

# Theorem-prover based Testing with HOL-TestGen

Achim D. Brucker<sup>1</sup>   Lukas Brügger<sup>2</sup>   Burkhart Wolff<sup>3</sup>

<sup>1</sup>SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany  
achim.brucker@sap.com

<sup>2</sup>Information Security, ETH Zürich, Switzerland  
lukas.bruegger@inf.ethz.ch

<sup>3</sup>Universität des Saarlandes, 66041 Saarbrücken, Germany  
wolff@wjpserver.cs.uni-sb.de

A Tutorial at NII  
Tokyo, 9th June 2008

# Outline

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics
- 3 Advanced Test Scenarios
- 4 Case Studies
- 5 Conclusion

# Outline

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics
- 3 Advanced Test Scenarios
- 4 Case Studies
- 5 Conclusion

# 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)

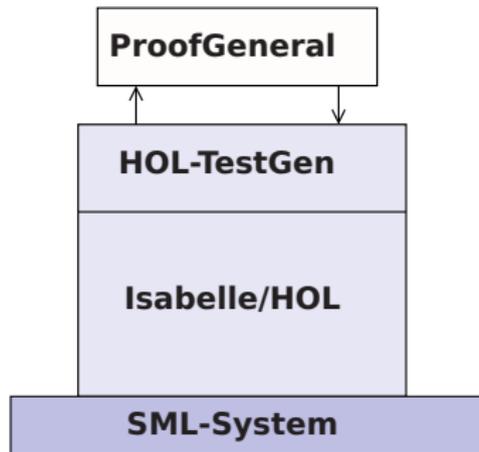
# Our Second Vision

- **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  $\Rightarrow$  bool"
```

```
primrec
```

```
"is_sorted [] = True"
```

```
"is_sorted (x#xs) = case xs of
```

```
    []  $\Rightarrow$  True
```

```
  | y#ys  $\Rightarrow$  ((x < y)  $\vee$  (x = y))
```

```
     $\wedge$  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 hypotheses later in great detail.

## Step IV: Test Data Generation

- On the test theorem, all sorts of logical messages 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  $\Rightarrow$  bool"
  primrec "is_sorted [] = True"
           "is_sorted (x#xs) = case xs of
                                   []  $\Rightarrow$  True
                                   | y#ys  $\Rightarrow$  ((x < y)  $\vee$  (x = y))
                                    $\wedge$  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!

The screenshot shows the HOL-TestGen tool interface. On the left, a terminal window displays the results of 12 test cases. On the right, the Emacs editor shows the Isabelle code for the test suite.

```

emacs@nakagawa.inf.ethz.ch
File Edit Options Buffers Tools Index Isabelle Proof-General X-S
State Context Goal Retract Undo Next Use Goto G.E.D. Find

test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
  -> (blackinv (prog (y,t)))"
apply (gen_test_cases "prog")
store_test_thm "red-and-black-inv"
testgen_params [iterations=100]
gen_test_data "red-and-black-inv"

thm "red-and-black-inv.test_data"

subsection (* An Alternative Approach with a Little Theorem Proving *)
-1: ** HOL_test.thy 428 (126,33) SVN-16263 (Isar_script MMM XS:isabelle)
RSF ==> blackinv (prog (31, T B (T B (T R E -45 E) 81 E) 15 E))
RSF ==> blackinv (prog (94, T B (T B E 99 E) -56 E))
blackinv (prog (-45, T B (T B E -92 E) -45 (T B E -11 E)))
blackinv (prog (-11, T B (T R E -11 E) 19 (T R E 98 E)))
blackinv (prog (39, T B (T R E 8 E) 16 (T R E 39 E)))[]

-1:-- *isabelle-response* Bot (13,53) (response)---6:22 Mail-----

Test 2 - SUCCESS, re
Test 3 - ** WARNING: pr
Test 4 - ** WARNING: pr
Test 5 - SUCCESS, re
Test 6 - SUCCESS, re
Test 7 - ** WARNING: pr
Test 8 - ** WARNING: pr
Test 9 - *** FAILURE: po
Test 10 - SUCCESS, r
Test 11 - SUCCESS, r

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

Figure: HOL-TestGen Using Proof General at one Glance

# Outline

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics**
- 3 Advanced Test Scenarios
- 4 Case Studies
- 5 Conclusion

# 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.,

$$\text{Suc}(x + y) = x + \text{Suc}(y)$$

$$x + y = y + x$$

$$\text{Suc}(x) \neq 0$$

- Logic and Set Theory, e. g.,

$$\forall x. (P x \wedge Q x) = (\forall x. P x) \wedge (\forall x. Q x)$$

$$\bigcup_{x \in S}. (P x \cup Q x) = (\bigcup_{x \in S}. P x) \cup (\bigcup_{x \in S}. Q x)$$

$$\llbracket A = A'; A \implies B = B' \rrbracket \implies (A \wedge B) = (A' \wedge B')$$

# The Core Tableaux-Calculus

- **Safe Introduction** Rules for logical connectives:

$$\begin{array}{c}
 \frac{}{t = t} \quad \frac{}{\text{true}} \quad \frac{P \quad Q}{P \wedge Q} \quad \frac{[\neg Q] \quad \vdots \quad P}{P \vee Q} \quad \frac{[P] \quad \vdots \quad Q}{P \rightarrow Q} \quad \frac{[P] \quad \vdots \quad \text{false}}{\neg P} \quad \dots
 \end{array}$$

- **Safe Elimination** Rules:

$$\begin{array}{c}
 \frac{\text{false}}{P} \quad \frac{P \wedge Q \quad R}{R} \quad \frac{[P, Q] \quad \vdots \quad R}{R} \quad \frac{P \vee Q \quad R \quad R}{R} \quad \frac{[P] \quad [Q] \quad \vdots \quad \vdots \quad R \quad R}{R} \quad \frac{P \rightarrow Q \quad R \quad R}{R} \quad \frac{[\neg P] \quad [Q] \quad \vdots \quad \vdots \quad R \quad R}{R} \quad \dots
 \end{array}$$

# The Core Tableaux-Calculus

- Safe Introduction Quantifier rules:

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

- Safe Quantifier Elimination
 
$$\frac{\exists x. P x \quad \bigwedge x. \begin{matrix} [P x] \\ \vdots \\ Q \end{matrix}}{Q}$$

- 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-structures?
- What is the connection between test-cases and test statements and the test theorems?
- Two problems, one answer: Introducing test hypothesis “on the fly”:

THYP : bool  $\Rightarrow$  bool

THYP(x)  $\equiv$  x

# Taming Infinity I: Regularity Hypothesis

- What to do with infinite data-structures of type  $\tau$ ?  
Conceptually, we split the set of all data of type  $\tau$  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:

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

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

- 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:

$$n) \quad \llbracket C1\ x; \dots; C_m\ x \rrbracket \implies TS\ x$$

is transformed into:

$$n) \quad \llbracket C1\ ?x; \dots; C_m\ ?x \rrbracket \implies TS\ ?x$$

$$n+1) \quad \text{THYP}((\exists x. C1\ x \dots C_m\ x \longrightarrow TS\ x) \\ \longrightarrow (\forall x. C1\ x \dots C_m\ x \longrightarrow TS\ x))$$

# Testcase Generation by NF Computations

Test-theorem is computed out of the test specification by

- a heuristics 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.:

$$\llbracket \neg(\text{prog } x = c); \neg(\text{prog } x = d) \rrbracket \Longrightarrow A$$

is transformed equivalently into

$$\llbracket \neg A \rrbracket \Longrightarrow (\text{prog } x = c) \vee (\text{prog } x = d)$$

- 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  $\implies$  prog x S"
```

```
apply(gen_testcase 0 0)
```

1)  $\text{prog } x [x]$

2)  $\bigwedge b. \text{prog } x [x,b]$

3)  $\bigwedge a. a \neq x \implies \text{prog } x [a,x]$

4)  $\text{THYP}(3 \leq \text{size } (S))$

$\longrightarrow \forall x. x \text{ mem } S$

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

# Sample Derivation of Test Theorems

## Example

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

# Sample Derivation of Test Theorems

## Example

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

is transformed via data-separation lemma to:

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

# Sample Derivation of Test Theorems

## Example

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

canonization leads to:

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

# Sample Derivation of Test Theorems

## Example

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

erasure for unsatisfiable constraints and rewriting conditionals yields:

$$2. \bigwedge a. a = x \vee (a \neq x \wedge \text{false})$$

$$\implies \text{prog } x \text{ } [a]$$

$$3. \bigwedge a \ b. a = x \vee (a \neq x \wedge x \text{ mem } [b]) \implies \text{prog } x \text{ } [a,b]$$

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

# Sample Derivation of Test Theorems

## Example

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

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

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

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

3'.  $\bigwedge a \text{ } b. \llbracket a \neq x; x \text{ mem } [b] \rrbracket \implies \text{prog } x \text{ } [a,b]$

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

# Sample Derivation of Test Theorems

## Example

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

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

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

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

3'.  $\bigwedge a \text{ } b. a \neq x \implies \text{ prog } x \text{ } [a, x]$

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

# 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:

$$\llbracket TC_1; \dots; TC_n; \text{THYP } H_1; \dots; \text{THYP } H_m \rrbracket \implies TS$$

where the **test cases**  $TC_i$  have the form:

$$\llbracket C_1x; \dots; C_mx; \text{THYP } H_1; \dots; \text{THYP } H_m \rrbracket \implies P x (\text{prog } x)$$

and where the **test-hypothesis** are either uniformity or regularity hypotheses.

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

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics
- 3 Advanced Test Scenarios**
- 4 Case Studies
- 5 Conclusion

# 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:  
⇒ 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.

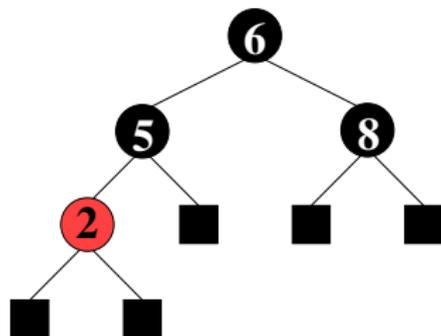
- part of the SML standard library
- widely used internally in the sml/NJ compiler, e. g., for providing efficient implementation for Sets, Bags, . . . ;
- very hard to generate (balanced) instances randomly

# 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 ( $\alpha$  tree) ( $\beta::\text{ord item}$ ) ( $\alpha$  tree)

# Modeling Red-black Trees II

- Red-Black Trees: Test Theory

## consts

redinv :: tree  $\Rightarrow$  bool

blackinv :: tree  $\Rightarrow$  bool

**recdef** blackinv measure ( $\lambda$  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))))

recdef redinv measure ...

# Red-black Trees: Test Specification

- Red-Black Trees: Test Specification

## **test\_spec:**

```
"isord t ∧ redinv t ∧ blackinv t
  ∧ isin (y::int) t
  →
  (blackinv(prog(y,t)))"
```

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

- Using the standard-workflows results, among others:

```
RSF → 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.

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

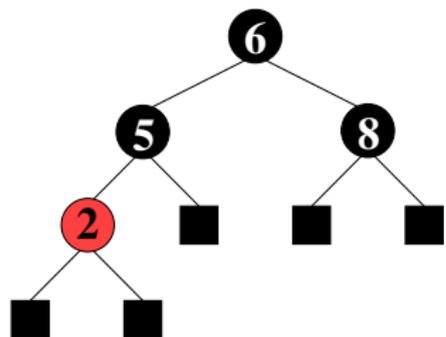
$$\begin{aligned}
 (\max\_B\_height \ x = 0) = \\
 (x = E \ \vee \exists a \ y \ b. \ x = T \ R \ a \ y \ b \wedge \\
 \max(\max\_B\_height \ a) \\
 (\max\_B\_height \ b) = 0)
 \end{aligned}$$

- 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`.

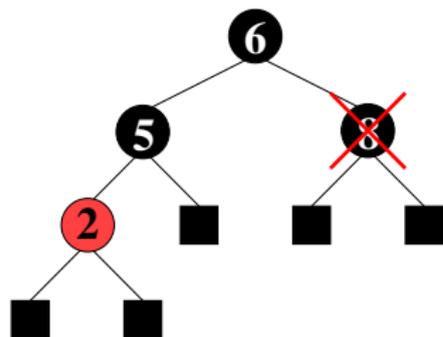
# Red-black Trees: sml/NJ Implementation



(a) pre-state

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

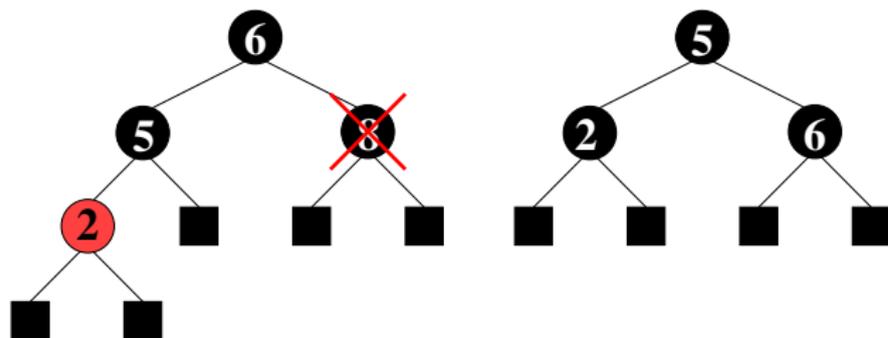
# Red-black Trees: sml/NJ Implementation



(b) pre-state: delete "8"

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

# Red-black Trees: sml/NJ Implementation

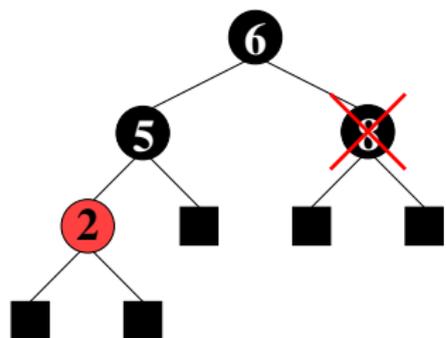


(b) pre-state: delete "8"

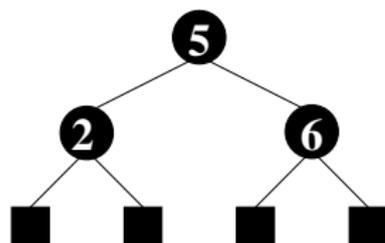
(c) correct result

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

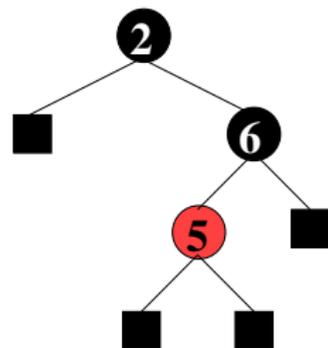
# Red-black Trees: sml/NJ Implementation



(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:

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

- 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  $\rightarrow P(\text{Mfold trace } \sigma_0 \text{ prog})$

where

$\text{Mfold [] } \sigma = \text{Some } \sigma$

$\text{MFold (input::R) = case prog(input, } \sigma) \text{ of}$

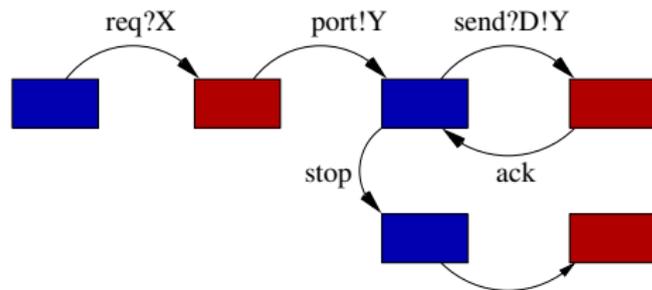
$\text{None } \Rightarrow \text{None}$

$| \text{Some } \sigma' \Rightarrow \text{Mfold R } \sigma' \text{ prog}$

- 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:

$$\begin{aligned} \text{req?}X \rightarrow \text{port!}Y[Y < X] \rightarrow \\ (\text{rec } N. \text{send!}D.Y \rightarrow \text{ack} \rightarrow N \\ \square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP}) \end{aligned}$$

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:

$$\text{req?}X \rightarrow \text{port!}Y[Y < X] \rightarrow$$

$$(\text{rec } N. \text{send!}D.Y \rightarrow \text{ack} \rightarrow N$$

$$\square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})$$

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 ioprogram  $\equiv$

$(\lambda \text{ input. } (\lambda (\sigma, \sigma'). \mathbf{let} \text{ input}' = \text{substitute } \sigma \text{ input in}$   
 $\text{case ioprogram input}' \sigma' \mathbf{of}$

None  $\Rightarrow$  None (*\* ioprogram failure – eg. timeout ... \**)

| Some (output,  $\sigma'''$ )  $\Rightarrow \mathbf{let} \sigma'' = \text{rebind } \sigma \text{ output in}$

(if postcond ( $\sigma'', \sigma'''$ ) input' output

then Some( $\sigma'', \sigma'''$ )

else None (*\* postcond failure \** ))))"

# Example: A Reactive Test IV

- Reactive Test-Specification Pattern:

accept *trace*  $\rightarrow$

$P(\text{Mfold } \textit{trace} \sigma_0 (\text{observer rebind subst postcond } \textit{ioprogram}))$

- for reactive systems!

# Outline

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics
- 3 Advanced Test Scenarios
- 4 Case Studies**
- 5 Conclusion

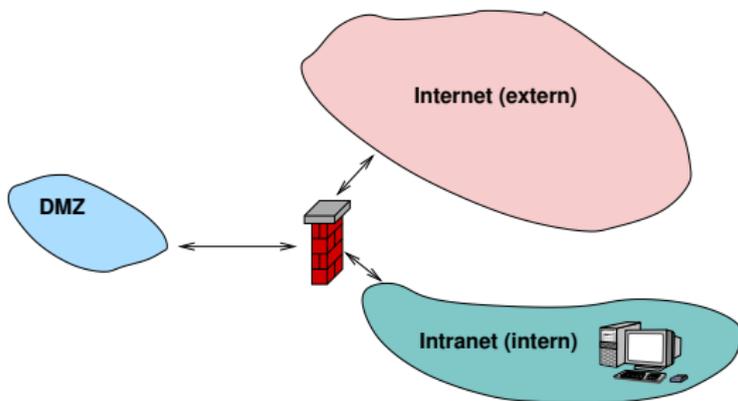
# Specification-based Firewall Testing

**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



→	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all protocols except smtp
DMZ	∅	-	smtp
Internet	∅	http,smtp	-

# A Bluffers Guide to Firewalls

- A Firewall is a
  - state-less or
  - state-fullpacket 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**  $(\alpha, \beta)$  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)

$\alpha$  src ( $\alpha$  dest): source (destination) address, e.g., using IPv4:

**types**

ipv4\_ip = "(int  $\times$  int  $\times$  int  $\times$  int)"

ipv4 = "(ipv4\_ip  $\times$  int)"

$\beta$  content: content of a packet

# The State-less Firewall Model II

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

## **datatype**

$\alpha$  out = accept  $\alpha$  | deny

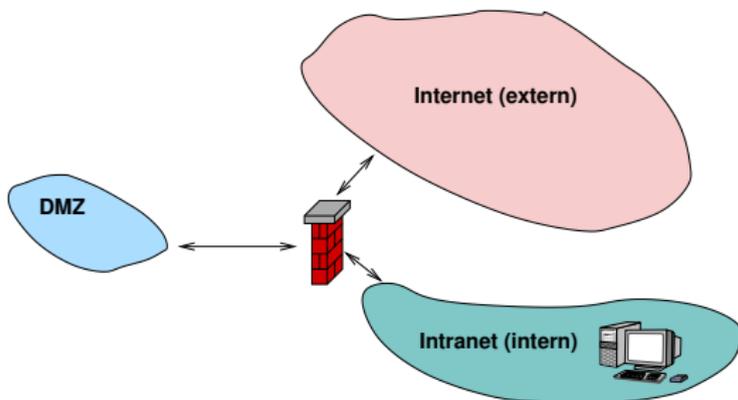
- A **policy** is a map from packet to packet out:

## **types**

$(\alpha, \beta)$  Policy = " $(\alpha, \beta)$  packet  $\rightarrow$  (( $\alpha, \beta$ ) packet) out"

- Writing policies is supported by a specialised combinator set

# Testing State-less Firewalls: An Example I



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

# Testing State-less Firewalls: An Example II

<b>src</b>	<b>dest</b>	<b>protocol</b>	<b>action</b>
Internet	DMZ	http	<i>accept</i>
Internet	DMZ	smtp	<i>accept</i>
⋮	⋮	⋮	⋮
*	*	*	<i>deny</i>

**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:

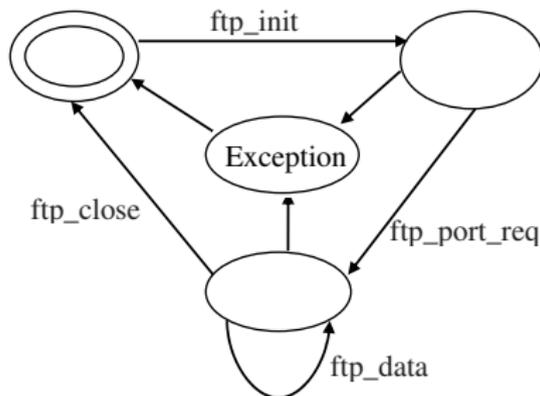
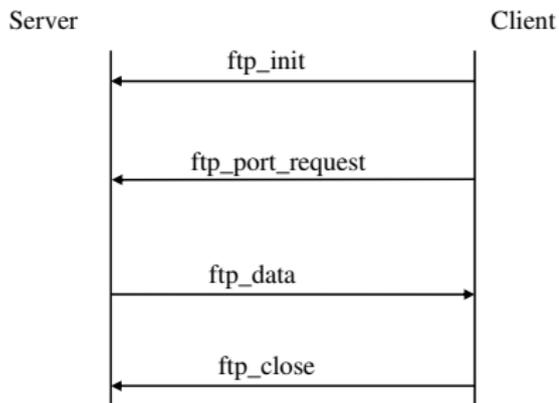
- FUT

(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**  $(\alpha, \beta, \gamma)$  FWState = " $\alpha \times (\beta, \gamma)$  Policy"

- where FWStateTransition maps an incoming packet to a new state

**types**  $(\alpha, \beta, \gamma)$  FWStateTransition =  
 " $((\beta, \gamma)$  In\_Packet  $\times (\alpha, \beta, \gamma)$  FWState)  $\rightarrow$   
 $((\alpha, \beta, \gamma)$  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

- 1 Motivation and Introduction
- 2 From Foundations to Pragmatics
- 3 Advanced Test Scenarios
- 4 Case Studies
- 5 Conclusion

# 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

- Test Hypothesis explicit and controllable by the user (can even be verified !)
- 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 . . . )

## Conclusion II

- Test Hypothesis explicit and controllable by the user (can even be verified !)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Unit Test**:

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

- 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 . . . )

## Conclusion II

- Test Hypothesis explicit and controllable by the user (can even be verified !)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Sequence Test**:

$$\text{accept } trace \implies P(\text{Mfold } trace \ \sigma_0 \text{prog})$$

- 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 . . . )

## Conclusion II

- Test Hypothesis explicit and controllable by the user (can even be verified !)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Reactive Sequence Test**:

$$\text{accept } trace \implies P(\text{Mfold } trace \sigma_0$$

$$\quad (\text{observer observer rebind subst } prog))$$

- 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 . . . )

# Bibliography I



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.



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.

# Bibliography II

 Achim D. Brucker and Burkhart Wolff.

HOL-TestGen 1.0.0 user guide.

Technical Report 482, ETH Zurich, April 2005.

 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.

 The HOL-TestGen Website.

<http://www.brucker.ch/projects/hol-testgen/>.

# Part II

## Appendix

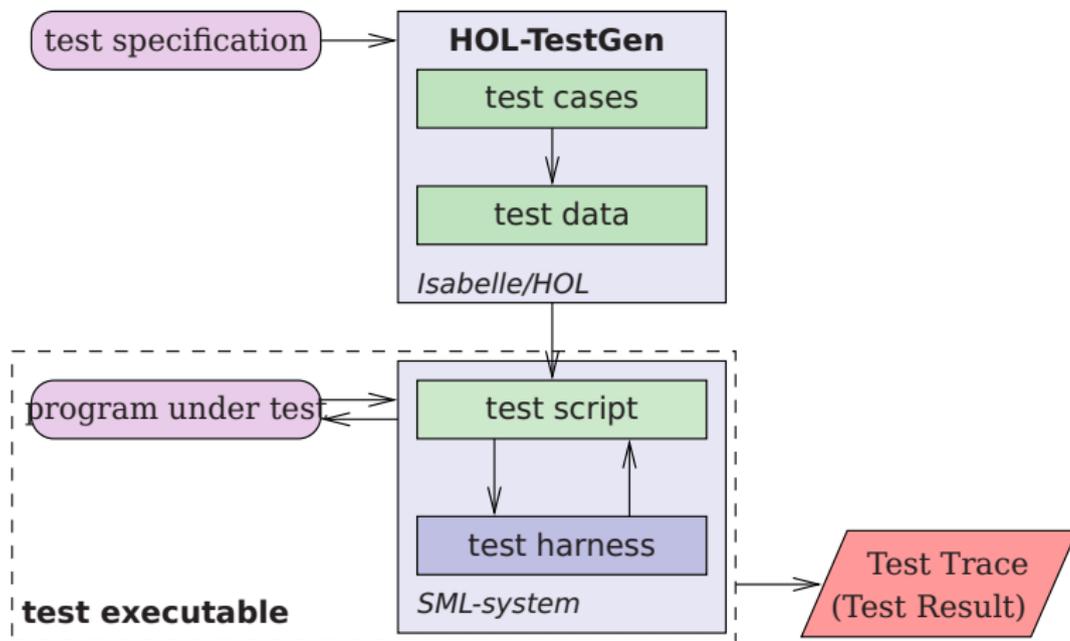
# Outline

- 6 The HOL-TestGen System
- 7 A Hands-on Example

# 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 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
- 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.:

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

```
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

```
1  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);
6   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;
11  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];
16  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

=====