The *Circus* Testing Theory Revisited in Isabelle/HOL

Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel and Burkhart Wolff

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

Abstract Formal specifications provide strong bases for testing and bring powerful techniques and technologies. Expressive formal specification languages combine large data domain and behavior. Thus, symbolic methods have raised particular interest for test generation techniques.
Integrating formal testing in proof environments such as Isabelle/HOL is referred to as "theorem-prover based testing". Theorem-prover based testing can be adapted to a specific specification language via a representation of its formal semantics, paving the way for specific support of its constructs. The main challenge of this approach is to reduce the gap between pen-and-paper semantics and formal mechanized theories.
In this paper we consider testing based on the *Circus* specification language. This language integrates the notions of states and of complex data in a Z-like fashion with communicating processes inspired from CSP. We present a machine-checked formalization in Isabelle/HOL of this language and its testing theory. Based on this formal representation of the semantics we revisit the original associated testing theory.
We discovered unforeseen simplifications in both definitions and symbolic computations. The approach lends itself to the construction of a tool, that directly uses semantic definitions of the language as well as derived rules of its testing theory, and thus provides some powerful symbolic computation machinery to seamlessly implement them both in a technical environment.

Keywords:
Formal Testing, Symbolic Computations, Isabelle/HOL, *Circus*.

1 Introduction

Test generation from formal specifications is an active research area. Several theoretical frameworks and tools have been proposed for various kinds of formal testing techniques and their resulting conformance and coverage notions [9].

We develop a formal test generation framework and tool for the *Circus* specification language. This language combines elements for the description of complex data and behavior specifications, via an integration of Z and CSP with a refinement calculus[18]. *Circus* has a denotational semantics [14], which is based on UTP [10], and an operational semantics that can be found in [2]. Here we present an environment for *Circus* -based testing in the line of [2].

Formal environments like Isabelle/HOL [12] are usually applied for formal proof developments. Integrating formal testing in such environments is referred to as "theorem-prover based testing". Theorem-prover based testing can be adapted to a specific specification language via a representation of its formal semantics, paving the way for specific support of its constructs. The main challenge of this approach is to reduce the gap between pen-and-paper semantics and formal mechanized theories, especially to one amenable to efficient symbolic computing. We present such a semantic theory for *Circus*, develop formally its testing theory in Isabelle/HOL and integrate the result in an own testing environment called HOL-TestGen/*CirTA*. Our environment comes with some basic test selection criteria, and was applied to an industrial case study extracted from an operational remote sensing system.

The main contribution of this paper is the representation of the symbolic testing theory of *Circus* in Isabelle using the symbolic infrastructure of the prover. We show how the original formalization can be much simplified using shallow symbolic computations.

The paper is organised as follows: Section 2 briefly recalls the context of this work, namely essential issues on *Circus* and *Circus*-based testing, the Isabelle/HOL formal environment, and Isabelle/Circus, an embedding of the denotational semantics of *Circus* in Isabelle/HOL, which makes it possible to reason on *Circus* specifications; Section 3 discusses and describes our main choices for embedding the symbolic notions necessary for generating symbolic tests from *Circus* specifications; on these bases, Section 4 explains how the operational semantics and the *Circus* testing theories have been formulated as Isabelle/HOL definitions and theories, leading to some Isabelle/HOL tactics for symbolic test generation; moreover, the definition of two basic test selection criteria is presented, as well as the instantiation of symbolic tests into concrete ones.

2 Context

2.1 *Circus* and *Circus*-based testing

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 [14] presented in terms of the UTP [10], and a corresponding operational semantics [2]. UTP is essential for providing a seamless semantic framework for states and processes. A simple example of a *Circus* specification is given in fig. 1; it describes a Fibonacci-number generator.

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 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

channel $out : \mathbb{N}$
process $Fibonacci \mathrel{\widehat=} \textbf{begin}$
state $FibState == [x, y : \mathbb{N}]$
$InitFibState \mathrel{\widehat=} x := 1;\ y := 1$
$InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow InitFibState$
$OutFibState \mathrel{\widehat=} \textbf{var}\ temp : \mathbb{N} \bullet (temp := y;\ y := x + y;\ x := temp)$
$OutFib \mathrel{\widehat=} \mu\, X \bullet out!(x + y) \rightarrow OutFibState;\ X$
$\bullet\ InitFib;\ OutFib$
end

Figure 1. The Fibonacci generator in *Circus*

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.

Testability hypotheses are assumptions on the SUT that are essential to prove that the success of a testing campaign entails correctness. In the *Circus* testing theory, the first testability hypothesis is that the SUT behaves like some unknown *Circus* process SUT_{Circus}. This means that, in any environment, the execution of the SUT and SUT_{Circus} give the same observations. In this context, even though the SUT is not a *Circus* process, one can use refinement to compare it to a given *Circus* specification. This requires, however, that events used in the specification are perceived as atomic and of irrelevant duration in the SUT.
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).
- $\overline{csinitials}$: given a process P and one of its cstraces (st, c), the set $\overline{csinitials}$ 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).

Examples of a constrained symbolic trace of *Fibonacci* and of a constrained symbolic event in $\overline{csinitials}$ after this cstrace is:

$$([out.1, out.1, out.a, out.b], a = 2 \wedge b = 3) \qquad (out.a, a \neq 5)$$

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 $\overline{csinitials}$ 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. A test that, if successful, deadlocks at the end, is used to check that forbidden traces cannot be executed. An example of a test for *traces inclusion* for *Fibonacci* in given by:

$$([out.1, out.1, out.a, out.b], a = 2 \wedge b \neq 3)$$

Given a *Circus* process P the set of *all* the symbolic tests described above is a symbolic *exhaustive* test set with respect to *traces inclusion*: a SUT that would pass all the instances of all the symbolic tests is a *traces inclusion* of P, assuming some basic testability hypotheses that are given in [2], where the proof can also be found.

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:

$$([out.1, out.1, out.a], a = 2)\ \{out.3\}$$

Given a *Circus* process P the set of all the symbolic tests described above is a symbolic exhaustive test set with respect to *deadlocks reduction* [2] under the same testability hypotheses as above. This is also proved in [2].

2.2 The Isabelle/HOL formal environment

Isabelle [12] is a generic theorem prover implemented in SML. Built upon a small trusted logical kernel, it is possible to provide logical and technical extensions by user-programmed procedures in a logically safe way. These days, the most commonly used logical extension is **Isabelle/HOL** supporting classical Higher-order Logics (HOL), i. e. a logic based on typed λ-calculus including a Haskell-style type-system. HOL provides the usual logical connectives as well as the object-logical quantifiers; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions of type $\alpha \Rightarrow \beta$. Isabelle/HOL comes with large libraries, where thousands of theorems have been proven from definitional axioms; this covers theories for sets, pairs, lists, relations, partial functions, orderings, and arithmetics. We use the HOL-notation throughout this paper (instead of, for example, the Z notation) in order to avoid confusion. The empty list is written [] and the constructor #; lists of the form $a\#b\#[]$ were denoted $[a, b]$. The @-operator denotes list concatenation, the projections into lists are the usual $hd[a, b] = a$ and $tl[a, b] = [b]$. **Isabelle/HOL-TestGen**[1] is a technical extension providing support for formal test generation.

Isabelle/*Circus* is a formalization of UTP and the denotational semantics of the *Circus* language[6] in Isabelle/HOL. For the work presented in this paper, we

will use the operational semantics and the testing theory of *Circus*, and formalize it on top of Isabelle/*Circus*. We will discuss in detail the impact of the symbolic representation of the language for symbolic execution.

3 Shallow Symbolic Computations with Isabelle

The test sets introduced in section 2.1 are defined in [2] using symbolic variables and traces. *Symbolic variables* are syntactic names that represent some values without any type information. These symbolic variables are introduced to represent a set of values (or a single, loosely defined, value), possibly constrained by a predicate. An alphabet a is associated to all symbolic definitions of the testing theory. This alphabet enumerates the symbolic variable names.

A deep symbolic representation would require the definition of these symbolic notions on top of Isabelle/HOL. This would rather be heavy to realize and may introduce some inconsistency in the theory. The main problem is that symbolic variables are just names. They are syntactic not typed entities and the type information recorded in *Circus* variables is not present at this stage. Moreover, constraints would also be syntactic entities, and thus, have to be presented in a side-calculus mimicking *Circus* substitution and type-checking.

As an alternative to this deep symbolic execution, we opt for a so-called *shallow embedding*. This embedding is based directly on the Isabelle symbolic representation and computation facilities. Isabelle, as a formal framework, provides powerful symbolic computation facilities that can be reused directly for our purpose. This requires symbolic variables to be HOL variables, which are semantic typed entities manipulated by the prover. Expressions over these variables are written using HOL predefined operators or logical connectives, constraints are entities directly represented as HOL predicates. With our representation, all the symbolic execution is carried out by Isabelle's symbolic computations.

This representation choice is natural since symbolic computations and higher-order manipulations (definitions, theories, rules, ...) are not of the same nature. They correspond to two different abstraction levels. This is not the case for deep symbolic execution, which would be represented at higher order abstraction level. In shallow representations, low-level symbolic computations are the basis of high-level formal definitions.

This choice of embedding strongly influences the definition and representation of the operational semantics and testing theory. The impacts of this choice are explained in various places in the following sections. In the sequel, we use "symbolic execution" to refer to the explicit (deep) symbolic manipulations as defined in [2]; we use "symbolic computations" to refer to the (shallow) implementation in Isabelle of these symbolic notions. A more detailed development of these issues can be found in [5].

4 Revisiting the *Circus* Testing Theories

The embedding of the testing theories of *Circus* essentially depends on its operational semantics. Thus, we start by introducing a shallow embedding of the *Circus* symbolic operational semantics in Isabelle/HOL.

4.1 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 [2], the transition relation of the operational semantics is defined symbolically. The symbolic execution systemis 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 [2] is a syntactic characterization of some *Circus* actions. This corresponds to the syntax of actions defined in the denotational semantics. In our representation of the operational semantics, the action 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 type parameters are instantiated for concrete processes.

Labels. 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 Section 2.1, we will have the `datatype Fibonacci_channels = out int`, where `Fibonacci_channels` is the concrete instance of the channel alphabet Θ, and `out` the only typed channel constructor of the `Fibonacci`-process. A symbolic event is obtained by applying the corresponding channel constructor to a HOL term, thus `out(3)` or `out(a)`. Labels are then defined either by one symbolic event or by ε.

States. In the *Circus* testing theory [2], 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.

As explained in section 3, symbolic variables are represented by HOL terms. Consequently, the symbolic state can be represented as a symbolic binding (variable name $\mapsto$ HOL term). 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 [5].

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.

4.2 Testing theories

As seen in Section 2.1, testing from *Circus* specifications is defined for two conformance relations: *traces inclusion* and *deadlocks reduction*. These conformance relations are based on the notion of *cstraces*. As explained in section 3, we will represent the "symbolic" by the "shallow"; consequently, all symbolic notions defined in [2] are mapped to shallow computations from Isabelle's point of view.

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 "$\Longrightarrow$" defined by:

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

where cf_1, cf_2 and cf_3 are configurations.

The *cstraces* set definition is given in [2] using the relation (*) as follows:

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

$$cstraces^a(c_0, s_0, P) = \{(st, \exists(\alpha c \setminus \alpha st) \bullet c) \mid s\ P_1 \bullet \alpha st \leq a \wedge (c_0 \mid s_0 \models P) \stackrel{st}{\Longrightarrow} (c \mid s \models P_1)\}$$
$$cstraces^a(\textbf{begin}\ state[x : T]P \bullet \textbf{end}) = cstraces^a(w_0 \in T, x := w_0, P)$$

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.

4.3 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 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 in Section 2.1, 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 [2] 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.

***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 do 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.

4.4 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.

***csacceptances* definition.** 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

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

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 [2] as follows:

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

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

where

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

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

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 $\restriction$ 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.

4.5 The *CirTA* system

CirTA stands for *Circus* Testing Automation, which is a test-generation environment for *Circus*. It defines some general tactics for generating, *cstraces* and test-cases for the two conformance relations introduced earlier.

cstraces **generation tactic.** 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:

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

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 reach a certain normal form of clauses, in our case, when we reach logical formulas of the form: `C ⟹ 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 shallow symbolic definition of *cstraces* makes it possible to simplify the set membership operator into a predicate in the premises. The final step of the generation produces a list of propositions, describing the generated traces stored by the free variable *Prog*. The trace generation tactic is described by the following algorithm:

Data: k : the maximum length of traces
Simplify the test specification using the *cstraces* Definition 2;
while *length* $\leq k \wedge$ *more traces can be generated* **do**
  Apply the rules of (*) on the current goal;
  Apply the rules of the operational semantics on the resulting subgoals;
end

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*

given in 2. The application of the elimination rules (*) 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.

***csinitials* generation tactic.** 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:

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

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

***$\overline{csinitials}$* generation tactic.** 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 $\overline{csinitials}$ are generated and added as a subgoal in the final generation state. This tactic can be represented in the following algorithm:

Data: k : the maximum length of tests
Generate *cstraces* using trace generation tactic for a length k;
foreach *generated trace tr* **do**
  Simplify the test specification (2) using the $\overline{csinitials}$ Definition 6;
  Generate the *csinitials* after *tr* using *csinitials set* generation tactic;
  Apply case-splitting and simplification rules to generate the $\overline{csinitials}$;
end

***csacceptances* generation tactic.** 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 generalized Cartesian product is computed from all resulting *IOcsinitials*. The generation tactic is defined in the following algorithm:

Data: k : the maximum length of tests
Generate *cstraces* using trace generation tactic for a length k;
foreach *generated trace tr* **do**
  Simplify the test specification using the $csacceptances_{min}$ Definition 10;
  Generate all stable configurations after *tr* using the derived rules;
  foreach *generated stable configuration cf* **do**
    Generate all *IOcsinitials* after this configuration cf;
  end
  Introduce the definition of $\bigotimes$ for the resulting set;
  Apply simplification rules to generate the sets $csacceptances_{min}$;
end

4.6 Some Test Selection Hypotheses

Symbolic tests cannot be used directly for testing. A finite number of concrete (executable) tests must be instantiated from them. However, in some situations, there is an infinite number of instances: it may come from infinite types, or from symbolic tests with unbounded length, as mentioned in section 4.5. Some selection criteria must be used to choose a finite subset of concrete finite tests. They are formalized as test selection hypotheses on the SUT: assuming these hypotheses the selected tests form an exhaustive test set [2, 8].

Selection hypothesis that can be used in the case of unbounded tests are *regularity hypotheses*. The simplest one allows to bound the traces length: it states that if the SUT behaves correctly for traces shorter than a given length, it will then behave correctly for all the traces. Other selection criteria are needed to choose a finite subset of concrete tests among the instances of symbolic tests. *uniformity hypotheses* can be used to state that the SUT will behave correctly for all the instances if it behaves correctly for some subset of them. Such a subset can be obtained using on-the-fly constraint solving as, for instance, in [1]

Test selection hypotheses can be explicitly stated in our test-generation framework *CirTA*. Currently, the classical regularity hypothesis on traces length is used, where the maximum regularity length is provided as parameter. Moreover, for each resulting symbolic test, a uniformity hypothesis is stated to extract a *witness* value for each symbolic value in the test. Concrete (*witness*) values are represented by Isabelle *schematic variables* representing arbitrary (but constrained) values. These uniformity and regularity hypotheses are respectively defined as introduction rules as follows:

$$\frac{P\ ?x_1 ... ?x_n \quad THYP((\exists\, x_1, ..., x_n \bullet P\ x_1 ... x_n) \rightarrow (\forall\, x_1, ..., x_n \bullet P\ x_1 ... x_n))}{\forall\, x_1, ..., x_n \bullet P\ x_1 ... x_n}$$

$$\frac{\begin{array}{c}[length(tr) < k] \\ \vdots \\ P\ (tr)\end{array} \qquad THYP((\forall\, tr \mid length(tr) < k \bullet P\ (tr)) \rightarrow (\forall\, tr \bullet P\ (tr)))}{\forall\, tr \bullet P\ (tr)}$$

P is the predicate of a (symbolic) test case, tr is a (symbolic) trace and $THYP$ is a constant used to preserve test hypotheses from automatic simplifications. Schematic variables are represented in Isabelle with ? prefixing their name.

4.7 Test Instantiations

The last step of test-generation is the selection of actual witness values corresponding to schematic variables produced by the uniformity hypothesis. Constraint solvers that are integrated with Isabelle are used for this instantiation, in the same way as what was done in [1]. Two kind of solvers can be used: random solvers and SMT solvers. The random constraint solving is performed using QuickCheck, that instantiates randomly the values of the schematic variables. An integration of QuichCheck with the Isabelle simplifier defined for HOL-TestGen can also be used for more efficient random solving. The second kind of integrated constraint solvers are SMT solvers and especially Z3 [4].

5 Conclusion

Related Work. There exists quite a variety of tools for supporting test generation. Symbolic evaluation and constraint solving are widely used, as well as model checkers or similar techniques. The LOFT tool performed test generation from algebraic specifications, essentially based on narrowing. TGV [11] 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 [3]. The FDR model-checker was used [13] for generating test cases from CSP specifications for a conformance relation similar to *ioco*. In Spec Explorer [16], the underlying semantic framework are abstract state machines (ASM) and the conformance relation is alternating refinement. The techniques are similar to those used for explicit model-checking. The ASM framework provides foundation to deal with arbitrarily complex states, but the symbolic extension, based on constraint solving, is still experimental. JavaPathFinder [17] 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 very strong tool in this line of white-box test systems using symbolic execution and model-checking is the Pex tool [15].

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 had not been 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. We have described the machine-checked formalization *CirTA* of the operational semantics and testing theory of *Circus*. Our experience has been developed for Isabelle/HOL, but could be reused for other HOL systems (like HOL4). Our formal reconstruction of the *Circus* theory lead to unforeseen simplifications of notions like channels and configurations, and, last but not least, to the concept of typing and binding inside the operational semantics rules, as well as the derived rules capturing the deductive construction of symbolic traces. In fact, since the original *Circus* theory is untyped, in a sense, Isabelle/*Circus* is an extension, and the question of the "faithfulness" of our semantic representation has to be raised. While a direct, formal "equivalence proof" between a machine-checked theory on the one hand and a mathematically rigorous paper-and-pencil development on the other is inherently impossible, nevertheless, we would argue that *CirTA* captures the *essence* of the *Circus* testing theory. Besides hands-on simulations in concrete examples, there is the entire architecture of similar definitions leading to closely related theorems and proofs that does establish a correspondence between these two. This correspondence would be further strengthened if we would complete the theory by a (perfectly feasible, but laborious) equivalence proof between the operational and denotational semantics (for the time being, such a proof does neither exist on paper nor in Isabelle). The correspondence could again be strengthened, if the existing paper-and-pencil proof of equivalence between the conformance relations and the refinement relation (given in [2]) could be reconstructed inside *CirTA*.

CirTA has been validated by a concrete case study. We developed, for a message monitoring module stemming from an industrial partner, an Isabelle/*Circus* model and derived tests for the real system. The component under test is embedded in not less than 5k lines of Java code. It binds together a variety of devices and especially patients pacemaker controllers, via sophisticated data structures and operations which was the main source of complexity when testing. More details about this case study can be found in [5, 7].

Isabelle/HOL is a mature theorem prover and easily supports our requirements for add-on tools for symbolic computation, but substantial efforts had to be invested for building our formal testing environment nonetheless. With regard to the experience of the last 10–20 years of the interactive theorem proving community, this initially steep ascend is in fact quite common, and we can anticipate eventual pay-off for more complex examples at the next stage. HOL as a logic opens a wide space of rich mathematical modeling, and Isabelle/HOL as a tool environment supports many mathematical domains by proof tools, say for simplification and constraint solving. Many of these Isabelle tools already incorporate other external proof tools, such as Z3. Thus we can benefit from this rich collection of formal reasoning tools for our particular application of *Circus* testing, and exploit the full potential of theorem prover technology for our work.

The Isabelle/HOL source code of Isabelle/*Circus* is already available in the Archive of Formal Proofs [1]. The source code of the *CirTA* environment will be distributed with the next release of HOL-TestGen.

Future Work. Besides the perspective to complete *CirTA* by the discussed equivalence theorems, our short term perspectives is to validate the environment on larger *Circus* specifications, and then integrate the *Circus* test generation framework with HOL-TestGen in order to benefit of its techniques for data-oriented case-splitting, test-driver generation and (on-the-fly) constraint-solving techniques. Moreover, we plan to study, develop and experiment with various test selection strategies and criteria for *Circus*.

Acknowledgment This research was partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n318353 (EURO-MILS project: http://www.euromils.eu).

References

[1] A. D. Brucker and B. Wolff. On Theorem Prover-based Testing. *Formal Aspects of Computing (FAOC)*, 2012.
[2] A. Cavalcanti and M.-C. Gaudel. Testing for refinement in circus. *Acta Inf.*, 48(2):97–147, Apr. 2011.
[3] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS 2002*, volume 2280 of *LNCS*. Springer, 2002.
[4] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS'08/ETAPS'08*, pages 337–340. Springer-Verlag, 2008.
[5] A. Feliachi. *Semantics-Based Testing for Circus*. PhD thesis, Université Paris-Sud 11, 2012.
[6] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/*Circus*: A process specification and verification environment. In *VSTTE*, volume 7152 of *LNCS*. Springer, 2012.
[7] A. Feliachi, M.-C. Gaudel, and B. Wolff. Exhaustive testing in hol-testgen/cirta – a case study. Technical Report 1562, LRI, July 2013.
[8] M.-C. Gaudel and P. L. Gall. Testing data types implementations from algebraic specifications. In Hierons et al. [9], pages 209–239.
[9] R. M. Hierons, J. P. Bowen, and M. Harman, editors. *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008.
[10] C. Hoare and J. He. *Unifying theories of programming*. Prentice Hall, 1998.
[11] C. Jard and T. Jéron. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *STTT*, 6, October 2004.
[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
[13] S. Nogueira, A. Sampaio, and A. Mota. Guided test generation from CSP models. In *ICTAC 2008*, volume 5160 of *LNCS*, pages 258–273, 2008.
[14] M. Oliveira, A. Cavalcanti, and J. Woodcock. A denotational semantics for Circus. *Electron. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
[15] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, Sept. 2005.

[1] `http://afp.sourceforge.net/entries/Circus.shtml`

[16] M. Veanes et al. Formal methods and testing. chapter Model-based testing of object-oriented reactive systems with spec explorer. Springer, 2008.
[17] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA 2004*, pages 97–107. ACM, 2004.
[18] J. Woodcock and A. Cavalcanti. The semantics of circus. In *ZB '02*, pages 184–203, London, UK, UK, 2002. Springer-Verlag.