Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff[1]

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

M2R: Test des Systemes Informatiques
Orsay, 15th Dec 2009

Outline

1. Motivation and Introduction
2. A Sample Workflow
3. From Foundations to Pragmatics
4. A Sample Derivation of a Test Theorem
5. Summary

Outline

1 Motivation and Introduction

2 A Sample Workflow

3 From Foundations to Pragmatics

4 A Sample Derivation of a Test Theorem

5 Summary

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

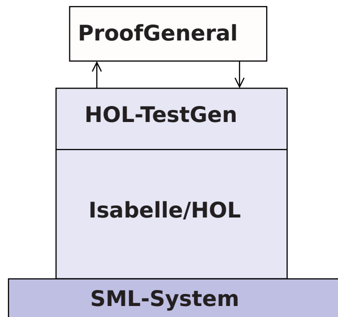


Figure: The Components of HOL-TestGen

A Sample Workflow

1 Motivation and Introduction
2 A Sample Workflow
3 From Foundations to Pragmatics
4 A Sample Derivation of a Test Theorem
5 Summary

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**):

  $\text{THYP}((\exists x.\ \text{is_sorted (prog [x])}) \longrightarrow (\forall x.\ \text{is_sorted(prog [x])}))$
  . . .
  $\text{THYP}((\forall l.\ 4 < |l| \longrightarrow \text{is_sorted(prog l)}))$

- 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!

```
emacs@nakagawa.inf.ethz.ch
File Edit Options Buffers Tools Index Isabelle Proof-General X-S
State Context Goal Retract Undo Next Use Goto Q.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:** RBT_test.thy  42% (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-------
```

```
brucker@nakagawa: /home
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 A Sample Workflow

3 From Foundations to Pragmatics

4 A Sample Derivation of a Test Theorem

5 Summary

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:

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

- **Safe Elimination** Rules:

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

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

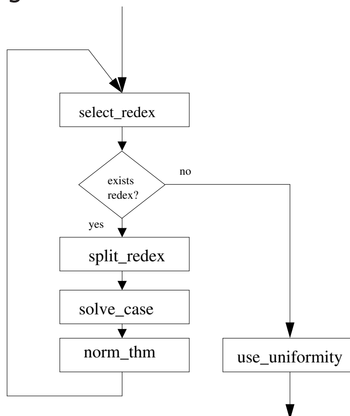
$$\frac{\exists x.\ P\ x \qquad \bigwedge x.\ \begin{array}{c}[P\ x]\\ \vdots \\ Q\end{array}}{Q}$$

- Critical Rewrite Rule:

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

The Generic Procedure

`gen_test_cases` :



Chooser: selects a splitting redex (e.g. free variables)

Splitter: applies splitting rules (e.g. regularity hypothesis, see below)

Normalizer: Applies global simplification and tableaux calculi of *E*, i. e. the previously decribed underlying ruleset

Solver: Attempts to eliminate unsatisfiable constraints

Finalizer: Applies minimization and uniformity hypothesis (see below).

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":

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

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, $k = 3$):

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

- 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) $\llbracket$ 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.

A Sample Derivation of a Test Theorem

1 Motivation and Introduction
2 A Sample Workflow
3 From Foundations to Pragmatics
4 A Sample Derivation of a Test Theorem
5 Summary

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 ⟹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

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

is transformed via data-separation lemma to:

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

Sample Derivation of Test Theorems

Example

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

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

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

which is reduced via the equation for mem:

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

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

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

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

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

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

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

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

Sample Derivation of Test Theorems

Example

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

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

A Sample Derivation of a Test Theorem

1 Motivation and Introduction

2 A Sample Workflow

3 From Foundations to Pragmatics

4 A Sample Derivation of a Test Theorem

5 Summary

Test Case Generation (I)

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

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

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

$$\exists x. C_1 x \wedge \ldots \wedge C_m x \Longrightarrow P\ x\ (prog\ x)$$

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.

Test Case Generation (II)

- 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 !!!**

Using Constraint Solving

Test data generation is now a constraint satisfaction problem.

- We eliminate the existential quantifiers (or equivalently: 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)

Correctness of a Test-Theorem

A Test-Theorem is *correct* iff the implication:

$$TC_1 \wedge \ldots \wedge TC_n \wedge \mathrm{THYP}\ H_1 \wedge \ldots \wedge \mathrm{THYP}\ H_m \Longrightarrow TS$$

is logically valid.
Well, actually correctness is assumed if we speak of a correctness-*theorem*.

Completeness of a Test-Theorem

A Test-Theorem is *complete* iff the implication:

$$TS \Longrightarrow (TC_1 \wedge \ldots \wedge TC_n \wedge \mathrm{THYP}\ H_1 \wedge \ldots \wedge \mathrm{THYP}\ H_m)$$

is logically valid.

Minimality of a Test-Theorem

A Test-Theorem is *minimal* iff the test cases are pairwise disjoint, i. e.

$$\{x.Ci_1 x \wedge \ldots \wedge Ci_m x\} \cap \{x.Cj_1 x \wedge \ldots \wedge Cj_n x\} = \{\}$$

is logically valid for all $i \neq j$. This means that the partitions of input are disjoint.

Theoretical Properties: The Case for HOL-TestGen

- generated test-theorems are correct by construction
- ... and complete (by meta-theoretic arguments)
- ... but not necessarily minimal (although, in practice, for data-type-oriented specifications, not far from minimality).

Bibliography I