Tactic Program-based Testing and Bounded Verification in Isabelle/HOL

Chantal Keller

Tactic Program-based Testing and Bounded Verification in Isabelle/HOL

Chantal Keller
Chantal.Keller@lri.fr

LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay, UMR8623
Orsay, F-91405, France

Abstract Program-based test-generation methods (also called "white-box" tests) are conventionally described in terms of a control flow graph and the generation of path conditions along the paths in this graph. In this paper, we present an alternative formalization based on state-exception monads that allows for direct derivations of path conditions from program presentations in them; the approach lends itself both for program-based testing procedures — designed to meet classical coverage criteria — and bounded verification. Our formalization is implemented in the Isabelle/HOL interactive theorem prover, where symbolic execution can be processed through tactics implementing test-generation strategies for various coverage criteria. The resulting environment is a major step towards testing support for the development of invariants and post-conditions in C verification environments similar to Isabelle/AutoCorres.

Keywords: White-box testing; Bounded verification; Symbolic execution; Coverage criteria; Interactive Theorem Proving

1 Introduction

In this paper, we present a range of program-based ("white-box") test generation methods inside an interactive theorem prover. Conventional implementations [3, 7, 8] convert the abstract-syntax tree of the source program into a control flow graph (CFG for short) defining a set of paths, giving rise to various path-coverage criteria. In contrast, we base our work on a shallow embedding of programs in the state-exception monad. This presentation can be seen as a minimalistic imperative core language tuned to program verification. As a side-effect, our test-generation procedure meeting different coverage criteria is implemented by a semantically neutral annotation process combined with tactical decomposition based on derived rules; it is therefore a verified tool by construction.

The contributions are the following. First, we propose to perform a symbolic execution of programs using the semantic rules of a state-exception monad. Compared to conventional presentations, it provides a lightweight environment for white-box test generation (around 1500 LOC, including proofs). Second, we embed the process into the Isabelle/HOL proof assistant, to offer:

– a formal verification of symbolic execution, *via* the correctness of the state-exception monad rules;
– reasonably efficient automatic engines for various coverage criteria, *via* Isabelle's support for term manipulation;
– bounded model checking, *via* the possibility to formally prove the validity of the abstract test cases.

Our paper proceeds as follows. After recalling the CFG-based approach for white-box testing on a running example, we detail our novel approach based on monads. We demonstrate the resulting symbolic execution rules by example, and explain their use both for bounded verification and for testing by injecting different forms of test-hypothesis. We conclude by tactics — implemented in the Isabelle/HOL interactive theorem prover — achieving various coverage criteria by construction, and illustrate the approach on a few examples.

All the material can be found at `https://www.lri.fr/~keller/TAP18`. For readability reasons, the definitions in the paper are written in Higher Order Logic (as defined by Church's Simple Type Theory [5]), expressed in a ML-like language with pre-defined symbols such as implication ($\Longrightarrow$), set comprehension ($\{\_ \mid \_\}$), ..., and a type constructor **theorem** that can be applied only to valid expressions (validity being proved interactively by Isabelle/HOL tactics). The notation

assumes H: P
shows Q

means **theorem** P $\Longrightarrow$ Q (giving the name H to hypothesis P).

2 The Classical Approach to White-box Testing

In order to contrast our approach to "the classical one", we will briefly present the latter using a running example: an algorithm for computing the integer square root of an integer. We use a vanilla imperative language in order to represent our example program:

```
1  int squareroot(int a):
2  -- pre : 0 ≤ a
3  -- post: result² ≤ a ∧ a < (result+1)²
4  { int tm = 1; int sqsum = 1; int i = 0;
5    while sqsum ≤ a {
6       i  := i + 1;
7       tm := tm + 2;
8       sqsum := tm + sqsum;
9    };
10   return(i)
11 }
```

The `return` command assigns its argument to the implicitly declared return value `result` that is in the variable scope of the post-condition.

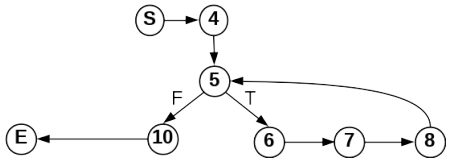

Figure 1. The CFG for the integer squareroot program

The algorithm computes the sum of odd numbers and exploits the well-known fact:

$$\sum_{x=0}^{i-1}(2x+1) = 1+3+5+\ldots+(2i-1) = i^2$$

Thus, the impairs are accumulated in the variable `sqsum` to a series of squares (`i`+1)^2. If `sqsum` becomes larger than the input `a`, `i` must be its integer squareroot.

In this article, we focus on program-based test methods, that make use of both the specification and the program itself to automatically build concrete tests in order to check if the program meets the specification on those tests, preferably given various coverage criteria.

The classical approach transforms our example program into a control-flow graph (CFG) as shown in Figure 1. Except for the start-node `S` and the end-node `E`, the nodes are labeled with corresponding program line numbers; these nodes represent the state-set that is reachable after a program point. These state-sets can be characterized by formulas associated to nodes; for example: $4 \mapsto \{\sigma | \sigma.tm = 1\}$, $5 \mapsto \{\sigma | \sigma.tm = 1 \wedge \sigma.sqsum = 1\}$, etc; the notation $\sigma.tm$ standing for "the value of tm in state σ".

The node labeled `7` is a *decision node* where the left outgoing arc represents the computations where the evaluation of the condition yielded false (`F`), while the right outgoing arc represents the evaluation to true (`T`) leading to one more traversal of the loop.

On the basis of a CFG, the notion of an execution path can be established: the path `[S,4,5,10,E]` is the path that does not traverse the loop, `[S,4,5,6,7,8,5,10,E]` the path that traverses the loop exactly once, etc. The formula ϕ_π characterizing the set of states that will lead to an execution along a path π is called a *path condition*; for the case $\pi =$ `[S,4,5,10,E]`, for example, ϕ_π is $\sigma.a = 0$. Path conditions can be constructed automatically by a symbolic execution process (a variant thereof will be presented later in the paper); however, a path condition can be unsatisfiable reflecting the fact that it does not necessarily represent a computation that is actually possible. This may happen due to conflicting conditions in decision nodes, for example. Whenever ϕ_π is unsatisfiable, π is called *infeasible*.

While the set of execution paths is infinite whenever the program contains unbounded loops, CFG's and the resulting notion of paths lend themselves naturally to *coverage criteria* which are fairly easy to understand and which found

their way into industrial applications and technical standards for software quality such as ISO 25119 [13]. For example a coverage criterion could constrain the set of all (if possible feasible) paths of a CFG to a path set covering all decision nodes (`allcond`), a path set covering all transitions (`alltrans`), or the set of paths that traverse all loops at most k times ($\texttt{allpath}_k$). In particular, variants based on $\texttt{allpath}_3$ are used in many industrial development processes since they have empirically shown a reasonably good compromise between cost and error detection capacity.

3 Instead of CFG's: Symbolic execution on Monads

3.1 Basic Definitions

We base this work on the Monad theory distributed with the HOL-TestGen testing framework [4]. This presentation is geared towards testing (see Section 8 for a comparison to other monad theories).

We define the monad-type as a transition function from a state of type σ to a successor state and some output of type 'o:

type ('o,' σ) MON_{SE} = 'σ →('o * 'σ) option

The composition operator on monads *bind* and the neutral element *unit* are standard:

let bind_{SE} : ('o,'σ) MON_{SE} →('o →('o,'σ) MON_{SE}) →('o,'σ) MON_{SE}
= **fun** f g → ($\lambda\sigma$. **case** f σ **of** None →None | Some (out, σ') → g out σ')

let unit_{SE} : 'o →('o,' σ) MON_{SE} = **fun** e →($\lambda\sigma$. Some(e,σ))

We will use alternative notations for the bind_{SE} combinator: the notation x ←f; g stands for bind_{SE} f (**fun** x →g) and f;– g for _ ←f; g . It is straightforward to prove the fundamental unit and associativity monad laws.

3.2 The Enriched Monad Infrastructure

As standard, in addition to these two basic blocks, it is convenient to declare common constructions used to define and manipulate monadic programs.

It is possible to add combinators for exception raising and handling and other usual programming constructs. We focus only on the conditional and the slightly more tricky case of loop definitions:

let **if**$_{SE}$: ('σ → bool) → ('o,' σ) MON_{SE} →('o,'σ) MON_{SE} →('o,'σ) MON_{SE}
= **fun** c E F → ($\lambda\sigma$. **if** c σ **then** E σ **else** F σ)

let while_{SE} : ('σ → bool) → (unit ,' σ) MON_{SE} →(unit,'σ) MON_{SE}
= **fun** c B → (lfp (Γ c B))"

In the definition of while_{SE}, lfp is a least-fixpoint operator, and Γ b cd is a state relation defined by:

```
let Γ b cd : 'σ ∗ 'σ
= fun cw →{(s,t) | if b s then (s,t) ∈ (cd O cw) else s = t}
```

where _ O _ is the relation composition (remind that {_ | _} is set comprehension). Informally, it means that the initial and final states of a loop are equal if the condition is false, and related by one more application of the loop body otherwise.

The proof of the unfold theorem:

```
theorem while_SE_unfold:
  (while_SE b do c od) = (if_SE b then c;−while_SE b do c od else return() fi)
```

is relatively complicated, but folklore (it is described in Winskell's book [18]; a first formal treatment in Isabelle we know of is [12]). Since our objective is to perform symbolic execution, the while operator can be decorated with a natural integer that limits the number of loops unrolling:

```
theorem while_n_unfold:
  (while[Suc n] b do c od) = (if_SE b then c;−while[n] b do c od else return() fi)
```

where while[n] is defined as while$_{SE}$ by ignoring the decoration n.

Note furthermore that we will embed assume$_{SE}$ and assert$_{SE}$ to model pre- and post-conditions, respectively. The construction assume$_{SE}$ puts a program in a initial state that satisfies some predicate P (if such a state exists), and assert$_{SE}$ checks if the final state of a program satisfies some predicate (by returning the program that succeeds if and only if its state satisfies P).

```
let assume_SE : ('σ →bool) → (unit ,' σ) MON_SE
= fun P →(λσ. if ∃σ. P σ then Some((), SOME σ. P σ) else None)

let assert_SE : ('σ → bool) → (bool ,' σ) MON_SE
= fun P →(λσ. if P σ then Some(True,σ) else None)
```

Here, the construction SOME σ. P σ returns a state σ that satisfies P (note that it is guarded by the fact that P is satisfiable).

3.3 Symbolic Execution Rules for the Monad

Instead of the syntax-based concept "execution path", we define the semantic concept of a *valid test-sequence* as a valid monad execution of a particular format: it consists of a series of monad computations $m_1 \ldots m_n$ applied to inputs $i_1 \ldots i_n$ and a post-condition P wrapped in a return depending on observed output. Validity is formally defined as follows:

```
let valid_SE : 'σ →(bool ,' σ) MON_SE →bool"
= fun σ m →(m σ ≠None) and fst(the (m σ))"
```

where the operator the is defined such that the (Some x) returns x (again, it is guarded by the fact that m $\sigma \neq$None). We will write $\sigma \models$m for valid$_{SE}$ σ m. Since each individual computation m_i may fail, the concept of a valid test-sequence corresponds to a feasible path in a non-deterministic

automaton, that leads to a state in which the observed output satisfies P. Using the notation introduced in Section 3.1, we will write an entire sequence as follows:

$$\sigma \models \mathsf{o}_1 \leftarrow \mathsf{m}_1\ \mathsf{i}_1;\ \ldots;\ \mathsf{o}_n \leftarrow \mathsf{m}_n\ \mathsf{i}_n;\ \mathsf{return}_{SE}\ (\mathsf{P}\ \mathsf{o}_1\ \ldots\ \mathsf{o}_n)$$

Note that since the m_i can be conditionals or a while loop, a sequence represents the stack of executions yet to be executed, and the o_j the intermediate results stored on way (if any).

The notion of a valid test-sequence has two facets. On the one hand, it is executable, i.e., a *program*, if and only if $\mathsf{m}_1, \ldots, \mathsf{m}_n, \mathsf{P}$ are. Thus, a code generator can map a valid test-sequence statement to code. In particular, in Isabelle/HOL, depending on the configuration, the code generator can map the calls to the m_i to Isabelle/HOL-defined operations or to external code, i.e.. some code to be tested. On the other hand, and this is a major strength of this monadic approach, valid test-sequences can be treated by a standard and simple family of symbolic executions rules, characterized by the following schema (for all monadic operations m of a system, which can be seen as its step-functions):

$$\sigma \models \mathsf{return}_{SE}\,\mathsf{P} = \mathsf{P} \tag{1}$$

$$\mathsf{m}\ \mathsf{i}\ \sigma = \mathsf{None} \Longrightarrow (\sigma \models \mathsf{s} \leftarrow \mathsf{m}\ \mathsf{i};\mathsf{m'}\ \mathsf{s}) = \mathit{False} \tag{2}$$

$$\mathsf{m}\ \mathsf{i}\ \sigma = \mathsf{Some}(\mathsf{b},\sigma') \Longrightarrow (\sigma \models \mathsf{s} \leftarrow \mathsf{m}\ \mathsf{i};\mathsf{m'}\ \mathsf{s}) = (\sigma' \models \mathsf{m'}\ \mathsf{b}) \tag{3}$$

$$(\sigma \models (\ \mathbf{if}_{SE}\mathsf{b}\ \mathbf{then}\ \mathsf{c}\ \mathbf{else}\ \mathsf{d}\ \mathsf{fi});{-}\mathsf{m}) = (b\ \sigma \wedge \sigma \models \mathsf{c};{-}\mathsf{m}) \vee (\neg b\ \sigma \wedge \sigma \models \mathsf{d};{-}\mathsf{m}) \tag{4}$$

$$\begin{array}{r}(\sigma \models \mathsf{while}\,[\mathsf{Suc\ n}]\ \mathsf{b}\ \mathsf{do}\ \mathsf{c}\ \mathsf{od}\ ;{-}\ \mathsf{m}) = \\ (\sigma \models (\ \mathbf{if}_{SE}\mathsf{b}\ \mathbf{then}\ \mathsf{c};{-}\mathsf{while}[\mathsf{n}]\ \mathsf{b}\ \mathsf{do}\ \mathsf{c}\ \mathsf{od}\ \mathbf{else}\ \mathsf{return}_{SE}()\mathsf{fi});{-}\mathsf{m})\end{array} \tag{5}$$

$$(\sigma \models \mathsf{assume}_{SE}\mathsf{P}\ ;{-}\ \mathsf{m}) = (\forall\sigma' \in \{\sigma' | \mathsf{P}\ \sigma'\}.(\sigma' \models \mathsf{m})) \tag{6}$$

$$(\sigma \models \mathsf{assert}_{SE}\mathsf{P}\ ;{-}\ \mathsf{m}) = (\mathsf{P}\ \sigma \wedge (\sigma \models \mathsf{m})) \tag{7}$$

This kind of rules is usually specialized for concrete operations m; if they contain pre-conditions C_{m} (constraints on i and state), or conditions, this calculus will just accumulate them and construct a constraint system to be treated by a solver (see next section for an example).

A technical improvement specific to Isabelle is to use its meta-logic (which is based on an intuitionistic fragment of Higher Order Logic) instead of Isabelle/HOL connectives. It gives better performance compared to rewriting the rules presented above. For example, the conditional rule is expressed as case-splitting as follows, similar to a disjunction elimination rule:

$$\frac{\sigma \models \mathbf{if}_{SE}\mathsf{b}\ \mathbf{then}\ \mathsf{c}\ \mathbf{else}\ \mathsf{d}\ \mathsf{fi}\ ;{-}\ \mathsf{m} \qquad \begin{array}{c}[\sigma \models \mathsf{c}\ ;{-}\ \mathsf{m},\ \mathsf{b}\ \sigma] \\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[\sigma \models \mathsf{d}\ ;{-}\ \mathsf{m},\ \neg\mathsf{b}\ \sigma] \\ \vdots \\ Q\end{array}}{Q} \tag{8}$$

4 Representing Programs and Symbolic Execution

We are ready to undertake the final steps to actually represent imperative programs as a symbolic evaluation problem (that will be used in bounded verification and testing scenarios later).

We will introduce a notation for the assignment, which is modeled to never fail in our core language:

```
let assign : ('σ → 'σ) → (unit,'σ) MON_SE = fun f σ→Some ((), f σ)
```

for which we derive the desired destruction rule:

$$\sigma \models \mathsf{assign}\ \mathsf{f}\ ;\!-\ \mathsf{m} = \mathsf{f}\,\sigma \models \mathsf{m}$$

With respect to the representation of state, we follow the idea of Isabelle/SIMPL [14] to reuse records where the record fields represent the program variables. For our running example, this means that we define the state as:

```
type state = {tm   : int,
              i    : int,
              sqsum : int,
              a    : int}
```

Note that the variable a could also be modeled as a parameter not represented in the state since it is not modified. As standard, from this record, one can generate the accessor functions a, sqsum, i and tm; update operations like $\sigma(\!|\mathsf{tm} := \mathsf{E}|\!)$; and a memory-theory with rules like $\mathsf{tm}(\sigma(\!|\mathsf{tm} := \mathsf{E}|\!)) = \mathsf{E}$ and $\mathsf{sqsum}(\sigma(\!|\mathsf{tm} := \mathsf{E}|\!)) = \mathsf{sqsum}\ \sigma$. (In Isabelle, they are all generated automatically.) As standard, we extend the notation for updates to chains of updates such as

$$\sigma(\!|\mathsf{tm}{:=}1,\ \mathsf{sqsum}{:=}1,\ \mathsf{i}{:=}0|\!)$$

where the rightmost "wins" when applied to the same record field. Note that the types of record fields can be arbitrary HOL types; here, we profit largely from our compact shallow embedding representation. Moreover, other memory-models could be used as well.

The right-hand side of assignments, assertions, and conditions in $\mathbf{if}_{SE}$ and while_{SE} are represented as state-transition functions or as state-to-bool predicates. However, we will use notations such as $\langle \mathsf{sqsum} \leq \mathsf{a} \rangle$ for $\mathbf{fun}\ \sigma \rightarrow (\mathsf{sqsum}\ \sigma) \leq (\mathsf{a}\ \sigma)$, i.e. $\langle \_ \rangle$ represents a parser that applies any record field name to a bound variable (for the state) that is λ-abstracted at the topmost level. Similarly, $\langle \mathsf{i} := \mathsf{i} + 1 \rangle$ represents $\lambda\sigma.\ \sigma(\!|\ \mathsf{i} := (\mathsf{i}\ \sigma\ + 1)|\!)$.

To semi-interactively perform symbolic execution of our programs under test, we state the pre- and post-conditions as Isabelle theorems (which will allow us to apply manually or systematically the correctness rules of the state-exception monad, as explained in the remaining of the article). Thus, we can represent our squareroot example program in the following format:

```
assumes annotated_program:
        σ 0 ⊨assume_SE ⟨0 ≤ a⟩ ;−
```

$\langle \mathsf{tm} := 1\rangle$;–
$\langle \mathsf{sqsum} := 1\rangle$;–
$\langle \mathsf{i} := 0\rangle$;–
$(\mathsf{while}_{SE}\ \langle \mathsf{sqsum} \leq \mathsf{a}\rangle\ \mathsf{do}$
$\langle \mathsf{i} := \mathsf{i} + 1\rangle$;–
$\langle \mathsf{tm} := \mathsf{tm} + 2\rangle$;–
$\langle \mathsf{sqsum} := \mathsf{tm} + \mathsf{sqsum}\rangle$
od) ;–
$\mathsf{assert}_{SE}(\lambda\sigma.\ \sigma = \sigma_R)$
shows $\sigma_R \models \mathsf{assert}_{SE}\langle\, \mathsf{i}^2 \leq \mathsf{a} \wedge \mathsf{a} < (\mathsf{i}+1)^2\rangle$

Note that σ_R is a free variable in this goal denoting the "result state" after the execution of our squareroot program; it is the purpose of the entire assumption to construct this state (symbolically). However, in the conclusion, we require that the post-condition is to hold in this state which expresses our verification notion.

We will run a little simulation of our rule set in order to show how everything fits together; we will automate the entire process in subsequent sections, targeting different objectives.

Assume that we want to explore the program up to all paths of the depth 3. Then we rewrite while_{SE} by $\mathsf{while}\,[\mathsf{Suc}(\mathsf{Suc}(\mathsf{Suc}\ 0)))]$ and apply subsequently the destruction of assume_{SE} (rule 6) and repeatedly the destruction of assign (rule 7). This transforms our goal into:

$\forall\sigma.\ 0 \leq \mathsf{a}\ \sigma \implies$
$\sigma(\!|\mathsf{tm} := 1, \mathsf{sqsum} := 1, \mathsf{i} := 0|\!) \models$
$(\mathsf{while}\,[\mathsf{Suc}(\mathsf{Suc}(\mathsf{Suc}\ 0)))]\ \langle \mathsf{sqsum} \leq \mathsf{a}\rangle\ \mathsf{do}$
$\langle \mathsf{i} := \mathsf{i} + 1\rangle$;–
$\langle \mathsf{tm} := \mathsf{tm} + 2\rangle$;–
$\langle \mathsf{sqsum} := \mathsf{tm} + \mathsf{sqsum}\rangle$
od) ;–
$\mathsf{assert}_{SE}(\lambda\sigma.\ \sigma = \sigma_R)$

Further repetitive applications of destruction of $\mathsf{while}\,[\_]$ and $\mathbf{if}_{SE}$ (rule 4, rule 5) as well as assign (rule 7) leave us basically with a proof state of the following form:

1. $\forall\sigma.\ 9 \leq \mathsf{a}\ \sigma \implies$
$\sigma(\!|\mathsf{i} := 3,\ \mathsf{tm} := 7,\ \mathsf{sqsum} := 16|\!) \models$
$(\mathsf{while}\,[0]\ \langle \mathsf{sqsum} \leq \mathsf{a}\rangle\ \mathsf{do}$
$\langle \mathsf{i} := \mathsf{i}+1\rangle$;–
$\langle \mathsf{tm} := \mathsf{tm}+2\rangle$;–
$\langle \mathsf{sqsum} := \mathsf{tm} + \mathsf{sqsum}\rangle$ od) ;–
$\mathsf{assert}_{SE}\ (\lambda\sigma.\ \sigma = \sigma_R) \implies$
$\sigma_R \models \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \leq \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}+1)^2\rangle$
2. $\forall\sigma.\ 4 \leq \mathsf{a}\ \sigma \implies \neg\, 9 \leq \mathsf{a}\ \sigma \implies$
$\sigma_R = \sigma(\!|\mathsf{i}:=2,\ \mathsf{tm}:=5,\ \mathsf{sqsum}:=9|\!) \implies$
$\sigma(\!|\mathsf{i}:=2,\ \mathsf{tm}:=5,\ \mathsf{sqsum}:=9|\!)$
$\models \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \leq \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}+1)^2\rangle$
3. $\forall\sigma.\ 1 \leq \mathsf{a}\ \sigma \implies \neg\, 4 \leq \mathsf{a}\ \sigma \implies$

$\sigma_R = \sigma(\!|\mathsf{i} := 1,\ \mathsf{tm} := 3,\ \mathsf{sqsum} := 4|\!) \Longrightarrow$
$\sigma\,(\!|\mathsf{i}{:=}1,\ \mathsf{tm}{:=}3,\ \mathsf{sqsum}{:=}4|\!)$
$\models \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \leq\ \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2 \rangle$

4. $\forall \sigma.\ 0 \leq \mathsf{a}\ \sigma \Longrightarrow \neg\, 1 \leq \mathsf{a}\ \sigma \Longrightarrow$
$\sigma_R = \sigma(\!|\mathsf{tm}{:=}1,\ \mathsf{sqsum}{:=}1,\ \mathsf{i}{:=}0|\!) \Longrightarrow$
$\sigma\,(\!|\mathsf{tm}{:=}1,\ \mathsf{sqsum}{:=}1,\ \mathsf{i}{:=}0|\!)$
$\models \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \leq\ \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2 \rangle$

This proof state contains now by construction:

- in the last sub-goal, the path condition for never entering the loop (essentially $\mathsf{a}\ \sigma\ = 0$),
- in the third sub-goal, the path condition for entering the loop exactly once ($1 \leq\ \mathsf{a}\ \sigma < 4$),
- in the second sub-goal, the path condition for entering the loop twice ($4 \leq\ \mathsf{a}\ \sigma < 9$),
- in the first sub-goal, the path condition for traversing the loop more than twice ($9 \leq\ \mathsf{a}\ \sigma$).

In the first sub-goal, the remaining $\mathsf{while}\,[0]$ represents the class of all possible remaining executions; therefore, for this class no elimination of the σ_R can be achieved via application of the one-point-rule.

Note that the decoration on $\mathsf{while}\,[\_]$ allows one to unroll nested loops at different depths.

In the remaining of the paper, sub-goals containing assumptions of the form: $\sigma\ \models(\mathsf{while}\,[0]\ \ ...\ \ \mathsf{do}\ ...\ ;{-}\ ...\ )$ will be called *incomplete*, and the others will be called *complete*. Obviously, the latter represent execution paths through the program where the resulting equation $\sigma_R = \mathsf{E}(\sigma)$ bounds σ_R to the result of the symbolic execution.

5 Verification vs. Testing

At this stage, we have obtained the result of symbolic executions up to a certain depth. Once again, the embedding into the Isabelle/HOL proof assistant offers a lightweight framework to perform bounded verification or testing, or a combination of the two.

To this end, the ideas of the Isabelle/HOL-TestGen system [4] can be reused. HOL-TestGen exploits the concept that two types of hypotheses are used to express the differences between a proof of a property P and its test [9]:

- the *uniformity hypothesis* assumes that if a test passes for one instance of a partition of the input-output relation of a program specification P, then P will hold for the whole partition, and
- the *regularity hypothesis* assumes that if a test passes with sufficiently "deep" or "complex" input data for P, then P will always hold.

HOL-TestGen generates from test specifications of the format:

$$\text{pre}\ x \rightarrow \text{post}\ x\ (\text{PUT}\ x)$$

for the (black-box) program under test PUT, which is logically just an uninterpreted constant, a partitioning of the input-output relation, and *explicit* test hypotheses which were added to test-property on the fly as a consequence of the targeted test-criterion. In a last step, HOL-TestGen generates test-drivers — basically test-oracles from post-conditions — that are linked to the actual code of PUT. Note again that it is not necessary to construct an invariant in a white-box testing approach, the precondition for filtering illegal input and the post-condition for generating oracles suffices.

In this section, we explain how these ideas apply in our setting. For the engineering part consisting in transforming goals into test hypotheses and test cases, we refer the reader to the description of the HOL-TestGen system [4].

5.1 Strategy: Bounded Verification (also called Bounded Model-Checking)

If we adopt this overall concept to white-box testing, we need only to find an equivalent to the regularity hypothesis. The strategy for bounded verification consists of:

- attempting to prove the *complete goals* automatically. For the case 2) in squareroot, for example, this boils down to proving:

$$\begin{array}{l} 4 \leq \mathsf{a}\ \sigma \implies \neg\ 9 \leq \mathsf{a}\ \sigma \implies \\ \sigma\ (\!|\, \mathsf{i}{:=}2,\ \mathsf{tm}{:=}5,\ \mathsf{sqsum}{:=}9 \,|\!) \models \\ \qquad\qquad \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \leq\ \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2 \rangle \end{array}$$

  which falls into a fragment decided by many automated provers. (In Isabelle, this goals is automatically discharged by the auto command.)
- admitting the *incomplete goals*. They would require an invariant for their proofs. Rather than attempting to prove them, we turn them into an explicit test hypothesis of the form:

$$\begin{array}{l} \mathsf{THYP}(\forall \sigma.\ 9 \leq \mathsf{a}\ \sigma\ \rightarrow \\ \qquad \sigma\ (\!|\, \mathsf{i}\ :=\ 3,\ \mathsf{tm}\ :=\ 7,\ \mathsf{sqsum}\ :=\ 16 \,|\!) \models \\ \qquad (\mathsf{while}_{SE}\ \langle \mathsf{sqsum} \leq\ \mathsf{a} \rangle\ \mathsf{do} \\ \qquad\quad \langle \mathsf{i}\ :=\ \mathsf{i}{+}1 \rangle\ ;{-} \\ \qquad\quad \langle \mathsf{tm}\ :=\ \mathsf{tm}{+}2 \rangle\ ;{-} \\ \qquad\quad \langle \mathsf{sqsum}\ :=\ \mathsf{tm}\ +\ \mathsf{sqsum} \rangle\ \mathsf{od})\ ;{-} \\ \qquad \mathsf{assert}_{SE}\ (\ \langle \mathsf{i}^2 \leq\ \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2 \rangle)) \end{array}$$

  where $\mathsf{THYP}(\mathsf{x}) \equiv \mathsf{x}$ just serves as a semantically neutral marker to control the tactic process. This form of explicit test-hypothesis states "beyond our analysis depth, we assume that the program is correct" and represents a regularity hypothesis adapted to program-based testing.

Adding the explicit regularity hypothesis to the assumptions of the original goal (thus weakening it logically) allows for a formal proof of the modified goal making explicit under which assumptions our program (model) satisfies the specification.

5.2 Strategy: White-box Testing

Testing differs from bounded verification in basically two ways: first, we use additionally the uniformity hypothesis, stating that for each partition of the input-output relation (i.e. the path conditions), we assume that the program is correct provided that we found one instance in this partition where it behaves correctly (in ISO 25119 [13], this assumption is used for what is called "equivalence class testing"). Second, the concrete instance of a partition, called concrete test-case and usually constructed by a constraint solver, can be converted into test driver code that is run against the real program, not just a model of it. This turns testing into a validation method that covers also hardware, the underlying operating system, the compiler, etc, of the program under test. Therefore, evaluators in formal evaluation schemes like Common Criteria insist on tests *validating* a (code) model against "the real thing"; verifications based on immanent arguments over models are acceptable as complements, but not as a complete replacement of tests.

In our running example, the uniformity test-hypothesis will be, for example:

$$\begin{aligned}&\mathsf{THYP}((\exists\ \sigma.\ 4 \le \mathsf{a}\ \sigma \wedge \neg\, 9 \le \mathsf{a}\ \sigma\ \wedge\\ &\qquad \sigma\ \models\ \mathsf{squareroot}\ ;-\ \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \le \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2\rangle) \rightarrow\\ &\qquad (\forall\ \sigma.\ 4 \le \mathsf{a}\ \sigma \Longrightarrow \neg\, 9 \le \mathsf{a}\ \sigma \longrightarrow\\ &\qquad \sigma\ \models\ \mathsf{squareroot}\ ;-\ \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \le \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2\rangle))\end{aligned}$$

where squareroot is an abbreviation for our program code.

A constraint solver might find the solution $\mathsf{a}\ \sigma = 7$ for the path-condition above, thus permitting us to construct automatically from the uniformity hypothesis the concrete test:

$$\sigma\,(\!|\mathsf{a} := 7|\!) \models \mathsf{squareroot}\ ;-\ \mathsf{assert}_{SE}\ \langle \mathsf{i}^2 \le \mathsf{a} \wedge\ \mathsf{a} < (\mathsf{i}{+}1)^2\rangle$$

6 Support for Coverage Criteria

In the end of Section 4, we detailed how a manual application of the rules of the state-exception monad (via Isabelle basic tactics) perform step-by-step symbolic execution. In this section, we explain how to build new tactics that automate the process, with support for various coverage criteria.

These tactics are based on the following observations. Repeated applications of the rule for symbolic execution of $\mathbf{if}_{SE}$ (rule 4) guarantee *branch coverage*, since each branch of each control structure is covered. In our small imperative language, this is equivalent to *decision coverage*, since we do not handle function calls yet [15]. Similarly, repeated applications of the rule for while_{SE} loops (rule 5) immediately followed by the rule for $\mathbf{if}_{SE}$ guarantee loop coverage up to a certain depth.

The Isabelle tactical language, called Eisbach [11], allows us to design various tactics that apply and combine the symbolic execution rules, together with simplifications of the goal:

– the tactic branch_and_loop_coverage simply relies on the two coverage criteria described above;
– the tactic mcdc_and_loop_coverage covers more: it also performs *Modified Condition/Decision Coverage* (MC/DC for short), meaning that each condition (i.e. Boolean sub-expression) appearing in a decision affects the decision outcome independently;
– conversely, the tactic loop_coverage_positive_branch covers less: it performs loop coverage but, for branches, always chooses the first branch (other choices, such as random, could as well be implemented). This may be useful if one wants to explore loop unrolling without a combinatorial explosion (see e.g. Section 7.1).

For instance, applied to the annotated_program of Sec. 4, the tactic

```
apply (branch_and_loop_coverage "Suc (Suc (Suc 0))")
```

directly leads to the proof state presented in the end of the section.

The Eisbach tactical language defines new tactics by combining existing ones using the syntax of regular expression. For instance, the first two tactics are programmed from a single parametric method defined as follows:

```
method loop_coverage for n::nat methods simp_mid simp_end =
  (bound_while n)?, loop_coverage_steps simp_mid, simp_end?
```

The syntax means the following. loop_coverage is the name of the tactic, parameterized by a natural number n and two other tactics named simp_mid and simp_end. This tactic sequentially performs:

1. (bound_while n)?: every occurrence (if there is any, represented by the ?) of while_{SE} is replaced by while[n], justified by the definition of while[n].
2. loop_coverage_steps simp_mid: the equations of Sec. 4 are repeatedly applied: the auxiliary tactic loop_coverage_steps is a simple loop that, as much as possible, applies those equations, and simplifies intermediate results by the (abstract) tactic simp_mid. Note that, at each step, at most one equation can be applied, depending on the first instruction remaining in the program.
3. simp_end?: if possible, the result is simplified by the (abstract) tactic simp_end.

The choice of the simplifications leads to the first two variants.

– In the case of branch coverage, only basic simplifications are performed:

```
method branch_and_loop_coverage for n::nat =
                loop_coverage n memory_simp simp_all
```

where memory_simp is the memory-theory presented in Sec. 4 and simp_all is the standard Isabelle simplifier.
– As for MC/DC, simplifications should also split conjunctive and disjunctive hypotheses according to their elimination rules. To this end, we can use the Isabelle auto tactic[1]:

[1] A specific tactic that only calls the simplifier and applies elimination rules of connectives would work as well and be less powerful.

```
method mcdc_and_loop_coverage for n::nat = loop_coverage n auto auto
```

The third tactic loop_coverage_positive_branch is similar but the rule for $\mathbf{if}_{SE}$ is applied only after loop unrollings. Other branches are symbolically executed using a weaker rule:

$$(\sigma \models (\ \mathbf{if}_{SE} \mathsf{b}\ \mathbf{then}\ \mathsf{c}\ \mathbf{else}\ \mathsf{d}\ \mathsf{fi});{-}\mathsf{m}) = (b\ \sigma \wedge \sigma \models \mathsf{c};{-}\mathsf{m}) \vee (\mathsf{opaque}(\neg b\ \sigma \wedge \sigma \models \mathsf{d};{-}\mathsf{m}))$$

where opaque is a tag that forbids further application of rules.

These prototype tactics are already reasonably efficient: it takes less than 30s to unroll the loop 100 times with MC/DC on our running example, searching counter-examples up to 10000.

7 Examples

This section illustrates the flexibility and expressivity of the monadic approach (Section 7.1) and the tactics we just presented (subsection 7.1 and 7.2). These examples and more have been implemented in Isabelle/HOL and are fully automated using the tactics; they can be found in the online material.

7.1 Maximum of an array

Symbolic execution of programs manipulating usual data types can be faithfully performed using standard functional representation. We give here the example of a function computing the maximum of an array: the array can be represented as a function whose domain is non-negative integers together with a length. The state thus contains

```
type state = {arr : nat → int,   (* The array is represented as a function ... *),
              l   : nat,         (* ... and a length *)
              i   : nat,         (* The loop index *)
              res : int          (* The result *)
             }
```

and the program is the usual one:

```
assumes annotated_program:
        σ 0 ⊨assume_SE ⟨1 ≤l⟩ ;−
              ⟨res := arr 0⟩ ;−
              ⟨i := 1⟩ ;−
              (while_SE ⟨i < l⟩ do
                 ( if_SE ⟨res < arr i⟩ then ⟨res := arr i⟩ else skip_SE fi) ;−
                 ⟨i := i + 1⟩
              od) ;−
              assert_SE(λσ. σ=σ_R)
shows σ_R ⊨assert_SE ⟨(∀ k < l. res ≥arr k) ∧(∃ k < l. res = arr k)⟩
```

The post-condition is also standard, stating that the result is greater or equal than all the elements of the array, and equal to one of them, under the pre-condition that the length is at least 1. Such a property is well-suited for testing or bounded checking.

On this example, full bounded symbolic execution presented in this article (and performed by the branch_and_loop_coverage and mcdc_and_loop_coverage tactics) unrolls the loop a given amount of time and explores both branches inside the loop, leading to an exponential blow-up where the maximum could be anywhere in the array. Under the regularity hypothesis though, it may be sufficient to test only one or a few possible cases for the maximum at each length, which is obtained by the tactic loop_coverage_positive_branch (or any variant that executes only one branch at each step). We refer the reader to the online material for an executable comparison.

In any case, this example actually generates abstract test cases that precisely determinate the position of the maximum of the array.

Other standard data-structures can be modeled such as hash tables, or lists (using Isabelle/HOL lists).

7.2 Median of three integers

To illustrate MC/DC vs. branch coverage, we take the example of a program computing the median of three integers:

$$
\begin{aligned}
&\textbf{assumes}\ \mathsf{annotated\_program}:\\
&\qquad \sigma_0 \models (\textbf{if}_{SE}\ \langle(b \le a \wedge a \le c) \vee (c \le a \wedge a \le b)\rangle\ \textbf{then}\ \langle res := a\rangle\ \textbf{else}\\
&\qquad\qquad (\ \textbf{if}_{SE}\ \langle(a \le b \wedge b \le c) \vee (c \le b \wedge b \le a)\rangle\ \textbf{then}\ \langle res := b\rangle\ \textbf{else}\\
&\qquad\qquad \langle res := c\rangle\ \ \mathsf{fi}\ )\ \ \mathsf{fi}\ )\ ;-\\
&\qquad\qquad \mathsf{assert}_{SE}(\lambda\sigma.\ \sigma{=}\sigma_R)\\
&\textbf{shows}\ \sigma_R \models \mathsf{assert}_{SE}\ \langle(res = a \vee res = b \vee res = c)\\
&\qquad\qquad\qquad\qquad \wedge\ (res > a \longrightarrow res \le b \wedge res \le c)\rangle
\end{aligned}
$$

The post-condition only consider one case but can be completed by adding the second conjunct for every permutations of a, b and c.

The branch_and_loop_coverage tactic generates only three abstract test cases whose premises are:

- $(b \le a \wedge a \le c) \vee (c \le a \wedge a \le b)$
- $(a \le b \wedge b \le c) \vee (c \le b \wedge b \le c)$
- the negation of the first two

whereas, as required, the mcdc_and_loop_coverage tactic generates six abstract test cases corresponding to the possible modified conditions, of the shape $(b \le a \wedge a \le c)$ for every permutation of a, b and c.

8 Related Work

Using monads is a standard technique for representing stateful computations. Leaving aside existing implementations in HOL4 and Coq, the Is-

abelle/HOL library alone defines a standardized monad-syntax for both a non-deterministic (Kleisli-like) and a deterministic monad. Other libraries include Isabelle/Simpl [14], Isabelle/ORCA [2] and Isabelle/AutoCorres [10]. These infrastructures are geared towards program-proofs or program refinement proofs, include bool's to capture termination, and, in the case of Imperative_HOL, a very specific heap-memory model used in the AutoCorres Tool for verifying C Programs. Besides having improved syntax support, the used monad here is geared to pure partial program semantics and optimized forms of partial evaluation therein. In particular, it is agnostic to a particular memory model. The presented loop-unfold theorem is not available in neither of mentioned monad theories.

Generating tests by counterexample generators is an active research area. There are basically two approaches. One is to take an input formula, try to construct a family of finite models, usually by bit-blasting into SAT problems, and to construct a counter-example on this basis [1, 16]. The other one is to interpret the input formula as a filter, i.e. to compile it to program for a Boolean function, and stimulate it by random values until a hit is found. This concept going back to [6] is known as QuickCheck and leads to a wealth of implementations for various languages meanwhile. Both approaches suffer from their generality when it comes to the generation of counterexamples for *programs* with pre-conditions. Moreover, they cannot compete with white-box testers with respect to the depth of program exploration as well as the coverage of given criteria imposed by standards such as [13].

As currently most developed white-box testers we mention Pex [7] and PathCrawler [17]. They present direct algorithmic implementations working on CFG's and scale well for realistic sizes of programs. In contrast, our approach is based on a shallow representation of a semantics and derived rules. It enjoys the following two advantages:

- the code is very small (around a 1500 LOC, including proofs of correctness) but first experiments show that it is reasonably efficient (Section 6);
- our implementation is based on derived rules and can therefore guarantee correctness.

Regarding expressivity, the programming language handled by our approach is Turing complete; we leave for future work to handle convenient paradigms such as local states (e.g. by replacing the memory model by a stack), function calls and recursion.

The aforementioned state-of-the-art testers have been combined with techniques for borderline analysis, regular expression constraint-solvers and test-execution environments supporting virtual system calls. The approach presented in the paper is being integrated in the HOL-TestGen[2] [4] framework, to make use of its concrete test generator and code extraction. In addition, it explicitly constructs the test hypotheses.

[2] See `https://www.brucker.ch/projects/hol-testgen` for more details, in particular the TestSequence theory.

9 Conclusion

We have shown an approach to model white-box testing of block-structured imperative programs. We used a shallow, monad-style embedding of the language. We believe this allows for a particularly concise and elegant formalization of the symbolic execution process, which is traditionally described on a control-flow graph: the trick is done by just eight rules with little deductive cost (first-order matching). We have shown that the process can be easily wrapped up in a tactic process.

The approach lends itself to precisely study the borderlines between deductive verification, bounded verification and testing in a uniform setting. By re-using HOL-TestGen's concept of explicit test-hypothesis, the approach allows us to establish a precise link between test and proof.

It was not our objective to develop in this paper a full-blown tool (for that, we would have to integrate it into, say, Isabelle/SIMPL which necessitates to cope with much more features and machinery). Still, the shown experiments indicate that our approach does scale fairly well. Therefore, we believe that our technique has the potential for a tool that effectively tests pre- and post-conditions as well as invariants for realistic program verification attempts.

Acknowledgments The author would like to thank Burkhart Wolff for setting up the foundations of this work. She also thanks the anonymous reviewers for valuable and detailed comments on how to improve the article.

Bibliography

[1] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[2] J. A. Bockenek. *An Extension of Isabelle/UTP with Simpl-like Control Flow*. PhD thesis, Virginia Polytechnic Institute and State University, 2017.

[3] B. Botella, M. Delahaye, S. H. T. Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with pathcrawler. In *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada, May 18-19, 2009.*, pages 70–78, 2009.

[4] A. D. Brucker and B. Wolff. On Theorem Prover-based Testing. *Formal Asp. Comput. (FAOC)*, 25(5):683–721, September 2013.

[5] A. Church. A set of pustulates for the foundation of logic (1). *Annals of Mathematics*, 1932.

[6] K. Claessen and J. Hughes. Testing monadic code with quickcheck. *SIGPLAN Not.*, 37(12):47–59, Dec. 2002.

[7] J. de Halleux and N. Tillmann. Parameterized unit testing with pex. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer, 2008.

[8] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder: A brief summary. In *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Deutschland*, pages 39–40, 2014.

[9] M. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[10] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM.

[11] D. Matichuk, M. Wenzel, and T. Murray. The Eisbach user manual. *Isabelle Community*, 2015.

[12] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[13] W. G. . W. of the ISO/IEC JTC1/SC7 Software and S. E. committee. ISO/IEC/IEEE 29119 Software Testing: The international standard for software testing., 2007-2014.

[14] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[15] F. Team et al. What is a "decision" in application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC). *Technical Report position paper*, 2002.

[16] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[17] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[18] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.