Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff\(^1\)

\(^1\)Université Paris-Sud, LRI, Orsay, France
wolff@lri.fr

A Tutorial at the LRI
Orsay, 15th Jan 2009
Outline

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 ⇒ 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:

  ```isabelle
  apply (gen_test_cases "prog")
  ```

- Concluded by the command:

  ```isabelle
  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):
  
  \[
  \text{is\_sorted (prog [])}
  \]
  
  \[
  \text{is\_sorted (prog [?X1X17])}
  \]
  
  \[
  \text{is\_sorted (prog [?X2X13, ?X1X12])}
  \]
  
  \[
  \text{is\_sorted (prog [?X3X7, ?X2X6, ?X1X5])}
  \]

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

  \[
  \text{THYP((}\exists x. \text{is\_sorted (prog [x]))} \rightarrow (\forall x. \text{is\_sorted(prog [x])))}
  \]
  
  \[
  \ldots
  \]
  
  \[
  \text{THYP((}\forall l. 4 < |l| \rightarrow \text{is\_sorted(prog l))}
  \]

- We will discuss these hypotheses later in great detail.
Step IV: Test Data Generation

- On the test theorem, all sorts of logical 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])
  ```
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
theory List_test
imports Main begin
consts is_sorted:: "('a::ord) list ⇒ bool"
primrec "is_sorted [] = True"
"is_sorted (x#xs) = case xs of
  [] ⇒ True
  | y#ys ⇒ ((x < y) ∨ (x = y)) ∧ is_sorted xs"

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

gen_test_data "test_sorting"
gen_test_script "test_lists.sml" list prog
end
Testing an Implementation

Executing the generated test script may result in:

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

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

Overall result: failed
Tool-Demo!

Figure: HOL-TestGen Using Proof General at one Glance
Outline

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.,
  \[
  \forall x. (P x \land Q x) = (\forall x. P x) \land (\forall x. P x) \\
  \bigcup x \in S. (P x \cup Q x) = (\bigcup x \in S. P x) \cup (\bigcup x \in S. Q x) \\
  [A = A'; A \implies B = B'] \implies (A \land B) = (A' \land B')
  \]
The Core Tableaux-Calculus

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

  \[ t = t \quad \text{true} \quad P \land Q \quad P \lor Q \quad P \rightarrow Q \quad \neg P \]

- **Safe Elimination** Rules:

  \[ \{P, Q\} \quad [P] \quad [Q] \quad \{\neg P, Q\} \quad [P] \quad [Q] \]

  \[ \text{false} \quad P \land Q \quad R \quad P \lor Q \quad R \quad R \quad P \rightarrow Q \quad R \quad R \quad R \]

B. Wolff

HOL-TestGen: Theorem-prover based Testing

A Tutorial at the LRI 23
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
  \[
  \exists x. P x \quad \bigwedge x. Q
  \frac{}{Q} \\
  \]

- Critical Rewrite Rule:
  \[
  \text{if } P \text{ then } A \text{ else } B = (P \rightarrow A) \land (\neg P \rightarrow B)
  \]
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”:

\[
\text{THYP} : \text{bool} \Rightarrow \text{bool} \\
\text{THYP}(x) \equiv x
\]
What to do with infinite data structures of type $\tau$?

Conceptually, we split the set of all data of type $\tau$ into

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

1) \( |x :: \tau| < 2 = (x = []) \lor (\exists \ a. x = [a]) \)
2) \( |x :: \tau| < 3 = (x = []) \lor (\exists \ a. x = [a]) \lor (\exists \ a b. x = [a,b]) \)
3) \( |x :: \tau| < 4 = (x = []) \lor (\exists \ a. x = [a]) \lor (\exists \ a b. x = [a,b]) \lor (\exists \ a b c. x = [a,b,c]) \)
Taming Infinity I: Data Separation Rules

This motivates the (derived) data-separation rule:

1. \((\tau = \alpha \text{ list}, k = 3)\):

\[
\begin{align*}
\text{[}x = []\text{]} & \quad \text{[}x = [a]\text{]} & \quad \text{[}x = [a, b]\text{]} \\
\text{.} & \quad \text{[}x = [a]\text{]} & \quad \text{[}x = [a, b]\text{]} \\
P & \quad \bigwedge a. & \quad \bigwedge a \; b. & \quad P & \quad \text{THYP } M \\
\hline
P & \quad P & \quad P & \quad P & \quad P
\end{align*}
\]

2. Here, \(M\) is an abbreviation for:

\[
\forall x. \ k < |x| \rightarrow P \; x
\]
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 ...*
Using the **uniformity hypothesis**, a test case:

n) \[ \left[ C_1 x; \ldots; C_m x \right] \implies TS x \]

is transformed into:

n) \[ \left[ C_1 ?x; \ldots; C_m ?x \right] \implies TS ?x \]

n+1) \( THYP((\exists x. C_1 x \ldots C_m x \implies TS x) \implies (\forall x. C_1 x \ldots C_m x \implies TS x)) \)
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.:

\[
\left\lbrack \neg (\text{prog } x = c); \neg (\text{prog } x = d) \right\rbrack \Rightarrow A
\]

is transformed equivalently into

\[
\left\lbrack \neg A \right\rbrack \Rightarrow (\text{prog } x = c) \lor (\text{prog } x = d)
\]

- 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 ⇒ prog x S"
apply(gen_testcase 0 0)
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \text{ S} \]
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow_{\text{prog}} x \text{ mem } S \]

is transformed via data-separation lemma to:

1. \[ S = [] \Rightarrow x \text{ mem } S \rightarrow_{\text{prog}} x \text{ mem } S \]

2. \[ \forall a. \ S = [a] \Rightarrow x \text{ mem } S \rightarrow_{\text{prog}} x \text{ mem } S \]

3. \[ \forall a \ b. \ S = [a,b] \Rightarrow x \text{ mem } S \rightarrow_{\text{prog}} x \text{ mem } S \]

4. THYP(\[ \forall S. \ 3 \leq |S| \rightarrow x \text{ mem } S \rightarrow_{\text{prog}} x \text{ mem } S \])
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \text{ S} \]

canonization leads to:

1. \[ x \text{ mem } [] \rightarrow \text{prog } x \text{ []} \]

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

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

4. \[ \text{THYP}(\forall S. 3 \leq |S| \rightarrow x \text{ mem } S \rightarrow \text{prog } x \text{ S}) \]
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \text{ S} \]

which is reduced via the equation for \( \text{mem} \):

1. \( \text{false} \rightarrow \text{prog } x \cdot \]

2. \( \bigwedge a. \text{ if } a = x \text{ then True} \)
   \hspace{1cm} \text{else } x \text{ mem } [] \rightarrow \text{prog } x \cdot [a]\]

3. \( \bigwedge a \ b. \text{ if } a = x \text{ then True} \)
   \hspace{1cm} \text{else } x \text{ mem } [b] \rightarrow \text{prog } x \cdot [a,b]\]

4. \( \text{THYP}(3 \leq |S| \rightarrow x \text{ mem } S \rightarrow \text{prog } x \cdot S)\]
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \text{ S} \]

erasure for unsatisfyable constraints and rewriting conditionals yields:

2. \( \forall a. \ a = x \lor (a \neq x \land \text{false}) \rightarrow \text{prog } x [a] \)

3. \( \forall a \ b. \ a = x \lor (a \neq x \land x \text{ mem } [b]) \rightarrow \text{prog } x [a,b] \)

4. \( \forall S. \ 3 \leq |S| \rightarrow x \text{ mem } S \rightarrow \text{prog } x \text{ S} \)
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \ S \]

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

2. \( \bigwedge a. \text{prog } a \ [a] \)

3. \( \bigwedge a\ b. \ a = x \rightarrow \text{prog } x \ [a, b] \)
3’. \( \bigwedge a\ b. \ [a \neq x; \ x \text{ mem } [b]] \rightarrow \text{prog } x \ [a, b] \)
4. \( \text{THYP}(\forall \ S. \ 3 \leq |S| \rightarrow x \text{ mem } S \rightarrow \text{prog } x \ S) \)
Sample Derivation of Test Theorems

Example

\[
\begin{align*}
x \ mem \ S & \rightarrow prog \ x \ S \\
\ldots \text{which is reduced by canonization and rewriting of mem to:}
\end{align*}
\]

2. \( \bigwedge a. \ prog \ x \ [x] \)

3. \( \bigwedge a \ b. \ prog \ x \ [x,b] \)

3'. \( \bigwedge a \ b. \ a \neq x \Rightarrow prog \ x \ [a,x] \)

4. \( \text{THYP}(\forall \ S. \ 3 \leq |S| \rightarrow x \ mem \ S \rightarrow prog \ x \ S) \)
Sample Derivation of Test Theorems

Example

\[ x \text{ mem } S \rightarrow \text{prog } x \text{ S} \]

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

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

... 

7. \( \text{THYP}(\forall S. 3 \leq |S| \rightarrow x \text{ mem } S \rightarrow \text{prog } x \text{ S}) \)
Summing up:

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

$$\left[ TC_1; \ldots; TC_n; THYP \ H_1; \ldots; THYP \ H_m \right] \implies TS$$

where the test cases $TC_i$ have the form:

$$\left[ C_1x; \ldots; C_mx; THYP \ H_1; \ldots; THYP \ H_m \right] \implies P \ x \ (prog \ x)$$

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:
  \[ \Rightarrow \] More Rules, better results . . .
What makes a Test-case “Bad”

- redundancy.
- many unsatisfiable constraints.
- many constraints with unclear logical status.
- constraints that are difficult to solve. (like arithmetics).
Case Studies: Red-black Trees

Motivation

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

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

Red-Black Trees:

**Red Invariant:** each red node has a black parent.

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

datatype
color = R | B
tree = E | T color (α tree) (β::ord item) (α tree)
Modeling Red-black Trees II

Red-Black Trees: Test Theory

**consts**
- redinv :: tree ⇒ bool
- blackinv :: tree ⇒ bool

**recdef** blackinv measure (∧ t. (size t))
- blackinv E = True
- blackinv (T color a y b) =
  - ((blackinv a) ∧ (blackinv b)
  ∧ ((max B (height a)) = (max B (height b))))

recdev redinv measure ...
Red-black Trees: Test Specification

- Red-Black Trees: Test Specification

**test_spec:**
"isord t \land redinv t \land blackinv t
\land isin (y::int) t
\rightarrow
(blackinv(prog(y,t)))"

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

- Using the standard-workflows results, among others:

  RSF \rightarrow blackinv (prog (100, T B E 7 E))
  blackinv (prog (-91, T B (T R E -91 E) 5 E))
Red-black Trees: A first Summary

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

- On the one hand:
  - random-solving is nearly impossible for solutions which are “difficult” to find
  - only a small fraction of trees with depth $k$ are balanced

- On the other hand:
  - we can quite easily construct valid red-black trees interactively.
Red-black Trees: A first Summary

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

- On the one hand:
  - random-solving is nearly impossible for solutions which are "difficult" to find
  - only a small fraction of trees with depth $k$ are balanced

- On the other hand:
  - we can quite easily construct valid red-black trees interactively.

Question:
Can we improve the test-data generation by using our knowledge about red-black trees?
Red-black Trees: Hierarchical Testing I

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

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

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

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

- But this is tedious . . .
Red-black Trees: Hierarchical Testing I

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

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

  **lemma** maxB\_0\_1:
  "max\_B\_height (E:: int tree) = 0"

  **lemma** maxB\_0\_5:
  "max\_B\_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"

- But this is tedious . . . and error-prone
How to Improve Test-Theorems

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

\[(\text{max}_B\text{ height } (T \times t1 \text{ val } t2) = 0) \Rightarrow (x = R)\]

\[(\text{max}_B\text{ height } x = 0) =
\ (x = E \lor \exists a y b. x = T R a y b \land \\ \quad \quad \quad \quad \quad \text{max}(\text{max}_B\text{ height } a)
\ (\text{max}_B\text{ height } b) = 0)\]

- 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

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

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

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

\[ \text{pre} \ x \ \rightarrow \ \text{post} \ x \ (\text{prog} \ x) \]

- 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 \textit{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 \textit{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:
  
  \[ \text{accept trace} \rightarrow \text{P}(\text{Mfold trace } \sigma_0 \text{ prog}) \]

where

\[
\text{Mfold } [\sigma] \sigma = \text{Some } \sigma \\
\text{MFold } (\text{input}::R) = \text{case } \text{prog}(\text{input}, \sigma) \text{ of} \\
\quad \text{None } \Rightarrow \text{None} \\
\quad | \text{ Some } \sigma' \Rightarrow \text{Mfold R } \sigma' \text{ prog}
\]

- 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:

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

  a channel is requested within a bound \(X\), a channel \(Y\) is chosen by the server, the client communicates along this channel . . .
Example: A Reactive System I

A toy client-server system:

\[ \text{req}\ X \rightarrow \text{port}\! Y[Y < X] \rightarrow \]
\[ (\text{rec}\ N. \text{send}\! D.Y \rightarrow \text{ack} \rightarrow N \]
\[ \Box \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP} ) \]

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:

\[ X \text{ and } Y \text{ are only known at runtime!} \]

- Mfold is a program that manages a state at test run time.
- use an environment that keeps track of the instances of \( X \) and \( Y \)?

**Infrastructure:** An **observer** maps abstract events (req \( X \), port \( Y \), ...) in traces to concrete events (req 4, port 2, ...) in runs!
Example: A Reactive System

- **Infrastructure**: the observer

  observer rebind substitute postcond ioprog \( \equiv \)
  
  \((\lambda \text{input}. (\lambda (\sigma, \sigma'). \textbf{let} \text{input'} = \text{substitute} \sigma \text{input} \textbf{in} \text{case} \text{ioprog} \text{input'} \sigma' \textbf{of} \))

  \[
  \begin{align*}
  \text{None} & \Rightarrow \text{None} (* \text{ioprog failure} \quad \text{eg. timeout} \ldots *) \\
  | \quad \text{Some} (\text{output}, \sigma'') & \Rightarrow \textbf{let} \sigma'' = \text{rebind} \sigma \text{output} \textbf{in} \\
  & (\text{if} \ \text{postcond} (\sigma'', \sigma''') \\text{input'} \ \text{output} \\
  & \text{then} \ \text{Some}(\sigma'', \sigma''') \\
  & \text{else} \ \text{None} (* \text{postcond failure} *) )
  \end{align*}
  \]
Example: A Reactive Test IV

- Reactive Test-Specification Pattern:
  
  $\text{accept } trace \rightarrow P(M\text{fold } trace \sigma_0 (\text{observer rebind subst postcond } \text{ioprog}))$

- for reactive systems!
Motivation

- So far, we have used HOL-TestGen only for test specifications of the form:
  
  \[ \text{pre } x \rightarrow \text{post } x \ (\text{prog } x) \]

- We have seen, this does not exclude to model reactive sequence test in HOL-TestGen.

- However, this seems still exclude the HOL-TestGen approach from program-based testing approaches (such as JavaPathfinder-SE or Pexx).
How to Realize White-box-Tests in HOL-TestGen?

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

- We embed an imperative core-language — called IMP — into HOL-TestGen, by defining its syntax and semantics.
- We add a specification mechanism for IMP: Hoare-Triples.
- We derive rules for symbolic evaluation and loop-unfolding.
The (abstract) IMP syntax is defined in Com.thy.

Com = Main +

**typedecl** loc

types

val = nat (*arb.*)
state = loc⇒val
aexp = state⇒val
bexp = state⇒bool

**datatype** com =

SKIP

| ":==" loc aexp (**infixl** 60)
| Semi com com ("_ ; _"[60, 60]10)
| Cond bexp com com

(" IF _ THEN _ ELSE _"60)
| While bexp com ("WHILE _ DO _"60)

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

\[ \begin{align*}
\text{tm} & :== \lambda s. 1; \\
\text{sum} & :== \lambda s. 1; \\
\text{i} & :== \lambda s. 0; \\
\text{WHILE} & \quad \lambda s. (s \text{ sum}) <= (s \ a) \ \text{DO} \\
& \quad (\text{i} \quad :== \lambda s. (s \text{ i}) + 1; \\
& \text{tm} \quad :== \lambda s. (s \text{ tm}) + 2; \\
& \text{sum} \quad :== \lambda s. (s \text{ tm}) + (s \text{ sum}))
\end{align*} \]

How does this program work?

Note: There is the implicit assumption, that \( \text{tm}, \text{sum} \) and \( \text{i} \) are distinct locations, i.e. they are not aliases from each other!
IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin
**IMP Semantics I: (Natural Semantics**

*Natural semantics going back to Plotkin*

**idea:** programs relates states.

```
state  a ::= b  \rightarrow  state'

state'  \rightarrow  state''  \rightarrow  state'''

WHILE...

SKIP
```
**IMP Semantics I: (Natural Semantics**

*Natural semantics going back to Plotkin*

**idea:** programs relates states.

\[
\text{state} \xrightarrow{a := b} \text{state}' \quad \text{WHILE} \ldots \quad \text{SKIP} \quad \text{state}''
\]

\textbf{consts} evalc :: (com \times \text{state} \times \text{state}) set

\textbf{translations} "\langle c, s \rangle \rightarrow_c s' " \equiv "(c, s, s') \in \text{evalc}"
**IMP Semantics I: (Natural Semantics)**

**Natural semantics going back to Plotkin**

**idea**: programs relates states.

\[
\text{state} \xrightarrow{a := b} \text{state}' \quad \text{WHILE} \ldots \quad \text{state}''
\]

\[
\text{state}'' \xrightarrow{\text{SKIP}} \text{state}'''
\]

**consts** \(\text{evalc} :: (\text{com} \times \text{state} \times \text{state}) \text{ set}\)

**translations** "\(\left\langle c, s \right\rangle \xrightarrow{c} s'\)" \(\equiv \left( c, s, s' \right) \in \text{evalc}\)"
The transition relation of natural semantics is inductively defined.
The transition relation of natural semantics is inductively defined.

This means intuitively: The evaluation steps defined by the following rules are the only possible steps.
The transition relation of natural semantics is inductively defined.

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

Let’s go . . .
The natural semantics as inductive definition:

\textbf{inductive} evalc

\begin{itemize}
  \item \textbf{intrs}
  \item \textbf{Skip:} \( \langle \text{SKIP}, s \rangle \xrightarrow{c} s \)
  \item \textbf{Assign:} \( \langle x := a, s \rangle \xrightarrow{c} s[x \mapsto a \ s] \)
\end{itemize}
The natural semantics as inductive definition:

**inductive** evalc

intrs

Skip: \(\langle \text{SKIP}, s \rangle \xrightarrow{c} s\)

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

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

\[
\text{update} \ s \ x \ v \equiv \lambda y. \text{if } y = x \text{ then } v \text{ else } s \ y
\]
The natural semantics as inductive definition:

**inductive** evalc

```
intrs

Skip:  ⟨SKIP,s⟩ →[c] s
Assign:  ⟨x ::= a,s⟩ →[c] s[x ↦ a s]
```

Note that s[x ↦ a s] is an abbreviation for update s x (a s), where

```
update s x v ≡ λy. if y=x then v else s y
```

Note that a is of type aexp or bexp.
Excursion: A minimal memory model:

\[(s[x \mapsto E]) \cdot x = E\]
\[x \neq y \implies (s[x \mapsto E]) \cdot y = s \cdot y\]

This small memory theory contains the *typical* rules for updating and memory-access. Note that this rewrite system is in fact executable!
The semantics for the sequential composition of statements can be described as follows:

\[
\text{Semi: } \left[ \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c} s'' \right] \implies \langle c; c', s \rangle \xrightarrow{c} s''
\]
The semantics for the sequential composition of statements can be described as follows:

\[
\text{Semi: } \left[ \langle c,s \rangle \xrightarrow{c} s' ; \langle c',s' \rangle \xrightarrow{c} s'' \right] \implies \langle c;c', s \rangle \xrightarrow{c} s''
\]

Rationale of natural semantics:

- if you can “jump” via \( c \) from \( s \) to \( s' \), \( \ldots \)
The semantics for the sequential composition of statements can be described as follows:

\[
\text{Semi: } \left[ \langle c, s \rangle \xrightarrow{c} s' ; \langle c', s' \rangle \xrightarrow{c} s'' \right] \implies \langle c ; c', s \rangle \xrightarrow{c} s''
\]

Rationale of natural semantics:

- if you can “jump” via \( c \) from \( s \) to \( s' \), ...
- and if you can “jump” via \( c' \) from \( s' \) to \( s'' \) ...
The semantics for the sequential composition of statements can be described as follows:

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

Rationale of natural semantics:
- if you can “jump” via \(c\) from \(s\) to \(s'\), . . .
- and if you can “jump” via \(c'\) from \(s'\) to \(s''\) . . .
- then this means that you can “jump” via the composition \(c; c'\) from \(c\) to \(c''\).
The other constructs of the language are treated analogously:

**IfTrue:**  
\[
\begin{align*}
&\left[ \ b \ s; \ \langle c,s \rangle \xrightarrow{c} s' \right] \\
\implies &\langle \text{IF } b \ \text{THEN } c \ \text{ELSE } c', s \rangle \xrightarrow{c} s'
\end{align*}
\]

**IfFalse:**  
\[
\begin{align*}
&\left[ \neg b \ s; \ \langle c',s \rangle \xrightarrow{c} s' \right] \\
\implies &\langle \text{IF } b \ \text{THEN } c \ \text{ELSE } c', s \rangle \xrightarrow{c} s'
\end{align*}
\]

**WhileFalse:**  
\[
\left[ \neg b \ s \right] \\
\implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \xrightarrow{c} s
\]

**WhileTrue:**  
\[
\begin{align*}
&\left[ b \ s; \ \langle c,s \rangle \xrightarrow{c} s'; \langle \text{WHILE } b \ \text{DO } c,s' \rangle \xrightarrow{c} s'' \right] \\
\implies &\langle \text{WHILE } b \ \text{DO } c, s \rangle \xrightarrow{c} s''
\end{align*}
\]

Note that for non-terminating programs no final state can be derived!
The transition semantics is inspired by abstract machines.
The transition semantics is inspired by abstract machines.

idea: programs relate “configurations”.

\[ a ::= b; X, state \quad X, state' \quad X'', state'' \quad X''', state'''

\]
The transition semantics is inspired by abstract machines.

idea: programs relate “configurations”.

\[ a :== b; X, state \rightarrow X, state' \rightarrow X'', state'' \]

consts \( \text{evalc1} :: ((\text{com} \times \text{state}) \times (\text{com} \times \text{state})) \text{ set} \)

translations "\( cs \rightarrow cs'' \equiv (cs, cs') \in \text{evalc1} \)"
**inductive** evalc1

**intro**

Assign: \((x:=a,s) \rightarrow (\text{SKIP}, s[x \mapsto a \ s])\)

Semi1: \((\text{SKIP};c,s) \rightarrow (c,s)\)

Semi2: \((c,s) \rightarrow (c'',s')\)

\[\implies (c;c',s) \rightarrow (c'';c',s')\]
**inductive** evalc1

intro

Assign: \( (x:=a,s) \rightarrow 1 \rightarrow (\text{SKIP}, s[x \rightarrow a \, s]) \)

Semi1: \( (\text{SKIP};c,s) \rightarrow 1 \rightarrow (c,s) \)

Semi2: \( (c,s) \rightarrow 1 \rightarrow (c'',s') \)

\[ \Rightarrow (c;c',s) \rightarrow 1 \rightarrow (c'';c',s') \]

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .
**inductive** evalc1

intro

Assign: \((x:=a,s) \rightarrow (\text{SKIP}, s[x \rightarrow a s])\)

Semi1: \((\text{SKIP};c,s) \rightarrow (c,s)\)

Semi2: \((c,s) \rightarrow (c'',s')\)

\[\Rightarrow (c;c',s) \rightarrow (c'';c',s')\]

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .
- this stack can also be seen as a *program counter* . . .
- transition semantics is close to an abstract machine.
IfTrue:
\[ b \ s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) -1 \rightarrow (c', s) \]

IfFalse:
\[ \neg b \ s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) -1 \rightarrow (c'', s) \]

WhileFalse:
\[ \neg b \ s \implies (\text{WHILE } b \text{ DO } c, s) -1 \rightarrow (\text{SKIP}, s) \]

WhileTrue:
\[ b \ s \implies (\text{WHILE } b \text{ DO } c, s) -1 \rightarrow (c; \text{WHILE } b \text{ DO } c, s) \]
IfTrue:
\[ b \ s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'',s) -1 \rightarrow (c',s) \]

IfFalse:
\[ \neg b \ s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'',s) -1 \rightarrow (c'',s) \]

WhileFalse:
\[ \neg b \ s \implies (\text{WHILE } b \text{ DO } c,s) -1 \rightarrow (\text{SKIP},s) \]

WhileTrue:
\[ b \ s \implies (\text{WHILE } b \text{ DO } c,s) -1 \rightarrow (c;\text{WHILE } b \text{ DO } c,s) \]

A non-terminating loop always leads to successor configurations ...
IMP Semantics III: (Denotational Semantics)

Idea:

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

As semantic domain we choose the state relation:

types $\text{com}_{\text{den}} = (\text{state} \times \text{state}) \text{ set}$

and declare the semantic function:

consts $\text{C} :: \text{com} \Rightarrow \text{com}_{\text{den}}$

The semantic function $\text{C}$ is defined recursively over the syntax.
IMP Semantics III: (Denotational Semantics)

Idea:

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

Idea:

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

\[
\text{types com_den} = (\text{state } \times \text{state}) \text{ set}
\]
Idea:

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

As semantic domain we choose the state relation:

\[
\text{types } \text{com\_den} = (\text{state} \times \text{state}) \text{ set}
\]

and declare the semantic function:

\[
\text{consts } C :: \text{com } \Rightarrow \text{com\_den}
\]

The semantic function \( C \) is defined recursively over the syntax.
primrec

\[ C(\text{SKIP}) = \text{Id} \] (* \equiv \text{idenity relation} *)

\[ C(x \triangleright= a) = \{(s,t). \; t = s[x \mapsto a \; s]\} \]

\[ C(c \;;\; c') = C(c') \circ C(c) \] (* \equiv \text{seq. composition} *)

\[ C(\text{IF } b \text{ THEN } c' \text{ ELSE } c'') = \]
\[ \{ (s,t). \; (s,t) \in C(c') \land b(s) \} \cup \]
\[ \{ (s,t). \; (s,t) \in C(c'') \land \neg b(s) \}\]

\[ C(\text{WHILE } b \text{ DO } c) = \text{lfp} (\Gamma b (C(c)))\]
**primrec**

\[
\begin{align*}
    C(\text{SKIP}) &= \text{Id} \\
    C(x := a) &= \{(s,t). t = s[x \mapsto a] \} \\
    C(c ; c') &= C(c') \circ C(c) \\
    C(\text{IF } b \text{ THEN } c' \text{ ELSE } c'') &= \\
    &\{(s,t). (s,t) \in C(c') \land b(s) \} \cup \\
    &\{(s,t). (s,t) \in C(c'') \land \neg b(s) \}\end{align*}
\]

\[
C(\text{WHILE } b \text{ DO } c) = \text{lfp } (\Gamma b (C(c)))
\]

where:

\[
\Gamma b c \equiv (\lambda \varphi. \{(s,t). (s,t) \in (\varphi \circ c) \land b(s) \} \cup \\
\{(s,t). s=t \land \neg b(s) \})
\]

and where the least-fixpoint-operator \( \text{lfp } F \) corresponds in this special case to:

\[
\bigcup_{n \in \mathbb{N}} F^n
\]

(* \( \equiv \text{identity relation } \) * )

(* \( \equiv \text{seq. composition } \) * )
Theorem: Natural and Transition Semantics Equivalent

\[(c, s) \rightarrow^* (\text{SKIP}, t) = (\langle c, s \rangle \rightarrow_c t)\]

where \(cs \rightarrow^* cs' \equiv (cs, cs') \in \text{evalc1}^*\), i.e. the new arrow denotes the transitive closure over old one.
IMP Semantics: Theorems I

Theorem: Natural and Transition Semantics Equivalent

\[(c, s) \rightarrow^{*} (\text{SKIP}, t) = (\langle c, s \rangle \rightarrow^{c} t)\]

where \(cs \rightarrow^{*} cs' \equiv (cs, cs') \in \text{eval}_{c1}^*,\) i.e. the new arrow denotes the transitive closure over old one.

Theorem: Denotational and Natural Semantics Equivalent

\[((s, t) \in C c) = (\langle c, s \rangle \rightarrow^{c} t)\]
Theorem: Natural and Transition Semantics Equivalent

\[(c, s) \xrightarrow{*} (\text{SKIP}, t) = (\langle c, s \rangle \xrightarrow{c} t)\]

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

Theorem: Denotational and Natural Semantics Equivalent

\[((s, t) \in C c) = (\langle c, s \rangle \xrightarrow{c} t)\]

i.e. all three semantics are closely related!
**IMP Semantics: Theorems II**

**Theorem:** Natural Semantics can be evaluated equationally !!!

\[
\begin{align*}
\langle \text{SKIP}, s \rangle \xrightarrow{c} s' &= (s' = s) \\
\langle x := a, s \rangle \xrightarrow{c} s' &= (s' = s[x \rightarrow a s]) \\
\langle c; c', s \rangle \xrightarrow{c} s' &= (\exists s''. \langle c, s \rangle \xrightarrow{c} s'' \land \langle c', s'' \rangle \xrightarrow{c} s') \\
\langle \text{IF } b \text{ THEN } c \text{ ELSE } c', s \rangle \xrightarrow{c} s' &= (b s \land \langle c, s \rangle \xrightarrow{c} s') \lor \\
& \quad (\neg b s \land \langle c', s \rangle \xrightarrow{c} s')
\end{align*}
\]

**Note:** This is the key for evaluating a program symbolically !!!
Example: “a:==2;b:==2*a”

\[
\langle a:==\lambda s. 2; b:==\lambda s. 2 \ast (s \ a), s \rangle \xrightarrow{c} s'
\]

\[
\equiv (\exists s''. \langle a:==\lambda s. 2, s \rangle \xrightarrow{c} s'' \land \langle b:==\lambda s. 2 \ast (s \ a), s'' \rangle \xrightarrow{c} s')
\]

\[
\equiv (\exists s''. \ s'' = s[a\mapsto(\lambda s. 2) \ s] \land s' = s''[b\mapsto(\lambda s. 2 \ast (s \ a)) \ s''])
\]

\[
\equiv (\exists s''. \ s'' = s[a\mapsto 2] \land s' = s''[b\mapsto 2 \ast (s'' \ a)])
\]

\[
\equiv s' = s[a\mapsto 2][b\mapsto 2 \ast (s[a \mapsto 2] \ a)]
\]

\[
\equiv s' = s[a\mapsto 2][b\mapsto 2 \ast 2]
\]

\[
\equiv s' = s[a\mapsto 2][b\mapsto 4]
\]

Note:

1. The \(\lambda\)-notation is perhaps a bit irritating, but helps to get the nitty-gritty details of substitution right.

2. The forth step is correct due to the “one-point-rule”
   \((\exists x. \ x = e \land P(x)) = P(e)\).

3. This does not work for the loop and for recursion...
IMP Semantics: Theorems III

Denotational semantics makes it easy to prove facts like:

\[ C(\text{WHILE } b \text{ DO } c) = C(\text{IF } b \text{ THEN } c; \text{ WHILE } b \text{ DO } c \text{ ELSE SKIP}) \]
\[ C(\text{SKIP}; c) = C(c) \]
\[ C(c; \text{ SKIP}) = C(c) \]
\[ C((c; d); e) = C(c;(d;e)) \]
\[ C((\text{IF } b \text{ THEN } c \text{ ELSE } d); e) = C(\text{IF } b \text{ THEN } c ; e \text{ ELSE } d ; e) \]

etc.
Program Annotations: Assertions revisited.

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

```plaintext
types  assn = state ⇒ bool
consts valid :: (assn × com × assn) ⇒ bool ("|= {__} _ {__}")
defs
  |= {P}c{Q} ≡ ∀ s. ∀ t. (s,t) ∈ C(c) → P s → Q t
```

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

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

\[ \forall s s'. \langle c,s \rangle \xrightarrow{c} s' \land P s \rightarrow Q s' \]

\[ \Rightarrow \models \{P\}c\{Q\} \]

or in more formal, natural-deduction notation:

\[ \left[ \langle c, s \rangle \rightarrow_{c} s', P s \right]_{s,s'} \]

\[ \vdots \]

\[ Q s' \]

\[ \models \{P\} c \{Q\} \]

Applied in backwards-inference, this rule *generates* the constraints for the states that were amenable to equational evaluation rules shown before.
**Example:** \(|= \{0 \leq x\} \ a :== x \ ; \ b :== 2 \ast a \ {0 \leq b}\)"

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

\(\iff \ s’ = s[a \mapsto s \ x][b \mapsto 2 \ast (s[a \mapsto s \ x] \ a)] \land 0 \leq s \ x \longrightarrow 0 \leq s’ \ b\)

\(\equiv \ s’ = s[a \mapsto s \ x][b \mapsto 2 \ast (s \ x)] \land \text{"PRE s’" } \longrightarrow \text{"POST s’" }\)

\(\equiv \ \text{"PRE s’" } \longrightarrow \text{"POST (s[a \mapsto s \ x][b \mapsto 2 \ast (s \ x)]) "}\)

**Note:**

- **Note:** the logical constraint
  \(s’ = s[a \mapsto s \ x][b \mapsto 2 \ast s \ x] \land 0 \leq s \ x\) consists of the constraint that functionally relate pre-state \(s\) to post-state \(s’\) and the **Path-Condition** (in this case just "PRE s’").

- This also works for conditionals ... Revise!

- The implication is actually the core validation problem: It means that for a certain path, we search for the solution of a path condition that validates the post-condition. We can decide to 1) keep it as test hypothesis, 2) test \(k\) witnesses and add a uniformity hypothesis, or 3) verify it.
Validation of Post-Conditions for a Given Path:

Ad 1 : Add $\text{THYP}(\text{PRE } s \rightarrow \text{POST}(s[a \mapsto s \cdot x][b \mapsto 2 \times (s \cdot x)]))$
(is: $\text{THYP}(0 \leq s \cdot x \rightarrow 0 \leq 2 \times s \cdot x))$ as test hypothesis.

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

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

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

Control-heuristics necessary.
Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP’s, sequentials, and conditionals.
Handling Loops (and Recursion).

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

What to do with loops ???
Handling Loops (and Recursion).

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

What to do with loops ???

Answer: Unfolding to a certain depth.
Handling Loops (and Recursion).

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

What to do with loops ???

Answer: Unfolding to a certain depth.

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

consts unwind :: "nat × com ⇒ com"

recdef unwind "less_than <*lex*> measure(λ s. size s)"
"unwind(n, SKIP) = SKIP"
"unwind(n, a ::= E) = (a ::= E)"
"unwind(n, IF b THEN c ELSE d) = IF b THEN unwind(n,c) ELSE unwind(n,d)"
"unwind(n, WHILE b DO c) =
  if 0 < n
    then IF b THEN unwind(n,c) @@ unwind(n− 1, WHILE b DO c) ELSE SKIP
    else WHILE b DO unwind(0,c))"
"unwind(n, SKIP; c) = unwind(n, c)"
"unwind(n, c ; SKIP) = unwind(n, c)"
"unwind(n, (IF b THEN c ELSE d) ; e) =
  ( IF b THEN (unwind(n,c;e)) ELSE (unwind(n,d;e)))"
"unwind(n, (c ; d); e) = (unwind(n, c;d)) @@ (unwind(n,e))"
"unwind(n, c ; d) = (unwind(n, c)) @@ (unwind(n, d))"
Handling Loops (and Recursion).

where the primitive recursive auxiliary function \( c@@d \) appends a command \( d \) to the last command in \( c \) that is reachable from the root via sequential composition modes.

```plaintext
consts "@@" :: "[com,com] ⇒ com" (infixr 70)
primrec
  "SKIP @@ c = c"
  "(x:= E) @@ c = ((x:= E); c)"
  "(c;d) @@ e = (c; d @@ e)"
  "( IF b THEN c ELSE d) @@ e = (IF b THENc @@ e ELSEEd @@ e)"
  "(WHILE b DO c) @@ e = ((WHILE b DOc);e)"
```
Handling Loops (and Recursion).

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

**Theorem: Unwind and Concat correct**

\[ C(c @@ d) = C(c;d) \] and \[ C(\text{unwind}(n,c)) = C(c) \]
Handling Loops (and Recursion).

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

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

\[ \forall s, s'. \langle \text{unwind}(n,c), s \rangle \overset{c}{\rightarrow} s' \land P s \rightarrow Q s' \]
\[ \Rightarrow \models \{P\} c \{Q\} \]

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

Or in natural deduction notation:

\[
\begin{array}{c}
\langle \text{unwind}(n, c), s \rangle \overset{c}{\rightarrow} s', P s \\
\vdots \\
Q s' \\
\hline
\models \{P\} c \{Q\}
\end{array}
\]
Handling Loops (and Recursion).

**Example:**

“$\models \{ \text{True} \} \ integer\_\text{squareroot} \ \{i^2 \leq a \land a \leq (i + 1)^2\}$”

Setting the depth to $n = 3$ and running the process yields:
Handling Loops (and Recursion).

Example:

“$$\models \{ \text{True} \} \text{ integer_squareroot } \{ i^2 \leq a \land a \leq (i + 1)^2 \}$$”

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

1. $$\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
9 \leq s \ a ; \langle \text{ WHILE } \lambda s. s \ \text{ sum } \leq s \ a \\
\begin{array}{l}
\text{ DO } i := \lambda s. \text{ Suc } (s \ i) ; \\
\text{ (tm := } \lambda s. \text{ Suc } (\text{ Suc } (s \ tm)) ; \\
\text{ sum := } \lambda s. s \ \text{ tm } + s \ \text{ sum } , \\
\text{ s(i := 3, tm := 7, sum := 16) } \rightarrow s' \\
\end{array}
\end{array}
\text{ } \rightarrow \text{ post } s'
\end{array}
\end{array}
\]$$

2. $$\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
4 \leq s \ a ; 8 < s \ a ; s' = s (i := 2, tm := 5, sum := 9) \} \rightarrow \text{ post } s'
\end{array}
\end{array}
\end{array}
\]$$

3. $$\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
1 \leq s \ a ; s \ a < 4 ; s' = s (i := 1, tm := 3, sum := 4) \} \rightarrow \text{ post } s'
\end{array}
\end{array}
\end{array}
\]$$

4. $$\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
s \ a = 0 ; s' = s(tm := 1, sum := 1, i := 0) \} \rightarrow \text{ post } s'
\end{array}
\end{array}
\end{array}
\]$$

which is a neat enumeration of all path-conditions for paths up to $$n = 3$$ times through the loop, except subgoal 1, which is:
Explicit test-Hypothesis in White-Box-Tests:

1. THYP(9 ≤ s a ∧ (∃ s. s sum ≤ s a
   D0 i ::= λs. Suc (s i);
   (tm ::= λs. Suc (Suc (s tm));
   sum ::= λs. s tm + s sum ),
   s(i := 3, tm := 7, sum := 16) ↝ s’
   → post s’)

... a kind of “structural” regularity hypothesis!
Summary: Program-based Tests in HOL-TestGen:

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

Open Questions:

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Intranet</th>
<th>DMZ</th>
<th>Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intranet</td>
<td>-</td>
<td>smtp, imap</td>
<td>all protocols except smtp</td>
</tr>
<tr>
<td>DMZ</td>
<td>∅</td>
<td>-</td>
<td>smtp</td>
</tr>
<tr>
<td>Internet</td>
<td>∅</td>
<td>http, smtp</td>
<td>-</td>
</tr>
</tbody>
</table>

**Diagram:**

- **DMZ**
- **Internet (extern)**
- **Intranet (intern)**

**Network Flows:**
- Intranet → DMZ
- DMZ → Intranet
- Intranet → Internet (extern)
- Internet (extern) → Intranet
- Internet (extern) → DMZ
- DMZ → Internet (extern)
A Bluffers Guide to Firewalls

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

packet filter.

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

First, we model a packet:

\[
\text{types} \ (\alpha, \beta) \ \text{packet} = "id \times \text{protocol} \times \alpha \text{src} \times \alpha \text{dest} \times \beta \text{content}"
\]

where

- **id**: a unique packet identifier, e.g., of type Integer
- **protocol**: the protocol, modeled using an enumeration type (e.g., ftp, http, smtp)
- **\(\alpha \text{ src} (\alpha \text{ dest})\)**: source (destination) address, e.g., using IPv4:

  \[
  \text{types} \\
  \text{ipv4}_\text{ip} = "(\text{int} \times \text{int} \times \text{int} \times \text{int})" \\
  \text{ipv4} = "(\text{ipv4}_\text{ip} \times \text{int})"
  \]

- **\(\beta \text{ content}\)**: content of a packet
The State-less Firewall Model II

- A firewall (packet filter) either accepts or denies a packet:
  
  \[
  \text{datatype} \\
  \alpha \text{ out } = \text{ accept } \alpha | \text{ deny }
  \]

- A policy is a map from packet to packet out:
  
  \[
  \text{types} \\
  (\alpha, \beta) \text{ Policy } = "(\alpha, \beta) \text{ packet } \rightarrow ((\alpha, \beta) \text{ packet) out"
  \]

- Writing policies is supported by a specialised combinator set
### Testing State-less Firewalls: An Example I

<table>
<thead>
<tr>
<th></th>
<th>Intranet</th>
<th>DMZ</th>
<th>Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intranet</td>
<td>-</td>
<td>smtp, imap</td>
<td>all protocols except smtp</td>
</tr>
<tr>
<td>DMZ</td>
<td>∅</td>
<td>-</td>
<td>smtp</td>
</tr>
<tr>
<td>Internet</td>
<td>∅</td>
<td>http, smtp</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Diagram:

- **DMZ**
- **Internet (extern)**
- **Intranet (intern)**

Arrows indicate directions of communication:
- **Intranet** → **DMZ**
- **DMZ** → **Intranet**
- **Internet** → **Intranet**
- **Intranet** → **Internet**
Testing State-less Firewalls: An Example II

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
<th>protocol</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td>DMZ</td>
<td>http</td>
<td>accept</td>
</tr>
<tr>
<td>Internet</td>
<td>DMZ</td>
<td>smtp</td>
<td>accept</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>deny</td>
</tr>
</tbody>
</table>

\textbf{constdefs} Internet_DMZ :: "(ipv4, content) Rule"

"Internet_DMZ \equiv 
(allow_prot_from_to smtp internet dmz) ++ 
(allow_prot_from_to http internet dmz)"

The policy can be modelled as follows:

\textbf{constdefs} test_policy :: "(ipv4,content) Policy"

"test_policy \equiv deny_all ++ Internet_DMZ ++ ..."
Testing State-less Firewalls: An Example III

- Using the test specification

  \texttt{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

Server

ftp_init

ftp_port_request

ftp_data

ftp_close

Client

ftp_data

ftp_close

ftp_port_request

ftp_init

Exception

ftp_data

ftp_close

ftp_port_req
**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:

  \[ \text{types} \ (\alpha, \beta, \gamma) \ FWState = "\alpha \times (\beta, \gamma) \ Policy" \]

- where FWStateTransition maps an incoming packet to a new state

  \[ \text{types} \ (\alpha, \beta, \gamma) \ FWStateTransition = \\
  "((\beta, \gamma) \ In\_Packet \times (\alpha, \beta, \gamma) \ FWState) \rightarrow \\
  ((\alpha, \beta, \gamma) \ FWState)" \]
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 3),
     (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 Addresses)
- Sequence Testing used for Stateful Firewalls
- Realistic, but amazingly concise model in HOL!
Conclusion I

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

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

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

B. Wolff

HOL-TestGen: Theorem-prover based Testing

A Tutorial at the LRI
Conclusion II

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

\[
\text{pre } x \quad \rightarrow \quad \text{post } x(\text{prog } x)
\]

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

B. Wolff
HOL-TestGen: Theorem-prover based Testing
A Tutorial at the LRI 105
Conclusion II

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

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

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

B. Wolff  
HOL-TestGen: Theorem-prover based Testing  
A Tutorial at the LRI
Conclusion II

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

\[
\text{accept } \text{trace} \implies P(\text{Mfold } \text{trace} \sigma_0 \\
\text{(observer observer rebind subst prog)})
\]

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

B. Wolff
HOL-TestGen: Theorem-prover based Testing
A Tutorial at the LRI


The HOL-TestGen Website.


Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder.
Outline

6. The HOL-TestGen System

7. A Hands-on Example
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 System

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

```isabelle
theory max_test = Testing:
...
end
```
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.
Now, abstract test cases for our test specification can (automatically) generated, e.g. by issuing

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

The generated test cases can be further processed, e.g., simplified using the usual Isabelle/HOL tactics.

After generating the test cases (and test hypothesis’) you should store your results, e.g.:

```
store_test_thm "max_test"
```
Test Data Selection

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

`gen_test_data "max_test"`
Test Script Generation

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

```
generate_test_script "test_max.sml" "max_test" "prog"
"myMax.max"
```
A Simple Testing Theory: max

theory max_test = Testing:

test_spec "prog a b = max a b"
  apply (gen_test_cases 1 3 "prog" simp: max_def)
store_test_thm "max_test"
gen_test_data "max_test"
genenerate_test_script "test_max.sml" "max_test" "prog"
  "myMax.max"
end
A (Automatically Generated) Test Script

```ml
structure TestDriver : sig end = struct

  val return = ref ~63;

  fun eval x2 x1 = let val ret = myMax.max x2 x1
                   in ((return := ret); ret) end

  fun retval () = SOME(! return);

  fun toString a = Int.toString a;

  val testres = [];
  val pre_0   = [];
  val post_0  = fn () => ( (eval ~23 69 = 69));
  val res_0   = TestHarness.check retval pre_0 post_0;

  val testres = testres@[res_0];
  val pre_1  = [];
  val post_1 = fn () => ( (eval ~11 ~15 = ~11));
  val res_1  = TestHarness.check retval pre_1 post_1;
  val testres = testres@[res_1];

  _ = TestHarness.printList toString testres;
end
```
Building the Test Executable

- Assume we want to test the SML implementation

```sml
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:

```sml
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
==============