Unit- and Sequence Test Generation with HOL-TestGen

Tests et Methodes Formelles

Prof. Burkhart Wolff
Univ - Paris-Sud / LRI
Overview

- HOL-TestGen and its Business-Case
- The Standard Workflow for Unit Testing
- Demo
- The Workflow for Sequence Tests
HOL-TestGen and its Business-Case

- HOL-TestGen is somewhat unusual test-Tool:
  - implemented as “PlugIn” in a major Interactive Theorem Proving Environment: Isabelle/HOL
  - conceived as formal testcase-generation method based on symbolic execution of a model (in HOL)
  - Favors Expressivity and emphasizes Test-Plans as formal entities; emphasis on Interactivity
HOL-TestGen as Plugin in the Isabelle Architecture

- Tools: HOL-OCL, HOL-TestGen, Simpl, ...
- ATP: Z3
- PIDE / jEdit
- Scala System Interface
  - integrators (sledge, smt)
  - components: datatype record, ...
  - code gen.
- proof procedures (simp, fast, auto, etc...)
- nano-kernel + kernel
- ML running on multi-core arch
  - C1, C2, C3, C4

- Argo/UML
- C-to-Simpl
HOL-TestGen as Plugin in the Isabelle Architecture

Advantage:

- **Reuse** of powerful components in unique, interactive integrated environment
- **Seamless integration** of test and proof activities

Tools:
HOL-OCL, HOL-TestGen, Simpl, ...

Scala System Interface
- integrators
  - sledge, smt
- components:
  - datatype
  - record, ...
- code gen.

Proof procedures
- simp, fast, auto, etc...

nano-kernel + kernel

ML running on multi-core arch

ATP : Z3

Arro/UML
C-to-Simpl

C1 C2 C3 C4
HOL-TestGen's Business Case

• If you have already a system model in Isabelle or Coq, you might want to link it to a real implementation.

• This has particular relevance in a Certification project (for example: Common Criteria EAL 5 - 7)
HOL-TestGen's Business Case

- NICTA seL4 Verified Project

- Security Model
  (“Good” Traces and Policies)

- Functional Model
  (System State & Atomic Steps)

- OS API
  (System Calls, Atomic Steps)

- Hardware & Environment

- Proof
  (Isabelle)

- Proof
  (Isabelle/Simpl)

- Tested C
  Compilation
  (ITP 08)
EUROMILS PikeOS Project

- CC EAL 5+; Linking the FM model of the certification to real code
  - Security Model ("Good" Traces and Policies)
  - Functional Model (System State & Atomic Steps)
  - OS API (System Calls, Atomic Steps)
  - Hardware & Environment

Proof (Isabelle)
Test Gen. (Isabelle/TestGen)
Test Gen (Isabelle/TestGen TAP13)
theory AVL_def
imports Testing Main
begin

datatype 'a tree = ET | MKT 'a "a tree" "a tree"

fun height :: "a tree ⇒ nat"
where
  "height ET = 0"
| "height (MKT n l r) = 1 + max (height l) (height r)"

fun is_in :: "'a ⇒ 'a tree ⇒ bool"
where
  "is_in k ET = False"
| "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"
theory AVL_def
imports Testing Main
begin

datatype 'a tree = ET | MKT 'a "'a tree" "'a tree"

fun height :: "'a tree ⇒ nat"
where
  "height ET = 0"
| "height (MKT n l r) = 1 + max (height l) (height r)"

fun is_in :: " 'a ⇒ 'a tree ⇒ bool"
where
  "is_in ET = False"
| "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"
Parallel, Asynchronous execution and validation in the jEdit - GUI

IDE look and feel;
Very attractive work environment.
(Better than Eclipse ;-)

theory Example
imports Main
begin

inductive path for rel :: "'a ⇒ 'a ⇒ bool" where
  base: "path rel x x"
| step: "rel x y ⇒ path rel y z ⇒ path rel x z"

theorem example:
  fixes x z :: 'a assumes "path rel x z" shows "P x z"
  using assms
  proof induct
    case (base x)
    show "P x x" by auto
  next
    case (step x y z)
    note "rel x y" and "path rel y z"
    moreover note "P y z"
    ultimately show "P x z" by auto
  qed

end
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory (the “model”)
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory (the “model”)

Example: Sorting in HOL

```isar
primrec is_sorted :: "int list ⇒ bool"
where  "is_sorted [] = True"
  "is_sorted (x#xs) = case xs of
    [] ⇒ True
  | (y#ys) ⇒ (x≤y) ∧ is_sorted ys"
```

16.6.2015  B.Wolff - HOL-TestGen
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory (the “model”)

Example: Sorting in HOL

```haskell
primrec  is_sorted :: "int list ⇒ bool"
where  "is_sorted [] = True"
       | "is_sorted (x#xs) = case xs of
           [] ⇒ True
           | (y#ys) ⇒ (x ≤ y) ∧ is_sorted ys"
```
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS

```
testspec " is_sorted(PUT x) \\
  ∧ asc(x, PUT x)"
```
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS

pattern:

testspec “pre $x \rightarrow$ post $x$ ($PUT x$)”
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS

example:

\[
\text{test\_spec} \ "\text{is\_sorted } x \ \rightarrow \ \text{is\_sorted } (\text{prog } a \ x)"
\]

or

\[
\text{test\_spec} \ "\text{is\_sorted } (\text{PUT } l)"
\]
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

(“Testcase Generation”)
Black-Box Testing:
“The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
  (“Testcase Generation”)

apply(gen_test_cases 3 1 “PUT”)
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory

• Writing a test-specification TS

• Conversion into test-theorem

  (“Testcase Generation”)

\[ TC_1 \Rightarrow \ldots \Rightarrow TC_n \Rightarrow THYP(H_1) \Rightarrow \ldots \Rightarrow THYP(H_m) \Rightarrow TS \]

• where testcases \( TC_i \) have the form

\[ \text{Constraint}_1(x) \Rightarrow \ldots \Rightarrow \text{Constraint}_k(x) \Rightarrow P(prog\ x) \]

• and where \( THYP(H_i) \) are test-hypothesis
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

Example:

\textit{is\_sorted (PUT l)}
1: \textit{is\_sorted(PUT [])}
2: \textit{is\_sorted(PUT [?X])}
3: \textit{THYP(\exists x. is\_sorted(PUT [x]) \rightarrow \forall x. is\_sorted(PUT [x])})
4: \textit{is\_sorted(PUT [?X, ?Y])}
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

...  

5: THYP(∃ x y. is_sorted(PUT[x,y]) →
   ∀ x y. is_sorted(PUT[x,y]))

6: is_sorted(PUT[?X, ?Y, ?X])

7: THYP(∃ x y z. is_sorted(PUT [x,y,z]) →
   ∀ x y z. is_sorted(PUT [x,y,z]))

8: THYP(3 < |l| → is_sorted(PUT l))
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data

    gen_test_data “…”
Black-Box Testing: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data

  is_sorted(PUT 1 [])
  is_sorted(PUT 1 [0])
  is_sorted(PUT 1 [2])
  is_sorted(PUT 1 [1,2])
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness
Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness
- Run of testharness and generation of test-document
Midi Example: Red Black Trees

Red-Black-Trees: Test Specification

testspec :
(redinv t ∧
  blackinv t)

→

(redinv (delete x t) ∧
  blackinvv (delete x t))

where delete is the program under test.
HOL-TestGen Workflow

Demo
Black-Box Sequence Testing:

• HOL is a state-less language; how to model and test stateful systems?

• How to test systems where you have only control over the initial state?

• How to test concurrent programs implementing a model?
How to model and test stateful systems?

- Use Monads !!!

  The transition in an automaton \((\sigma, (I, O), \sigma)\) set can isomorphically represented by:

  \[ I \Rightarrow \sigma \Rightarrow (O, \sigma) \text{ set} \]

  or for a deterministic transition function:

  \[ I \Rightarrow \sigma \Rightarrow (O, \sigma) \text{ option} \]

... which category theorists or functional programmers would recognize as a Monad function space
How to model and test stateful systems?

- **Use Monads !!!**

  - The transition in an automaton \((\sigma, (\iota, o), \sigma)\) set can isomorphically represented by:

    \[
    \iota \Rightarrow (o \times \sigma) \text{ Mon}_{SBE}
    \]

    or for a deterministic transition function:

    \[
    \iota \Rightarrow (o \times \sigma) \text{ Mon}_{SE}
    \]

    ... which category theorists or functional programmers would recognize as a **Monad function space**
How to model and test stateful systems?

- Monads must have two combination operations bind and unit enjoying three algebraic laws.

- For the concrete case of $\text{Mon}_{SE}$:

\[
\text{definition } \text{bind}_{SE} ::= "('o,'\sigma)\text{MON}_{SE} \Rightarrow ('o \Rightarrow ('o,'\sigma)\text{MON}_{SE}) \Rightarrow ('o,'\sigma)\text{MON}_{SE}"
\]

where

\[
\text{bind}_{SE} \ f \ g = (\lambda\sigma. \ \text{case } f \ \sigma\text{of } \text{None } \Rightarrow \text{None } \mid \text{Some } (\text{out}, \ \sigma') \Rightarrow g \ \text{out} \ \sigma')"
\]

\[
\text{definition } \text{unit}_{SE} ::= "'o \Rightarrow ('o, '\sigma)\text{MON}_{SE}" ("(\text{return } \_ )" 8)
\]

where

\[
\text{unit}_{SE} \ e = (\lambda\sigma. \ \text{Some}(e,\sigma))"
\]

- and write $\sigma \leftarrow m; m' \sigma$ for $\text{bind}_{SE} m (\lambda o. m' o)$

and $\text{return}$ for $\text{unit}_{SE}$
How to model and test stateful systems?

- Valid Test Sequences:
  \[
  \sigma \models o_1 \leftarrow m_1 \tau_1; \ldots; o_n \leftarrow m_n \tau_n; \text{return}(P \ o_1 \cdots o_n)
  \]

- ... can be generated to code
- ... can be symbolically executed ...

\[
\frac{C_m \vdash \sigma \quad m \vdash \sigma = \text{None}}{(\sigma \models (s \leftarrow m \tau; m' s))) = \text{False}}
\]

\[
\frac{C_m \vdash \sigma \quad m \vdash \sigma = \text{Some}(b, \sigma')}{(\sigma \models s \leftarrow m \tau; m' s) = (\sigma' \leftarrow (m' b))}
\]
How to model and test stateful systems?

- Test Refinements for a step-function SPEC and a step function SUT:
  \[\sigma \models o_1 \leftarrow \text{SPEC}_1 \ t_1; \ldots; o_n \leftarrow \text{SPEC}_n \ t_n; \text{return}(res = [o_1 \cdots o_n])\]
  \[\rightarrow\]
  \[\sigma \models o_1 \leftarrow \text{SUT}_1 \ t_1; \ldots; o_n \leftarrow \text{SUT}_n \ t_n; \text{return}(res = [o_1 \cdots o_n])\]

- The premis is reduced by symbolic execution to constraints over res; a constraint solver (Z3) produces an instance for res. The conclusion is compiled to a test-driver/test-oracle linked to SUT.
How to test concurrent programs implementing a model?

- Assumption: Code compiled for LINUX and instrumented for debugging (gcc -d)
- Assumption: No dynamic thread creation (realistic for PikeOS); identifiable atomic actions in the code;
- Assumption: Mapping from abstract atomic actions in the model to code-positions known.
- Abstract execution sequences were generated to .gdb scripts forcing explicit thread-switches of the SUT executed under gdb.
How to test concurrent programs implementing a model?

\[ \sigma \models o_1 \leftarrow \text{SUT}_1 \ \iota_1; \ldots ; o_n \leftarrow \text{SUT}_n \ \iota_n; \text{return}(res = [o_1 \ldots o_n]) \]
Conclusion

• HOL-TestGen is an Advanced Model-based Testing Environment built on top of Isabelle/HOL

• Allows to establish a Link between a formal System Model in Isabelle/HOL and Real Code by (semi)-automated generation of tests.

• Smooth Integration of Test and Proof!