Interactive Testing with HOL-TestGen

Achim D. Brucker and Burkhart Wolff

Information Security, ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland.
{brucker, bwolff}@inf.ethz.ch

Abstract HOL-TestGen is a test environment for specification-based unit testing build upon the proof assistant Isabelle/HOL. While there is considerable skepticism with regard to interactive theorem provers in testing communities, we argue that they are a natural choice for (automated) symbolic computations underlying systematic tests. This holds in particular for the development on non-trivial formal test plans of complex software, where some parts of the overall activity require inherently guidance by a test engineer. In this paper, we present the underlying methods for both black box and white box testing in interactive unit test scenarios. HOL-TestGen can also be understood as a unifying technical and conceptual framework for presenting and investigating the variety of unit test techniques in a logically consistent way.
Keywords: symbolic test case generations, black box testing, white box testing, theorem proving, interactive testing

1 Introduction

HOL-TestGen [1, 5, 6] is a test environment for unit testing based on the proof assistant Isabelle/HOL. Its design rationale is remarkably different from the mainstream of other symbolic testing tools which are designed to be fully automatic: In our view, the development of tests is an *interactive* activity, where the form of test specifications, the abstraction levels used in a test, the solution of generated logical constraints (for path-conditions, etc.), and the parameters of the test data selection must be experimented with and adopted up to the point where the generated tests are sufficiently "good" with respect to an underlying test adequacy criteria.

Aiming at a fully automatic tool for specification-based test has a number of consequences: be it for the specification language and for the degree of abstraction of test specifications, be it for the theories of the underlying data structures, and be it, last but not least, on the way how not automatically resolvable logical constraints of a test are finally treated. It may be the case that most of the generated constraints can (and, of course, should!) be solved by an automated theorem prover or constraint solver, however, what happens to the remaining rest? Raising this question and putting the interactive element from the periphery into the center leads in our experience to different answers with respect to the usable specification language, to the design of the system architecture as well as to the testing methodology.

W. Grieskamp and C. Weise (Eds.): FATES 2005, LNCS 3997, pp. 87–102, 2006.
© 2006 Springer-Verlag. This is the author's version of the work. It is posted at http://www.brucker.ch/bibliography/abstract/brucker.ea-interactive-2005 by permission of Springer-Verlag for your personal use. The definitive version was published with doi: 10.1007/11759744_7.

This paper focuses on the latter issue. HOL-TestGen has been used to find non-trivial bugs in "real" software based on highly automated symbolic computation processes [6], where theory and implementation are described. But what is the underlying methodology leading to this result? And are there other ways to profit from the inactive potentials of HOL-TestGen? We answer these questions in the main sections of this paper: in Sec. 4, we describe the "best practices" developed in previous specification-based black box tests. In particular, we show how the highly automated standard workflow for generating test data can be enhanced by mixing it with more or less ingenious intermediate theorem proving steps. In Sec. 5, we will exploit the underlying generality of Isabelle for a different testing technique in the style of Pathfinder [11], SpecExplorer [8], and Korat [4]. The approach is based on a suitable semantic presentation of a programming language (a "logical embedding"), which can be used to both derive semantic constraints underlying a test as well as solving them in an integrated way. Moreover, our approach allows for logging explicit test hypotheses, a concept developed by the authors [6].

We would like to emphasize that all our symbolic computations are based entirely on conservative theory extensions of HOL and derived rules from them, such that HOL-TestGen is in fact a proven correct tool (assuming the consistency of HOL and its correct implementation in Isabelle). We believe that proving the crucial rules helps to develop a simple, semantically clean and integrated support of testing techniques.

The contributions and the plan of this paper are as follows: First, we outline an overall system presentation of HOL-TestGen (Sec. 3), second, we will develop the interactive methodology of generating specification-based black box tests (Sec. 4), and finally, we develop a proof of concept for white box testing in our framework. In particular, we show how our concept of explicit test hypotheses can be applied in this context (Sec. 5).

2 Foundations

2.1 Isabelle

Isabelle [10] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic, Zermelo-Fränkel set theory and HOL, which we choose as framework for HOL-TestGen.

While Isabelle/HOL is usually coined as "proof assistant", we use it as symbolic computation environment. Implementations on Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution and rewriting, and the overall environment provides a large collection of components ranging from documentation generators and code-generators to (generic) decision procedures for datatypes and Presburger Arithmetic.

Isabelle can easily be controlled by a programming interface on its implementation level in SML in a logically safe way, as well as in the Isar level, i.e., a tactic

proof language in which interactive and automated proofs can be mixed arbitrarily. Documents in the Isar format, enriched by the commands provided by HOL-TestGen, can be processed incrementally within Proof General (see Sec. 3) as well as in batch mode. These documents can be seen as formal and technically checked test plan of a program under test.

2.2 Higher-order Logic

Higher-order logic (HOL) [7, 3] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e.g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

Isabelle/HOL provides also a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories. Furthermore, it provides the means for defining data types and recursive function definitions over them in a style similar to a functional programming language.

Isabelle/HOL processes rules and theorems of the form $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, also denoted as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$. They can be understood as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". In particular, the presentation of sub-goals uses this format. We will refer to assumptions also as *constraints* in this paper.

3 The HOL-TestGen System

HOL-TestGen is an *interactive* (semi-automated) test tool for specification based unit tests. Its theory and implementation has been described in [5], here, we briefly review main concepts and outline the standard workflow. The latter is divided into four phases: writing the *test specification*, generation of *test cases* along with a *test theorem*, generation of *test data* (TD), and the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test. Once a test theory is completed, documents can be generated that represent a formal test plan. See Fig. 1 for the overall workflow.

The properties of *program under test* are specified in HOL in the *test specification* (TS). The system will decompose the test specification in the test case generation phase into a semantically equivalent *test theorem* which has the form:

$$[\![\mathsf{TD}_1; ...; \mathsf{TD}_n; \mathsf{THYP}\ \mathsf{H}_1; ...; \mathsf{THYP}\ \mathsf{H}_m]\!] \Longrightarrow \mathsf{TS}$$

where THYP is a constant (semantically: an identity) used to mark the *test hypotheses* that are underlying this test. At present, HOL-TestGen uses only uniformity and regularity Hypothesis; for example, a uniformity hypothesis means informally "if the program conforms to one instance of a case to TS, it conforms

[1] to be more specific: *parametric polymorphism*

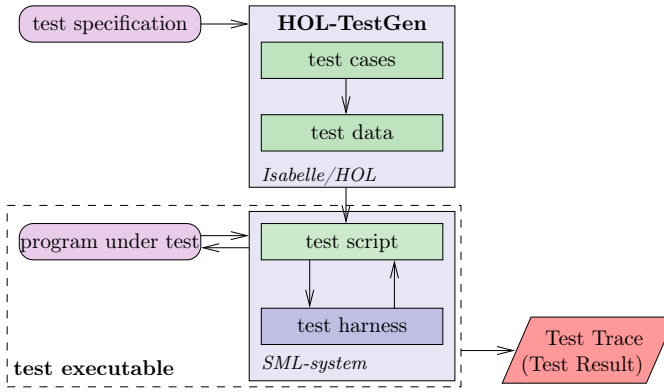

Figure 1. Overview of the Standard Workflow of HOL-TestGen

to all instances of this case to TS" (see Sec.4.2 for a formal presentation). Thus, a test theorem has the following meaning:

If the program under test passes the tests for all TD_i *successfully, and if it satisfies all test hypothesis, it conforms to the test specification.*

In this sense, a test theorem bridges the gap between test and verification. h The theory containing test theory, test specifications, configurations of the test data and test script generation, possibly extended by proofs for rules that support

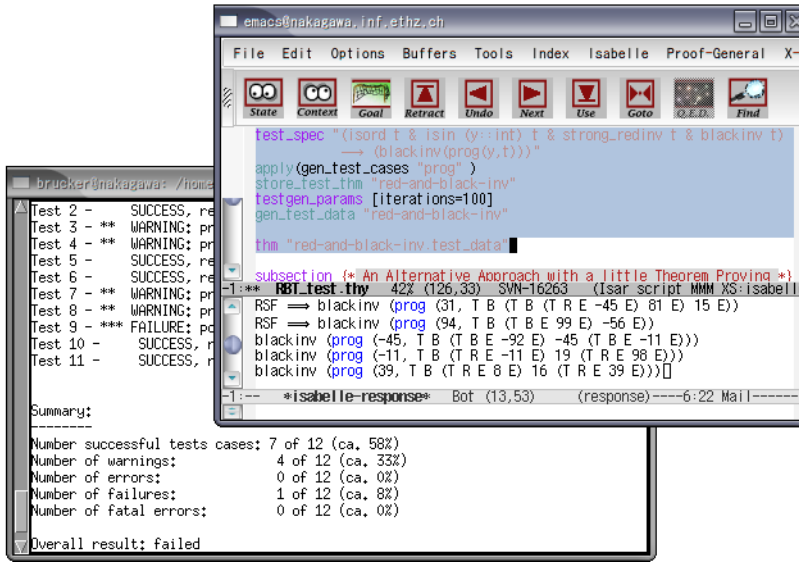

Figure 2. A HOL-TestGen Session Using Proof General

the overall process, is written in an extension of the Isar language [12]. It can be processed in batch mode, but also using the Proof General interface interactively, see Fig. 2. This interface allows for interactively stepping through a test theory (in the upper sub-window) and the sub-window below shows the corresponding system state. A system state may be a proof state in a test theorem development, or the result of inspections of generated test data or a list of test hypothesis.

After test data generation, HOL-TestGen can produces a *test script* driving the test using the provided *test harness*. The test script together with the test harness stimulate the code for the program under test built into the *test executable*. Executing the *test executable* runs the test and results in a *test trace* showing possible errors in the implementation (see lower window in Fig. 2).

4 Interactive Black Box Testing

In this section we present the method for the current main application of HOL-TestGen: generating test data for black box testing of side-effect free programs. As running example we chose the red-black trees already used in [5] to find an error in the "real" sml/NJ library. However, this time we will show *how* errors were found and how test data can be generated that actually explores the program under test to a satisfactory degree.

4.1 The Test Specification

Red-black trees store the balancing information in one additional bit per node, which is called the "color of a node". This is either red or black. A valid (balanced) red-black tree must fulfill the following three invariants:

1. *Red Invariant:* each red node has a black parent.
2. *Strong Red Invariant:* the root is red *and* the red invariant holds.
3. *Black Invariant:* each path from the root to a leaf has the same number of black nodes.

An invariant can be represented as recursive predicate; for the red invariant this looks as follows:

```
types     'a item = "'a::ord_key"
datatype  color = R | B
datatype 'a tree = E | T color "'a tree" "'a item" "'a tree"


consts redinv :: "'a tree ⇒ bool"
recdef redinv "measure (λt. (size t))"
              "redinv E = True"
              "redinv (T B a y b) = (redinv a ∧ redinv b)"
              "redinv (T R (T R a x b) y c) = False"
              "redinv (T R a x (T R b y c)) = False"
              "redinv (T R a x b) = (redinv a ∧ redinv b)"
```

Assume we want to test that insertion or deletion (summarized by the placeholder prog) fulfill the black invariant. Hence, we are searching for test data fulfilling the premise of the following test specification:

```
test_spec "(isord t∧isin y t∧strong_redinv t∧ blackinv t)⟶ (blackinv(prog(y,t)))"
```

which we found after some experimenting (weaker preconditions lead to understandable exceptions of the program under test).

4.2 First Attempt: The "Standard Workflow"

Test Case Generation Now we can automatically generate *test cases* in a model checking-like fashion by applying the gen_test_cases method. The method generates data-structures (here: trees) up to a certain depth and performs case splitting over all possible cases; remaining constraints are simplified. The default depth-parameter of the method is set to 3. Finally, the resulting test theorem is stored in a *test environment*:

```
  apply(gen_test_cases "prog")
store_test_thm "red_and_black_inv"
```

This fairly simple setup generates already 25 subgoals containing 12 test cases, altogether with non-trivial constraints, among them:

```
1. ⟦ x1=x2 ⟧ ⟹ blackinv (prog (x1, T B E x2 E))
2. ⟦ x1=x6; max_B_height (T x5 x4 x3 x2) = 0; blackinv x2;
     max_B_height x4 = max_B_height x2; blackinv x4; redinv (T x5 x4 x3 x2);
      ∀x. (x= x3 ⟶x6 < x) ∧ ( isin x x4 ⟶x6 < x) ∧ ( isin x x2 ⟶x6 < x);
      ∀x. isin x x2 ⟶x3 < x; ∀x. isin x x4 ⟶x < x3; isord x2; isord x4⟧
      ⟹ blackinv (prog (x1, T B E x6 (T x5 x4 x3 x2)))
```

An example for a generated uniformity test hypothesis is:

```
THYP ((∃x xa. x = xa ⟶blackinv (prog (x, T B E xa E)) ⟶
       (∀x xa. x = xa ⟶blackinv (prog (x, T B E xa E))));
```

Test Data Generation Generating concrete test data already takes a remarkable length of time, as it's quite unlikely that the random solver generates values that fulfill these ordering constraints. Therefore we restrict the attempts (*iterations*) the random solver takes for solving a single test case to 40

```
testgen_params [iterations=40]
gen_test_data "red_and_black_inv"
```

which is sadly not sufficient to solve all conditions, e.g., we obtain test cases like

```
RSF ⟶ blackinv (prog (100, T B E 7 E))
RSF ⟶ blackinv (prog (83, T B (T B (T B E −8 E) 57 (T R E 13 E)) −62 E))
blackinv (prog (−91, T B (T R E −91 E) 5 E))
RSF ⟶ blackinv (prog (−33, T B (T R E −2 E) 37 E))
```

were RSF marks unsolved cases. Analyzing the generated test data reveals that only very few had been resolved and therefore lead to inconclusive tests. To compute more conclusive test data, we can interactively increase the number of iterations, which reveals that we need to set iterations to more than 100 to find a suitable set of test data reliably.

Test Script Generation Now we generate the test script for `PUT` being implemented by `wrapper.del`:

```
gen_test_script "rbt_script.sml" "red_and_black_inv" "PUT" "wrapper.del"
```

In principle, any SML-system should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML implementations, e.g., Java, or C, is supported. Depending on the SML-system, the test execution can be done within an interpreter or using a compiled test executable. Testing implementations written in SML is straightforward. For testing non-SML implementations it is in most cases sufficient to provide a quite simple "wrapper" doing some datatype conversion.

Test Result Verification Running the test executable for red_and_black_inv results in an output similar to Tab. 1, showing successful test cases, failures (i.e., the implementation violates the post condition) and warning caused by unresolved cased (where the random solver returns `RSF` as pre-condition). In the latter case, the `PUT` is still executed (and throws in our example an exception). Already in this highly automatic set-up, we were able to produce the reported

```
Test Results:
 Test  0 -     SUCCESS, result: E
 Test  1 -     SUCCESS, result: T(R,E,67,E)
 Test  2 -     SUCCESS, result: T(B,E,~88,E)
 Test  3 - **  WARNING: pre cond. false (exception during post cond.)
 Test  4 - **  WARNING: pre cond. false (exception during post cond.)
 Test  5 -     SUCCESS, result: T(R,E,30,E)
 Test  6 -     SUCCESS, result: T(B,E,73,E)
 Test  7 - **  WARNING: pre cond. false (exception during post cond.)
 Test  8 - **  WARNING: pre cond. false (exception during post cond.)
 Test  9 - *** FAILURE: post cond. false, result: T(B,T(B,E,~92,E),~11,E)
 Test 10 -     SUCCESS, result: T(B,E,19,T(R,E,98,E))
 Test 11 -     SUCCESS, result: T(B,T(R,E,8,E),16,E)

Summary:
 Number successful tests cases: 7 of 12 (ca. 58%)
 Number of warnings:            4 of 12 (ca. 33%)
 Number of errors:              0 of 12 (ca. 0%)
 Number of failures:            1 of 12 (ca. 8%)
 Number of fatal errors:        0 of 12 (ca. 0%)
```

Table 1. RBT Test trace

error in the SML library. However, based on the test depth 3 (which represents the limit of the standard approach if we restrict ourselves to a time investment of 10 minutes for the overall run) we cannot have trees with more than three nodes on the level of the test case generation. Of course, random solving increases the

depth of the trees sporadically, as can be seen from the test result, but in an unsystematic way. Thus, the program under test has obviously not been tested satisfactorily, and we need means to treat test data sets with higher depth.

4.3 Second Attempt: Using "Abstract Test Data"

By inspection of the constraints of the test theorem, one immediately identifies predicates for which solutions are difficult to find by a random process (a measure for this difficulty could be the percentage of trees up to depth k, that make this predicate valid. One can easily convince oneself, that this percentage is decreasing asymptotically).

Repeatedly, ground instances were needed for terms of the form:

1. max_B_height x = 0
2. max_B_height y = max_B_height z
3. blackinv x
4. redinv x
5. isord x

How can the constraint resolution be helped by user guidance? The idea is to establish ground instances by hand and to feed them into the resolution process as *abstract test cases*, see [6]. But which of the patterns should we choose? It turns out that max_B_height X = 0 (is the number of black nodes on any path 0?) has many candidates, but after depth 2 they are all ruled out by redinv). Thus, we picked redinv and provided ground instances for it by hand: redinv E, redinv (T R E (5::int) E), redinv (T B E (5::int) E), redinv (T R E 2 (T B E (5::int) E)), redinv ((T R (T B E (5::int) E) 6 E)), redinv(T R (T B E 3 E)4 (T B E (5::int) E)), etc. Each of these ground instances is in fact established by an automatic proof:

```
lemma redinv_6[test "red_and_black_inv"]:
      "redinv(T R (T B E 3 E) 4 (T B E (5::int) E))"  by auto
```

The pragma [test "red_and_black_inv"] is used to associate this theorem as abstract test data to the data generation

```
gen_test_data "red_and_black_inv"
```

An analysis of the test results (omitted here for space reasons) reveals that the tests are now a more complete set of trees of depth 4.

Note, however, that the "samples" of abstract data had been chosen with hindsight to the overall test: they all represent ordered trees that happen to fulfill the black invariant, generated within the time frame of 10 minutes as in the previous run. Abstract test data that do not fulfill all the other possible constraints represent dead ends and are no help for the constraint solving phase.

4.4 Third Approach: Using a Little Theorem Proving

The question arises how this problematic aspect of ingeniously added abstract test data can be overcome and be systematized for our example. One answer is a characterization theorem of redinv:

```
lemma redinv_enumerate:
  "redinv x =((x = E)
            ∨ (∃ a y b. x = T B a y b ∧ redinv a ∧ redinv b)
            ∨ (∃ y. x = T R E y E)
            ∨ (∃ y am an ao. x = T R E y (T B am an ao) ∧
                             redinv (T B am an ao))
            ∨ (∃ ae af ag y. x = (T R (T B ae af ag) y E)
                             ∧ redinv (T B ae af ag))
            ∨ (∃ ae af ag y T B bg bh bi.
                             x = (T R (T B ae af ag) y (T B bg bh bi)) ∧
                             (redinv (T B ae af ag) ∧ redinv (T B bg bh bi))))"
```

The precise form of this lemma can be inferred when inspecting the rule set generated by Isabelle from the redinv-definition. The proof is a routine induction proof which nevertheless needs knowledge about theorem proving in general and Isabelle in particular. This lemma is used to improve the form of the test theorem. To be a bit more precise, we insert after the test case generation a sequence of Isar-methods that resolve in any constraint of the form redinv x the above lemma, recomputes the TNF and repeats this process once. The resulting test is now of depth 5 and constitutes now a quite extensive test of our program (again in the time-frame of 10 minutes for a complete run).

4.5 Summing Up

In our experience, increasing the number of iterations also increases remarkably the time needed for test data generation. On the other side, this underpins the usual criticism with respect to random testing: deeply nested (either in the sense of data or execution paths) program structures cannot be tested seriously using pure random tests; guidance generated by test cases is crucially needed.

Further, our results show that highly automated approaches yield useful "first shots" but heavily profit from more or less ingenious user interaction. A trade-off must be made here between the time needed to run a test (including generation), the quality of the test and the time and experience needed in advanced techniques such as Isabelle theorem proving.

5 Imperative White Box Tests

Our framework is not restricted to black box test of side-effect free programs. Using a *logical embedding* (a representation in HOL comprising syntax and semantics) for an imperative language, it can be used to implement and analyze various white-box test techniques.

5.1 The Language IMP: An Overview

The Isabelle distribution comes already with various logical embeddings: IMP, IMPP, NanoJava, or MicroJava, and more are available in the literature. For the sake of this presentation, we chose the simplest one, IMP, which is intended as

formalization of a textbook on programming language semantics [13, 9], and provides as such a particularly clean and complete collection of several semantics of IMP (natural semantics, transition semantics, denotational semantics, axiomatic semantics), proofs of their relations (e.g., denotational is equivalent to natural) and proofs of crucial meta-properties (axiomatic semantics is sound and relative complete).

The basic concepts of IMP are *values* val (just natural numbers, for example), and *states* state = loc ⇒ val. *Boolean expressions* bexp and *atomic expressions* (aexp) are represented as functions from state to val or bool. Thus, IMP has in fact no syntax of its own, but just inherits the expression language of HOL at this place[2]. The syntax of IMP commands com is then defined as data type:

```
datatype com = SKIP
             | ":==" loc aexp          ( infixl 60)
             | Semi com com            ("_ ; _" [60, 60] 10)
             | Cond bexp com com       (" IF _ THEN _ ELSE _" 60)
             | While bexp com          (" WHILE _ DO _" 60)
```

where the text in the parenthesis are just pragmas for the powerful Isabelle syntax engine to allow the usual infix/mixfix notation.

One of the operational semantics of IMP is a relation of triples evalc :: (com × state × state) set ((cm,s,s') ∈ evalc is denoted $\langle cm,s\rangle \xrightarrow[c]{} s'$) which is inductively defined as follows:

```
inductive evalc intros
  "⟨SKIP,s⟩ →c s"
  "⟨x :== a,s⟩ →c s[x:=(a s)]"
  "⟦ ⟨c0,s⟩ →c s1; ⟨cs1,s1⟩ →c s2 ⟧ ⟹ ⟨c0;cs1, s⟩ →c s2"
  "⟦ b s; ⟨c0,s⟩ →c s1 ⟧ ⟹ ⟨ IF b THEN c0 ELSE c_1, s⟩ →c s1"
  "⟦ ¬b s; ⟨c1,s⟩ →c s1 ⟧ ⟹ ⟨ IF b THEN c0 ELSE c1, s⟩ →c s1"
  "⟦¬b s⟧ ⟹ ⟨ WHILE b DO c, s⟩ →c s"
  "⟦ b s; ⟨c,s⟩→c s1; ⟨ WHILE b DO c,s1⟩→c s2 ⟧ ⟹ ⟨ WHILE b DO c, s⟩ →c s2"
```

The usual notation s[x:=v] is defined by λ y. if y=x then v else s y. For these inductive rules, an alternative rule set is derived that can be processed by the efficient Isabelle rewriter directly:

```
⟨ SKIP,s⟩ →c s' = (s' = s)
⟨ x :== a, s⟩ →c s' = (s' = s[x := a s])
b s ⟹ ⟨ IF b THEN d ELSE e,s⟩ →c s' = ⟨ d,s⟩ →c s'
. . .
```

We omit the definition of the denotational semantics reflecting the partial correctness C :: com ⇒(state × state) set (see [2] for details), but it is linked to the operational semantics via the theorem ((s, t) ∈ C c) = $\langle c,s\rangle \xrightarrow[c]{} t$. On the denotational level, program transformation rules relevant for the next section can be shown easily:

[2] This technique is also called a "shallow embedding"

```
C(SKIP;c) = C(c)          C(c;SKIP) = C(c)        C((c;d);e) = C(c;(d;e))
C(( IF b THEN c ELSE d);e) = C( IF b THEN c;e ELSE d;e)
C( WHILE b DO c)           = C( IF b THEN c; WHILE b DO c ELSE SKIP)
```

On the level of the denotational semantics, the usual notion of "valid Hoare triple" is formalized as:

$$\models \{P\}\ c\ \{Q\} \equiv \forall s\ t.\ (s,\ t) \in C\ c \longrightarrow P\ s \longrightarrow Q\ t$$

where P, Q are *assertions*, i.e., functions from state to bool.

5.2 Unwinding IMP Programs

To perform white box tests in the style of Pathfinder [11], SpecExplorer [8], or Korat [4], it is necessary to make the program paths explicit in the program representation and amenable to the rules of the operational semantics. Therefore, a pre-processing step is necessary that unfolds all `WHILE`-loops up to a certain limit, the *unwind-factor* k. This principle can also be applied in a language extension with procedure calls such as IMPP, also available in the Isabelle distribution. Additionally, the program should be transformed into a certain normal form to be efficiently processed (left associative sequential compositions must be avoided since they lead to an existentially quantified intermediate states which are more difficult to process in the symbolic computation). We define two recursive functions on com-terms that perform both these normalizations as well as the unwinding up to k. Note, that we will not program this function outside the logic as (tactic), i.e., a control program in SML, but inside HOL, such that we can also prove its correctness with respect to the IMP semantics:

```
consts "@@" :: "[com,com] ⇒com" (infixr 70)
primrec "SKIP @@ c = c"
        "(x:== E) @@ c = ((x:== E); c)"
        "(c;d) @@ e = (c; d @@ e)"
        "( IF b THEN c ELSE d) @@ e = ( IF b THEN c @@ e ELSE d @@ e)"
        "( WHILE b DO c) @@ e = (( WHILE b DO c);e)"

consts unwind :: "nat ×com ⇒com"
recdef unwind "less_than <*lex*> measure(λ s. size s)"
   "unwind(n, SKIP)      = SKIP"
   "unwind(n, a :== E) = (a :== E)"
   "unwind(n, IF b THEN c ELSE d) = IF b THEN unwind(n,c) ELSE unwind(n,d)"
   "unwind(n, WHILE b DO c) =
           (if 0 < n
            then IF b THEN unwind(n,c)@@unwind(n− 1,WHILE b DO c) ELSE SKIP
            else WHILE b DO unwind(0, c))"
   "unwind(n, SKIP ; c) = unwind(n, c)"
   "unwind(n, c ; SKIP) = unwind(n, c)"
   "unwind(n, ( IF b THEN c ELSE d) ; e) =
               ( IF b THEN (unwind(n,c;e)) ELSE (unwind(n,d;e)))"
   "unwind(n, (c ; d); e) = (unwind(n, c;d))@@(unwind(n,e))"
   "unwind(n, c ; d) = (unwind(n, c))@@(unwind(n, d))"
```

The primitive recursive auxiliary function c@@d appends a command d to the last command in c that is reachable from the root via sequential composition modes. The more tricky unwind function unfolds `WHILE`-loops as long as the unwind factor is positive and performs the program normal form computation along the program equivalences as discussed in Sec. 5.1.

The Isabelle Recursion Package adopts a "First Fit" pattern matching strategy (similar to SML). This means that in overlapping cases, the first is taken into account with higher priority—this is reflected on the level of the rewrite rule set generated from this definition. Thus, the last equation in the recursive definition is a catch-all rule for sequential composition.

Now we derived the following facts over these definitions:

Lemma 1 (Termination:). *Both functions terminate.*

Proof. In the case of @@ this is trivial due to machine checked primitive recursion; in case of unwind a proof has to be performed that the lexicographic composition of the standard ordering $\_ < \_$ and the standard term ordering is well-founded and respected by the inner calls in this recursive definition. This proof is done fully automatically.

Lemma 2 (Correctness:). *C(c @@ d)= C(c;d) and C(unwind(n,c))= C(c)*

Proof. For @@, a straight-forward induction suffices. As for unwind, the proof is non-trivial, but routine (generalization over n, induction over c, intricate case splitting, application of semantic equivalences of Sec. 5.1).

5.3 Generating Path Conditions

As example program, we chose a little program that computes the square-root of a natural number. In Isabelle/IMP syntax, we can define it as follows:

```
constdefs squareroot :: "[loc, loc, loc, loc] ⇒ com"
        "squareroot tm sum i a ≡ ((  tm  :== λs. 1);
                                  (( sum :== λs. 1);
                                   ((i   :== λs. 0);
                                     WHILE λs. (s sum) <= (s a) DO
                                       (( i   :== λs. (s i) + 1);
                                        ((tm  :== λs. (s tm) + 2);
                                         (sum :== λs. (s tm) + (s sum)))))))
```

where the locations (references) are the input into the program to express semantically constraints on them, as we will see later. The shallow embedding of the expressions has the consequence that program variable accesses must be represented as explicit application of the state s (at this program point) to a location representing this variable. Hence, we implicitly require a pre-parser that makes these bindings of program variables explicit.

We need one further derived rule If_split, which is necessary to expand the case splits produced for each path:

$$[\![b\ s \Longrightarrow \langle c, s\rangle \xrightarrow{}_{c} s';\ \neg\, b\ s \Longrightarrow \langle d, s\rangle \xrightarrow{}_{c} s']\!] \Longrightarrow \langle\ \texttt{IF}\ \ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d, s\rangle \xrightarrow{}_{c} s'$$

Putting everything together, we can now formulate the generation of symbolic states for the program squareroot as follows:

```
lemma derive_test_cases: assumes no_alias : . . .
  shows "⟨unwind(3, squareroot tm sum i a), s⟩ →c s'"
```

where the omitted technical side-condition no_alias specifies that the locations tm,sum,i,a are pairwise disjoint. Now, the canonical tactic script:

```
apply(simp add: squareroot_def)
apply( rule If_split , simp_all add: update_def no_alias)+
```

unfolds the definition of squareroot, and then enters in a loop that performs the computation of unwind (including path normalization), the case splitting along the If_split rule discussed above, the evaluation of state constraints and the simplification of the arithmetic constraints until no further changes can be achieved. The resulting proof-state consists of the following goals:[3]

```
1. 9 ≤s a ⟹ ⟨ WHILE  λs. s sum ≤s a
                DO  i :== λs. Suc (s i) ;
                  (tm :== λs. Suc (Suc (s tm)) ;
                    sum :== λs. s tm + s sum ),
               s(i := 3, tm := 7, sum := 16)⟩ →c s'
2. ⟦4 ≤s a; 8 < s a ⟧     ⟹ s' = s (i := 2, tm := 5, sum := 9)
3. ⟦ 1 ≤s a; s a < 4⟧     ⟹ s' = s (i := 1, tm := 3, sum := 4)
4.  s a = 0               ⟹ s' = s(tm := 1, sum := 1, i := 0)
```

The resulting proof state enumerates the possible symbolic states including their path conditions.[4]

5.4 Treating Assertions and Test Hypothesises

Traditional pre and post conditions can be expressed via the validity relation for Hoare Triple, e.g.: $\models$ {pre} squareroot tm sum i a {post a i} where pre is just λx. True and post a i is λ s. (s i)∗(s i)≤(s a) ∧ s a < (s i + 1)∗(s i + 1).

The setup of a specification based white box test is now produced by the derived rule:

$$\models\{P\}\ c\ \{Q\} = \forall s\ t.\quad \langle \text{unwind}\ (n,\ c), s\rangle \xrightarrow[c]{}\ t \longrightarrow P\ s \longrightarrow Q\ t$$

The result of this rule application is piped into the previous process which conjoins the preconditions with the path conditions and attempts to solve them; the post condition is then constructed over the post state constructed by the natural semantics.

Assertions can be introduced into our language as follows: First, we declare an uninterpreted constant STOP as command of the language. Then, a construct like ASSERT b c can be introduced as abbreviation for ASSERT b c ≡ IF b THEN c ELSE STOP, and further constructs like an annotated while loop AWHILE b inv c are introduced analogously.

[3] the presentation has been slightly syntactically simplified

[4] The computing time for unwind-factor 10 based on this simplistic tactic remains under a few seconds, including pretty-printing.

It remains to show how white box testing fits *methodically* into our framework, where we try to generate *test hypothesis* that make the "logical difference" between a test and the verification of the test specification explicit. Obviously, the only new element related to white box test is the unwinding parameter; if exhausted, this leads to program fragments that represent the "set of untested execution paths" of a program under test. In our running example, this lead to the first sub-goal in the final proof state. Turned into an explicit *unwinding k test hypothesis*, this condition for resulting from the test theorem: $\models\{\text{pre}\}$ squareroot tm sum i a {post a i} looks as follows:

```
1. THYP(9 ≤s a ⟶ ⟨ WHILE  λs. s sum ≤s a
                   DO     i :== λs. Suc (s i) ;
                      (tm :== λs. Suc (Suc (s tm)) ;
                      sum :== λs. s tm + s sum ),
                  s(i := 3, tm := 7, sum := 16)⟩ →c s'
                  ∧ post a i s')
```

Testing a program in this setting means that all symbolic state transitions including their path conditions must satisfy the post condition whenever the pre condition holds. This is the case in our example, and the system will find the satisfiability of the generated constraints without need for random solving in this case. The only remaining assumption is the test hypothesis shown above which reflects that we have tested the program and not verified it.

To sum up, we described a symbolic computation process for white box tests in the language IMP, that generates from a given, potentially annotated program a test theorem including the test hypotheses automatically. This test theorem can be fed into the test data generation phase to find ground instances for particular paths as before.

5.5 Blowing up IMP

The reader might object that the language IMP, having only Boolean and arithmetic side-effect free expressions and non-recursive, macro-like procedures, is too academic to be of practical importance. In contrast, we argue that IMP is a reasonable core language which can be "blown-up" fairly easy to larger languages, in large parts without adding further complexity to the symbolic computation process presented so far.

We discuss three extensions of IMP, two more straight-forward, one more involved, to give an impression over the potential of our approach:

1. *Mutual recursion:* Just apply our approach to embedding IMPP.
2. *Arbitrary expressions:* exchange val in the IMP semantics by a universe which is a sum of the HOL data types.
3. *Objects:* Extend our approach to an embedding like NanoJava.

In more detail, extension 2 requires that program variables must be presented as triples $(\text{loc}, \text{emb}::\alpha \longrightarrow \text{val}, \text{proj}::\text{val} \longrightarrow \alpha)$ consisting of the traditional location, and a pair of functions (representing the typing of the program and variable) that

allow for injection and projection of HOL-values into the val universe of IMP. Program variable accesses, which has been encoded by s a so far, will be s!a where s!(a,emb,prj) is defined by prj(s a). The assignment semantics of IMP must be adopted analogously. This technique paves the way for lists, options, strings, and further user-defined data types. Expressions over user-defined HOL data-types can now be processed by the gen_test_cases-method which is at the heart of HOL-TestGen. As a result of these extensions, we have an SML-like language with data-types and HOL-expressions inside.

The extension 3 involves sub-classing, method calls with late-binding and object creation; as such, a lot more machinery is therefore involved whose tactical control will be feasible in our opinion, but require substantial more work.

6 Conclusion

We have shown the pragmatics of our Isabelle/HOL-based testing tool HOL-TestGen [1, 5] gained from previous experiences for specification based black box tests. While some aspects of the symbolic computations are fully automatic (like data separation lemma generation, generation of test hypothesis, TNF-computations, test data generations and solving), other aspects like constraint solving may profit from some theorem proving and experiments with "appropriate" formulations of test specifications/test theorems. We have also developed a method to use HOL-TestGen for specification based white box tests.

The symbolic computation process is fully presented inside HOL, so no tool integration and conversion issues are involved which may be critical both for correctness and efficiency. Since the necessary symbolic transformation processes can be based on derived rules,[5] HOL-TestGen can be used as a tool for a seamless *conceptual study* of these techniques including formal correctness proofs, their *prototypical implementation* and even their *industry strength implementation*. The latter, will require substantial effort in tactic programming and tool integration.

Although the example for imperative white-box test is based on a conceptual language and therefore merely a proof of concept than a proof of technology, we believe that the approach can scale up with respect to size of the supported language while maintaining reasonable efficiency of the underlying symbolic computations. Thus, we believe that HOL-TestGen can be seen as unifying framework in which a wide range of unit test techniques can be presented in a mathematically clean way.

References

[1] HOL-TestGen. `http://www.brucker.ch/projects/hol-testgen/`.
[2] IMP. `http://isabelle.in.tum.de/library/HOL/IMP/Denotation.html`.
[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, Orlando, May 1986.

[5] Inserting such rules as *axioms* is trivial, but endangers correctness, of course

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the international symposium on Software testing and analysis*, pages 123–133, 2002.
[5] A. D. Brucker and B. Wolff. HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zürich, Apr. 2005.
[6] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, Linz, 2005.
[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
[8] W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, 2004.
[9] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.
[12] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002.
[13] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.