprises first-class syntactic support for the definition of concrete *Circus* processes in applications, see subsection 2.5.
2. A collection of libraries defining the operational semantics.
3. The testing theories of *Circus*, introducing notions for symbolic traces, their configuration and transformation; This layer is discussed in more detail in the next section.

4. The top-most layer, the test-generation engine is built by different symbolic test-generation procedures implemented as Isabelle tactics.

Both basic layers consist of theory developments based on definitional axioms and derived rules.[1]

2.5 Example: The demultiplexer in CirTA

We introduce in the following a *Circus* specification (a slight abstraction) of the multiplexer module studied in this paper, given in the syntax of Isabelle/*Circus*. Note that the Z-notation used to specify general state-transitions is parsed away directly into HOL, so Z schemas of the form $[a : T1, b : T2 \mid P\ a\ b]$ are directly parsed to sets of records `{(| a::T1, b::T2|). P a b}`, and operation schemas $[\Delta\sigma \mid Q\ a\ b\ a'\ b']$ are just transition relations: `{(σ , σ'). Q (σ a) (σ b) (σ' a) (σ' b)}`.

```
circusprocess Demux =
  alphabet = [x::thread-id×data, y::thread-id×data]
  state = [queue::(thread-id×data) list, active::(thread-id×data) set]
  channel = [get thread-id×data, put thread-id×data,
                  finish thread-id×data]
schema InitQueue = queue' = [] ∧active' = {}
schema EnQueuePacket = queue' := queue@[x]
schema Choose = (∃(tid,_)∈set queue. tid ∉fst ‘active) ∧
         y := hd (filter (λ(tid,_).∀(tid',_)∈active. tid ≠tid') queue)
schema Activate = active' = active ∪{y}
schema Remove = active' := active - {x}
action Put = put?x →(Schema EnQueuePacket)
action Get = (fst‘(set queue)) - (fst‘active)) ≠{})
                  &  ((Schema Choose); get!y)
action Finish = finish?x∈active →(Schema Remove)
where (Schema InitQueue); μX ·(Put □ Get □  Finish; X)
```

where `thread-id` and `data` are enumerable infinite types (thus fixed by *nat* in our test-generation phase).

A *Circus* - process can have a collection of channels (as in CSP) along which a typed event-alphabet that can be communicated. A process can have a global state, in our case a queue of data, and a set of active communications. Note that unbounded queues are difficult, in general impossible to handle in model-checkers such as FDR. The declaration part of a *Circus* process consists of schema-declations, action declarations and a process initialization. A particular *Circus* feature are schema-declarations; they allow for silent transitions over the process that can be modeled in a fully declarative way (in contrast to CSP). These transitions were described by Z schemata (thus arbitrary predicates over the pair of pre and post state). Actions are processes, which can contain communications (an event communicated along a channel) or references to schemata. Note that *Circus* binding conventions for variables (inspired by Z) make that the `x` in, e.g., `EnqueuePacket`, refers to the same value as the communicated one in `put?x` in action `Put`, where `EnQueuePacket` is used in the same scope. As in CSP, predicates can be used to restrict the set of possible communications.

[1] *CirTA* is available at `www.lri.fr/~wolff/tmp/cirta.gz`. Major components are also published in the Archive of Formal Proofs AFP, `2012-05-27:Isabelle/Circus`.

3 Revisiting the Circus Testing Theories

The embedding of the testing theories of *Circus* essentially depends on its operational semantics as discussed in subsection 2.3. It is the main instrument to decompose *Circus* processes and construct symbolic traces, i. e. traces containing terms with free variables which are restricted by symbolic constraints accumulated during composition. Instead of a randomly choosing values for variables, or model-check constraints interpreted over finite (and small) models, these constraints can be normalized and solved by constraint-solvers such as Z3 directly before (or even in the case of online-testing: during) the test-execution phase rather during modeling or test-case generation.

3.1 Testing theories

Formal testing theories for languages like CSP or *Circus* are based on the testability assumption, that both the Model SPEC and the system under test SUT are processes, where the latter is in a kind of refinement relation to the former like[21]:

$$\text{SPEC} \sqsubseteq_{FD} \text{SUT}$$

it is this refinement relation which can be checked, for example, by model-checkers such as FDR.

Now, model-based testing is based on two fundamental assumptions which motivate to speak rather of *conformance* than of refinement: First, the structure of the SUT is unknown or "black-box"; we only reveal the behaviour of SUT by combining it with a *tester* that feeds it with more or less pre-computed test-stimuli. Second, refinement notions are based on the comparison of infinite sets of traces and refusals, wheras a test must necessarily be based on a finite set of stimuli and observations, which can be, however, chosen to meet certain coverage criteria over SPEC.

Testing from *Circus* specifications is defined for two conformance relations: *traces inclusion* "SPEC conf_T SUT" and *deadlocks reduction* "SPEC conf_D SUT".

The former is based on the idea "all traces in SUT should be possible in SPEC", the latter follows the goal "whenever SPEC refuses an input *in* after a trace t, SUT should do so as well".

In the following, we develop the test-derivation strategy for both conformance notions. As a pre-requisite, we need the notion of symbolic traces or *cstraces*.

Symbolic traces definition.

Let $cstraces(P)$ the set of constrained symbolic traces of the process P. A *cstrace* is a list of symbolic events associated with a constraint defined as a predicate over the symbolic variables of the trace. Events are given by the labels, different from ε, of the operational semantics transitions. Let us consider the relation noted "$\Longrightarrow$" given by:

$$\frac{}{cf_1 \overset{[]}{\Longrightarrow} cf_1} \qquad \frac{cf_1 \overset{\varepsilon}{\longrightarrow} cf_2 \quad cf_2 \overset{st}{\Longrightarrow} cf_3}{cf_1 \overset{st}{\Longrightarrow} cf_3} \qquad \frac{cf_1 \overset{e}{\longrightarrow} cf_2 \quad cf_2 \overset{st}{\Longrightarrow} cf_3 \quad e \neq \varepsilon}{cf_1 \overset{e\#st}{\Longrightarrow} cf_3} \qquad (*)$$

where cf_1, cf_2 and cf_3 are configurations.

The *cstraces* set definition is given in Ref. [3] using the relation (*) as follows:

Definition 1. for a given process P, an initial constraint c_0, an initial state s_0

$$cstraces^a(c_0, s_0, P) = \{(st, \exists(\alpha c \setminus \alpha st) \bullet c) \mid s\, P_1 \bullet \alpha st \leqslant a \wedge (c_0 \mid s_0 \models P) \overset{st}{\Longrightarrow} (c \mid s \models P_1)\}$$

$$cstraces^a(\textbf{begin}\ state[x : T]P \bullet \textbf{end}) = cstraces^a(w_0 \in T, x := w_0, P)$$

One can read: the constrained symbolic traces of a given configuration are the constrained symbolic traces that can be reached using the operational semantics rules starting from this configuration.

The shallow symbolic representation of this definition is simpler since the symbolic alphabet a is not addressed explicitly. It is also the case for the symbolic constraint because it is described by the characteristic predicate of the set of these traces. Therefore, the *cstraces* set is defined in our theory as follows:

Definition 2.

```
cstraces P = {st. ∃s P1. (s0 ⊨P) =st⇒ (s ⊨P1)}
```

Since the operational semantics rules contain premises that ensure the validity of the target constraint, the trace constraint is embedded in the set predicate: in our formalization, a constrained symbolic trace is seen as a concrete trace, i. e. a trace with symbolic HOL variables, restricted by rules premises. Thus, the constraint of a constrained symbolic trace can be retrieved using set membership.

3.2 Test-generation for traces inclusion

The first studied conformance relation for *Circus*-based testing corresponds to the traces-inclusion refinement relation. This relation states that all the traces of the System Under Test (SUT) belong to the traces set of the specification, or in other words, the SUT should not engage in traces that are not traces of the specification.

As seen previously, a forbidden *cstrace* is defined by a prefix which is a valid *cstrace* of the specification followed by a forbidden symbolic event (continuation). The set of forbidden continuations is called $\overline{csinitials}$, the set of valid continuations is *csinitials*. Because of the constrained symbolic nature of the *cstraces* and events, $\overline{csinitials}$ is not exactly the complement of *csinitials*.

$csinitials$ definition.

csinitials is the set of constrained symbolic events a system may perform after a given trace. It is defined in Ref. [3] as follows:

Definition 3. For every $(st, c) \in cstraces^a(P)$

$$csinitials^a(P, (st, c)) = \{(se, c \wedge c_1) \mid (st@[se], c_1) \in cstraces^a(P) \wedge (\exists\, a \bullet c \wedge c_1)\}$$

Symbolic initials after a given constrained symbolic trace are symbolic events that, concatenated to this trace, yield valid constrained symbolic traces. Only events whose constraints are compatible with the trace constraint are considered.

We introduce the shallow symbolic representation of this definition as follows:

Definition 4.

```
csinitials (P, tr) = {e. tr@[e] ∈ cstraces (P)}
```

All explicit symbolic manipulations are removed, since they are implicitly handled by the prover. The constraint of the trace is not considered, since at this level *tr* is considered as a single concrete trace.

$\overline{csinitials}$ definition.

In order to generate tests for the *traces inclusion* relation, we need to introduce the definition of $\overline{csinitials}$. This set contains the constrained symbolic events the system must refuse to perform after a given trace. These elements are used to lead the SUT to execute a prohibited trace, and to detect an error if the SUT does so.

Definition 5. for every $(st, c) \in cstraces^a(P)$

$$\overline{csinitials}^a(P,(st,c)) = \left\{ (d.\alpha_0, c_1) \mid \begin{pmatrix} \alpha_0 = a(\#st+1) \wedge \\ c_1 = c \wedge \neg \bigvee\{c_2 \mid (d.\alpha_0, c_2) \in csinitials^a(P,(st,c))\} \end{pmatrix} \right\}$$

The $\overline{csinitials}$ set is built from the *csinitials* set: if an event is not in *csinitials* it is added to $\overline{csinitials}$, constrained with the constraint of the trace. If the event is in *csinitials* it is added with the negation of its constraint. The new symbolic variable α_0 is defined as a fresh variable in the alphabet a, the next after the symbolic variables used in the symbolic trace *st*.

In our theories, the symbolic execution is carried out by the symbolic computations of the prover. Consequently, all explicit symbolic constructs are removed in the representation of $\overline{csinitials}$. This representation is introduced as follows:

Definition 6.

```
csinitialsb (P,tr) = {e. ¬Sup {e ∈csinitials(P,tr)}}
```

where the *Sup* operator is the supremum of the lattice of booleans which is predefined in the HOL library, i. e. generalized set union. No constraint is associated to the trace *tr* because it is globally constrained in the context. Symbolic $\overline{csinitials}$ are represented by sets of events where the constraint can be retrieved by negating set membership over the *csinitials* set.

3.3 Test-generation for deadlocks reduction

The *deadlocks reduction* conformance relation, also known as *conf*, states that all the deadlocks must be specified. Testing this conformance relation aims at verifying that all specified deadlock-free situations are dead-lock free in the SUT. A deadlock-free situation is defined by a cstrace followed by the choice among a set of events the system must not refuse, i. e. if the SUT is waiting for an interaction after performing a specified trace, it must accept to perform at least one element of the proposed *csacceptances* set of this trace.

Definition of *csacceptances*.

In order to distinguish input symbolic events from output symbolic events in the symbolic acceptance sets, the set *IOcsinitials* is defined. This set contains, for a given configuration, the constrained symbolic initials where input and output information is recorded. Since inputs and outputs are considered separately in the labels of the transition relation, the set of *IOcsinitials* is easy to define. It contains the set of labels (different from ε) of all possible transitions of a given configuration.

Definition 7. for a given process P_1

$$IOcsinitials^a_{st}(c_1, s_1, P_1) = \left\{ \begin{array}{l} (l, \exists(\alpha c_2 \setminus (\alpha(st@[l])) \bullet c_2) \mid s_2,\ P_2 \bullet \\ \quad (c_1 \mid s_1 \models P_1) \xrightarrow{l} (c_2 \mid s_2 \models P_2) \wedge l \neq \varepsilon \wedge \alpha(st@[l]) \leqslant a \end{array} \right\}$$

A symbolic acceptance set after a given trace must contain at least one symbolic event from each *IOcsinitials* set obtained from a stable configuration after this trace. In our representation of this definition the alphabets a and $\alpha(st)$ are not addressed explicitly, and the constraint is defined as the set predicate.

Definition 8.

```
IOcsinitials cf = {e. ∃cf'. cf -e→ cf' ∧e ≠ε}
```

The general definition of *csacceptances* was introduced in Ref. [3] as follows:

Definition 9. for every $(st, c) \in cstraces^a(P_1)$ we define

$$csacceptances^a(c_1, s_1, P_1, (st, c)) = \left\{ SX \mid \left(\begin{array}{l} \forall\, c_2, s_2, P_2 \bullet \left(\begin{array}{l} (c_1 \mid s_1 \models P_1) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models P_2) \wedge \\ (\exists\, a \bullet c_2 \wedge c) \wedge stable(c_2 \mid s_2 \models P_2) \end{array} \right) \bullet \\ \exists\, iose \in SX \bullet iose \in IOcsinitials^a_{st}(c_2, s_2, P_2) \upharpoonright^a c \end{array} \right) \right\}$$

where

$$stable(c_1 \mid s_1 \models P_1) = \neg\ \exists\, c_2, s_2, P_2 \bullet (c_1 \mid s_1 \models P_1) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models P_2)$$

$$S \upharpoonright^a c = \{(se, c \wedge c_1) \mid (se, c_1) \in S \wedge (\exists\, a \bullet c \wedge c_1)\}$$

The *csacceptances* are computed using the *IOcsinitials* after a given stable configuration of the specification. A configuration is stable if no internal silent evolution is possible directly for its action. Only *IOcsinitials* whose constraints are compatible with the constraint of the tested trace are considered. A filter function $\upharpoonright$ is introduced in order to remove unfeasible initials.

The *csacceptances* set defined above is infinite and contains redundant elements since any superset of a set in *csacceptances* is also in *csacceptances*. A minimal symbolic acceptances set $csacceptances_{min}$ can be defined to avoid this problem. The $csacceptances_{min}$ set after a given *cstrace* must contain exactly one element from each *IOcsinitials* set. Unlike *csacceptances*, the $csacceptances_{min}$ contain only elements that are possible *IOcsinitials*. It is defined as follows:

Definition 10.

```
csacceptances_min tr s A =
      cart (⋃{SX. ∃t∈(after_trace tr s A). SX ∈IOcsinitials t})
```

where `after_trace` is defined by:
`after_trace tr s A = {t. (s ⊨A) =tr⇒A t ∧stable t}`
and `cart` operator defined below is a generalized Cartesian product whose elements are sets, rather than tuples. It takes a set of sets SX as argument, and defines also a set of sets, characterized as follows:
`cart SX = {s1. (∀s2∈SX. s2 ≠{} ⟶(∃e. s2 ∩s1 = {e}))`
`∧ (∀e∈s1. ∃s2∈SX. e∈s2)}`

The resulting $csacceptances_{min}$ of this definition is minimal (not redundant), but can still be infinite. This can come from some unbound internal nondeterminism in the specification that leads to infinite possibilities. In this case, the set cannot be restricted and all elements must be considered.

Each element of the resulting $csacceptances_{min}$ set is a set of symbolic events. A symbolic acceptance event is represented as a set of concrete events. The instantiation of these sets is done using the membership operator.

3.4 The concepts in terms of our running example

We will demonstrate the key-concepts

- *traces inclusion* ("in which traces should the SUT at least engage?"), and
- *deadlocks reduction* ("in which traces should the SUT at most engage?").

in terms of our running Multiplexer example shown in subsection 2.5.

An example of a *constrained symbolic trace* in terms of the Multiplexer and a *constrained symbolic event* after this trace is given by:

$$([\mathtt{put}(tid1, d1), \mathtt{put}(tid2, d3), \mathtt{put}(tid1, d5), \mathtt{get}(c)], \\ tid1 = tid(c) \wedge pktno(d5) > pktno(d1))$$

Note that we adopt the convention that for HOL-constants like `put` we use typewriter font (the channels in our *Circus* specification were compiled to a data-type) while for HOL-variables like $tid1$ we use an italic font.

Furthermore, note that we are interested in the implicit "Queue"-property in this symbolic traces in our case study, since they represent a safety-critical property in our case-study: the order of sent messages should never be switched. The schema operations ensure this order in the specification, thus, the generated tests will cover situations in which the order is not preserved.

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 csevent *cse* belonging to the set $\overline{csinitials}$ associated with *cst* in P. An example of a symbolic test for *traces inclusion* is given by:

$$([\mathtt{put}(tid1, a), \mathtt{put}(tid1, b), \mathtt{get}(c)], tid(c) \neq tid1 \vee data(c) \neq a)$$

deadlocks reduction (also called *conf* in the literature) 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. Here is an example of a test for a *deadlocks reduction* test-trace:

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

In Ref. [10], we presented a formalization of all these notions in Isabelle/*Circus*. This formalization represents the second layer of the *CirTA* system, and it is the basis of the test generation tactics defined in the top-most (and novel) layer.

The CirTA 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 tests 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:

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

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 reaches some normal form of clauses, in our case, when we reach logical formulas of the form: $C \implies Prog(tr)$, 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*; Isabelle's decision procedures will to a large extent close these subgoals automatically, (the remaining cases were proven infeasible by user interaction).

Recursive process specifications may imply traces of unbounded length and thus an unbounded number of symbolic traces. The generation is then limited by a given trace length k, defined as a parameter of the generation process. The list of subgoals corresponds to all possible traces of 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 by:

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

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. Transforming this set, the feasible $\overline{csinitials}$ were generated and booked as new subgoal into the final generation state.

Test-generation for *deadlocks reduction.*

Test-generation in this case is based on the generation of the $csacceptances_{min}$ 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 Ref. [10].

4 The Application: A Message Demultiplexer in a Medical Home-Monitoring System

Our case study addresses a part of a remote monitoring system used in a worldwide health-care network, where a variety of devices — most notably pacemaker controllers — where connected via a network. 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 very large number of message-packets then recollects them to messages transferred to their corresponding "workers" in order to be processed. The demultiplexer is implemented by highly complex, highly efficient concurrent Java-code, and a key security property became a major concern: in each communication thread (which are actually prioritized, so the demultiplexer can choose higher-prior `get`'s to lower-prior ones), the FIFO property of message packets must be respected in any circumstance.

An overview of the remote monitoring system is given in Fig. 2.

Figure 2. 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 consists of a number of elementary packages (characterized with a device identifier) and its actual content. The queue receives and stores messages, and then forwards them to an assigned processing ("worker") service. A crucial safety property of this arrangement is that the packet order between packets belonging to the same message is maintained and that no confusion between packets belonging to different messages arises. The main operations of the *implementation* that can be performed by the multiplexer are:

- *PUT:* The queue receives the messages using the PUT operation and stores them with the *new* status. Messages are ordered following their reception.
- *GET:* The processing services can retrieve *new* messages from the queue and mark them as *active*. The queue decides which message it provides to the processing service according to two conditions. First, the message must be the oldest available message in the queue. Second, there is no active message with the same identifier as the returned message.
- *FINISH:* When a processing service successfully completes a message processing, this *active* message is completely removed from the queue.

The reader will notice the direct match to the *interface* of our queue in subsection 2.5, where the *Circus* specification represents the *abstract test model* from which the concrete implementation (consisting of several thousands of dense concurrent Java-code) is tested.

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.

5 Testing in CirTA

The testing procedure is very similar to the standard HOL-TestGen procedure described in Ref. [1] — albeit now generating tests for *Circus* processes rather than just data-types. First, a combined HOL and *Circus* formula representing the *test specification* must be introduced describing the test goal. Inference rules are then used to derive tests from this specification following some predefined tactics. Finally, testers are generated from the resulting tests and executed against the SUT.

5.1 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.

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

$$\texttt{tr} \in \texttt{cstraces Demux} \Longrightarrow \texttt{prog tr}$$

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

For each generated trace, different tests 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 Demux ∧e ∈csinitialsb Demux 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 Demux ∧e ∈csacceptances Demux tr ⟹  prog tr e
```

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

5.2 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 5.3).

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[7] 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 6.

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 Demux [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` $\neq$ `a` and `bf` $\neq$ `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 tests, 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 tests. 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* tests 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* tests, a uniformity hypothesis, is stated for each test. 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* tests 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 tests 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.

5.3 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 and then translated into JUnit tests. The resulting tests are then directly executed on the given implementation.

In order to execute the concrete tests against the provided Java implementation of the queue, these tests 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 presented previously, produce the following JUnit method:

```
1  public void testqueue1_1() throws Exception {
2      AbstractQueueableObject o_3_1 = new NamedEntry("topic", 3, 1);
3      AbstractQueueableObject o_0_0 = new NamedEntry("topic", 0, 0);
4      AbstractQueueableObject o_0_1 = new NamedEntry("topic", 0, 1);
5      AbstractQueueableObject o = null;
6      tm.begin();
7      try { queueManager.put(o_3_1); tm.commit(); }
8      catch (Exception e) { System.out.println("inconclusive"); return;}
9      try { queueManager.put(o_0_0); tm.commit(); }
10     catch (Exception e) { System.out.println("inconclusive"); return;}
11     o = queueManager.get(NamedEntry.class, "topic");
12     assertFalse(equals(o_0_1, o));
13 }
```

Deadlock reduction.

The generated tests for the second conformance relation are also automatically translated into Java methods. 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 generated previously produces the following test method:

```
1  public void testqueue1_1() throws Exception {
2      AbstractQueueableObject o_1_1 = new NamedEntry("topic", 1, 1);
3      AbstractQueueableObject o_10_5 = new NamedEntry("topic", 10, 5);
4      AbstractQueueableObject o_0_2 = new NamedEntry("topic", 0, 2);
5      AbstractQueueableObject o = null;
6      tm.begin();
7      try { queueManager.put(o_1_1); tm.commit(); }
8      catch (Exception e) { System.out.println("inconclusive"); return;}
9      try { queueManager.put(o_10_5); tm.commit(); }
```

```
10      catch (Exception e) { System.out.println("inconclusive"); return;}
11      try { o = queueManager.get(NamedEntry.class, "topic"); tm.commit(); }
12      catch (Exception e)
13          { tm.rollback(); queueManager.put(o_0_2); tm.commit(); }
14      if (o == null || !equals(o_1_1, o)) {
15          tm.rollback(); queueManager.put(o_0_2); tm.commit();
16      }
17  }
```

6 Test Evaluation

Just at the beginning: a warning. Figures referring to the number of symbolic states were often compared to non-symbolic approaches, where blind enumeration of data leads to state systems with billions and trillions of states; it is a desired feature that the number of states in symbolic approaches is relatively small, and not a conclusive proof that the approach "does not scale".[2] Note, furthermore, that some of our symbolic states use formulas of HOL.

The following experiments used 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.

Table 1 Statistics of test generation for trace inclusion

Trace length	Traces time (s) / number	Symbolic Tests time (s) / number	Schematic Tests time (s)	Instantiation time (s)
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	-	-

The generation time and the number of generated tests 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 tests. In order to generate longer tests, one can introduce more specific selection hypotheses. This will produce less but more focused tests.

The exhaustive test generation for length at most 8 produced a total of 4396 tests using the 297 traces of length up to 7. All the generated tests are concrete, where all communicated values are instantiated. As explained before, all these tests 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).

[2]For example, a symbolic automata that accepts correct UTF-8 encodings can be done with 7(!) symbolic states, while corresponding automatas have 2^{1024} states.

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

Table 2 Statistics of test generation for deadlocks reduction

Trace length	Traces time (s) / number	Symbolic Tests time (s) / number	Schematic Tests time (s)	Instantiation time (s)
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	-	-

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 tests in less time.

Test execution.

For the trace-inclusion conformance relation, the execution of the 4396 test methods ended without finding errors (the error that we detected during the modeling phase was already corrected in the code that we tested). Since we used a so-called offline-test-method (test-data is completely computed before the test execution) and since we did *not* instrument the operating system scheduler of the SUT, we ended up with 1024 inconclusive tests, i. e. tests where the concrete test trace is finally not chosen by the SUT due to non-deterministic scheduling. In the second case of deadlock reduction relation, no errors and 494 inconclusive tests resulted from executing the 2473 tests. The component 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 tests 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 tests, 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, but more than 1840 tests were inconclusive. Different mutations were also applied, mainly by inverting some conditions. All these mutants were killed by some tests of the trace inclusion conformance relation. Due to the nature of our mutations, no errors were detected by the tests 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.

7 Conclusion

Related Work.

Symbolic evaluation and constraint solving are widely used, as well as model checking or similar techniques. The LOFT tool[17] performed test generation from algebraic specifications, essentially based on narrowing. TGV[16] performs test generation from IOLTS (Input Output LTS) and test purposes for the *ioco* conformance relation. TGV considers finite transition systems, thus enumerative techniques are used to deal with finite data types. Some symbolic extension of TGV, STG has been enriched by constraint solving and abstract interpretation techniques[5]. The FDR model-checker was used[19] for generating tests from CSP specifications for a conformance relation similar to *ioco*. In Spec Explorer[24], the underlying semantic framework are abstract state machines (ASM) and the conformance relation is alternating refinement. The ASM framework provides foundation to deal with arbitrarily complex states, but the symbolic extension, based on constraint solving, is still experimental. JavaPathFinder[25] has been used for generating test input from descriptions of method preconditions. The approach combines model checking, symbolic execution, constraint solving and improves coverage of complex data structures in Java programs. A strong tool in this line of white-box test systems using symbolic execution and constraint-solcving is the Pex tool[22].

Symbolic evaluation and constraint solving are widely used, as well as model checking or similar techniques. The LOFT tool performed test generation from algebraic specifications, essentially based on narrowing. TGV[16] performs test generation from IOLTS (Input Output LTS) and test purposes for the *ioco* conformance relation. TGV considers finite transition systems, thus enumerative techniques are used to deal with finite data types. Some symbolic extension of TGV, STG has been enriched by constraint solving and abstract interpretation techniques[5]. The FDR model-checker was used[19] for generating tests from CSP specifications for a conformance relation similar to *ioco*. In Spec Explorer[24], the underlying semantic framework are abstract state machines (ASM) and the conformance relation is alternating refinement. The ASM framework provides foundation to deal with arbitrarily complex states, but the symbolic extension, based on constraint solving, is still experimental. JavaPathFinder[25] has been used for generating test input from descriptions of method preconditions. The approach combines model checking, symbolic execution, constraint solving and improves coverage of complex data structures in Java programs. A strong tool in this line of white-box test systems using symbolic execution and model-checking is the Pex tool[22].

In our case, the use of a theorem prover, namely Isabelle/HOL, is motivated by the fact that test generation from rich specification languages such as *Circus* can greatly benefit from the automatic and interactive symbolic computations and proof technology to define sound and flexible test generation techniques. Actually, this is

extremely useful and convenient to deal with infinite state spaces. TGV does not possess symbolic execution techniques and is thus limited to small data models. Our approach has much in common with STG, however its development was abandoned since the necessary constraint solving technologies were not available at that time. In contrast, *CirTA* uses most recent deduction technology in a framework that guarantees its seamless integration. On the other hand, Symbolic JavaPathFinder and Pex are white-box testing tools which are both complementary to our black-box approach.

Summary.

This paper presents a case study of the *CirTA* test generation environment. A first case study covers experience on a process-oriented black-box testing method of a "real" safety-critical component. The system under test (SUT) is a Demux module, implemented in highly concurrent Java, integrated in a conventional JUnit testbed who's test-code has been generated by *CirTA*. Based on an abstract behavioral description of the system component in *Circus*, our environment generates symbolic traces, then tests for two conformance relations, and finally JUnit Testsuites of offline testing of the system under test. Some basic mutation analysis was also performed to evaluate error-detection capacity of the generated tests.

The test generation procedure is an *automatic* solution for any *Circus* specification. However, *CirTA* is an interactive environment allowing not only to modify and adapt the specification, but also the test-generation process; in particular customizations for the simplification or elimination of symbolic test-traces are possible and in practice necessary for unfeasible constraints that could not be proven false automatically and for improving the overall efficiency.

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 problem-adapted test hypotheses, thus some guidance of the test-generation tactics.

References

[1] Brucker AD, Wolff B. On theorem prover-based testing. Formal Aspects of Computing (FAOC), 2012.

[2] Camilleri AJ. Mechanizing csp trace theory in higher order logic. IEEE Trans. on Software Engineering , September 1990, 16(9): 993–1004.

[3] Cavalcanti A, Gaudel M-C. Testing for refinement in circus. Acta Informatica, April 2011, 48(2): 97–147.

[4] Church A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[5] Clarke D, Jéron T, Rusu V, Zinovieva E. STG: A symbolic test generation tool. TACAS 2002. Springer-Verlag, 2002, volume 2280 of LNCS.

[6] Cavalcanti ALC, Woodcock JCP. A tutorial introduction to CSP in unifying theories of programming. Refinement Techniques in Software Engineering. Springer-Verlag. 2006, volume 3167 of LNCS. 220–268.

[7] Dick J, Faivre A. Automating the generation and sequencing of test cases from model-based specifications. FME. 1993. 268–284.

[8] De Nicola R, Hennessy M. Testing equivalences for processes. Theor. Comput. Sci., 1984, 34: 83–133.

[9] De Moura L, Bjørner N. Z3: An efficient smt solver. TACAS. Springer-Verlag, 2008. 337–340.

[10] Feliachi A. Semantics-Based Testing for Circus[PhD thesis]. Université Paris-Sud 11, 2012.
[11] Feliachi A, Gaudel M-C, Wolff B. Isabelle/circus: A process specification and verification environment. VSTTE. 2012. 243–260.
[12] Feliachi A, Gaudel M-C, Wenzel M, Wolff B. The circus testing theory revisited in isabelle/hol. ICFEM. 2013. 131–147.
[13] Hoare CAR, He J. Unifying theories of programming. Prentice Hall, 1998, volume 14.
[14] Hoare CAR. Communicating Sequential Processes. Prentice-Hall, 1985.
[15] Isobe Y, Roggenbach M. A generic theorem prover of csp refinement. TACAS. 2005. 108–123.
[16] Jard C, Jéron T. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT. October 2004, 6.
[17] Marre B. Loft: A tool for assisting selection of test data sets from algebraic specifications. Proc. of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, TAPSOFT '95. London, UK. UK. Springer-Verlag. 1995. 799–800.
[18] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Springer-Verlag. 2002, volume 2283 of LNCS.
[19] Nogueira S, Sampaio A, Mota A. Guided test generation from CSP models. ICTAC 2008. 2008, volume 5160 of LNCS. 258–273.
[20] Oliveira M, Cavalcanti A, Woodcock J. A denotational semantics for Circus. Electron. Notes Theor. Comput. Sci., 2007, 187: 107–123.
[21] Roscoe AW. Theory and Practice of Concurrency. Prentice Hall, 1998.
[22] Tillmann N, Schulte W. Parameterized unit tests. SIGSOFT Softw. Eng. Notes, September 2005, 30(5): 253–262.
[23] Tej H, Wolff B. A corrected failure divergence model for csp in isabelle/hol. FME. 1997. 318–337.
[24] Veanes M, et al. Formal methods and testing. Chapter Model-based Testing of Object-oriented Reactive Systems with Spec Explorer. Springer, 2008.
[25] Visser W, Pasareanu CS, Khurshid S. Test input generation with Java Path Finder. ISSTA 2004. ACM, 2004. 97–107.
[26] Woodcock J, Cavalcanti A. The semantics of circus. ZB '02. London, UK. Springer-Verlag. 200. 184–203 2.