Program-based Testing: A Proof-of-Technology

Burkhart Wolff

October 13, 2012

Abstract

In this paper, 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

Contents

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.

MORE TO COME (COMMENTED OUT IN LATEX)

2 Appendix

3 Syntax of Commands

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

typedecl *loc*
— an unspecified (arbitrary) type of locations (adresses/names) for variables

types
val = *nat* — or anything else, *nat* used in examples
state = *loc* ⇒ *val*
aexp = *state* ⇒ *val*
bexp = *state* ⇒ *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

4 Natural Semantics of Commands

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

4.1 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*] => *maplet* (- /|->/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[|->]/ -)
:: *maplet* => *maplets* (-)
-Maplets :: [*maplet*, *maplets*] => *maplets* (-,/ -)
-MapUpd :: ['*a* ~=> '*b*, *maplets*] => '*a* ~=> '*b* (-/'(-') [*900*,*0*]*900*)
-Map :: *maplets* => '*a* ~=> '*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:

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

($\bigwedge$ 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*!]: $\langle \texttt{SKIP},s\rangle \longrightarrow_c s'$
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*!]: $\langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s'$
inductive-cases *whileE* [*elim*]: $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c s'$

The next proofs are all trivial by rule inversion.

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

lemma *ifTrue*:
$b\ s \Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s' = \langle c0,s\rangle \longrightarrow_c s'$
by *auto*

lemma *ifFalse*:
$\neg b\ s \Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s' = \langle c1,s\rangle \longrightarrow_c s'$
by *auto*

lemma *whileFalse*:
$\neg\ b\ s \Longrightarrow \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c s' = (s' = s)$
by *auto*

lemma *whileTrue*:
$b\ s \Longrightarrow$
$\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s\rangle \longrightarrow_c s' =$

$(\exists s''.\ \langle c,s\rangle \longrightarrow_c s'' \wedge \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s''\rangle \longrightarrow_c s')$
by *auto*

Again, Isabelle may use these rules in automatic proofs:

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

4.2 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* ⇒ *com* ⇒ *bool* (- ∼ - [*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*!]:
$(\bigwedge s\ s'.\ \langle c,\ s\rangle \longrightarrow_c s' = \langle c',\ s\rangle \longrightarrow_c s') \Longrightarrow c \sim c'$
by (*unfold equiv-c-def*) *blast*

lemma *equivD1*:
$c \sim c' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s'$
by (*unfold equiv-c-def*) *blast*

lemma *equivD2*:
$c \sim c' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s'$
by (*unfold equiv-c-def*) *blast*

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

lemma *unfold-while*:
$(\texttt{WHILE}\ b\ \texttt{DO}\ c) \sim (\texttt{IF}\ b\ \texttt{THEN}\ c;\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{ELSE SKIP})$ (**is** *?w* ∼ *?if*)
proof −
— to show the equivalence, we look at the derivation tree for
— each side and from that construct a derivation tree for the other side
{ fix *s s'* **assume** *w*: $\langle ?w,\ s\rangle \longrightarrow_c s'$
— as a first thing we note that, if *b* is *False* in state *s*,
— then both statements do nothing:
hence $\neg b\ s \Longrightarrow s = s'$ **by** *blast*
hence $\neg b\ s \Longrightarrow \langle ?if,\ s\rangle \longrightarrow_c s'$ **by** *blast*
moreover
— on the other hand, if *b* is *True* in state *s*,
— then only the *WhileTrue* rule can have been used to derive $\langle ?w,\ s\rangle \longrightarrow_c s'$

```
    { assume b: b s
      with w obtain s'' where
        ⟨c, s⟩ ⟶c s'' and ⟨?w, s''⟩ ⟶c s' by (cases set: evalc) auto
      — now we can build a derivation tree for the IF
      — first, the body of the True-branch:
      hence ⟨c; ?w, s⟩ ⟶c s' by (rule Semi)
      — then the whole IF
      with b have ⟨?if, s⟩ ⟶c s' by (rule IfTrue)
    }
    ultimately
    — both cases together give us what we want:
    have ⟨?if, s⟩ ⟶c s' by blast
  }
  moreover
  — now the other direction:
  { fix s s' assume if: ⟨?if, s⟩ ⟶c s'
    — again, if b is False in state s, then the False-branch
    — of the IF is executed, and both statements do nothing:
    hence ¬b s ⟹ s = s' by blast
    hence ¬b s ⟹ ⟨?w, s⟩ ⟶c s' by blast
    moreover
    — on the other hand, if b is True in state s,
    — then this time only the IfTrue rule can have be used
    { assume b: b s
      with if have ⟨c; ?w, s⟩ ⟶c s' by (cases set: evalc) auto
      — and for this, only the Semi-rule is applicable:
      then obtain s'' where
        ⟨c, s⟩ ⟶c s'' and ⟨?w, s''⟩ ⟶c s' by (cases set: evalc) auto
       — with this information, we can build a derivation tree for the
WHILE
      with b
      have ⟨?w, s⟩ ⟶c s' by (rule WhileTrue)
    }
    ultimately
    — both cases together again give us what we want:
    have ⟨?w, s⟩ ⟶c s' by blast
  }
  ultimately
  show ?thesis by blast
qed
```

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

```
lemma
  (WHILE b DO c) ∼ (IF b THEN c; WHILE b DO c ELSE SKIP)
by blast

lemma triv-if:
  (IF b THEN c ELSE c) ∼ c
```

by *blast*

lemma *commute-if*:
(*IF b1 THEN* (*IF b2 THEN c11 ELSE c12*) *ELSE c2*)
∼
(*IF b2 THEN* (*IF b1 THEN c11 ELSE c2*) *ELSE* (*IF b1 THEN c12 ELSE c2*))
by *blast*

lemma *while-equiv*:
⟨*c0*, *s*⟩ ⟶$_c$ *u* ⟹ *c* ∼ *c'* ⟹ (*c0* = *WHILE b DO c*) ⟹ ⟨*WHILE b DO c'*, *s*⟩ ⟶$_c$ *u*
by (*induct rule*: *evalc.induct*) (*auto simp add*: *equiv-c-def*)

lemma *equiv-while*:
c ∼ *c'* ⟹ (*WHILE b DO c*) ∼ (*WHILE b DO c'*)
by (*simp add*: *equiv-c-def*) (*metis equiv-c-def while-equiv*)

Program equivalence is an equivalence relation.

lemma *equiv-refl*:
c ∼ *c*
by *blast*

lemma *equiv-sym*:
c1 ∼ *c2* ⟹ *c2* ∼ *c1*
by (*auto simp add*: *equiv-c-def*)

lemma *equiv-trans*:
c1 ∼ *c2* ⟹ *c2* ∼ *c3* ⟹ *c1* ∼ *c3*
by (*auto simp add*: *equiv-c-def*)

Program constructions preserve equivalence.

lemma *equiv-semi*:
c1 ∼ *c1'* ⟹ *c2* ∼ *c2'* ⟹ (*c1*; *c2*) ∼ (*c1'*; *c2'*)
by (*force simp add*: *equiv-c-def*)

lemma *equiv-if*:
c1 ∼ *c1'* ⟹ *c2* ∼ *c2'* ⟹ (*IF b THEN c1 ELSE c2*) ∼ (*IF b THEN c1' ELSE c2'*)
by (*force simp add*: *equiv-c-def*)

lemma *while-never*: ⟨*c*, *s*⟩ ⟶$_c$ *u* ⟹ *c* ≠ *WHILE* (λ*s*. *True*) *DO c1*
apply (*induct rule*: *evalc.induct*)
apply *auto*
done

lemma *equiv-while-True*:
(*WHILE* (λ*s*. *True*) *DO c1*) ∼ (*WHILE* (λ*s*. *True*) *DO c2*)
by (*blast dest*: *while-never*)

4.3 Execution is deterministic

This proof is automatic.

theorem $\langle c,s\rangle \longrightarrow_c t \Longrightarrow \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = t$
by (*induct arbitrary*: *u rule*: *evalc.induct*) *blast+*

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 $u = t$
  using *assms*
proof (*induct arbitrary*: *u set*: *evalc*)
  fix *s u* **assume** $\langle$`SKIP`$,s\rangle \longrightarrow_c u$
  thus $u = s$ **by** *blast*
next
  fix *a s x u* **assume** $\langle x :== a,s\rangle \longrightarrow_c u$
  thus $u = s[x \mapsto a\ s]$ **by** *blast*
next
  fix *c0 c1 s s1 s2 u*
  assume *IH0*: $\bigwedge u.\ \langle c0,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
  assume *IH1*: $\bigwedge u.\ \langle c1,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

  assume $\langle c0;c1,\ s\rangle \longrightarrow_c u$
  then obtain s' **where**
      c0: $\langle c0,s\rangle \longrightarrow_c s'$ **and**
      c1: $\langle c1,s'\rangle \longrightarrow_c u$
    by *auto*

  from *c0 IH0* **have** $s'=s2$ **by** *blast*
  with *c1 IH1* **show** $u=s1$ **by** *blast*
next
  fix *b c0 c1 s s1 u*
  assume *IH*: $\bigwedge u.\ \langle c0,s\rangle \longrightarrow_c u \Longrightarrow u = s1$

  assume *b s* **and** $\langle$`IF` *b* `THEN` *c0* `ELSE` *c1*$,s\rangle \longrightarrow_c u$
  hence $\langle c0,\ s\rangle \longrightarrow_c u$ **by** *blast*
  with *IH* **show** $u = s1$ **by** *blast*
next
  fix *b c0 c1 s s1 u*
  assume *IH*: $\bigwedge u.\ \langle c1,s\rangle \longrightarrow_c u \Longrightarrow u = s1$

  assume $\neg b\ s$ **and** $\langle$`IF` *b* `THEN` *c0* `ELSE` *c1*$,s\rangle \longrightarrow_c u$
  hence $\langle c1,\ s\rangle \longrightarrow_c u$ **by** *blast*
  with *IH* **show** $u = s1$ **by** *blast*
next
  fix *b c s u*
  assume $\neg b\ s$ **and** $\langle$`WHILE` *b* `DO` *c*$,s\rangle \longrightarrow_c u$
  thus $u = s$ **by** *blast*
next

fix *b c s s1 s2 u*
assume IH_c: $\bigwedge u.\ \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
assume IH_w: $\bigwedge u.\ \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume *b s* **and** $\langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s\rangle \longrightarrow_c u$
then obtain s' **where**
c: $\langle c,s\rangle \longrightarrow_c s'$ **and**
w: $\langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s'\rangle \longrightarrow_c u$
by *auto*

from *c* IH_c **have** $s' = s2$ **by** *blast*
with *w* IH_w **show** $u = s1$ **by** *blast*
qed

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 $u = t$
using *assms*
proof (*induct arbitrary*: *u*)
— the simple `SKIP` case for demonstration:
fix *s u* **assume** $\langle \mathtt{SKIP},s\rangle \longrightarrow_c u$
thus $u = s$ **by** *blast*
next
— and the only really interesting case, `WHILE`:
fix *b c s s1 s2 u*
assume IH_c: $\bigwedge u.\ \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
assume IH_w: $\bigwedge u.\ \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume *b s* **and** $\langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s\rangle \longrightarrow_c u$
then obtain s' **where**
c: $\langle c,s\rangle \longrightarrow_c s'$ **and**
w: $\langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c,s'\rangle \longrightarrow_c u$
by *auto*

from *c* IH_c **have** $s' = s2$ **by** *blast*
with *w* IH_w **show** $u = s1$ **by** *blast*
qed *blast*+ — prove the rest automatically

end

5 Inductive Definition of Hoare Logic

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*↦*a* *s*])} *x*:==*a* {*P*}
| *semi*: [| |– {*P*}*c*{*Q*}; |– {*Q*}*d*{*R*} |] ==> |– {*P*} *c*;*d* {*R*}
| *If*: [| |– {%*s*. *P s* & *b s*}*c*{*Q*}; |– {%*s*. *P s* & ~*b s*}*d*{*Q*} |] ==>
|– {*P*} `IF` *b* `THEN` *c* `ELSE` *d* {*Q*}
| *While*: |– {%*s*. *P s* & *b s*} *c* {*P*} ==>
|– {*P*} `WHILE` *b* `DO` *c* {%*s*. *P s* & ~*b s*}
| *conseq*: [| !*s*. *P*′ *s* –-> *P s*; |– {*P*}*c*{*Q*}; !*s*. *Q s* –-> *Q*′ *s* |] ==>
|– {*P*′}*c*{*Q*′}

lemma *strengthen-pre*: [| !*s*. *P*′ *s* –-> *P s*; |– {*P*}*c*{*Q*} |] ==> |– {*P*′}*c*{*Q*}
by (*blast intro*: *conseq*)

lemma *weaken-post*: [| |– {*P*}*c*{*Q*}; !*s*. *Q s* –-> *Q*′ *s* |] ==> |– {*P*}*c*{*Q*′}
by (*blast intro*: *conseq*)

lemma *While*′:
assumes |– {%*s*. *P s* & *b s*} *c* {*P*} **and** *ALL s*. *P s* & ¬ *b s* ⟶ *Q s*
shows |– {*P*} `WHILE` *b* `DO` *c* {*Q*}
by(*rule weaken-post*[*OF While*[*OF assms*(*1*)] *assms*(*2*)])

lemmas [*simp*] = *skip ass semi If*

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

end

6 Soundness and Completeness wrt Operational Semantics

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

definition
hoare-valid :: [*assn*,*com*,*assn*] => *bool* (|= {(*1*-)}/ (-)/ {(*1*-)} *50*)
where
|= {*P*}*c*{*Q*} = (!*s t*. ⟨*c*,*s*⟩ ⟶$_c$ *t* –-> *P s* –-> *Q t*)

lemma *hoare-sound*: |– {*P*}*c*{*Q*} ==> |= {*P*}*c*{*Q*}
proof(*induct rule*: *hoare.induct*)

case (*While P b c*)
{ fix *s t*
assume $\langle$*WHILE b DO c*,*s*$\rangle \longrightarrow_c t$
hence $P\ s \longrightarrow P\ t \wedge \neg\ b\ t$
proof(*induct WHILE b DO c s t*)
case *WhileFalse* **thus** *?case* **by** *blast*
next
case *WhileTrue* **thus** *?case*
using *While*(*2*) **unfolding** *hoare-valid-def* **by** *blast*
qed

}
thus *?case* **unfolding** *hoare-valid-def* **by** *blast*
qed (*auto simp*: *hoare-valid-def*)

definition
wp :: *com* => *assn* => *assn* **where**
wp c Q = (%*s*. !*t*. $\langle$*c*,*s*$\rangle \longrightarrow_c t$ --> *Q t*)

lemma *wp-SKIP*: *wp* `SKIP` *Q* = *Q*
by (*simp add*: *wp-def*)

lemma *wp-Ass*: *wp* (*x*:==*a*) *Q* = (%*s*. *Q*(*s*[*x*↦*a s*]))
by (*simp add*: *wp-def*)

lemma *wp-Semi*: *wp* (*c*;*d*) *Q* = *wp c* (*wp d Q*)
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-If*:
wp (`IF` *b* `THEN` *c* `ELSE` *d*) *Q* = (%*s*. (*b s* --> *wp c Q s*) & (~*b s*
--> *wp d Q s*))
by (*rule ext*) (*auto simp*: *wp-def*)

lemma *wp-While-If*:
wp (`WHILE` *b* `DO` *c*) *Q s* =
wp (*IF b THEN c*;`WHILE` *b* `DO` *c ELSE SKIP*) *Q s*
unfolding *wp-def* **by** (*metis equivD1 equivD2 unfold-while*)

lemma *wp-While-True*: *b s* ==>
wp (`WHILE` *b* `DO` *c*) *Q s* = *wp* (*c*;`WHILE` *b* `DO` *c*) *Q s*
by(*simp add*: *wp-While-If wp-If wp-SKIP*)

lemma *wp-While-False*: ~*b s* ==> *wp* (`WHILE` *b* `DO` *c*) *Q s* = *Q s*
by(*simp add*: *wp-While-If wp-If wp-SKIP*)

lemmas [*simp*] = *wp-SKIP wp-Ass wp-Semi wp-If wp-While-True wp-While-False*

lemma *wp-is-pre*: |- {*wp c Q*} *c* {*Q*}
proof(*induct c arbitrary*: *Q*)

```
  case SKIP show ?case by auto
next
  case Assign show ?case by auto
next
  case Semi thus ?case by(auto intro: semi)
next
  case (Cond b c1 c2)
  let ?If = IF b THEN c1 ELSE c2
  show ?case
  proof(rule If)
    show |- {λs. wp ?If Q s ∧ b s} c1 {Q}
    proof(rule strengthen-pre[OF - Cond(1)])
      show ∀ s. wp ?If Q s ∧ b s ⟶ wp c1 Q s by auto
    qed
    show |- {λs. wp ?If Q s ∧ ¬ b s} c2 {Q}
    proof(rule strengthen-pre[OF - Cond(2)])
      show ∀ s. wp ?If Q s ∧ ¬ b s ⟶ wp c2 Q s by auto
    qed
  qed
next
  case (While b c)
  let ?w = WHILE b DO c
  have |- {wp ?w Q} ?w {λs. wp ?w Q s ∧ ¬ b s}
  proof(rule hoare.While)
    show |- {λs. wp ?w Q s ∧ b s} c {wp ?w Q}
    proof(rule strengthen-pre[OF - While(1)])
      show ∀ s. wp ?w Q s ∧ b s ⟶ wp c (wp ?w Q) s by auto
    qed
  qed
  thus ?case
  proof(rule weaken-post)
    show ∀ s. wp ?w Q s ∧ ¬ b s ⟶ Q s by auto
  qed
qed

lemma hoare-relative-complete: assumes |= {P}c{Q} shows |- {P}c{Q}
proof(rule strengthen-pre)
  show ∀ s. P s ⟶ wp c Q s using assms
    by (auto simp: hoare-valid-def wp-def)
  show |- {wp c Q} c {Q} by(rule wp-is-pre)
qed

end
```

7 Verification Conditions

```
theory VC imports Hoare-Op begin
```

```
datatype  acom = Askip
          | Aass   loc aexp
          | Asemi  acom acom
          | Aif    bexp acom acom
          | Awhile bexp assn acom

primrec awp :: acom => assn => assn
where
  awp Askip Q = Q
| awp (Aass x a) Q = (λs. Q(s[x↦a s]))
| awp (Asemi c d) Q = awp c (awp d Q)
| awp (Aif b c d) Q = (λs. (b s-->awp c Q s) & (~b s-->awp d
Q s))
| awp (Awhile b I c) Q = I

primrec vc :: acom => assn => assn
where
  vc Askip Q = (λs. True)
| vc (Aass x a) Q = (λs. True)
| vc (Asemi c d) Q = (λs. vc c (awp d Q) s & vc d Q s)
| vc (Aif b c d) Q = (λs. vc c Q s & vc d Q s)
| vc (Awhile b I c) Q = (λs. (I s & ~b s --> Q s) &
                        (I s & b s --> awp c I s) & vc c I s)

primrec astrip :: acom => com
where
  astrip Askip = SKIP
| astrip (Aass x a) = (x:==a)
| astrip (Asemi c d) = (astrip c;astrip d)
| astrip (Aif b c d) = (IF b THEN astrip c ELSE astrip d)
| astrip (Awhile b I c) = (WHILE b DO astrip c)


primrec vcawp :: acom => assn => assn × assn
where
  vcawp Askip Q = (λs. True, Q)
| vcawp (Aass x a) Q = (λs. True, λs. Q(s[x↦a s]))
| vcawp (Asemi c d) Q = (let (vcd,wpd) = vcawp d Q;
                            (vcc,wpc) = vcawp c wpd
                        in (λs. vcc s & vcd s, wpc))
| vcawp (Aif b c d) Q = (let (vcd,wpd) = vcawp d Q;
                            (vcc,wpc) = vcawp c Q
                        in (λs. vcc s & vcd s,
                            λs.(b s --> wpc s) & (~b s --> wpd s)))
| vcawp (Awhile b I c) Q = (let (vcc,wpc) = vcawp c I
                           in (λs. (I s & ~b s --> Q s) &
                                   (I s & b s --> wpc s) & vcc s, I))
```

declare *hoare.conseq* [*intro*]

lemma *vc-sound*: $(ALL\ s.\ vc\ c\ Q\ s) \Longrightarrow \vdash \{awp\ c\ Q\}\ astrip\ c\ \{Q\}$
proof(*induct c arbitrary*: *Q*)
case (*Awhile b I c*)
show *?case*
proof(*simp*, *rule While'*)
from ‹$\forall s.\ vc\ (Awhile\ b\ I\ c)\ Q\ s$›
have *vc*: $ALL\ s.\ vc\ c\ I\ s$ **and** *IQ*: $ALL\ s.\ I\ s \wedge \neg\ b\ s \longrightarrow Q\ s$ **and**
awp: $ALL\ s.\ I\ s\ \&\ b\ s\ --{>}\ awp\ c\ I\ s$ **by** *simp-all*
from *vc* **have** $\vdash \{awp\ c\ I\}\ astrip\ c\ \{I\}$ **using** *Awhile.hyps* **by** *blast*
with *awp* **show** $\vdash \{\lambda s.\ I\ s \wedge b\ s\}\ astrip\ c\ \{I\}$
by(*rule strengthen-pre*)
show $\forall s.\ I\ s \wedge \neg\ b\ s \longrightarrow Q\ s$ **by**(*rule IQ*)
qed
qed *auto*

lemma *awp-mono*:
$(!s.\ P\ s\ --{>}\ Q\ s) =={>}\ awp\ c\ P\ s =={>}\ awp\ c\ Q\ s$
proof (*induct c arbitrary*: *P Q s*)
case *Asemi* **thus** *?case* **by** *simp metis*
qed *simp-all*

lemma *vc-mono*:
$(!s.\ P\ s\ --{>}\ Q\ s) =={>}\ vc\ c\ P\ s =={>}\ vc\ c\ Q\ s$
proof(*induct c arbitrary*: *P Q*)
case *Asemi* **thus** *?case* **by** *simp* (*metis awp-mono*)
qed *simp-all*

lemma *vc-complete*: **assumes** *der*: $\vdash \{P\}c\{Q\}$
shows $(\exists ac.\ astrip\ ac = c\ \&\ (\forall s.\ vc\ ac\ Q\ s)\ \&\ (\forall s.\ P\ s\ --{>}\ awp\ ac\ Q\ s))$
(**is** *? ac. ?Eq P c Q ac*)
using *der*
proof *induct*
case *skip*
show *?case* (**is** *? ac. ?C ac*)
proof show *?C Askip* **by** *simp* **qed**
next
case (*ass P x a*)
show *?case* (**is** *? ac. ?C ac*)
proof show *?C*(*Aass x a*) **by** *simp* **qed**
next
case (*semi P c1 Q c2 R*)
from *semi.hyps* **obtain** *ac1* **where** *ih1*: *?Eq P c1 Q ac1* **by** *fast*

from *semi.hyps* **obtain** *ac2* **where** *ih2*: *?Eq Q c2 R ac2* **by** *fast*
show *?case* (**is** *? ac. ?C ac*)
proof
show *?C*(*Asemi ac1 ac2*)
using *ih1 ih2* **by** *simp* (*fast elim*!: *awp-mono vc-mono*)
qed
next
case (*If P b c1 Q c2*)
from *If.hyps* **obtain** *ac1* **where** *ih1*: *?Eq* (%*s*. *P s* & *b s*) *c1 Q ac1* **by** *fast*
from *If.hyps* **obtain** *ac2* **where** *ih2*: *?Eq* (%*s*. *P s* & ~*b s*) *c2 Q ac2* **by** *fast*
show *?case* (**is** *? ac. ?C ac*)
proof
show *?C*(*Aif b ac1 ac2*)
using *ih1 ih2* **by** *simp*
qed
next
case (*While P b c*)
from *While.hyps* **obtain** *ac* **where** *ih*: *?Eq* (%*s*. *P s* & *b s*) *c P ac* **by** *fast*
show *?case* (**is** *? ac. ?C ac*)
proof show *?C*(*Awhile b P ac*) **using** *ih* **by** *simp* **qed**
next
case *conseq* **thus** *?case* **by**(*fast elim*!: *awp-mono vc-mono*)
qed

lemma *vcawp-vc-awp*: *vcawp c Q* = (*vc c Q*, *awp c Q*)
by (*induct c arbitrary*: *Q*) (*simp-all add*: *Let-def*)

end

8 Denotational Semantics of Commands

theory *Denotation* **imports** *Natural* **begin**

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

definition
Gamma :: [*bexp*,*com-den*] => (*com-den* => *com-den*) **where**
Gamma b cd = (λ*phi*. {(*s*,*t*). (*s*,*t*) ∈ (*cd O phi*) ∧ *b s*} ∪
{(*s*,*t*). *s*=*t* ∧ ¬*b s*})

primrec *C* :: *com* => *com-den*
where
C-skip: *C* `SKIP` = *Id*
| *C-assign*: *C* (*x* :== *a*) = {(*s*,*t*). *t* = *s*[*x*↦*a*(*s*)]}
| *C-comp*: *C* (*c0*;*c1*) = *C*(*c0*) *O C*(*c1*)

| *C-if*: C (`IF` b `THEN` $c1$ `ELSE` $c2$) = $\{(s,t).\ (s,t) \in C\ c1 \wedge b\ s\} \cup \{(s,t).\ (s,t) \in C\ c2 \wedge \neg b\ s\}$
| *C-while*: C(`WHILE` b `DO` c) = *lfp* (*Gamma* b (C c))

lemma *Gamma-mono*: *mono* (*Gamma* b c)
by (*unfold Gamma-def mono-def*) *fast*

lemma *C-While-If*: C(`WHILE` b `DO` c) = C(`IF` b `THEN` c;`WHILE` b `DO` c `ELSE SKIP`)
apply *simp*
apply (*subst lfp-unfold* [*OF Gamma-mono*]) — lhs only
apply (*simp add*: *Gamma-def*)
done

lemma *com1*: $\langle c,s\rangle \longrightarrow_c t \Longrightarrow (s,t) \in C(c)$

apply (*induct set*: *evalc*)
apply *auto*

apply (*unfold Gamma-def*)
apply (*subst lfp-unfold*[*OF Gamma-mono*, *simplified Gamma-def*])
apply *fast*
apply (*subst lfp-unfold*[*OF Gamma-mono*, *simplified Gamma-def*])
apply *auto*
done

lemma *com2*: $(s,t) \in C(c) \Longrightarrow \langle c,s\rangle \longrightarrow_c t$
apply (*induct c arbitrary*: *s t*)
apply *auto*
apply *blast*

apply (*erule lfp-induct2* [*OF - Gamma-mono*])
apply (*unfold Gamma-def*)
apply *auto*
done

lemma *denotational-is-natural*: $(s,t) \in C(c) = (\langle c,s\rangle \longrightarrow_c t)$
by (*fast elim*: *com2 dest*: *com1*)

end

9 A Proof-Of-Technology of Program-based Testing: The Framework

theory
program-based-testing
imports
IMP−2011/VC
IMP−2011/Denotation
Testing

begin

9.1 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*] ⇒ *com* (**infixr** @@ *70*)
where
conc-skip : *SKIP* @@ *c* = *c*
| *conc-ass* : (*x*:== *E*) @@ *c* = ((*x*:== *E*); *c*)
| *conc-semi* : (*c*;*d*) @@ *e* = (*c*; *d* @@ *e*)
| *conc-If* : (*IF b THEN c ELSE d*) @@ *e* =
(*IF b THEN c* @@ *e ELSE d* @@ *e*)
| *conc-while*: (*WHILE b DO c*) @@ *e* = ((*WHILE b DO c*);*e*)

lemma *C-skip-cancel1*[*simp*] : *C*(*SKIP*;*c*) = *C*(*c*)
by (*simp add*: *Denotation.C.simps Id-O-R R-O-Id*)

lemma *C-skip-cancel2*[*simp*] : *C*(*c*;*SKIP*) = *C*(*c*)
by (*simp add*: *Denotation.C.simps Id-O-R R-O-Id*)

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

```
    C((IF x THEN c ELSE d);e) = C(IF x THEN (c;e) ELSE (d;e))
  by auto

lemma comappend-correct [simp]: C(c @@ d) = C(c;d)
  apply(induct c)
  apply(simp-all only: C-If-semi conc-If)
  apply(simp-all add: Relation.O-assoc)
  done

fun unfold :: nat × com ⇒ 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)@@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 : ∀ x. C (unfold (x, c))   = C c
  shows       C(unfold(n,WHILE b DO c)) = C(WHILE b DO c)
proof (induct n)
  case 0 then show ?case
      by(simp add: Denotation.C.simps H)
next
  case (Suc n) then show ?case
      apply(subst uf-while,subst if-P, simp)
      apply(rule-tac s = n and t = Suc n − 1 in subst,arith)
      apply(simp only: Denotation.C.simps  comappend-correct)
      apply(simp only: Denotation.C.simps [symmetric] H)
      apply(simp only: Denotation.C-While-If)
      done
qed

declare uf-while [simp del]

lemma unfold-correct-aux2 :
  C(unfold(n,c;d))= C(unfold(n,c) ; unfold(n, d))
```

proof (*induct c*) **print-cases**
  case *SKIP* **then show** *?case* **by**(*simp*)
next
  case (*Assign loc E*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
next
  case (*Semi c1 c2*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
next
  case (*Cond cond then-branch else-branch*) **then show** *?case*
      apply (*case-tac d*, *simp-all*)
      apply (*simp-all only*: *C.simps*[*symmetric*] *C-If-semi*)
      by *auto*
next
  case (*While cond body*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
qed

lemma *unfold-correct* [*rule-format*]: ∀ *x*. (*C*(*unfold*(*x*,*c*)) = *C*(*c*))
proof(*induct c*)
  case *SKIP* **then show** *?case* **by** *simp*
next
  case (*Assign loc E*) **then show** *?case* **by** *simp*
next
  case (*Semi c1 c2*) **then show** *?case*
      by (*cases c1*, *cases c2*,
          simp-all add: *unfold-correct-aux2*)
next
 case (*Cond cond then-branch else-branch*) **then show** *?case* **by** *simp*
next
  case (*While cond body*) **then show** *?case*
      by (*intro allI unfold-correct-aux1*,*auto*)
qed

lemma *wp-unfold* : *wp* (*c*) (*p*) = *wp*(*unfold*(*n*,*c*)) (*p*)
 by(*simp add*: *wp-def unfold-correct denotational-is-natural*[*symmetric*])

lemma *wp-test* : ∀ σ. *P* σ ⟶ *wp* (*unfold*(*k*,*c*)) *Q* σ ⟹ |− {*P*} *c* {*Q*}
  apply (*rule Hoare.strengthen-pre*)
  apply (*simp add*: *wp-unfold*[*symmetric*])
  apply (*rule wp-is-pre*)
  done

9.2 Symbolic Evaluation Rule-Set

lemma *If-split*:
⟦ $b\ s \Longrightarrow \langle c0,s\rangle \longrightarrow_c s'$;
  $\neg\ b\ s \Longrightarrow \langle c1,s\rangle \longrightarrow_c s'$ ⟧
  $\Longrightarrow \langle IF\ b\ THEN\ c0\ ELSE\ c1,s\rangle \longrightarrow_c s'$
  by (*cases b s*, *simp-all*)

lemma *If-splitE*:
  ⟦ $\langle IF\ b\ THEN\ c\ ELSE\ d,s\rangle \longrightarrow_c s'$;
    ⟦ $b\ s$; $\langle c,s\rangle \longrightarrow_c s'$ ⟧ $\Longrightarrow P$;
    ⟦ $\neg\ b\ s$; $\langle d,s\rangle \longrightarrow_c s'$ ⟧ $\Longrightarrow P$ ⟧ $\Longrightarrow P$

  by(*cases b s*, *simp-all*)

9.3 Splitting Rule for program-based Tests

lemma *symbolic-eval-test* :
  ($\vdash$ {*Pre*} *c* {*Post*}) =
    ($\forall\ s\ t.\ \langle unfold\ (n,\ c),s\rangle \longrightarrow_c t \longrightarrow Pre\ s \longrightarrow Post\ t$)
proof −
  have *hoare-sound-complete* : ($\vdash$ {*Pre*} *c* {*Post*}) = ($\models$ {*Pre*} *c* {*Post*})
    by(*auto intro*!: *hoare-sound hoare-relative-complete*)
  show *?thesis*
    by(*simp only*: *hoare-sound-complete hoare-valid-def*
        denotational-is-natural[*symmetric*] *unfold-correct*)
qed

9.4 Tactic Set-up

ML⟪

```
fun contains-eval n thm =
    let fun T(Natural.evalc,-) = true | T - = false
    in  Term.exists-Const T (term-of(cprem-of thm n)) end
```

⟫
ML⟪
```
local open TestGen in

fun thyp-ify-partial-evaluations ctxt =
    (COND' contains-eval (thyp-ify ctxt) (K all-tac))

end
```

⟫

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

lemma *IF-split*:
⟨*IF b THEN c ELSE d*,*s*⟩ ⟶$_c$ *s'* =
((*b s* ∧ ⟨*c* ,*s*⟩ ⟶$_c$ *s'*) ∨ (¬ *b s* ∧ ⟨*d* ,*s*⟩ ⟶$_c$ *s'*))
by(*cases b s*, *auto*)

lemma *assign-sequence*:
⟨*a*:== *e*; *c*,*s*⟩ ⟶$_c$ *s'* = ⟨*c*,*s*[*a* ↦ *e s*]⟩ ⟶$_c$ *s'*
by(*simp only*:*Natural.semi Natural.assign one-point-rules*)

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

ML⟪

local open TestGen HOLogic in

fun generate-program-splitter ctxt simps depth no thm =
let val thy = *theory-of-thm thm*
val @{*term Hoare.hoare*} $ *PRE* $ *PROG* $ - = *dest-Trueprop*(*term-of*(*cprem-of thm 1*));
val S = (*Thm.trivial* (*cterm-of thy* (*mk-Trueprop*
(@{*term Hoare.hoare*} $ *PRE* $ *PROG* $
Free(*POSTCONDITION*,
@{*typ* (*Com.loc* ⇒ *nat*) ⇒ *bool*})))))
val S = *S* |$> (*res-inst-tac ctxt* (∗ [(*n1*, *Int.toString depth*)] ∗)
[((*n*,*1*),*Int.toString depth*)]
(@{*thm symbolic-eval-test*} *RS iffD2*) *no*)
|$> (*safe-tac ctxt*)
|$> (*asm-full-simp-tac*((*global-simpset-of thy*)
addsimps
(@{*thms Append.simps*} @

```
                @{thms unfold.simps} @
                [@{thm uf-while}])) 1)
            |$> (asm-full-simp-tac( HOL-ss
                addsimps
               (@{thms symbolic-evaluation2} @
                @{thms memory-model} @ simps @
                [@{thm update-def}])) 1)
            |$> (safe-tac ctxt)
            |$> (ALLGOALS(COND' contains-eval (thyp-ify ctxt)
(K all-tac)))

   in  thm |> (rtac (Drule.export-without-context S) 1)
  end

end (* local *)

⟫

end
```

10 Program-based Testing: The Squareroot-Example.

```
theory
  squareroot-test
imports
  ../../src/program-based-testing
begin
```

10.1 The Definition of the Integer-Squareroot Program

```
definition squareroot :: [loc,loc,loc,loc] ⇒ com
where     squareroot tm sqsum i a ==
                          ((  tm   :== (λs. 1));
                           (( sqsum  :== (λs. 1));
                            ((i    :== (λs. 0));
                              WHILE (λs. (s sqsum) <= (s a)) DO
                                (( i    :== (λs. (s i) + 1));
                                 ((tm   :== (λs. (s tm) + 2));
                                       (sqsum  :== (λs. (s tm) + (s
sqsum)))))))
                          )


definition  pre   :: assn where   pre  ≡  λ x. True
```

definition *post* :: [*loc*,*loc*] ⇒ *assn*
where *post a i* ≡ λ *s*. (*s i*)∗(*s i*)≤(*s a*) ∧ *s a* < (*s i* + *1*)∗(*s i* + *1*)

definition *inv* :: [*loc*,*loc*,*loc*,*loc*] ⇒ *assn*
where *inv i sqsum tm a* ≡ λ*s*.(*s i* + *1*) ∗ (*s i* + *1*) = *s sqsum*
∧ *s tm* = (*2* ∗ (*s i*) + *1*)
∧ (*s i*) ∗ (*s i*) <= (*s a*)

10.2 Computing Program Paths and their Path-Constraints

lemma *derive-pathconds*:
assumes *no-alias* : *sqsum* ≠ *i* ∧ *i* ≠ *sqsum* ∧ *tm* ≠ *sqsum* ∧
sqsum ≠ *tm* ∧ *sqsum* ≠ *a* ∧ *a* ≠ *sqsum* ∧
tm ≠ *i* ∧ *i* ≠ *tm* ∧ *tm* ≠ *a* ∧ *a* ≠ *tm* ∧
a ≠ *i* ∧ *i* ≠ *a*
shows ⟨*unfold*(*3*, *squareroot tm sqsum i a*), *s*⟩ ⟶$_c$ *s'*

apply(*simp add*: *squareroot-def uf-while*)
apply(*rule If-split*,*simp-all add*: *update-def no-alias*)+

The resulting proof state capturing the test hypothesis as well as the resulting 4 evaluation paths (no entry into loop, 1 pass, 2 passes and 3 passes through the loop) looks as follows:

1. *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))) ≤ *s a* ⟹
⟨`WHILE` λ*s*. *s sqsum*
≤ *s a* `DO` *i* :== λ*s*. *Suc* (*s i*) ; (*tm* :== λ*s*.
Suc (*Suc* (*s tm*)) ; *sqsum* :== λ*s*. *s tm* + *s sqsum*),*s*
(*i* := *Suc* (*Suc* (*Suc 0*)),
tm := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))),
sqsum :=
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))))⟩
⟶$_c$ *s'*
2. ⟦*Suc* (*Suc* (*Suc* (*Suc 0*))) ≤ *s a*;
¬ *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))) ≤ *s a*⟧
⟹ *s'* = *s*
(*i* := *Suc* (*Suc 0*), *tm* := *Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))),
sqsum := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
0)))))))))
3. ⟦*Suc 0* ≤ *s a*; ¬ *Suc* (*Suc* (*Suc* (*Suc 0*))) ≤ *s a*⟧
⟹ *s'* = *s*
(*i* := *Suc 0*, *tm* := *Suc* (*Suc* (*Suc 0*)),
sqsum := *Suc* (*Suc* (*Suc* (*Suc 0*))))
4. ¬ *Suc 0* ≤ *s a* ⟹ *s'* = *s*(*tm* := *Suc 0*, *sqsum* := *Suc 0*, *i* := *0*)

oops

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

10.3 Testing Specifications

thm *symbolic-evaluation2*

Slow Motion Interactive Version (for demonstrations).

lemma *whitebox-test*:
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 $\vdash \{pre\}$ *squareroot tm sqsum i a* $\{post\ a\ i\}$

apply(*simp add*: *squareroot-def pre-def*)
apply(*rule-tac n1 = 3* **in** *iffD2*[*OF symbolic-eval-test*])

apply(*safe*, *simp add*: *Append.simps unfold.simps uf-while*)
apply(*simp only*: *symbolic-evaluation2*
memory-model no-alias update-def,
safe)
apply(*tactic ALLGOALS*(*TestGen.COND'* *contains-eval*
(*TestGen.thyp-ify* @{*context*})
(*K all-tac*)))

apply(*simp-all*)

sorry

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 $\vdash \{pre\}$ *squareroot tm sqsum i a* $\{post\ a\ i\}$

apply(*simp add*: *squareroot-def pre-def*)
apply(*tactic generate-program-splitter* @{*context*} (@{*thms no-alias*})
3 1)

apply(*simp-all*)

The resulting proof state captures the essence of this white box test:

1. THYP

```
(∀ x xa xb xc xd xe xf.
    ⟨WHILE λs. s x
                ≤ s xa DO xb :== λs. Suc (s
  xb) ; (xc :== λs. Suc (Suc (s xc)) ; x :== λs. s xc + s x ),xf
    (xb := Suc (Suc (Suc 0)),
     xc := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
     x := Suc (Suc (Suc (Suc (Suc (Suc
 (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
    ⟶c xe ⟶
    Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ xf
xa ⟶
    xd xe)
2. ⋀s. ⟦Suc (Suc (Suc 0))) ≤ s a;
     ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤
s a⟧
    ⟹ post a i
         (s(i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc
0)))),
            sqsum :=
              Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))
3. ⋀s. ⟦Suc 0 ≤ s a; ¬ Suc (Suc (Suc 0))) ≤ s a⟧
    ⟹ post a i
         (s(i := Suc 0, tm := Suc (Suc (Suc 0)),
            sqsum := Suc (Suc (Suc (Suc 0)))))
4. ⋀s. ¬ Suc 0 ≤ s a ⟹ post a i (s(tm := Suc 0, sqsum := Suc 0,
i := 0))
```

Now testing all paths for compliance to post condition:

apply(*simp-all add*: *no-alias post-def*)

In this special case—arithmetic constraints—the system can even **verify** these constraints, i.e. the simplifier shows that all postconditions follow from the initial constraints and the computed relation between pre-state and post state.

```
1. THYP
    (∀ x xa xb xc xd xe xf.
        ⟨WHILE λs. s x
                    ≤ s xa DO xb :== λs. Suc (s
      xb) ; (xc :== λs. Suc (Suc (s xc)) ; x :== λs. s xc + s x ),xf
        (xb := Suc (Suc (Suc 0)),
         xc := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
         x := Suc (Suc (Suc (Suc (Suc (Suc
      (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
        ⟶c xe ⟶
        Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ xf
xa ⟶
        xd xe)
```

To say it loud and clearly: The white box test decomposes the original

specification into a test hypothesis for cases with $3^3 = 9 <= sa$ and all other cases (e.g. $2^2 = 4 <= sa \wedge sa < 9$). The latter have been proven automatically.

oops

10.4 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. ALL t. (s, t) : C ?c --> ?Q t

Hoare.wp_If:   wp (IF ?b THEN ?c ELSE ?d) ?Q = (%s. (?b s --> wp ?c ?Q s) &
                                                  (~ ?b s --> 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 $\vdash \{pre\}$ *squareroot tm sqsum i a* $\{post\ a\ i\}$

We fire the basic white-box scenario:

apply (*rule wp-test* [*of - 3*])

Given the concrete unfolding factor and the concrete program term, standard normalization yields an "Path Exhaustion Theorem" with the explicit test hypothesis:

apply(*auto simp*: *squareroot-def update-def no-alias uf-while*)
apply(*tactic ALLGOALS*(*TestGen.COND'* *contains-eval*
(*TestGen.thyp-ify* @{*context*})
(*K all-tac*)))

and we reach the following instantiation of a white-box test-theorem (with explicit test-hypothesis for the uncovered paths):

1. $\bigwedge\sigma.$ $[\![pre\ \sigma;\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))$
$\leq \sigma\ a]\!]$
$\Longrightarrow wp\ (\texttt{WHILE}\ \lambda s.\ s\ sqsum$
$\leq s\ a\ \texttt{DO}\ i :== \lambda s.$
$Suc\ (s\ i)\ ;\ (tm :== \lambda s.\ Suc\ (Suc\ (s\ tm))\ ;\ sqsum :== \lambda s.\ s\ tm + s$
$sqsum\ ))$
$(post\ a\ i)$
$(\sigma(i := Suc\ (Suc\ (Suc\ 0)),$
$tm := Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))),$
$sqsum :=$

$Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$(Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))))))))$
2. $\bigwedge\sigma.\ [\![pre\ \sigma;\ Suc\ (Suc\ (Suc\ (Suc\ 0)))) \leq \sigma\ a;$
$\neg\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))\ \leq$
$\sigma\ a]\!]$
$\Longrightarrow post\ a\ i$
$(\sigma(i := Suc\ (Suc\ 0),\ tm := Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$0)))),$
$sqsum :=$
$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))$
3. $\bigwedge\sigma.\ [\![pre\ \sigma;\ Suc\ 0 \leq \sigma\ a;\ \neg\ Suc\ (Suc\ (Suc\ (Suc\ 0)))) \leq \sigma\ a]\!]$
$\Longrightarrow post\ a\ i$
$(\sigma(i := Suc\ 0,\ tm := Suc\ (Suc\ (Suc\ 0)),$
$sqsum := Suc\ (Suc\ (Suc\ (Suc\ 0)))))$
4. $\bigwedge\sigma.\ [\![pre\ \sigma;\ \neg\ Suc\ 0 \leq \sigma\ a]\!]$
$\Longrightarrow post\ a\ i\ (\sigma(tm := Suc\ 0,\ sqsum := Suc\ 0,\ i := 0))$

Now we allso perform the "tests" by symbolic execution:

apply(*auto simp*: *no-alias pre-def post-def*)

which leaves us just with test-hypothesis case; for all paths *not* leading to a remaining while, the program is correct.

1. $\bigwedge\sigma.\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))) \leq \sigma\ a \Longrightarrow$
$wp\ (\texttt{WHILE}\ \lambda s.\ s\ sqsum$
$\leq s\ a\ \texttt{DO}\ i :== \lambda s.\ Suc\ (s\ i)\ ;\ (tm :== \lambda s.\ Suc\ (Suc\ (s\ tm))\ ;\ sqsum :== \lambda s.\ s\ tm + s\ sqsum))$
$(\lambda s.\ s\ i * s\ i \leq s\ a \wedge s\ a < Suc\ (s\ i + (s\ i + s\ i * s\ i)))$
$(\sigma(i := Suc\ (Suc\ (Suc\ 0)),$
$tm := Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))),$
$sqsum :=$
$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$(Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))))))))$

oops

end