Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff[1]

[1]Université Paris-Sud, LRI, Orsay, France
wolff@lri.fr

A Tutorial at the LRI
Orsay, 15th Jan 2009

Outline

Outline

State of the Art

“Dijkstra’s Verdict”:

Program testing can be used to show the presence of bugs, but never to show their absence.

- Is this always true?
- Can we bother?

Our First Vision

Testing and verification may converge,
in a precise technical sense:

- specification-based (black-box) unit testing
- generation and management of formal test hypothesis
- verification of test hypothesis (not discussed here)

Our Second Vision

- Observation:
  Any testcase-generation technique is based on and limited by underlying constraint-solution techniques.
- Approach:
  Testing should be integrated in an environment combining automated and interactive proof techniques.
- the test engineer must decide over, abstraction level, split rules, breadth and depth of data structure exploration ...
- we mistrust the dream of a push-button solution
- byproduct: a verified test-tool

Components of HOL-TestGen

- HOL (Higher-order Logic):
  - "Functional Programming Language with Quantifiers"
  - plus definitional libraries on Sets, Lists, . . .
  - can be used meta-language for Hoare Calculus for Java, Z, . . .
- HOL-TestGen:
  - based on the interactive theorem prover Isabelle/HOL
  - implements these visions
- Proof General:
  - user interface for Isabelle and HOL-TestGen
  - step-wise processing of specifications/theories
  - shows current proof states

Components-Overview

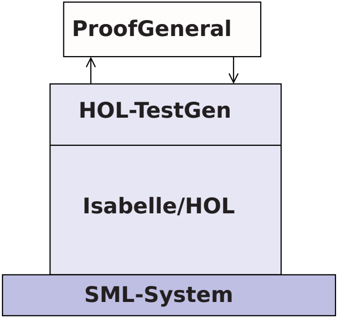


Figure: The Components of HOL-TestGen

The HOL-TestGen Workflow

The HOL-TestGen workflow is basically fivefold:

1. *Step I:* writing a test theory (in HOL)
2. *Step II:* writing a test specification (in the context of the test theory)
3. *Step III:* generating a test theorem (roughly: testcases)
4. *Step IV:* generating test data
5. *Step V:* generating a test script

And of course:

- building an executable test driver
- and running the test driver

Step I: Writing a Test Theory

- Write data types in HOL:

```
theory List_test
imports Testing
begin

datatype 'a list =
     Nil     ("[]")
   | Cons 'a "'a list"     (infixr "#" 65)
```

Step I: Writing a Test Theory

- Write recursive functions in HOL:

```
consts is_sorted:: "('a::ord) list ⇒bool"
primrec
  "is_sorted []       = True"
  "is_sorted (x#xs) = case xs of
                          [] ⇒ True
                        | y#ys ⇒((x < y) ∨(x = y))
                                  ∧ is_sorted xs"
```

Step II: Write a Test Specification

- writing a test specification (TS)
  as HOL-TestGen command:

test_spec "is_sorted (prog (l::('a list)))"

Step III: Generating Testcases

- executing the testcase generator in form of an Isabelle proof method:

  apply(gen_test_cases "prog")

- concluded by the command:

  store_test_thm "test_sorting"

  . . . that binds the current proof state as test theorem to the name `test_sorting`.

Step III: Generating Testcases

- The test theorem contains clauses (the test-cases):

  is_sorted (prog [])
  is_sorted (prog [?X1X17])
  is_sorted (prog [?X2X13, ?X1X12])
  is_sorted (prog [?X3X7, ?X2X6, ?X1X5])

- as well as clauses (the test-hypothesis):

  $THYP((\exists x.\ is\_sorted\ (prog\ [x])) \longrightarrow (\forall x.\ is\_sorted(prog\ [x])))$
  . . .
  $THYP((\forall l.\ 4 < |l| \longrightarrow is\_sorted(prog\ l))$

- We will discuss these hypothesises later in great detail.

Step IV: Test Data Generation

- On the test theorem,
  all sorts of logical massages can be performed.
- Finally, a test data generator can be executed:

  gen_test_data "test_sorting"

- The test data generator
  - extracts the testcases from the test theorem
  - searches ground instances satisfying the constraints (none in the example)
- Resulting in test statements like:

  is_sorted (prog [])
  is_sorted (prog [3])
  is_sorted (prog [6, 8])
  is_sorted (prog [0, 10, 1])

Step V: Generating A Test Script

- Finally, a test script or test harness can be generated:

  gen_test_script "test_lists.sml" list" prog

- The generated test script can be used to test an implementation, e. g., in SML, C, or Java

The Complete Test Theory

```
theory List_test
imports Main begin
  consts is_sorted:: "('a::ord) list ⇒bool"
  primrec "is_sorted []      = True"
          "is_sorted  (x#xs) = case xs of
                                  []  ⇒ True
                               |  y#ys ⇒((x < y) ∨(x = y))
                                        ∧ is_sorted xs"

  test_spec "is_sorted (prog (l::('a list)))"
    apply(gen_test_cases prog)
  store_test_thm "test_sorting"

  gen_test_data "test_sorting"
  gen_test_script "test_lists.sml" list" prog
end
```

Testing an Implementation

Executing the generated test script may result in:

```
Test Results:
Test 0 - *** FAILURE: post-condition false, result: [1, 0, 10]
Test 1 -     SUCCESS, result: [6, 8]
Test 2 -     SUCCESS, result: [3]
Test 3 -     SUCCESS, result: []

Summary:
Number successful tests cases:  3 of 4 (ca. 75%)
Number of warnings:             0 of 4 (ca.  0%)
Number of errors:               0 of 4 (ca.  0%)
Number of failures:             1 of 4 (ca. 25%)
Number of fatal errors:         0 of 4 (ca.  0%)

Overall result: failed
```

Tool-Demo!

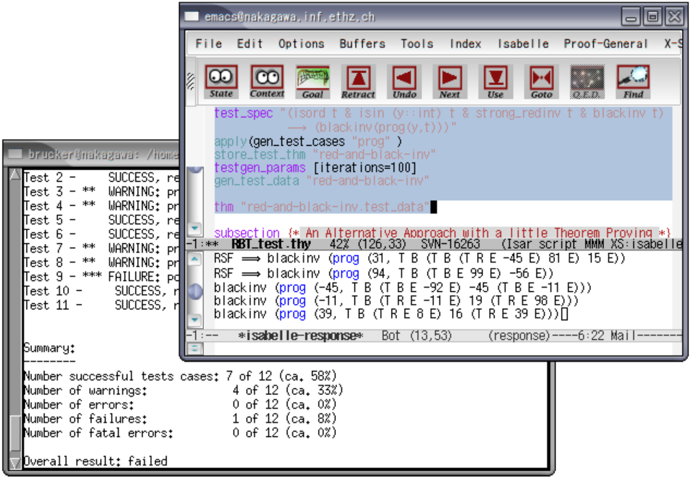


Figure: HOL-TestGen Using Proof General at one Glance

Outline

The Foundations of HOL-TestGen

- Basis:
  - Isabelle/HOL library: 10000 derived rules, . . .
  - about 500 are organized in larger data-structures used by Isabelle's proof procedures, . . .
- These Rules were used in advanced proof-procedures for:
  - Higher-Order Rewriting
  - Tableaux-based Reasoning — a standard technique in automated deduction
  - Arithmetic decision procedures (Coopers Algorithm)
- gen_testcases is an automated tactical program using combination of them.

Some Rewrite Rules

- Rewriting is a easy to understand deduction paradigm (similar FP) centered around equality
- Arithmetic rules, e. g.,

$$\mathrm{Suc}(x + y) = x + \mathrm{Suc}(y)$$
$$x + y = y + x$$
$$\mathrm{Suc}(x) \neq 0$$

- Logic and Set Theory, e. g.,

$$\forall x.\ (P\,x \wedge Q\,x) = (\forall x.\ P\,x) \wedge (\forall x.\ P\,x)$$
$$\bigcup x \in S.\ (P\,x \cup Q\,x) = (\bigcup x \in S.\ P\,x) \cup (\bigcup x \in S.\ Q\,x)$$
$$[\![A = A'; A \Longrightarrow B = B']\!] \Longrightarrow (A \wedge B) = (A' \wedge B')$$

The Core Tableaux-Calculus

- Safe Introduction Rules for logical connectives:

$$\frac{}{t = t} \qquad \frac{}{\text{true}} \qquad \frac{P \quad Q}{P \wedge Q} \qquad \frac{\begin{matrix}[\neg Q]\\ \vdots \\ P\end{matrix}}{P \vee Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \rightarrow Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ \text{false}\end{matrix}}{\neg P} \qquad \ldots$$

- Safe Elimination Rules:

$$\frac{\text{false}}{P} \qquad \frac{P \wedge Q \quad \begin{matrix}[P, Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \vee Q \quad \begin{matrix}[P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \rightarrow Q \quad \begin{matrix}[\neg P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \ldots$$

The Core Tableaux-Calculus

- Safe Introduction Quantifier rules:

$$\frac{P\ ?x}{\exists x.\ P\ x} \qquad \frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}$$

- Safe Quantifier Elimination

$$\frac{\exists x.\ P\ x \qquad \bigwedge x.\ \begin{array}{c}[P\ x]\\ \vdots\\ Q\end{array}}{Q}$$

- Critical Rewrite Rule:

$$\text{if } P \text{ then } A \text{ else } B = (P \rightarrow A) \wedge (\neg P \rightarrow B)$$

Explicit Test Hypothesis: The Concept

- What to do with infinite data-strucutures?
- What is the connection between test-cases and test statements and the test theorems?
- Two problems, one answer: Introducing test hypothesis "on the fly":

$$\mathrm{THYP} : \mathrm{bool} \Rightarrow \mathrm{bool}$$
$$\mathrm{THYP}(x) \equiv x$$

Taming Infinity I: Regularity Hypothesis

- What to do with infinite data-strucutures of type τ?
  Conceptually, we split the set of all data of type τ into

$$\{x :: \tau \mid |x| < k\} \cup \{x :: \tau \mid |x| \geq k\}$$

Taming Infinity I: Motivation

Consider the first set $\{X :: \tau \mid |x| < k\}$
for the case $\tau = \alpha$ list, $k = 2, 3, 4$.
These sets can be presented as:

1) $|x::\tau|<2 = (x = []) \vee (\exists\ a.\ x = [a])$
2) $|x::\tau|<3 = (x = []) \vee (\exists\ a.\ x = [a])$
   $\vee (\exists\ a\ b.\ x = [a,b])$
3) $|x::\tau|<4 = (x = []) \vee (\exists\ a.\ x = [a])$
   $\vee (\exists\ a\ b.\ x = [a,b]) \vee (\exists\ a\ b\ c.\ x = [a,b,c])$

Taming Infinity I: Data Separation Rules

This motivates the (derived) data-separation rule:

- ($\tau = \alpha$ list, $k = 3$):

$$\frac{\begin{matrix}[x = []] \\ \vdots \\ P\end{matrix} \qquad \bigwedge a. \begin{matrix}[x = [a]] \\ \vdots \\ P\end{matrix} \qquad \bigwedge a\, b. \begin{matrix}[x = [a, b]] \\ \vdots \\ P\end{matrix} \qquad \text{THYP } M}{P}$$

- Here, M is an abbreviation for:

$$\forall\, x.\ k < |x| \longrightarrow P\ x$$

Taming Infinity II: Uniformity Hypothesis

- What is the connection between test cases and test statements and the test theorems?
- Well, the "uniformity hypothesis":
- *Once the program behaves correct for one test case, it behaves correct for all test cases ...*

Taming Infinity II: Uniformity Hypothesis

- Using the uniformity hypothesis, a test case:

  n) $\llbracket$ C1 x; ...; Cm x$\rrbracket \Longrightarrow$ TS x

  is transformed into:

  n) $\llbracket$ C1 ?x; ...; Cm ?x$\rrbracket \Longrightarrow$ TS ?x

  n+1) THYP(($\exists$x. C1 x ... Cm x $\longrightarrow$ TS x) $\longrightarrow$ ($\forall$x. C1 x ... Cm x $\longrightarrow$ TS x))

Testcase Generation by NF Computations

Test-theorem is computed out of the test specification by

- a heuristicts applying Data-Separation Theorems
- a rewriting normal-form computation
- a tableaux-reasoning normal-form computation
- shifting variables referring to the program under test prog test into the conclusion, e.g.:

  $$[\![\neg(\text{prog } x = c); \neg(\text{prog } x = d)]\!] \Longrightarrow A$$

  is transformed equivalently into

  $$[\![\neg A]\!] \Longrightarrow (\text{prog } x = c) \vee (\text{prog } x = d)$$

- as a final step, all resulting clauses were normalized by applying uniformity hypothesis to each free variable.

Testcase Generation: An Example

theory TestPrimRec
imports Main
begin
primrec
  x mem [] = False
  x mem (y#S) = if y = x
    then True
    else x mem S

test_spec:
  "x mem S $\Longrightarrow$prog x S"
apply(gen_testcase 0 0)

1) prog x [x]
2) $\bigwedge$b. prog x [x,b]
3) $\bigwedge$a. a$\neq$x$\Longrightarrow$prog x [a,x]
4) THYP(3 $\leq$size (S)
  $\longrightarrow\forall$ x. x mem S
  $\longrightarrow$prog x S)

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

Sample Derivation of Test Theorems

Example

$$x \text{ mem } S \longrightarrow prog\ x\ S$$

is transformed via data-separation lemma to:

1. $S=[] \Longrightarrow x \text{ mem } S \longrightarrow prog\ x\ S$
2. $\bigwedge a.\ S=[a] \Longrightarrow x \text{ mem } S \longrightarrow prog\ x\ S$
3. $\bigwedge a\ b.\ S=[a,b] \Longrightarrow x \text{ mem } S \longrightarrow prog\ x\ S$
4. $THYP(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow prog\ x\ S)$

Sample Derivation of Test Theorems

Example

$$x \text{ mem } S \longrightarrow \text{prog } x\ S$$

canonization leads to:

1. $x \text{ mem } [] \Longrightarrow \text{prog } x\ []$
2. $\bigwedge a.\ x \text{ mem } [a] \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ x \text{ mem } [a,b] \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

which is reduced via the equation for mem:

1. $\text{false} \Longrightarrow \text{prog } x\ []$

2. $\bigwedge a.\ \text{if } a = x \text{ then True}$
   $\quad \text{else } x \text{ mem } [] \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ \text{if } a = x \text{ then True}$
   $\quad \text{else } x \text{ mem } [b] \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

erasure for unsatisfyable constraints and rewriting conditionals yields:

2. $\bigwedge a.\ a = x \vee (a \neq x \wedge \text{false}) \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ a = x \vee (a \neq x \wedge x \text{ mem } [b]) \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

. . . which is further reduced by tableaux rules and canconization to:

2. $\bigwedge a.\ \text{prog } a\ [a]$

3. $\bigwedge a\ b.\ a = x \Longrightarrow \text{prog } x\ [a,b]$

3'. $\bigwedge a\ b.\ [\![\ a \neq x;\ x \text{ mem } [b]\ ]\!] \Longrightarrow \text{prog } x\ [a,b]$

4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

x mem S $\longrightarrow$ prog x S

. . . which is reduced by canonization and rewriting of mem to:

2. $\bigwedge$a. prog x [x]

3. $\bigwedge$a b. prog x [x,b]

3'. $\bigwedge$a b. a$\neq$x $\Longrightarrow$ prog x [a,x]

4. THYP($\forall$ S. 3 $\leq$|S| $\longrightarrow$ x mem S $\longrightarrow$ prog x S)

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

...as a final step, uniformity is expressed:

1. $\text{prog } ?x1\ [?x1]$
2. $\text{prog } ?x2\ [?x2,?b2]$
3. $?a3 \neq ?x1 \Longrightarrow \text{prog } ?x3\ [?a3,?x3]$
4. $\text{THYP}(\exists x.\text{prog } x\ [x] \longrightarrow \text{prog } x\ [x]$

   ...

7. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Summing up:

The test-theorem for a test specification *TS* has the general form:

$$[\![TC_1; \ldots; TC_n; \text{THYP } H_1; \ldots; \text{THYP } H_m]\!] \Longrightarrow TS$$

where the test cases TC_i have the form:

$$[\![C_1 x; \ldots; C_m x; \text{THYP } H_1; \ldots; \text{THYP } H_m]\!] \Longrightarrow P\, x\, (prog\ x)$$

and where the test-hypothesis are either uniformity or regularity hypothethises.
The C_i in a test case were also called constraints of the testcase.

Summing up:

- The overall meaning of the test-theorem is:
  - if the program passes the tests for all test-cases,
  - and if the test hypothesis are valid for *PUT*,
  - then *PUT* complies to testspecification *TS*.
- Thus, the test-theorem establishes a formal link between test and verification !!!

Generating Test Data

Test data generation is now a constraint satisfaction problem.

- We eliminate the meta variables ?x , ?y, . . . by constructing values ("ground instances") satisfying the constraints. This is done by:
  - random testing (for a smaller input space!!!)
  - arithmetic decision procedures
  - reusing pre-compiled abstract test cases
  - . . .
  - interactive simplify and check, if constraints went away!
- Output: Sets of instantiated test theorems (to be converted into Test Driver Code)

Outline

Tuning the Workflow by Interactive Proof

Observations:

- Test-theorem generations is fairly easy ...
- Test-data generation is fairly hard ...
  (it does not really matter if you use random solving
  or just plain enumeration !!!)
- Both are scalable processes . . .
  (via parameters like depth, iterations, ...)
- There are bad and less bad forms of test-theorems !!!
- **Recall**: Test-theorem and test-data generation are normal form computations:
  $\Longrightarrow$ More Rules, better results . . .

What makes a Test-case “Bad”

- redundancy.
- many unsatisfiable constraints.
- many constraints with unclear logical status.
- constraints that are difficult to solve.
  (like arithmetics).

Case Studies: Red-black Trees

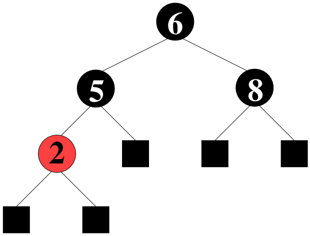
Motivation

Test a non-trivial and widely-used data structure.

- part of the SML standard library
- widely used internally in the sml/NJ compiler, e. g., for providing efficient implementation for Sets, Bags, . . . ;
- very hard to generate (balanced) instances randomly

Modeling Red-black Trees I

Red-Black Trees:



Red Invariant: each red node has a black parent.

Black Invariant: each path from the root to an empty node (leaf) has the same number of black nodes.

datatype

color = R | B

tree = E | T color (α tree) (β::ord item) (α tree)

Modeling Red-black Trees II

- Red-Black Trees: Test Theory

consts

redinv :: tree $\Rightarrow$ bool

blackinv :: tree $\Rightarrow$ bool

recdef blackinv measure (λ t. (size t))

blackinv E = True

blackinv (T color a y b) =

((blackinv a) $\wedge$(blackinv b)

$\wedge$ ((max B (height a)) = (max B (height b))))

recdev redinv measure ...

Red-black Trees: Test Specification

- Red-Black Trees: Test Specification

  test_spec:
  "isord t $\wedge$ redinv t $\wedge$blackinv t
  $\wedge$ isin (y::int) t
  $\longrightarrow$
  (blackinv(prog(y,t)))"

  where prog is the program under test (e. g., delete).

- Using the standard-workflows results, among others:

  RSF $\longrightarrow$blackinv (prog (100, T B E 7 E))
  blackinv (prog ($-$91, T B (T R E $-$91 E) 5 E))

Red-black Trees: A first Summary

Observation:

Guessing (i. e., random-solving) valid red-black trees is difficult.

- On the one hand:
  - random-solving is nearly impossible for solutions which are "difficult" to find
  - only a small fraction of trees with depth *k* are balanced
- On the other hand:
  - we can quite easily construct valid red-black trees interactively.

Red-black Trees: A first Summary

Observation:

Guessing (i. e., random-solving) valid red-black trees is difficult.

- On the one hand:
  - random-solving is nearly impossible for solutions which are "difficult" to find
  - only a small fraction of trees with depth *k* are balanced
- On the other hand:
  - we can quite easily construct valid red-black trees interactively.
- Question:
  Can we improve the test-data generation by using our knowledge about red-black trees?

Red-black Trees: Hierarchical Testing I

Idea:

Characterize valid instances of red-black tree in more detail and use this knowledge to guide the test data generation.

- First attempt:
  enumerate the height of some trees without black nodes

```
lemma maxB_0_1:
  "max_B_height (E:: int tree) = 0"

lemma maxB_0_5:
  "max_B_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"
```

- But this is tedious . . .

Red-black Trees: Hierarchical Testing I

Idea:

Characterize valid instances of red-black tree in more detail and use this knowledge to guide the test data generation.

- First attempt:
  enumerate the height of some trees without black nodes

  lemma maxB_0_1:
  "max_B_height (E:: int tree) = 0"

  lemma maxB_0_5:
  "max_B_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"

- But this is tedious . . . and error-prone

How to Improve Test-Theorems

- New simplification rule establishing unsatisfiability.
- New rules establishing equational constraints for variables.

$$(\text{max\_B\_height } (\text{T } x \text{ t1 val t2}) = 0) \Longrightarrow (x = \text{R})$$

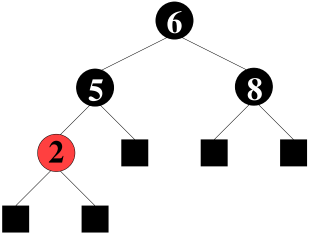
$$\begin{aligned}&(\text{max\_B\_height } x = 0) = \\ &(x = \text{E} \vee \exists\, a\ y\ b.\ x = \text{T R } a\ y\ b \wedge \\ &\qquad\qquad \text{max}(\text{max\_B\_height } a) \\ &\qquad\qquad\quad (\text{max\_B\_height } b) = 0)\end{aligned}$$

- Many rules are domain specific —
  few hope that automation pays really off.

Improvement Slots

- logical massage of test-theorem.
- in-situ improvements:
  add new rules into the context before gen_test_cases.
- post-hoc logical massage of test-theorem.
- in-situ improvements:
  add new rules into the context before **gen_test_data**.

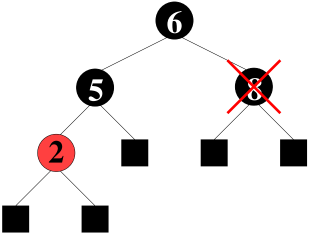
Red-black Trees: sml/NJ Implementation



(a) pre-state

Figure: Test Data for Deleting a Node in a Red-Black Tree

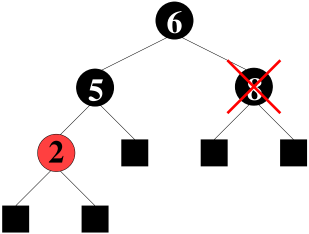
Red-black Trees: sml/NJ Implementation



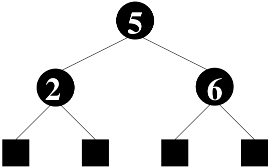
(b) pre-state: delete “8”

Figure: Test Data for Deleting a Node in a Red-Black Tree

Red-black Trees: sml/NJ Implementation



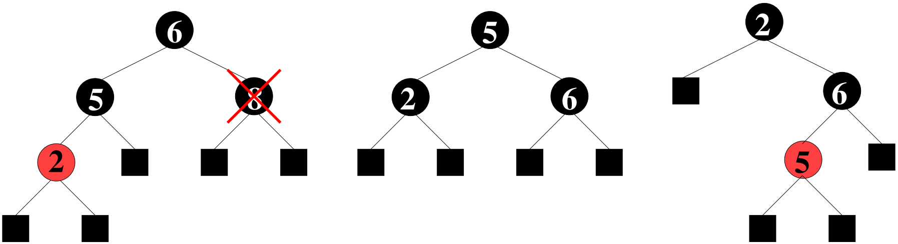
(b) pre-state: delete "8"



(c) correct result

Figure: Test Data for Deleting a Node in a Red-Black Tree

Red-black Trees: sml/NJ Implementation



(b) pre-state: delete "8" (c) correct result (d) result of sml/NJ

Figure: Test Data for Deleting a Node in a Red-Black Tree

Red-black Trees: Summary

- Statistics: 348 test cases were generated (within 2 minutes)
- One error found: crucial violation against red/black-invariants
- Red-black-trees degenerate to linked list (insert/search, etc. only in linear time)
- Not found within 12 years
- Reproduced meanwhile by random test tool

Motivation: Sequence Test

- So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- This seems to limit the HOL-TestGen approach to **UNIT**-tests.

Apparent Limitations of HOL-TestGen

- No Non-determinism.

Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.

Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- No Automata - No Tests for Sequential Behaviour.

Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.

- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .

Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
- No possibility to describe reactive tests.

Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
- HOL has Monads. And therefore means for IO-specifications.

Representing Sequence Test

- Test-Specification Pattern:

  accept trace $\rightarrow$P(Mfold trace σ_0 prog)

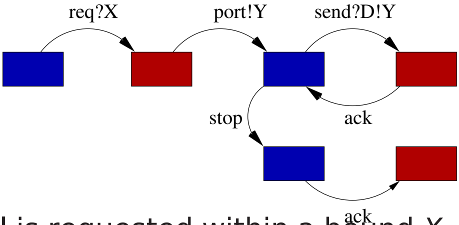
  where

  Mfold [] σ = Some σ
  MFold (input::R) = case prog(input, σ) **of**
  None $\Rightarrow$ None
  | Some σ‘$\Rightarrow$ Mfold R σ’ prog

- Can this be used for reactive tests?

Example: A Reactive System I

- A toy client-server system:



a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel . . .

Example: A Reactive System I

- A toy client-server system:

$$\begin{aligned} &\mathrm{req}?X \to \mathrm{port}!Y[Y < X] \to \\ &\qquad (\mathrm{rec}\, N.\ \mathrm{send}!D.Y \to \mathrm{ack} \to N \\ &\qquad\qquad \Box\ \mathrm{stop} \to \mathrm{ack} \to \mathrm{SKIP}) \end{aligned}$$

a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel . . .

Example: A Reactive System I

- A toy client-server system:

$$\begin{aligned} &\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\ &\qquad (\mathrm{rec}\, N.\, \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\ &\qquad\qquad \Box\, \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP}) \end{aligned}$$

a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel ...

Observation:

X and Y are only known at runtime!

Example: A Reactive System II

Observation:

X and Y are only known at runtime!

- Mfold is a program that manages a state at test run time.
- use an environment that keeps track of the instances of X and Y?
- Infrastructure: An **observer** maps
  abstract events (req X, port Y, ...) in traces
  to
  concrete events (req 4, port 2, ...) in runs!

Example: A Reactive System |||

- Infrastructure: the observer

observer rebind substitute postcond ioprog $\equiv$
(λ input. (λ (σ, σ'). **let** input'= substitute σinput **in**
case ioprog input' σ' **of**
None $\Rightarrow$None *(∗ ioprog failure − eg. timeout ... ∗)*
| Some (output, σ''') $\Rightarrow$**let** σ'' = rebind σoutput **in**
(if postcond (σ'',σ''') input' output
then Some(σ'', σ''')
else None *(∗ postcond failure ∗)*)))"

Example: A Reactive Test IV

- Reactive Test-Specification Pattern:

$$\text{accept}\, \mathit{trace} \longrightarrow P(\text{Mfold}\, \mathit{trace}\, \sigma_0\, (\text{observer rebind}\, \text{subst postcond}\, \mathit{ioprog}))$$

- for reactive systems!

Motivation

- So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- We have seen, this does not exclude to model reactive sequence test in HOL-TestGen.
- However, this seems still exclude the HOL-TestGen approach from program-based testing approaches (such as JavaPathfinder-SE or Pexx).

How to Realize White-box-Tests in HOL-TestGen?

- Fact: HOL is a powerful *logical framework* used to embed all sorts of specification and programming languages.
- Thus, we can embed the language of our choice in HOL-TestGen...
- and derive the necessary rules for symbolic execution based tests ...

The Master-Plan for White-box-Tests in HOL-TestGen?

- We embed an imperative core-language — called IMP — into HOL-TestGen, by defining its syntax and semantics
- We add a specification mechanism for IMP: Hoare-Triples
- we derive rules for symbolic evaluation and loop-unfolding.

IMP Syntax

The (abstract) IMP syntax is defined in Com.thy.

```
Com = Main +
typedecl loc
types
 val    = nat (*arb.*)
 state = loc⇒val
 aexp = state⇒val
 bexp = state⇒bool
```

```
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)
```

The type loc stands for *locations*. Note that expressions are represented as HOL-functions depending on state. The *datatype com* stands for commands (command sequences).

Example: The Integer Square-Root Program

```
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))
```

How does this program work?

Note: There is the implicit assumption, that tm, sum and i are distinct locations, i.e. they are not aliases from each other !

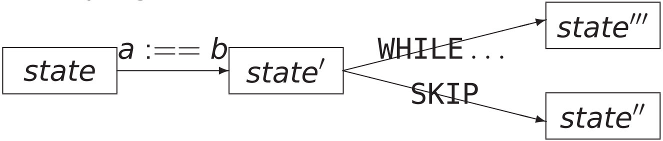
IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

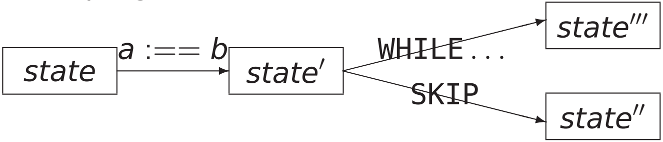
idea: programs relates states.



IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

idea: programs relates states.



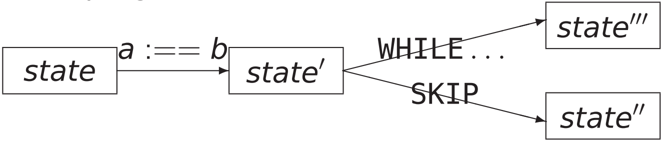
consts evalc :: (com ×state ×state) set

translations "$\langle$c,s$\rangle \underset{c}{\longrightarrow}$ s’ " $\equiv$ "(c,s,s’) $\in$evalc"

IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

idea: programs relates states.



consts evalc :: (com $\times$state $\times$state) set

translations "$\langle$c,s$\rangle \underset{c}{\longrightarrow}$ s' " $\equiv$ "(c,s,s') $\in$evalc"

The transition relation of natural semantics is inductively defined.

The transition relation of natural semantics is inductively defined.

This means intuitively: The evaluation steps defined by the following rules are the *only* possible steps.

The transition relation of natural semantics is inductively defined.

This means intuitively: The evaluation steps defined by the following rules are the *only* possible steps.

Let's go . . .

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \text{SKIP,s} \rangle \underset{c}{\longrightarrow} \text{s}$

Assign: $\langle \text{x :== a,s} \rangle \underset{c}{\longrightarrow} \text{s[x} \mapsto \text{a s]}$

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s\rangle \xrightarrow[c]{} s$

Assign: $\langle x :== a, s\rangle \xrightarrow[c]{} s[x \mapsto a\ s]$

Note that $s[x \mapsto a\ s]$ is an abbreviation for *update s x (a s)*, where

update s x v $\equiv \lambda$y. if y=x then v else s y

The natural semantics as inductive definition:

inductive evalc
intrs
Skip: $\langle \mathrm{SKIP},s\rangle \underset{c}{\longrightarrow} s$
Assign: $\langle x :== a,s\rangle \underset{c}{\longrightarrow} s[x \mapsto a\ s]$

Note that $s[x \mapsto a\ s]$ is an abbreviation for *update s x (a s)*, where

update s x v $\equiv \lambda$y. if y=x then v else s y

Note that a is of type aexp or bexp.

Excursion: A minimal memory model:

$$(s[x \mapsto E])\ x = E$$
$$x \neq y \Longrightarrow (s[x \mapsto E])\ y = s\ y$$

This small memory theory contains the *typical* rules for updating and memory-access. Note that this rewrite system is in fact executable!

The semantics for the sequential composition of statements can be described as follows:

Semi: $[\![\langle c,s\rangle \underset{c}{\longrightarrow} s'; \ \langle c',s'\rangle \underset{c}{\longrightarrow} s'' \ ]\!] \Longrightarrow \langle c;c', \ s\rangle \underset{c}{\longrightarrow} s''$

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } [\![\langle c,s\rangle \underset{c}{\longrightarrow} s';\ \langle c',s'\rangle \underset{c}{\longrightarrow} s'']\!] \Longrightarrow \langle c;c',\ s\rangle \underset{c}{\longrightarrow} s''$$

Rationale of natural semantics:

- if you can "jump" via c from s to s', . . .

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } [\![\langle c,s\rangle \xrightarrow[c]{} s';\ \langle c',s'\rangle \xrightarrow[c]{} s'']\!] \Longrightarrow \langle c;c',\ s\rangle \xrightarrow[c]{} s''$$

Rationale of natural semantics:

- if you can "jump" via c from s to s', ...
- and if you can "jump" via c' from s' to s'' ...

The semantics for the sequential composition of statements can be described as follows:

Semi: $[\![\langle c,s\rangle \underset{c}{\longrightarrow} s';\ \langle c',s'\rangle \underset{c}{\longrightarrow} s'']\!] \Longrightarrow \langle c;c',\ s\rangle \underset{c}{\longrightarrow} s''$

Rationale of natural semantics:

- if you can "jump" via c from s to s', . . .
- and if you can "jump" via c' from s' to s'' . . .
- then this means that you can "jump" via the composition c;c' from c to c''.

The other constructs of the language are treated analogously:

IfTrue: $[\![\ b\ s;\ \langle c,s\rangle \underset{c}{\longrightarrow} s'\ ]\!]$
$\Longrightarrow \langle\ \texttt{IF}\ \ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ c',\ s\rangle \underset{c}{\longrightarrow} s'$

IfFalse: $[\![\ \neg b\ s;\ \langle c',s\rangle \underset{c}{\longrightarrow} s'\ ]\!]$
$\Longrightarrow \langle\ \texttt{IF}\ \ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ c',\ s\rangle \underset{c}{\longrightarrow} s'$

WhileFalse: $[\![\neg b\ s]\!]$
$\Longrightarrow \langle \texttt{WHILE}\ \ b\ \texttt{DO}\ \ c,\ s\rangle \underset{c}{\longrightarrow} s$

WhileTrue: $[\![\ b\ s;\ \langle c,s\rangle \underset{c}{\longrightarrow} s';\langle \texttt{WHILE}\ \ b\ \texttt{DO}\ \ c,s'\rangle \underset{c}{\longrightarrow} s''\ ]\!]$
$\Longrightarrow \langle \texttt{WHILE}\ \ b\ \texttt{DO}\ \ c,\ s\rangle \underset{c}{\longrightarrow} s''$

Note that for non-terminating programs no final state can be derived !

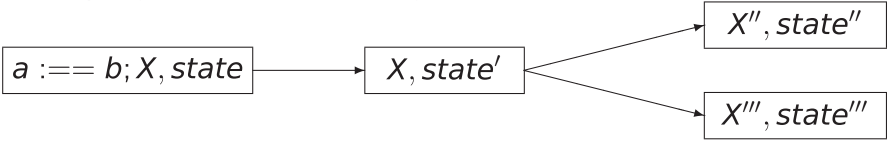
IMP Semantics II: (Transition Semantics)

The transition semantics is inspired by abstract machines.

IMP Semantics II: (Transition Semantics)

The transition semantics is inspired by abstract machines.

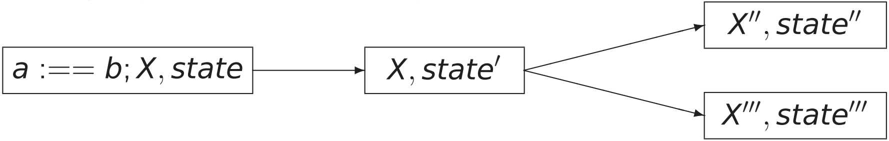
idea: programs relate “configurations”.



IMP Semantics II: (Transition Semantics)

The transition semantics is inspired by abstract machines.

idea: programs relate “configurations”.



consts evalc1 :: ((com $\times$state) $\times$(com $\times$state)) set

translations "cs $-$1$-$> cs’" $\equiv$"(cs,cs’) $\in$evalc1"

inductive evalc1

intro

Assign: (x:==a,s) −1−> (SKIP, s[x ↦a s])

Semi1: (SKIP;c,s) −1−> (c,s)

Semi2: (c,s) −1−> (c'',s')

$\Longrightarrow$ (c;c',s) −1−> (c'';c',s')

inductive evalc1

intro

```
Assign:  (x:==a,s) −1−> (SKIP, s[x ↦a s])
Semi1:   (SKIP;c,s) −1−> (c,s)
Semi2:   (c,s)      −1−> (c'',s')
         ⟹  (c;c',s) −1−> (c'';c',s')
```

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .

inductive evalc1

intro

Assign: (x:==a,s) −1−> (SKIP, s[x ↦a s])

Semi1: (SKIP;c,s) −1−> (c,s)

Semi2: (c,s) −1−> (c'',s')

$\Longrightarrow$ (c;c',s) −1−> (c'';c',s')

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .
- this stack can also be seen as a *program counter* . . .
- transition semantics is close to an abstract machine.

IfTrue:

$b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c', s)

IfFalse:

$\neg b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c'', s)

WhileFalse:

$\neg b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (SKIP, s)

WhileTrue:

$b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (c;WHILE b DO c, s)

IfTrue:

$b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c', s)

IfFalse:

$\neg b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c'', s)

WhileFalse:

$\neg b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (SKIP, s)

WhileTrue:

$b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (c;WHILE b DO c, s)

A non-terminating loop always leads to successor configurations . . .

IMP Semantics III: (Denotational Semantics)

Idea:

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.
As semantic domain we choose the state relation:

types com_den = (state $\times$state) set

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.
As semantic domain we choose the state relation:

types com_den = (state $\times$state) set
and declare the semantic function:

consts C :: com $\Rightarrow$com_den

The semantic function C is defined recursively over the syntax.

primrec

C(SKIP) = Id (* ≡identity relation *)

C(x :== a) = {(s,t). t = s[x↦ a s]}

C(c ; c') = C(c') O C(c) (* ≡seq. composition *)

C(IF b THEN c' ELSE c'') =

{(s,t). (s,t) ∈C(c') ∧b(s)} ∪

{(s,t). (s,t) ∈C(c'') ∧¬b(s)}"

C(WHILE b DO c) = lfp (Γ b (C(c)))"

primrec

C(SKIP)   = Id   (∗ ≡identity relation ∗)

C(x :== a) = {(s,t). t = s[x↦ a s]}

C(c ; c')   = C(c') O C(c)   (∗ ≡seq. composition ∗)

C(IF b THEN c' ELSE c'') =

{(s,t). (s,t) ∈C(c') ∧b(s)} ∪

{(s,t). (s,t) ∈C(c'') ∧¬b(s)}"

C(WHILE b DO c) = lfp (Γ b (C(c)))"

where:

Γ b c ≡(λφ. {(s,t). (s,t) ∈ (φ O c) ∧b(s)} ∪

{(s,t). s=t ∧¬b(s)})

and where the least-fixpoint-operator *lfp F* corresponds in this special case to:

$$\bigcup_{n \in N} F^n$$

IMP Semantics:Theorems I

Theorem: Natural and Transition Semantics Equivalent

$(c, s) -*-> (SKIP, t) = (\langle c,s \rangle \underset{c}{\longrightarrow} t)$

where $cs -*-> cs' \equiv (cs,cs') \in evalc1^*$, i.e. the new arrow denotes the transitive closure over old one.

IMP Semantics:Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c,\ s)\ -{*}-> (\mathrm{SKIP}, t) = (\langle c,s\rangle \underset{c}{\longrightarrow} t)$$

where $cs\ -{*}->\ cs' \equiv (cs,cs') \in evalc1^*$, i.e. the new arrow denotes the transitive closure over old one.

Theorem: Denotational and Natural Semantics Equivalent

$$((s,\ t)\ \in C\ c) = (\langle c,s\rangle \underset{c}{\longrightarrow} t)$$

IMP Semantics:Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c,\ s)\ -*-> (\text{SKIP}, t) = (\langle c,s\rangle \underset{c}{\longrightarrow} t)$$

where $cs\ -*->\ cs' \equiv (cs,cs') \in evalc1^*$, i.e. the new arrow denotes the transitive closure over old one.

Theorem: Denotational and Natural Semantics Equivalent

$$((s,\ t) \in C\ c) = (\langle c,s\rangle \underset{c}{\longrightarrow} t)$$

i.e. all three semantics are closely related !

IMP Semantics:Theorems II

Theorem: Natural Semantics can be evaluated equationally !!!

$\langle \texttt{SKIP}, s \rangle \underset{c}{\longrightarrow} s' \quad = (s' = s)$

$\langle x :== a, s \rangle \underset{c}{\longrightarrow} s' \quad = (s' = s[x \mapsto a\ s])$

$\langle c;\ c',\ s \rangle \underset{c}{\longrightarrow} s' \quad = (\exists s''.\ \langle c, s \rangle \underset{c}{\longrightarrow} s'' \wedge \langle c', s'' \rangle \underset{c}{\longrightarrow} s')$

$\langle \texttt{IF}\ b\ \text{THEN}\ c\ \text{ELSE}\ c',\ s \rangle \underset{c}{\longrightarrow} s' = (b\ s \wedge \langle c, s \rangle \underset{c}{\longrightarrow} s') \vee$
$(\neg b\ s \wedge \langle c', s \rangle \underset{c}{\longrightarrow} s')$

Note: This is the key for evaluating a program symbolically !!!

Example: “a:==2;b:==2*a”

$$\langle \text{a:==}\lambda \text{s. } 2;\ \text{b:==}\lambda\text{s. } 2 * (\text{s a}),\text{s}\rangle \xrightarrow[c]{} \text{s'}$$
$$\equiv (\exists\, \text{s''}.\ \langle \text{a:==}\lambda\text{s. } 2,\text{s}\rangle \xrightarrow[c]{} \text{s''} \wedge \langle \text{b:==}\lambda\text{s. } 2 * (\text{s a}),\text{s''}\rangle \xrightarrow[c]{} \text{s'})$$
$$\equiv (\exists\, \text{s''}.\ \text{s''} = \text{s[a}\mapsto(\lambda\text{s. } 2)\ \text{s]} \wedge \text{s'} = \text{s''[b}\mapsto(\lambda\text{s. } 2 * (\text{s a}))\ \text{s'']})$$
$$\equiv (\exists\, \text{s''}.\ \text{s''} = \text{s[a}\mapsto 2] \wedge \text{s'} = \text{s''[b}\mapsto 2 * (\text{s'' a})])$$
$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 2 * (\text{s[a}\mapsto 2]\ \text{a})]$$
$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 2 * 2]$$
$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 4]$$

Note:

1. The λ-notation is perhaps a bit irritating, but helps to get the nitty-gritty details of substitution right.
2. The forth step is correct due to the “one-point-rule” $(\exists x.\ x = e \wedge P(x)) = P(e)$.
3. This does not work for the loop and for recursion...

IMP Semantics:Theorems III

Denotational semantics makes it easy to prove facts like:

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

etc.

Program Annotations: Assertions revisited.

For our scenario, we need a mechanism to combine programs with their specifications.
The Standard: Hoare-Tripel with Pre- and Post-Conditions a special form of assertions.

types assn = state ⇒bool
consts valid :: (assn ×com ×assn) ⇒bool ("|= {_} _ {_}")

defs
|= {P}c{Q} ≡∀s. ∀t. (s,t) ∈C(c) ⟶P s ⟶ Q t"

Note that this reflects partial correctnes; for a non-terminating program c, i.e. $(s,t) \notin C(c)$, a Hoare-Triple does not enforce anything as post-condition !

Finally: Symbolic Evaluation.

For programs without loop, we have already anything together for symbolic evaluation:

$$\forall\, s\ s'.\ \langle c,s\rangle \underset{c}{\longrightarrow} s' \wedge \quad P\ s \rightarrow Q\ s'$$
$$\Longrightarrow \models \{P\}c\{Q\}$$

or in more formal, natural-deduction notation:

$$\dfrac{\begin{array}{c}[\langle c, s\rangle \rightarrow_c s', P\ s]_{s,s'} \\ \vdots \\ Q\ s'\end{array}}{\models \{P\}\ c\ \{Q\}}$$

Applied in backwards-inference, this rule *generates* the constraints for the states that were amenable to equational evaluation rules shown before.

Example: "$\models \{0 \leq x\}$a:==x;b:==2*a$\{0 \leq b\}$"

$$|= \{\lambda s.\ 0 \leq s\ x\}\ a{:}{=}{=}\lambda s.\ s\ x;\ b{:}{=}{=}\lambda s.\ 2 * (s\ a)\ \{\lambda s.\ 0 \leq s\ b\}$$

$$\Longleftarrow\ s' = s[a \mapsto s\ x][b \mapsto 2 * (s[a \mapsto s\ x]\ a)] \wedge 0 \leq s\ x \longrightarrow 0 \leq s'\ b$$

$$\equiv\ s' = s[a \mapsto s\ x][b \mapsto 2 * (s\ x)] \wedge \text{“PRE s''} \longrightarrow \text{“POST s' ''}$$

$$\equiv\ \text{“PRE s''} \longrightarrow \text{“POST } (s[a \mapsto s\ x][b \mapsto 2 * (s\ x)])\text{ ''}$$

Note:

- Note: the logical constaint $s' = s[a \mapsto s\ x][b \mapsto 2 * s\ x] \wedge 0 \leq s\ x$ consists of the constraint that functionally relate pre-state s to post-state s' and the Path-Condition (in this case just "PRE s'').
- This also works for conditionals ... Revise !
- The implication is actually the core validation problem: It means that for a certain path, we search for the solution of a path condition that validates the post-condition. We can decide to 1) keep it as test hypothesis, 2) test *k* witnesses and add a uniformity hypothesis, or 3) verify it.

Validation of Post-Conditions for a Given Path:

Ad 1 : Add $THYP(PRE\ s \longrightarrow POST(s[a \mapsto s\ x][b \mapsto 2 * (s\ x)]))$ (is: $THYP(0 \leq s\ x \longrightarrow 0 \leq 2 * s\ x)$) as test hypothesis.

Ad 2 : Find witness to $\exists s. 0 \leq s\ x$, run a test on this witness (does it establish the post-condition?) and add the uniformity-hypothesis:
$THYP(\exists s.\ 0 \leq s\ x \longrightarrow 0 \leq 2 * s\ x \longrightarrow \forall s.\ 0 \leq s\ x \longrightarrow 0 \leq 2 * s\ x)$.

Ad 3 : Verify the implication, which is in this case easy.

Option 1 can be used to model weaker coverage criteria than all statements and k loops, option 2 can be significantly easier to show than option 3, but as the latter shows, for simple formulas, testing is not *necessarily* the best solution.

Control-heuristics necessary.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Answer: Unfolding to a certain depth.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Answer: Unfolding to a certain depth.

In the sequel, we define an unfolding function, prove it semantically correct with respect to C, and apply the procedure above again.

Handling Loops (and Recursion).

```
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) ELSEunwind(n
"unwind(n, WHILEb DO c) =
    if  0 < n
      then IF  b THEN unwind(n,c)@@unwind(n− 1,WHILE b DOc) ELSESKI
      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))"
```

Handling Loops (and Recursion).

where 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.

```
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 THENc @@ e ELSEd @@ e)"
  "(WHILE  b DO  c)  @@ e = ((WHILE b DOc);e)"
```

Handling Loops (and Recursion).

Proofs for Correctness are straight-forward (done in Isabelle/HOL) based on the shown rules for denotationally equivalent programs ...

Theorem: Unwind and Concat correct

C(c @@ d) = C(c;d) and C(unwind(n,c)) = C(c)

Handling Loops (and Recursion).

This allows us (together with the equivalence of natural and denotational semantics) to generalize our scheme:

Handling Loops (and Recursion).

This allows us (together with the equivalence of natural and denotational semantics) to generalize our scheme:

$$\forall s\ s'.\ \langle\ \text{unwind}(n,c)\ ,s\rangle \xrightarrow[c]{} s' \wedge \quad P\ s \rightarrow Q\ s'$$
$$\Longrightarrow \models \{P\}c\{Q\}$$

for an arbitrary (user-defined!) *n* !

Or in natural deduction notation:

$$\dfrac{\begin{array}{c}[\langle unwind(n,c), s\rangle \rightarrow_c s', P\ s]_{s,s'} \\ \vdots \\ Q\ s'\end{array}}{\models \{P\}\ c\ \{Q\}}$$

Handling Loops (and Recursion).

Example:

"$\models \{True\}\ integer\_squareroot\ \{i^2 \leq a \wedge a \leq (i+1)^2\}$"

Setting the depth to $n = 3$ and running the process yields:

Handling Loops (and Recursion).

Example:

"$\models \{True\}\ integer\_squareroot\ \{i^2 \leq a \wedge a \leq (i+1)^2\}$"

Setting the depth to $n = 3$ and running the process yields:

```
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'
               ⟧     ⟹post s'
2. ⟦ 4 ≤s a; 8 < s a ; s'  = s (i  := 2, tm := 5, sum := 9) ⟧  ⟹post s'
3. ⟦ 1 ≤s a; s a < 4; s' = s (i  := 1, tm := 3, sum := 4) ⟧  ⟹post s'
4. ⟦ s a = 0 ; s'  = s(tm := 1, sum := 1, i := 0) ⟧  ⟹post s'
```

which is a neat enumeration of all path-conditions for paths up to $n = 3$ times through the loop, except subgoal 1, which is:

Explicit test-Hypothesis in White-Box-Tests:

1. THYP(9 $\leq$s a $\wedge\langle$`WHILE` λs. s sum $\leq$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)$\rangle \underset{c}{\longrightarrow}$s'
   $\longrightarrow$ post s')

... a kind of “structural” regularity hypothesis !

Summary: Program-based Tests in HOL-TestGen:

1. It is possible to do white-box tests in HOL-TestGen
2. Requisite: Denotational and Natural Semantics for a programming language
3. Proven correct unfolding scheme
4. Explicit Test-Hypotheses Concept also applicable for Program-based Testing
5. Can either verify or test paths ...

Summary (II) : Program-based Tests in HOL-TestGen:

Open Questions:

1. Does it scale for *large programs* ???
2. Does it scale for *complex memory models* ???
3. What heuristics should we choose ???
4. How to combine the approach with randomized tests?
5. How to design Modular Test Methods ???

Outline

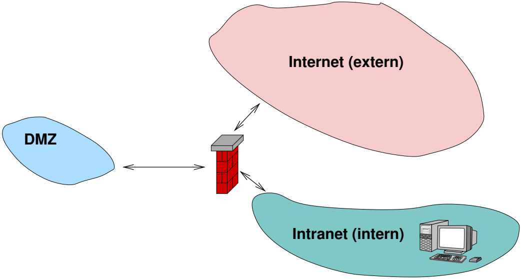
Specification-based Firewall Testing

Objective: test if a firewall configuration implements a given firewall policy

Procedure: as usual:

1. model firewalls (e.g., networks and protocols) and their policies in HOL
2. use HOL-TestGen for test-case generation

A Typical Firewall Policy



$\longrightarrow$	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all protocols except smtp
DMZ	$\emptyset$	-	smtp
Internet	$\emptyset$	http,smtp	-

A Bluffers Guide to Firewalls

- A Firewall is a
  - state-less or
  - state-full

  packet filter.
- The filtering (i.e., either accept or deny a packet) is based on the
  - source
  - destination
  - protocol
  - possibly: internal protocol state

The State-less Firewall Model I

First, we model a packet:

types (α,β) packet = "id $\times$protocol $\times\alpha$src $\times\alpha$dest $\times\beta$content"

where

id: a unique packet identifier, e. g., of type Integer

protocol: the protocol, modeled using an enumeration type (e.g., ftp, http, smtp)

α src (α dest): source (destination) address, e.g., using IPv4:

types
ipv4_ip = "(int $\times$int $\times$int $\times$int)"
ipv4 = "(ipv4_ip $\times$int)"

β content: content of a packet

The State-less Firewall Model II

- A firewall (packet filter) either accepts or denies a packet:

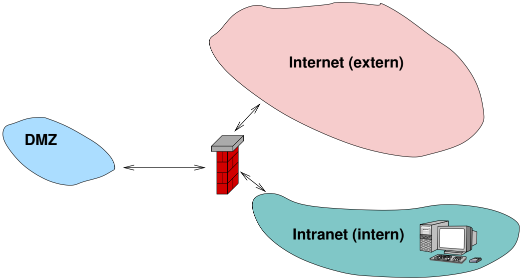
  datatype
  α out = accept α | deny

- A policy is a map from `packet` to `packet out`:

  types
  (α, β) Policy = "(α, β) packet $\rightharpoonup$ ((α, β) packet) out"

- Writing policies is supported by a specialised combinator set

Testing State-less Firewalls: An Example I



$\longrightarrow$	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all protocols except smtp
DMZ	$\emptyset$	-	smtp
Internet	$\emptyset$	http,smtp	-

Testing State-less Firewalls: An Example II

src	dest	protocol	action
Internet	DMZ	http	*accept*
Internet	DMZ	smtp	*accept*
⋮	⋮	⋮	⋮
$*$	$*$	$*$	*deny*

constdefs Internet_DMZ :: "(ipv4, content) Rule"
    "Internet_DMZ ≡
        (allow_prot_from_to smtp internet dmz) ++
        (allow_prot_from_to http internet dmz)"

The policy can be modelled as follows:

constdefs test_policy :: "(ipv4,content) Policy"
    "test_policy ≡ deny_all ++ Internet_DMZ ++ ..."

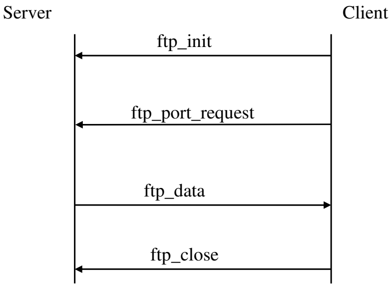
Testing State-less Firewalls: An Example III

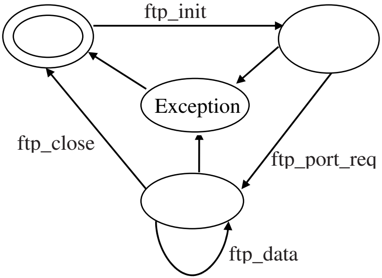
- Using the test specification

  test_spec "FUT x = test_policy x"

- results in test cases like:
  - `FUT (6,smtp,((192,169,2,8),25),((6,2,0,4),2),data) = Some (accept (6,smtp,((192,169,2,8),25),((6,2,0,4),2),data))`
  - `FUT (2,smtp,((192,168,0,6),6),((9,0,8,0),6),data) = Some deny`

State-full Firewalls: An Example (ftp) I





State-full Firewalls: An Example (ftp) II

- based on our state-less model:
  Idea: a firewall (and policy) has an internal state:
- the firewall state is based on the history and the current policy:

  types (α,β,γ) FWState = "$\alpha \times (\beta,\gamma)$ Policy"

- where FWStateTransition maps an incoming packet to a new state

  types (α,β,γ) FWStateTransition =
  "$((\beta,\gamma)$ In_Packet $\times (\alpha,\beta,\gamma)$ FWState) $\rightharpoonup$
  $((\alpha,\beta,\gamma)$ FWState)"

State-full Firewalls: An Example (ftp) III

HOL-TestGen generates test case like:

```
FUT [(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), close),
     (6, ftp, ((4, 7, 9, 8), 21), ((192, 168, 3, 1), 3), ftp_data),
     (6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), port_request
     (6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), init)] =
([(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), close),
  (6, ftp, ((4, 7, 9, 8), 21), ((192, 168, 3, 1), 3), ftp_data),
  (6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), port_request 3)
  (6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), init)],
new_policy)
```

Firewall Testing: Summary

- Successful testing if a concrete configuration of a network firewall correctly implements a given policy
- Non-Trivial Test-Case Generation
- Non-Trivial State-Space (IP Adresses)
- Sequence Testing used for Stateful Firewalls
- Realistic, but amazingly concise model in HOL!

Outline

Conclusion I

- Approach based on theorem proving
  - test specifications are written in HOL
  - functional programming, higher-order, pattern matching
- Test hypothesis explicit and controllable by the user (could even be verified!)
- Proof-state explosion controllable by the user
- Although logically puristic, systematic unit-test of a “real” compiler library is feasible!
- Verified tool inside a (well-known) theorem prover

Conclusion II

- Explicit Test Hypothesis are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!

- The Sequence Test Setting of HOL-TestGen is effective (see Firewall Test Case Study)
- HOL-Testgen is a verified test-tool (entirely based on derived rules . . .)
- The White-box Test offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- Explicit Test Hypothesis are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern Unit Test:

$$\text{pre}\, x \longrightarrow \text{post}\, x (\mathit{prog}\; x)$$

- The Sequence Test Setting of HOL-TestGen is effective (see Firewall Test Case Study)
- HOL-Testgen is a verified test-tool (entirely based on derived rules . . .)
- The White-box Test offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- Explicit Test Hypothesis are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern Sequence Test:

$$\text{accept}\, \mathit{trace} \Longrightarrow P(\text{Mfold}\, \mathit{trace}\, \sigma_0 \mathit{prog})$$

- The Sequence Test Setting of HOL-TestGen is effective (see Firewall Test Case Study)
- HOL-Testgen is a verified test-tool (entirely based on derived rules . . .)
- The White-box Test offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- Explicit Test Hypothesis are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern Reactive Sequence Test:

$$\text{accept } \textit{trace} \implies P(\text{Mfold } \textit{trace}\, \sigma_0 \\ (\text{observer observer rebind subst } \textit{prog}))$$

- The Sequence Test Setting of HOL-TestGen is effective (see Firewall Test Case Study)
- HOL-Testgen is a verified test-tool (entirely based on derived rules . . .)
- The White-box Test offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Bibliography I

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
Model-based firewall conformance testing.
In Kenji Suzuki and Teruo Higashino, editors, *Testcom/*FATES *2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

Achim D. Brucker and Burkhart Wolff.
Interactive testing using HOL-TestGen.
In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Testing of Software (FATES 05)*, LNCS 3997, pages 87–102. Springer-Verlag, Edinburgh, 2005.

Bibliography II

- Achim D. Brucker and Burkhart Wolff.
  Symbolic test case generation for primitive recursive functions.
  In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing (FATES)*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, Linz, 2005.

- Achim D. Brucker and Burkhart Wolff.
  HOL-TestGen 1.0.0 user guide.
  Technical Report 482, ETH Zurich, April 2005.

Bibliography III

- Achim D. Brucker and Burkhart Wolff.
  Test-sequence generation with HOL-TestGen – with an application to firewall testing.
  In Bertrand Meyer and Yuri Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science. Springer-Verlag, Zurich, 2007.

- Jeremy Dick and Alain Faivre.
  Automating the generation and sequencing of test cases from model-based specications.
  In J.C.P. Woodcock and P.G. Larsen, editors, *FME 93*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.

Bibliography IV

Marie-Claude Gaudel.
Testing can be formal, too.
In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.

Wolfgang Grieskamp, Nicolas Kicillof, Dave MacDonald, Alok Nandan, Keith Stobie, and Fred L. Wurden.
Model-based quality assurance of windows protocol documentation.
In *ICST*, pages 502–506, 2008.

Bibliography V

- The HOL-TestGen Website.
  `http://www.brucker.ch/projects/hol-testgen/`.
- Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson.
  Model-based testing of object-oriented reactive systems with spec explorer.
  In *Formal Methods and Testing*, pages 39–76, 2008.
- Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid.
  Test input generation with java pathfinder.
  In *ISSTA*, pages 97–107, 2004.

Bibliography VI

Hong Zhu, Patrick A.V. Hall, and John H. R. May.
Software unit test coverage and adequacy.
ACM Computing Surveys, 29(4):366–427, December 1997.

Part II

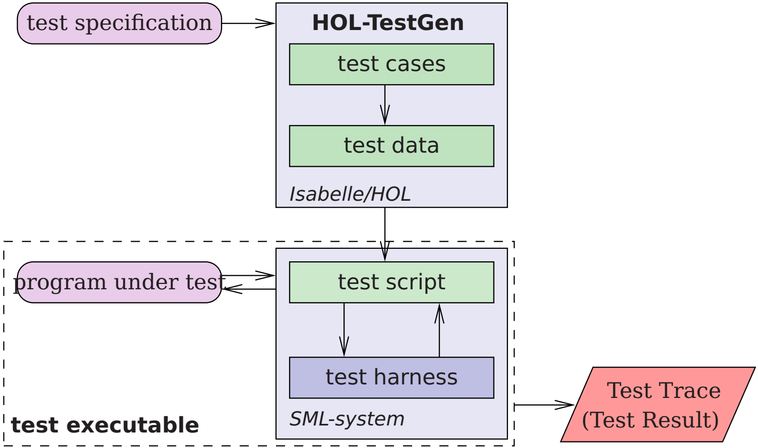
Appendix

Outline

Download HOL-TestGen

- available, including source at:
  `http://www.brucker.ch/projects/hol-testgen/`
- for a “out of the box experience,” try IsaMorph:
  `http://www.brucker.ch/projects/isamorph/`

The System Architecture of HOL-TestGen



The HOL-TestGen Workflow

We start by

1. writing a test theory (in HOL)
2. writing a test specification (within the test theory)
3. generating test cases
4. interactively improve generated test cases (if necessary)
5. generating test data
6. generating a test script.

And finally we,

1. build the test executable
2. and run the test executable.

Writing a Test Theory

For using HOL-TestGen you have to build your Isabelle theories (i.e. test specifications) on top of the theory Testing instead of Main:

theory max_test = Testing:

. . .

end

Writing a Test Specification

Test specifications are defined similar to theorems in Isabelle, e.g.

test_spec "prog a b = max a b"

would be the test specification for testing a a simple program computing the maximum value of two integers.

Test Case Generation

- Now, abstract test cases for our test specification can (automatically) generated, e.g. by issuing

  apply(gen_test_cases 3 1 "prog" simp: max_def)

- The generated test cases can be further processed, e.g., simplified using the usual Isabelle/HOL tactics.
- After generating the test cases (and test hypothesis') you should store your results, e.g.:

  store_test_thm "max_test"

Test Data Selection

In a next step, the test cases can be refined to concrete test data:

gen_test_data "max_test"

Test Script Generation

After the test data generation, HOL-TestGen is able to generate a test script:

generate_test_script "test_max.sml" "max_test" "prog" "myMax.max"

A Simple Testing Theory: max

```
theory max_test = Testing:

test_spec "prog a b = max a b"
  apply(gen_test_cases 1 3 "prog" simp: max_def)
  store_test_thm "max_test"
  gen_test_data "max_test"
  generate_test_script "test_max.sml" "max_test" "prog"
                       "myMax.max"
end
```

A (Automatically Generated) Test Script

```
structure TestDriver : sig end = struct
    val return     = ref  ~63;
    fun eval x2 x1 = let val ret = myMax.max x2 x1
                     in ((return := ret);ret) end
    fun retval ()  = SOME(!return);
    fun toString a = Int.toString a;
    val testres    = [];
  val pre_0   = [];
  val post_0  = fn () => ( (eval ~23 69 = 69));
  val res_0   = TestHarness.check retval pre_0 post_0;
  val testres = testres@[res_0];
  val pre_1   = [];
  val post_1  = fn () => ( (eval ~11 ~15 = ~11));
  val res_1   = TestHarness.check retval pre_1 post_1;
  val testres = testres@[res_1];
  val _ = TestHarness.printList toString testres;
end
```

Building the Test Executable

- Assume we want to test the SML implementation

```
structure myMax = struct
  fun max x y = if (x < y) then y else x
end
```

stored in the file `max.sml`.

- The easiest option is to start an interactive SML session:

```
use "harness.sml";
use "max.sml";
use "test_max.sml";
```

- It is also an option to compile the test harness, test script and our implementation under test into one executable.
- Using a foreign language interface we are able to test arbitrary implementations (e. g., C, Java or any language supported by the .Net framework).

The Test Trace

Running our test executable produces the following test trace:

```
Test Results:
=============
Test 0 -     SUCCESS, result:  69
Test 1 -     SUCCESS, result: ~11


Summary:
--------
Number successful tests cases:  2 of 2 (ca. 100%)
Number of warnings:             0 of 2 (ca. 0%)
Number of errors:               0 of 2 (ca. 0%)
Number of failures:             0 of 2 (ca. 0%)
Number of fatal errors:         0 of 2 (ca. 0%)

Overall result: success
===============
```