

Program-based Testing

A Proof-of-Technology

Achim D. Brucker and Burkhart Wolff

October 14, 2012

We present the underlying methods for 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

Further readings: This theory is an extended and updated version of the work described in the paper “Interactive testing using HOL-TestGen.”, LNCS 2997, Springer Verlag, 2005 [4].

Contents

1	Introduction to White Box Tests	2
1.1	The Language IMP: An Overview	2
1.2	Unwinding IMP Programs	3
1.3	Generating Path Conditions	4
1.4	Treating Assertions and Test Hypotheses	5
1.5	Blowing up IMP	6
2	Conclusion	7
3	Appendix	8
3.1	Execution of commands	8
3.2	Equivalence of statements	10
3.3	Execution is deterministic	12
3.4	Unfold and its Correctness	17
3.5	Symbolic Evaluation Rule-Set	18
3.6	Splitting Rule for program-based Tests	19
3.7	Tactic Set-up	19
3.8	The Definition of the Integer-Squareroot Program	20
3.9	Computing Program Paths and their Path-Constraints	20

3.10 Testing Specifications	20
3.11 An Alternative Approach with an On-The-Fly generated Explicit Test-Hyp.	21

1 Introduction to 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.

1.1 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 [10, 8], 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 \Rightarrow 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 place¹. The syntax of IMP commands `com` is then defined as data type:

```
datatype com = SKIP
  | "x ::= a" 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 \times state \times state) set` (`((cm,s,s') \in evalc` is denoted `$\langle cm,s \rangle \xrightarrow{c} s'$`) which is inductively defined as follows:

inductive evalc intros

```
"<SKIP,s>  $\xrightarrow{c}$  s"
"⟨x ::= a,s⟩  $\xrightarrow{c}$  s[x:=(a s)]"
"[[⟨c0,s⟩  $\xrightarrow{c}$  s1; ⟨cs1,s1⟩  $\xrightarrow{c}$  s2]]  $\implies$  ⟨c0;cs1, s⟩  $\xrightarrow{c}$  s2"
"[[b s; ⟨c0,s⟩  $\xrightarrow{c}$  s1]]  $\implies$  ⟨IF b THEN c0 ELSE c1, s⟩  $\xrightarrow{c}$  s1"
"[[¬b s; ⟨c1,s⟩  $\xrightarrow{c}$  s1]]  $\implies$  ⟨IF b THEN c0 ELSE c1, s⟩  $\xrightarrow{c}$  s1"
"[[¬b s]]  $\implies$  ⟨WHILE b DO c, s⟩  $\xrightarrow{c}$  s"
"[[b s; ⟨c,s⟩  $\xrightarrow{c}$  s1; ⟨WHILE b DO c,s1⟩  $\xrightarrow{c}$  s2]]  $\implies$  ⟨WHILE b DO c, s⟩  $\xrightarrow{c}$  s2"
```

¹This technique is also called a “shallow embedding”

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

$$\begin{aligned} \langle \text{SKIP}, s \rangle &\xrightarrow{c} s' = (s' = s) \\ \langle x := a, s \rangle &\xrightarrow{c} s' = (s' = s[x := a]) \\ b s \implies \langle \text{IF } b \text{ THEN } d \text{ ELSE } e, s \rangle &\xrightarrow{c} s' = \langle d, s \rangle \xrightarrow{c} s' \\ &\dots \end{aligned}$$

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

$$\begin{aligned} C(\text{SKIP}; c) &= C(c) & C(c; \text{SKIP}) &= C(c) & C((c; d); e) &= C(c; (d; e)) \\ C((\text{IF } b \text{ THEN } c \text{ ELSE } d); e) &= C(\text{IF } b \text{ THEN } c; e \text{ ELSE } d; e) \\ C(\text{WHILE } b \text{ DO } c) &= C(\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP}) \end{aligned}$$

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

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

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

1.2 Unwinding IMP Programs

To perform white box tests in the style of Pathfinder [9], SpecExplorer [7], or Korat [3], 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. 1.1.

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. \square

Lemma 2 (Correctness:). $C(c @@ d) = C(c;d)$ and $C(\text{unwind}(n,c)) = C(c)$

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. 1.1). \square

1.3 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 squarerooot :: "[loc,loc,loc,loc]  $\Rightarrow$  com"
"squarerooot tm sum i a  $\equiv$  (( tm ::=  $\lambda$ s. 1);
  (( sum ::=  $\lambda$ s. 1);
  (( i ::=  $\lambda$ s. 0);
  WHILE  $\lambda$ s. (s sum) <= (s a) DO

```

$$((i := \lambda s. (s\ i) + 1); \\ ((tm := \lambda s. (s\ tm) + 2); \\ (sum := \lambda s. (s\ tm) + (s\ sum))))))$$

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:

$$\llbracket b \rrbracket s \implies \langle c, s \rangle \xrightarrow{c} s'; \neg b \rrbracket s \implies \langle d, s \rangle \xrightarrow{c} s' \implies \langle \text{IF } b \text{ THEN } c \text{ ELSE } d, s \rangle \xrightarrow{c} s'$$

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) \xrightarrow{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:²

1. $9 \leq s\ a \implies \langle \text{WHILE } \lambda s. s\ \text{sum} \leq s\ a \\ \text{DO } i := \lambda s. \text{Suc } (s\ i); \\ (tm := \lambda s. \text{Suc } (\text{Suc } (s\ tm))); \\ \text{sum} := \lambda s. s\ tm + s\ \text{sum}), \\ s(i := 3, tm := 7, sum := 16) \rangle \xrightarrow{c} s'$
2. $\llbracket 4 \leq s\ a; 8 < s\ a \rrbracket \implies s' = s(i := 2, tm := 5, sum := 9)$
3. $\llbracket 1 \leq s\ a; s\ a < 4 \rrbracket \implies s' = s(i := 1, tm := 3, sum := 4)$
4. $s\ a = 0 \implies s' = s(tm := 1, sum := 1, i := 0)$

The resulting proof state enumerates the possible symbolic states including their path conditions.³

1.4 Treating Assertions and Test Hypotheses

Traditional pre and post conditions can be expressed via the validity relation for Hoare Triple, e.g.: $\models \{\text{pre}\} \text{ squareroot } tm\ sum\ i\ a\ \{\text{post } a\ i\}$ where `pre` is just $\lambda x. \text{True}$ and `post a i` is $\lambda s. (s\ i) * (s\ i) \leq (s\ a) \wedge s\ a < (s\ i + 1) * (s\ i + 1)$.

²the presentation has been slightly syntactically simplified

³The computing time for `unwind`-factor 10 based on this simplistic tactic remains under a few seconds, including pretty-printing.

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

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

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 $c \equiv \text{IF } b \text{ THEN } c \text{ ELSE STOP}$, and further constructs like an annotated while loop **AWHILE** b $\text{inv } c$ are introduced analogously.

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: $\models \{\text{pre}\} \text{sqrtroot } tm \text{ sum } i \text{ a } \{\text{post } a \text{ } i\}$ looks as follows:

1. $\text{THYP}(9 \leq s \text{ a} \longrightarrow \langle \text{WHILE } \lambda s. s \text{ sum} \leq s \text{ a}$
 $\quad \text{DO } i := \lambda s. \text{Suc } (s \text{ } i) ;$
 $\quad (tm := \lambda s. \text{Suc } (\text{Suc } (s \text{ } tm)) ;$
 $\quad \text{sum} := \lambda s. s \text{ } tm + s \text{ } \text{sum}),$
 $\quad s(i := 3, tm := 7, \text{sum} := 16) \rangle \xrightarrow{c} s'$
 $\quad \wedge \text{post } a \text{ } i \text{ } s')$

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.

1.5 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.

2 Conclusion

We have shown the pragmatics of our Isabelle/HOL-based testing tool HOL-TestGen [1, 6] 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,⁴ 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.

⁴Inserting such rules as *axioms* is trivial, but endangers correctness, of course

3 Appendix

theory *Com* **imports** *Main* **begin**

typedecl *loc*

— an unspecified (arbitrary) type of locations (addresses/names) for variables

types

val = *nat* — or anything else, *nat* used in examples

state = *loc* \Rightarrow *val*

aexp = *state* \Rightarrow *val*

bexp = *state* \Rightarrow *bool*

— arithmetic and boolean expressions are not modelled explicitly here,

— they are just functions on states

datatype

com = *SKIP*

| *Assign loc aexp* (*- ::= - 60*)

| *Semi com com* (*;- [60, 60] 10*)

| *Cond bexp com com* (*IF - THEN - ELSE - 60*)

| *While bexp com* (*WHILE - DO - 60*)

notation (*latex*)

SKIP (*SKIP*) **and**

Cond (*IF - THEN - ELSE - 60*) **and**

While (*WHILE - DO - 60*)

end

theory *Natural* **imports** *Com* **begin**

3.1 Execution of commands

We write $\langle c, s \rangle \longrightarrow_c s'$ for *Statement* *c*, *started in state* *s*, *terminates in state* *s'*. Formally, $\langle c, s \rangle \longrightarrow_c s'$ is just another form of saying *the tuple* (c, s, s') *is part of the relation* *evalc*:

definition

update :: (*'a* \Rightarrow *'b*) \Rightarrow *'a* \Rightarrow *'b* \Rightarrow (*'a* \Rightarrow *'b*) (*- / [- ::= / -] [900, 0, 0] 900*) **where**

update = *fun-upd*

notation (*xsymbols*)

update (*- / [- \mapsto / -] [900, 0, 0] 900*)

Disable conflicting syntax from HOL Map theory.

no-syntax

-maplet :: [*'a*, *'a*] \Rightarrow *maplet* (*- / | - > / -*)

-maplets :: [*'a*, *'a*] \Rightarrow *maplet* (*- / [| - >] / -*)

:: *maplet* \Rightarrow *maplets* (*-*)

$-Maplets :: [maplet, maplets] \Rightarrow maplets \ (-/-)$
 $-MapUpd :: ['a \rightsquigarrow 'b, maplets] \Rightarrow 'a \rightsquigarrow 'b \ (-/'(-) [900,0]900)$
 $-Map :: maplets \Rightarrow 'a \rightsquigarrow 'b \ \ ((1[-]))$

The big-step execution relation *evalc* is defined inductively:

inductive

evalc :: [com,state,state] \Rightarrow bool ($\langle -, - \rangle \longrightarrow_c - [0,0,60] \ 60$)

where

Skip: $\langle SKIP, s \rangle \longrightarrow_c s$

| *Assign*: $\langle x ::= a, s \rangle \longrightarrow_c s[x \mapsto a \ s]$

| *Semi*: $\langle c0, s \rangle \longrightarrow_c s'' \Longrightarrow \langle c1, s'' \rangle \longrightarrow_c s' \Longrightarrow \langle c0; c1, s \rangle \longrightarrow_c s'$

| *IfTrue*: $b \ s \Longrightarrow \langle c0, s \rangle \longrightarrow_c s' \Longrightarrow \langle IF \ b \ THEN \ c0 \ ELSE \ c1, s \rangle \longrightarrow_c s'$

| *IfFalse*: $\neg b \ s \Longrightarrow \langle c1, s \rangle \longrightarrow_c s' \Longrightarrow \langle IF \ b \ THEN \ c0 \ ELSE \ c1, s \rangle \longrightarrow_c s'$

| *WhileFalse*: $\neg b \ s \Longrightarrow \langle WHILE \ b \ DO \ c, s \rangle \longrightarrow_c s$

| *WhileTrue*: $b \ s \Longrightarrow \langle c, s \rangle \longrightarrow_c s'' \Longrightarrow \langle WHILE \ b \ DO \ c, s'' \rangle \longrightarrow_c s'$
 $\Longrightarrow \langle WHILE \ b \ DO \ c, s \rangle \longrightarrow_c s'$

lemmas *evalc.intros* [intro] — use those rules in automatic proofs

The induction principle induced by this definition looks like this:

$\llbracket \langle x1, x2 \rangle \longrightarrow_c x3; \bigwedge s. P \ SKIP \ s \ s; \bigwedge x \ a \ s. P \ (x ::= a) \ s \ (s[x \mapsto a \ s]);$
 $\bigwedge c0 \ s \ s'' \ c1 \ s'.$
 $\llbracket \langle c0, s \rangle \longrightarrow_c s''; P \ c0 \ s \ s''; \langle c1, s'' \rangle \longrightarrow_c s'; P \ c1 \ s'' \ s' \rrbracket$
 $\Longrightarrow P \ (c0; c1) \ s \ s';$
 $\bigwedge b \ s \ c0 \ s' \ c1. \llbracket b \ s; \langle c0, s \rangle \longrightarrow_c s'; P \ c0 \ s \ s' \rrbracket \Longrightarrow P \ (IF \ b \ THEN \ c0 \ ELSE \ c1) \ s \ s';$
 $\bigwedge b \ s \ c1 \ s' \ c0. \llbracket \neg b \ s; \langle c1, s \rangle \longrightarrow_c s'; P \ c1 \ s \ s' \rrbracket \Longrightarrow P \ (IF \ b \ THEN \ c0 \ ELSE \ c1) \ s \ s';$
 $\bigwedge b \ s \ c. \neg b \ s \Longrightarrow P \ (WHILE \ b \ DO \ c) \ s \ s;$
 $\bigwedge b \ s \ c \ s'' \ s'.$
 $\llbracket b \ s; \langle c, s \rangle \longrightarrow_c s''; P \ c \ s \ s''; \langle WHILE \ b \ DO \ c, s'' \rangle \longrightarrow_c s';$
 $P \ (WHILE \ b \ DO \ c) \ s'' \ s' \rrbracket$
 $\Longrightarrow P \ (WHILE \ b \ DO \ c) \ s \ s'$
 $\Longrightarrow P \ x1 \ x2 \ x3$

(\bigwedge and \Longrightarrow are Isabelle's meta symbols for \forall and \longrightarrow)

The rules of *evalc* are syntax directed, i.e. for each syntactic category there is always only one rule applicable. That means we can use the rules in both directions. This property is called rule inversion.

inductive-cases *skipE* [elim!]: $\langle SKIP, s \rangle \longrightarrow_c s'$

inductive-cases *semiE* [elim!]: $\langle c0; c1, s \rangle \longrightarrow_c s'$

inductive-cases *assignE* [elim!]: $\langle x ::= a, s \rangle \longrightarrow_c s'$

inductive-cases *ifE* [elim!]: $\langle IF \ b \ THEN \ c0 \ ELSE \ c1, s \rangle \longrightarrow_c s'$

inductive-cases *whileE* [elim!]: $\langle WHILE \ b \ DO \ c, s \rangle \longrightarrow_c s'$

The next proofs are all trivial by rule inversion.

inductive-simps

skip: $\langle \text{SKIP}, s \rangle \longrightarrow_c s'$
and *assign*: $\langle x := a, s \rangle \longrightarrow_c s'$
and *semi*: $\langle c0; c1, s \rangle \longrightarrow_c s'$

lemma ifTrue:

$b \text{ s} \implies \langle \text{IF } b \text{ THEN } c0 \text{ ELSE } c1, s \rangle \longrightarrow_c s' = \langle c0, s \rangle \longrightarrow_c s'$
 $\langle \text{proof} \rangle$

lemma ifFalse:

$\neg b \text{ s} \implies \langle \text{IF } b \text{ THEN } c0 \text{ ELSE } c1, s \rangle \longrightarrow_c s' = \langle c1, s \rangle \longrightarrow_c s'$
 $\langle \text{proof} \rangle$

lemma whileFalse:

$\neg b \text{ s} \implies \langle \text{WHILE } b \text{ DO } c, s \rangle \longrightarrow_c s' = (s' = s)$
 $\langle \text{proof} \rangle$

lemma whileTrue:

$b \text{ s} \implies$
 $\langle \text{WHILE } b \text{ DO } c, s \rangle \longrightarrow_c s' =$
 $(\exists s''. \langle c, s \rangle \longrightarrow_c s'' \wedge \langle \text{WHILE } b \text{ DO } c, s'' \rangle \longrightarrow_c s')$
 $\langle \text{proof} \rangle$

Again, Isabelle may use these rules in automatic proofs:

lemmas evalc-cases [simp] = skip assign ifTrue ifFalse whileFalse semi whileTrue

3.2 Equivalence of statements

We call two statements c and c' equivalent wrt. the big-step semantics when c started in s terminates in s' iff c' started in the same s also terminates in the same s' . Formally:

definition

equiv-c :: $com \Rightarrow com \Rightarrow bool$ ($- \sim -$ [56, 56] 55) **where**
 $c \sim c' = (\forall s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s')$

Proof rules telling Isabelle to unfold the definition if there is something to be proved about equivalent statements:

lemma equivI [intro!]:

$(\bigwedge s s'. \langle c, s \rangle \longrightarrow_c s' = \langle c', s \rangle \longrightarrow_c s') \implies c \sim c'$
 $\langle \text{proof} \rangle$

lemma equivD1:

$c \sim c' \implies \langle c, s \rangle \longrightarrow_c s' \implies \langle c', s \rangle \longrightarrow_c s'$
 $\langle \text{proof} \rangle$

lemma equivD2:

$c \sim c' \implies \langle c', s \rangle \longrightarrow_c s' \implies \langle c, s \rangle \longrightarrow_c s'$
 $\langle \text{proof} \rangle$

As an example, we show that loop unfolding is an equivalence transformation on programs:

lemma *unfold-while*:

$(\text{WHILE } b \text{ DO } c) \sim (\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP})$ (is ?w ~ ?if)
 $\langle \text{proof} \rangle$

Happily, such lengthy proofs are seldom necessary. Isabelle can prove many such facts automatically.

lemma

$(\text{WHILE } b \text{ DO } c) \sim (\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP})$
 $\langle \text{proof} \rangle$

lemma *triv-if*:

$(\text{IF } b \text{ THEN } c \text{ ELSE } c) \sim c$
 $\langle \text{proof} \rangle$

lemma *commute-if*:

$(\text{IF } b1 \text{ THEN } (\text{IF } b2 \text{ THEN } c11 \text{ ELSE } c12) \text{ ELSE } c2)$
 \sim
 $(\text{IF } b2 \text{ THEN } (\text{IF } b1 \text{ THEN } c11 \text{ ELSE } c2) \text{ ELSE } (\text{IF } b1 \text{ THEN } c12 \text{ ELSE } c2))$
 $\langle \text{proof} \rangle$

lemma *while-equiv*:

$\langle c0, s \rangle \longrightarrow_c u \implies c \sim c' \implies (c0 = \text{WHILE } b \text{ DO } c) \implies \langle \text{WHILE } b \text{ DO } c', s \rangle \longrightarrow_c u$
 $\langle \text{proof} \rangle$

lemma *equiv-while*:

$c \sim c' \implies (\text{WHILE } b \text{ DO } c) \sim (\text{WHILE } b \text{ DO } c')$
 $\langle \text{proof} \rangle$

Program equivalence is an equivalence relation.

lemma *equiv-refl*:

$c \sim c$
 $\langle \text{proof} \rangle$

lemma *equiv-sym*:

$c1 \sim c2 \implies c2 \sim c1$
 $\langle \text{proof} \rangle$

lemma *equiv-trans*:

$c1 \sim c2 \implies c2 \sim c3 \implies c1 \sim c3$
 $\langle \text{proof} \rangle$

Program constructions preserve equivalence.

lemma *equiv-semi*:

$c1 \sim c1' \implies c2 \sim c2' \implies (c1; c2) \sim (c1'; c2')$
 $\langle \text{proof} \rangle$

lemma *equiv-if*:

$c1 \sim c1' \implies c2 \sim c2' \implies (IF\ b\ THEN\ c1\ ELSE\ c2) \sim (IF\ b\ THEN\ c1'\ ELSE\ c2')$
 $\langle proof \rangle$

lemma *while-never*: $\langle c, s \rangle \longrightarrow_c u \implies c \neq WHILE\ (\lambda s. True)\ DO\ c1$
 $\langle proof \rangle$

lemma *equiv-while-True*:

$(WHILE\ (\lambda s. True)\ DO\ c1) \sim (WHILE\ (\lambda s. True)\ DO\ c2)$
 $\langle proof \rangle$

3.3 Execution is deterministic

This proof is automatic.

theorem $\langle c, s \rangle \longrightarrow_c t \implies \langle c, s \rangle \longrightarrow_c u \implies u = t$
 $\langle proof \rangle$

The following proof presents all the details:

theorem *com-det*:

assumes $\langle c, s \rangle \longrightarrow_c t$ **and** $\langle c, s \rangle \longrightarrow_c u$
shows $u = t$
 $\langle proof \rangle$

This is the proof as you might present it in a lecture. The remaining cases are simple enough to be proved automatically:

theorem

assumes $\langle c, s \rangle \longrightarrow_c t$ **and** $\langle c, s \rangle \longrightarrow_c u$
shows $u = t$
 $\langle proof \rangle$

end

theory *Hoare* **imports** *Natural* **begin**

types *assn* = *state* => *bool*

inductive

hoare :: *assn* => *com* => *assn* => *bool* ($|- \{ (1-) \} / (-) / \{ (1-) \}$) 50)

where

skip: $|- \{ P \} SKIP \{ P \}$
 $|$ *ass*: $|- \{ \%s. P(s[x \mapsto a\ s]) \} x ::= a \{ P \}$
 $|$ *semi*: $[| \vdash \{ P \} c \{ Q \}; \vdash \{ Q \} d \{ R \} |] \implies \vdash \{ P \} c; d \{ R \}$
 $|$ *If*: $[| \vdash \{ \%s. P\ s \ \&\ b\ s \} c \{ Q \}; \vdash \{ \%s. P\ s \ \&\ \sim b\ s \} d \{ Q \} |] \implies$
 $\quad \vdash \{ P \} IF\ b\ THEN\ c\ ELSE\ d \{ Q \}$
 $|$ *While*: $|- \{ \%s. P\ s \ \&\ b\ s \} c \{ P \} \implies$
 $\quad \vdash \{ P \} WHILE\ b\ DO\ c \{ \%s. P\ s \ \&\ \sim b\ s \}$
 $|$ *conseq*: $[| \vdash !s. P'\ s \longrightarrow P\ s; \vdash \{ P \} c \{ Q \}; \vdash !s. Q\ s \longrightarrow Q'\ s |] \implies$
 $\quad \vdash \{ P \} c \{ Q \}$

lemma *strengthen-pre*: $[| \text{!}s. P' s \dashrightarrow P s; \vdash \{P\}c\{Q\} |] \Rightarrow \vdash \{P'\}c\{Q\}$
 $\langle \text{proof} \rangle$

lemma *weaken-post*: $[| \vdash \{P\}c\{Q\}; \text{!}s. Q s \dashrightarrow Q' s |] \Rightarrow \vdash \{P\}c\{Q'\}$
 $\langle \text{proof} \rangle$

lemma *While'*:
assumes $\vdash \{\%s. P s \ \& \ b \ s\} \ c \ \{P\}$ **and** *ALL* $s. P s \ \& \ \neg b \ s \longrightarrow Q s$
shows $\vdash \{P\} \text{ WHILE } b \text{ DO } c \ \{Q\}$
 $\langle \text{proof} \rangle$

lemmas $[simp] = skip \ ass \ semi \ If$

lemmas $[intro!] = hoare.skip \ hoare.ass \ hoare.semi \ hoare.If$

end

theory *Hoare-Op* **imports** *Hoare* **begin**

definition
 $hoare-valid :: [assn, com, assn] \Rightarrow bool \ (| = \{(1-)\} / (-) / \{(1-)\} \ 50)$ **where**
 $| = \{P\}c\{Q\} = (\text{!}s \ t. \langle c, s \rangle \longrightarrow_c t \dashrightarrow P s \dashrightarrow Q t)$

lemma *hoare-sound*: $\vdash \{P\}c\{Q\} \Rightarrow | = \{P\}c\{Q\}$
 $\langle \text{proof} \rangle$

definition
 $wp :: com \Rightarrow assn \Rightarrow assn$ **where**
 $wp \ c \ Q = (\%s. \text{!}t. \langle c, s \rangle \longrightarrow_c t \dashrightarrow Q t)$

lemma *wp-SKIP*: $wp \ SKIP \ Q = Q$
 $\langle \text{proof} \rangle$

lemma *wp-Ass*: $wp \ (x ::= a) \ Q = (\%s. Q(s[x \mapsto a \ s]))$
 $\langle \text{proof} \rangle$

lemma *wp-Semi*: $wp \ (c; d) \ Q = wp \ c \ (wp \ d \ Q)$
 $\langle \text{proof} \rangle$

lemma *wp-If*:
 $wp \ (IF \ b \ THEN \ c \ ELSE \ d) \ Q = (\%s. (b \ s \dashrightarrow wp \ c \ Q \ s) \ \& \ (\sim b \ s \dashrightarrow wp \ d \ Q \ s))$
 $\langle \text{proof} \rangle$

lemma *wp-While-If*:
 $wp \ (\text{WHILE } b \text{ DO } c) \ Q \ s =$
 $wp \ (IF \ b \ THEN \ c; \text{WHILE } b \text{ DO } c \ ELSE \ SKIP) \ Q \ s$
 $\langle \text{proof} \rangle$

lemma *wp-While-True*: $b \ s ==>$
 $wp \ (WHILE \ b \ DO \ c) \ Q \ s = wp \ (c; WHILE \ b \ DO \ c) \ Q \ s$
 $\langle proof \rangle$

lemma *wp-While-False*: $\sim b \ s ==>$ $wp \ (WHILE \ b \ DO \ c) \ Q \ s = Q \ s$
 $\langle proof \rangle$

lemmas $[simp] = wp\text{-}SKIP \ wp\text{-}Ass \ wp\text{-}Semi \ wp\text{-}If \ wp\text{-}While\text{-}True \ wp\text{-}While\text{-}False$

lemma *wp-is-pre*: $\vdash \{wp \ c \ Q\} \ c \ \{Q\}$
 $\langle proof \rangle$

lemma *hoare-relative-complete*: **assumes** $\models \{P\}c\{Q\}$ **shows** $\vdash \{P\}c\{Q\}$
 $\langle proof \rangle$

end

theory *VC* **imports** *Hoare-Op* **begin**

datatype *acom* = *Askip*
 $\mid Aass \ \text{loc} \ aexp$
 $\mid Asemi \ \text{acom} \ \text{acom}$
 $\mid Aif \ \text{bexp} \ \text{acom} \ \text{acom}$
 $\mid Awhile \ \text{bexp} \ \text{assn} \ \text{acom}$

primrec *awp* :: *acom* \Rightarrow *assn* \Rightarrow *assn*
where
 $awp \ Askip \ Q = Q$
 $\mid awp \ (Aass \ x \ a) \ Q = (\lambda s. \ Q(s[x \mapsto a \ s]))$
 $\mid awp \ (Asemi \ c \ d) \ Q = awp \ c \ (awp \ d \ Q)$
 $\mid awp \ (Aif \ b \ c \ d) \ Q = (\lambda s. \ (b \ s \longrightarrow awp \ c \ Q \ s) \ \& \ (\sim b \ s \longrightarrow awp \ d \ Q \ s))$
 $\mid awp \ (Awhile \ b \ I \ c) \ Q = I$

primrec *vc* :: *acom* \Rightarrow *assn* \Rightarrow *assn*
where
 $vc \ Askip \ Q = (\lambda s. \ True)$
 $\mid vc \ (Aass \ x \ a) \ Q = (\lambda s. \ True)$
 $\mid vc \ (Asemi \ c \ d) \ Q = (\lambda s. \ vc \ c \ (awp \ d \ Q) \ s \ \& \ vc \ d \ Q \ s)$
 $\mid vc \ (Aif \ b \ c \ d) \ Q = (\lambda s. \ vc \ c \ Q \ s \ \& \ vc \ d \ Q \ s)$
 $\mid vc \ (Awhile \ b \ I \ c) \ Q = (\lambda s. \ (I \ s \ \& \ \sim b \ s \longrightarrow Q \ s) \ \& \ (I \ s \ \& \ b \ s \longrightarrow awp \ c \ I \ s) \ \& \ vc \ c \ I \ s)$

primrec *astrip* :: *acom* \Rightarrow *com*
where
 $astrip \ Askip = SKIP$
 $\mid astrip \ (Aass \ x \ a) = (x := a)$
 $\mid astrip \ (Asemi \ c \ d) = (astrip \ c; astrip \ d)$
 $\mid astrip \ (Aif \ b \ c \ d) = (IF \ b \ THEN \ astrip \ c \ ELSE \ astrip \ d)$

$$\mid \textit{astrip} \ (A\textit{while} \ b \ I \ c) = (\textit{WHILE} \ b \ \textit{DO} \ \textit{astrip} \ c)$$
$$\mathbf{primrec} \ vcawp :: acom \Rightarrow assn \Rightarrow assn \times assn$$

where

$$\begin{aligned}
& \text{vcawp } \text{Askip } Q = (\lambda s. \text{True}, Q) \\
& | \text{vcawp } (\text{Aass } x \ a) \ Q = (\lambda s. \text{True}, \lambda s. Q(s[x \mapsto a \ s])) \\
& | \text{vcawp } (\text{Asemi } c \ d) \ Q = (\text{let } (vcd, wpd) = \text{vcawp } d \ Q; \\
& \quad (vcc, wpc) = \text{vcawp } c \ wpd \\
& \quad \text{in } (\lambda s. vcc \ s \ \& \ vcd \ s, wpc)) \\
& | \text{vcawp } (\text{Aif } b \ c \ d) \ Q = (\text{let } (vcd, wpd) = \text{vcawp } d \ Q; \\
& \quad (vcc, wpc) = \text{vcawp } c \ Q \\
& \quad \text{in } (\lambda s. vcc \ s \ \& \ vcd \ s, \\
& \quad \lambda s. (b \ s \ \longrightarrow wpc \ s) \ \& \ (\sim b \ s \ \longrightarrow wpd \ s))) \\
& | \text{vcawp } (\text{Awhile } b \ I \ c) \ Q = (\text{let } (vcc, wpc) = \text{vcawp } c \ I \\
& \quad \text{in } (\lambda s. (I \ s \ \& \ \sim b \ s \ \longrightarrow Q \ s) \ \& \\
& \quad (I \ s \ \& \ b \ s \ \longrightarrow wpc \ s) \ \& \ vcc \ s, I))
\end{aligned}$$

```
declare hoare.conseq [intro]
```

lemma *vc-sound*: $(ALL\ s.\ vc\ c\ Q\ s) \implies |- \{awp\ c\ Q\}\ astrip\ c\ \{Q\}$
 $\langle proof \rangle$

lemma *awp-mono*:

$$\langle proof \rangle \quad (!s. P \ s \ \longrightarrow \ Q \ s) \implies awp \ c \ P \ s \implies awp \ c \ Q \ s$$

lemma *vc-mono*:

$$\langle proof \rangle \quad (!s. P \ s \ \longrightarrow \ Q \ s) \implies vc \ c \ P \ s \implies vc \ c \ Q \ s$$

lemma *vc-complete*: **assumes** *der*: $\vdash \{P\}c\{Q\}$

shows $(\exists ac. \text{astrip } ac = c \ \& \ (\forall s. \text{vc } ac \ Q \ s) \ \& \ (\forall s. P \ s \dashrightarrow \text{awp } ac \ Q \ s))$
(is $? \text{ac. } ?Eq \ P \ c \ Q \ ac)$
 $\langle \text{proof} \rangle$

lemma *vcawp-vc-awp*: $vcawp\ c\ Q = (vc\ c\ Q,\ awp\ c\ Q)$
⟨proof⟩

end

theory *Denotation* imports *Natural* begin

```
types com-den = (state × state)set
```

definition

$\text{Gamma} :: [\text{bexp}, \text{com-den}] \Rightarrow (\text{com-den} \Rightarrow \text{com-den})$ **where**
 $\text{Gamma } b \text{ } cd = (\lambda phi. \{(s,t). (s,t) \in (cd \text{ } O \text{ } phi) \wedge b \text{ } s\} \cup \{(s,t). s=t \wedge \neg b \text{ } s\})$

primrec $C :: \text{com} \Rightarrow \text{com-den}$

where

$C\text{-skip}: C \text{ } SKIP = Id$
 $| C\text{-assign}: C (x ::= a) = \{(s,t). t = s[x \mapsto a(s)]\}$
 $| C\text{-comp}: C (c0; c1) = C(c0) \text{ } O \text{ } C(c1)$
 $| C\text{-if}: C (IF \text{ } b \text{ } THEN \text{ } c1 \text{ } ELSE \text{ } c2) = \{(s,t). (s,t) \in C \text{ } c1 \wedge b \text{ } s\} \cup \{(s,t). (s,t) \in C \text{ } c2 \wedge \neg b \text{ } s\}$
 $| C\text{-while}: C(WHILE \text{ } b \text{ } DO \text{ } c) = \text{lfp } (\text{Gamma } b \text{ } (C \text{ } c))$

lemma $\text{Gamma-mono}: \text{mono } (\text{Gamma } b \text{ } c)$
 $\langle \text{proof} \rangle$

lemma $C\text{-While-If}: C(WHILE \text{ } b \text{ } DO \text{ } c) = C(IF \text{ } b \text{ } THEN \text{ } c; WHILE \text{ } b \text{ } DO \text{ } c \text{ } ELSE \text{ } SKIP)$
 $\langle \text{proof} \rangle$

lemma $\text{com1}: \langle c, s \rangle \longrightarrow_c t \implies (s, t) \in C(c)$

$\langle \text{proof} \rangle$

lemma $\text{com2}: (s, t) \in C(c) \implies \langle c, s \rangle \longrightarrow_c t$
 $\langle \text{proof} \rangle$

lemma $\text{denotational-is-natural}: (s, t) \in C(c) = (\langle c, s \rangle \longrightarrow_c t)$
 $\langle \text{proof} \rangle$

end

theory

program-based-testing

imports

IMP-2011/VC

IMP-2011/Denotation

Testing

begin

3.4 Unfold and its Correctness

The core of our white box testing function is the following “unwind” function, that “unfolds” while loops and normalizes the resulting program in order to expose it to the operational semantics (i.e. the “natural semantics” *evalc* up to an unwind factor *k*. Evaluating programs leads to accumulating path-conditions: If a remaining constraint (whose components essentially result from applications of the *If-split* rule), is satisfiable that a path through a program is traceable and results to a certain successor state.

This can be used to test program specifications: Hoare-Triples were checked against for all paths up to a certain depth.

primrec *Append* :: $[com, com] \Rightarrow com$ (**infixr** @@ 70)

where

conc-skip : $SKIP \text{ @@ } c = c$
| *conc-ass* : $(x ::= E) \text{ @@ } c = ((x ::= E); c)$
| *conc-semi* : $(c; d) \text{ @@ } e = (c; d \text{ @@ } e)$
| *conc-If* : $(IF\ b\ THEN\ c\ ELSE\ d) \text{ @@ } e =$
 $(IF\ b\ THEN\ c \text{ @@ } e\ ELSE\ d \text{ @@ } e)$
| *conc-while*: $(WHILE\ b\ DO\ c) \text{ @@ } e = ((WHILE\ b\ DO\ c); e)$

lemma *C-skip-cancel1*[simp] : $C(SKIP; c) = C(c)$

<proof>

lemma *C-skip-cancel2*[simp] : $C(c; SKIP) = C(c)$

<proof>

lemma *C-If-semi*[simp] :

$C((IF\ x\ THEN\ c\ ELSE\ d); e) = C(IF\ x\ THEN\ (c; e)\ ELSE\ (d; e))$

<proof>

lemma *comappend-correct* [simp]: $C(c \text{ @@ } d) = C(c; d)$

<proof>

fun *unfold* :: $nat \times com \Rightarrow com$

where

uf-skip : $unfold(n, SKIP) = SKIP$
| *uf-ass* : $unfold(n, a ::= E) = (a ::= E)$
| *uf-If* : $unfold(n, IF\ b\ THEN\ c\ ELSE\ d) =$
 $IF\ b\ THEN\ unfold(n, c)\ ELSE\ unfold(n, d)$
| *uf-while*: $unfold(n, WHILE\ b\ DO\ c) =$
 $(if\ 0 < n$
 $then\ IF\ b\ THEN\ unfold(n, c) \text{ @@ } unfold(n-1, WHILE\ b\ DO\ c)\ ELSE\ SKIP$
 $else\ WHILE\ b\ DO\ unfold(0, c))$
| *uf-semi1*: $unfold(n, SKIP ; c) = unfold(n, c)$
| *uf-semi2*: $unfold(n, c ; SKIP) = unfold(n, c)$
| *uf-semi3*: $unfold(n, (IF\ b\ THEN\ c\ ELSE\ d) ; e) =$

$(IF\ b\ THEN\ (unfold(n,c;e))\ ELSE\ (unfold(n,d;e)))$
 $|\ uf-semi4: unfold(n, (c ; d); e) = (unfold(n, c;d))@@(unfold(n,e))$
 $| \ uf-semi5: unfold(n, c ; d) = (unfold(n, c))@@(unfold(n, d))$

lemma *unfold-correct-aux1* :
assumes $H : \forall x. C\ (unfold\ (x, c)) = C\ c$
shows $C(unfold(n, WHILE\ b\ DO\ c)) = C(WHILE\ b\ DO\ c)$
 $\langle proof \rangle$

declare *uf-while* [simp del]

lemma *unfold-correct-aux2* :
 $C(unfold(n,c;d)) = C(unfold(n,c) ; unfold(n, d))$
 $\langle proof \rangle$ **print-cases**
 $\langle proof \rangle$

lemma *unfold-correct* [rule-format]: $\forall x. (C(unfold(x,c)) = C(c))$
 $\langle proof \rangle$

lemma *wp-unfold* : $wp\ (c)\ (p) = wp(unfold(n,c))\ (p)$
 $\langle proof \rangle$

lemma *wp-test* : $\forall \sigma. P\ \sigma \longrightarrow wp\ (unfold(k,c))\ Q\ \sigma \implies |- \{P\}\ c\ \{Q\}$
 $\langle proof \rangle$

3.5 Symbolic Evaluation Rule-Set

lemma *If-split*:
 $\llbracket b\ s \implies \langle c0, s \rangle \longrightarrow_c s' ;$
 $\neg b\ s \implies \langle c1, s \rangle \longrightarrow_c s' \rrbracket$
 $\implies \langle IF\ b\ THEN\ c0\ ELSE\ c1, s \rangle \longrightarrow_c s'$
 $\langle proof \rangle$

lemma *If-splitE*:
 $\llbracket \langle IF\ b\ THEN\ c\ ELSE\ d, s \rangle \longrightarrow_c s' ;$
 $\llbracket b\ s ; \langle c, s \rangle \longrightarrow_c s' \rrbracket \implies P ;$
 $\llbracket \neg b\ s ; \langle d, s \rangle \longrightarrow_c s' \rrbracket \implies P \rrbracket \implies P$
 $\langle proof \rangle$

3.6 Splitting Rule for program-based Tests

lemma *symbolic-eval-test* :
 $(\vdash \{Pre\} c \{Post\}) =$
 $(\forall s t. \langle unfold (n, c), s \rangle \longrightarrow_c t \longrightarrow Pre\ s \longrightarrow Post\ t)$
 $\langle proof \rangle$

3.7 Tactic Set-up

$\langle ML \rangle$

lemmas *one-point-rules* = *HOL.simp-thms*(39) *HOL.simp-thms*(40)

lemma *IF-split*:
 $\langle IF\ b\ THEN\ c\ ELSE\ d, s \rangle \longrightarrow_c s' =$
 $((b\ s \wedge \langle c, s \rangle \longrightarrow_c s') \vee (\neg b\ s \wedge \langle d, s \rangle \longrightarrow_c s'))$
 $\langle proof \rangle$

lemma *assign-sequence*:
 $\langle a ::= e; c, s \rangle \longrightarrow_c s' = \langle c, s[a \mapsto e\ s] \rangle \longrightarrow_c s'$
 $\langle proof \rangle$

lemmas *symbolic-evaluation* = *IF-split*
Natural.skip *Natural.assign*
Natural.semi *Natural.whileFalse*

thm *symbolic-evaluation*

lemmas *symbolic-evaluation2* = *IF-split* *assign-sequence*
Natural.skip *Natural.assign*
Natural.whileFalse

lemmas *memory-model* = *Fun.fun-upd-other* *HOL.simp-thms*(8)
Fun.fun-upd-same *Fun.fun-upd-triv*

$\langle ML \rangle$

end

theory
squareroot-test

```
imports
  ../.. /src/program-based-testing
begin
```

3.8 The Definition of the Integer-Squareroot Program

```

definition squareroot :: [loc,loc,loc,loc]  $\Rightarrow$  com
where      squareroot tm sqsum i a ==
              (( tm      := ( $\lambda s$ . 1));
               (( sqsum  := ( $\lambda s$ . 1));
               ((i       := ( $\lambda s$ . 0));
               WHILE ( $\lambda s$ . (s sqsum) <= (s a)) DO
               (( i      := ( $\lambda s$ . (s i) + 1));
               ((tm      := ( $\lambda s$ . (s tm) + 2));
               (sqsum := ( $\lambda s$ . (s tm) + (s sqsum)))))))))

```

definition $pre \quad :: \text{assn}$ **where** $pre \equiv \lambda x. \text{True}$

definition $post :: [loc, loc] \Rightarrow assn$

where $post\ a\ i \equiv \lambda\ s. (s\ i) * (s\ i) \leq (s\ a) \wedge s\ a < (s\ i + 1) * (s\ i + 1)$

definition $inv :: [loc, loc, loc, loc] \Rightarrow assn$

where

$$\begin{aligned} inv\ i\ sqsum\ tm\ a &\equiv \lambda s.(s\ i + 1) * (s\ i + 1) = s\ sqsum \\ &\wedge s\ tm = (2 * (s\ i) + 1) \\ &\wedge (s\ i) * (s\ i) <= (s\ a) \end{aligned}$$

3.9 Computing Program Paths and their Path-Constraints

lemma *derive-pathconds*:

$$\begin{aligned} \text{assumes } & \text{no-alias} : sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \wedge \\ & sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge \\ & tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \wedge \\ & a \neq i \wedge i \neq a \end{aligned}$$

shows $\langle \text{unfold}(\mathcal{B}, \text{squareroot } tm \text{ sqsum } i \ a), s \rangle \longrightarrow_c s'$

⟨proof⟩

Summary: With this approach, one can synthesize paths and their conditions.

3.10 Testing Specifications

thm *symbolic-evaluation2*

Slow Motion Interactive Version (for demonstrations).

lemma *whitebox-test*:

$$\begin{aligned} \text{assumes } \text{no-alias}[simp] : & \text{sqsum} \neq i \wedge i \neq \text{sqsum} \wedge \text{tm} \neq \text{sqsum} \wedge \\ & \text{sqsum} \neq \text{tm} \wedge \text{sqsum} \neq a \wedge a \neq \text{sqsum} \wedge \\ & \text{tm} \neq i \wedge i \neq \text{tm} \wedge \text{tm} \neq a \wedge a \neq \text{tm} \wedge \end{aligned}$$

$a \neq i \wedge i \neq a$

shows $|- \{pre\} \text{ squareroot } tm \text{ sqsum } i \ a \ \{post \ a \ i\}$

$\langle proof \rangle$

Automated Version:

lemma *whitebox-test2*:

assumes *no-alias*[*simp*] : $sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \wedge$
 $sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge$
 $tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \wedge$
 $a \neq i \wedge i \neq a$

shows $|- \{pre\} \text{ squareroot } tm \text{ sqsum } i \ a \ \{post \ a \ i\}$

$\langle proof \rangle$

3.11 An Alternative Approach with an On-The-Fly generated Explicit Test-Hyp.

Recall the rules for the computation of weakest preconditions:

Hoare.wp_def: $wp \ ?c \ ?Q == \%s. \text{ ALL } t. (s, t) : C \ ?c \ \rightarrow \ ?Q \ t$

Hoare.wp_If: $wp \ (IF \ ?b \ THEN \ ?c \ ELSE \ ?d) \ ?Q = (\%s. (\?b \ s \ \rightarrow \ wp \ ?c \ ?Q \ s) \ \& \ (\sim \ ?b \ s \ \rightarrow \ wp \ ?d \ ?Q \ s))$

Hoare.wp_Semi: $wp \ (?c; \ ?d) \ ?Q = wp \ ?c \ (wp \ ?d \ ?Q)$

Hoare.wp_Ass: $wp \ (?x \ := \ ?a) \ ?Q = (\%s. \ ?Q \ (s[?x \ := \ ?a \ s]))$

Hoare.wp_SKIP: $wp \ SKIP \ ?Q = ?Q$

lemma *path-exploration-test*:

assumes *no-alias* : $sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \wedge$
 $sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge$
 $tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \wedge$
 $a \neq i \wedge i \neq a$

shows $|- \{pre\} \text{ squareroot } tm \text{ sqsum } i \ a \ \{post \ a \ i\} \langle proof \rangle$

end

References

- [1] HOL-TestGen. <http://www.brucker.ch/projects/hol-testgen/>.
- [2] IMP. <http://isabelle.in.tum.de/library/HOL/IMP/Denotation.html>.
- [3] 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. ACM Press, 2002.

- [4] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Formal Approaches to Testing of Software*, number 3997 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2005.
- [5] A. D. Brucker and B. Wolff. Extensible universes for object-oriented data models. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, number 5142 in Lecture Notes in Computer Science, pages 438–462. Springer-Verlag, Heidelberg, 2008.
- [6] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, Heidelberg, 2009.
- [7] W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, 2004.
- [8] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
- [9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.
- [10] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.