HOL-TestGen
An Interactive Test-case Generation Framework

Achim D. Brucker[1]   Burkhart Wolff[2]

[1]SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
`achim.brucker@sap.com`

[2]Université Paris-Sud, Parc Club Orsay Université, 91893 Orsay Cedex, France
`wolff@wjpserver.cs.uni-sb.de`

ETAPS 2009
York, 27th March 2009

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

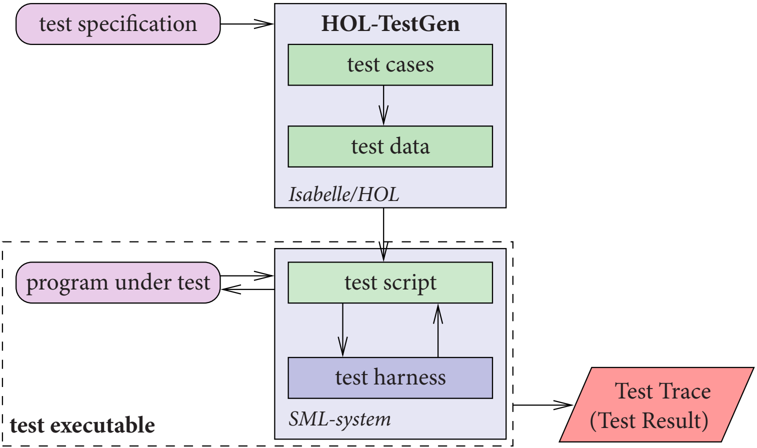
- **Observation:**
  Any testcase-generation technique is based on and limited by underlying constraint-solution techniques.
- **Approach:**
  Testing should be integrated in an environment combining **automated and interactive proof techniques**.
- the test engineer must decide over, abstraction level,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

Outline

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

The System Architecture of HOL-TestGen



Outline

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**):

  $$\begin{array}{l} \text{is\_sorted} \;\; (\text{prog} \;\; []) \\ \text{is\_sorted} \;\; (\text{prog} \;\; [?X1X17]) \\ \text{is\_sorted} \;\; (\text{prog} \;\; [?X2X13, \; ?X1X12]) \\ \text{is\_sorted} \;\; (\text{prog} \;\; [?X3X7, \; ?X2X6, \; ?X1X5]) \end{array}$$

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

  $$\text{THYP}((\exists\, x.\ \text{is\_sorted} \;\; (\text{prog} \;\; [x])) \longrightarrow (\forall\, x.\ \ \text{is\_sorted}\, (\text{prog} \;\; [x]))$$

  $$\vdots$$

  $$\text{THYP}((\forall\, l.\ 4 < |l| \longrightarrow \text{is\_sorted}\, (\text{prog} \;\; l\,))$$

- 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
```

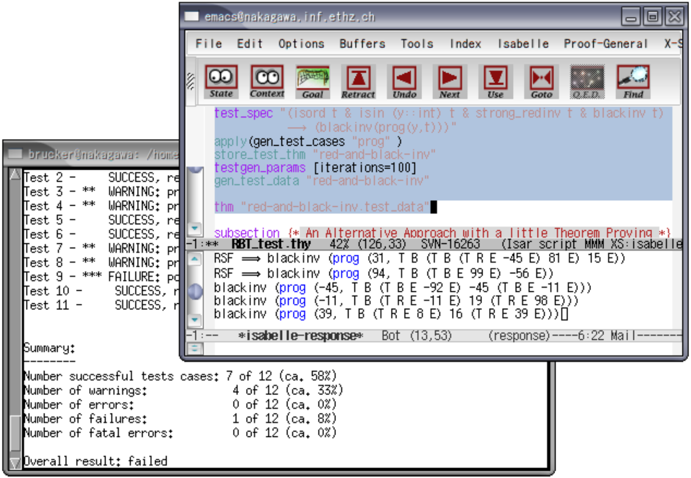
Testing an Implementation

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
```


emacs@nakagawa.inf.ethz.ch
File Edit Options Buffers Tools Index Isabelle Proof-General
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

Outline

Red-black trees

Goal:

Test if balancing property is preserved by the red-black tree operations.

- 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

Red-black Trees: Summary

- Statistics: 348 test cases were generated
- 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

Case Studies: Stateless Firewalls (Packet Filters)

Goal:

Test if a packet filter (firewall) configuration conforms to a given policy.

- A packet filter filters (e.g., rejects or denies) packets based on
  - source address destination address
  - protocol
- As usual
  - model firewalls (e.g., networks and protocols) and their policies in HOL
  - use HOL-TestGen for test-case generation

Case Studies: Stateful Firewalls

Goal:

Test if a stateful firewall supports stateful protocols correctly.

- Obvervation:
  - protocols like ftp and VoIP have an internal state
  - and need to be filtered (dynamically) based on their state
- Idea:
  - re-use our state-less model
  - model an observer using a monadic fold construction
  - this observers manages the state at the execution time
  - for many cases, an observer can be generated automatically

Firewall Testing: Summary

- Remark:
  - Stateless firwalls are a **unit testing** scenario
  - Statefull firwalls are a **sequence testing** scenario
- 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

- 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 (can be seen as proof-obligations)
- 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

Ongoing and Future Work

- Ongoing work includes the development of support for:
  - integration of SAT and SMT Solvers
  - domain-specific test case generation
  - theories for simplifying and transforming test theories
- Future works could include the development for:
  - test theories for three-valued specification (e.g., UML/OCL)
  - integration of unit- and sequence testing approaches
  - ...

Thank you
for your attention!

Any questions or remarks?

The HOL-TestGen can be downloaded from:
`http://www.brucker.ch/projects/hol-testgen/`
(including source, examples, and documentation)

Bibliography I

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
Verifying test-hypotheses: An experiment in test and proof.
Electronic Notes in Theoretical Computer Science, 220(1):15–27, 2008.
Proceedings of the Fourth Workshop on Model Based Testing (oo 2008).

Achim D. Brucker and Burkhart Wolff.
Symbolic test case generation for primitive recursive functions.
In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, 2004.

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

Bibliography II

Achim D. Brucker and Burkhart Wolff.
Interactive testing using HOL-TestGen.
In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Testing of Software*, number 3997 in Lecture Notes in Computer Science. Springer-Verlag, 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, pages 149–168. Springer-Verlag, 2007.

Part I

Appendix

Further Remarks

- In HOL, Sequence Testing and Unit Testing are the same!

- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Further Remarks

- 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 **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Further Remarks

- In HOL, Sequence Testing and Unit Testing are the same!
  TS pattern **Sequence Test**:

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

- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Further Remarks

- In HOL, Sequence Testing and Unit Testing are the same!
  TS pattern **Reactive Sequence Test**:

$$\text{accept } \mathit{trace} \Longrightarrow P(\text{Mfold } \mathit{trace}\ \sigma_0 \\ (\text{observer observer rebind subst } \mathit{prog}))$$

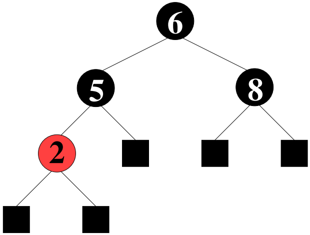
- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

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

$$\begin{array}{lll} \text{redinv} & :: & \text{tree} \Rightarrow \text{bool} \\ \text{blackinv} & :: & \text{tree} \Rightarrow \text{bool} \end{array}$$

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

blackinv E = True

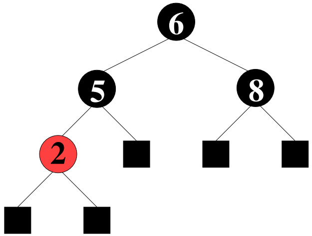
blackinv (T color a y b) =

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

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

recdev redinv measure ...

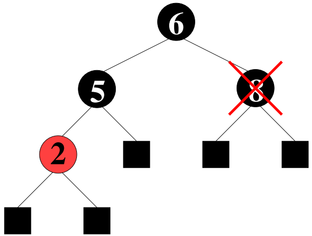
Red-black Trees: sml/NJ Implementation



(a) pre-state

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

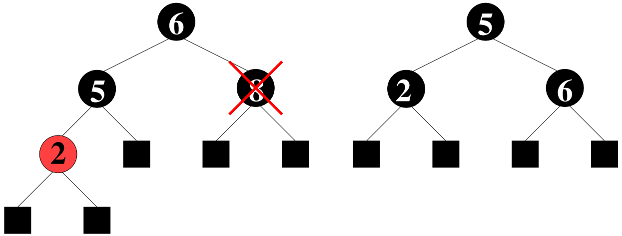
Red-black Trees: sml/NJ Implementation



(b) pre-state: delete "8"

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

Red-black Trees: sml/NJ Implementation

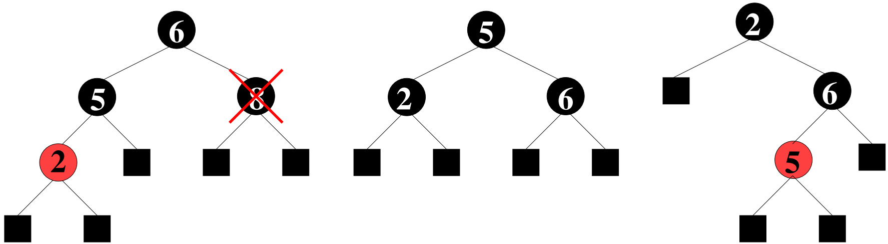


(b) pre-state: delete “8”

(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: Test Specification

- Red-Black Trees: Test Specification

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

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

- Using the standard-workflows results, among others:

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

The State-less Firewall Model I

First, we model a packet:

types (α,β) packet = "id × protocol × α src × α dest × β conten

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 × int × int × int)"
ipv4 = "(ipv4_ip × 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 $\rightharpoonup$ $((\alpha, \beta)$ packet$)$ out"

- Writing policies is supported by a specialised combinator set

State-full Firewalls: An Example (ftp)

- 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 $\times$ (α,β,γ) FWState) $\rightharpoonup$
  $((\alpha,\beta,\gamma)$ FWState)"