HOL-TestGen 1.4.0
User Guide

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

Achim D. Brucker
brucker@member.fsf.org

Burkhart Wolff
wolff@wjpserver.cs.uni-sb.de

June 19, 2008

Note:
This manual describes HOL-TestGen version 1.4.0 (build: 8174). The manual of version 1.0.0 is also available as technical report number 482 from the department of computer science, ETH Zurich.

Contents

1 Introduction

Today, essentially two validation techniques for software are used: *software verification* and *software testing*. Whereas verification is rarely used in "real" software development, testing is widely-used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Djikstra's verdict* [19, p.6]:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

Abstraction Techniques: model-checking raised interest in techniques to abstract infinite to finite models. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [11, 18].

Systematic Testing: the discussion over *test adequacy criteria* [28], i. e. criteria solving the question "when did we test enough to meet a given test hypothesis," led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [22, 20].

Specification Animation: constructing counter-examples has raised interest also in the theorem proving community, since combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [10, 23, 17].

The first two areas are motivated by the question "are we building the program right?" the latter is focused on the question "are we specifying the right program?" While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows, that it simply does not apply in practically important situations. In particular, if a formal model of the environment of a software system (e. g. based among others on the operation system, middleware or external libraries) must be reverse-engineered, testing ("experimenting") is without alternative (see [14]).

Following standard terminology [28], our approach is a *specification-based unit test*. In general, a test procedure for such an approach can be divided into:

Test Case Generation: for each operation of the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test Data Generation: (also: Test Data Selection) for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test Execution: the implementation is run with the selected test input data in order to determine the test output data.

Test Result Verification: the pair of input/output data is checked against the specification of the test case.

The development of HOL-TestGen has been inspired by [21], which follows the line of specification animation works. In contrast, we see our contribution in the development of techniques mostly on the first and to a minor extent on the second phase. Building on QuickCheck [17], the work presented in [21] performs essentially random test, potentially improved by hand-programmed external test data generators. Nevertheless, this work also inspired the development of a random testing tool for Isabelle [10]. It is well-known that random test can be ineffective in many cases; in particular, if preconditions of a program based on recursive predicates like "input tree must be balanced" or "input must be a typable abstract syntax tree" rule out most of randomly generated data. HOL-TestGen exploit these predicates and other specification data in order to produce adequate data. As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis are valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification, see [13] for details.

2 Preliminary Notes on Isabelle/HOL

2.1 Higher-order logic — HOL

Higher-order logic(HOL) [16, 9] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e. g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML (SML) or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

2.2 Isabelle

Isabelle [24, 2] is a *generic* theorem prover. New object logic's can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic (constructive and classical), Zermelo-Fränkel set theory and HOL, which we choose as the basis for the development of HOL-TestGen.

Isabelle consists of a logical engine encapsulated in an abstract data type *thm* in Standard ML; any *thm* object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (*tactics*) have been developed; namely a simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

We use the possibility to build on top of the logical core engine own programs performing symbolic computations over formulae in a logically safe (conservative) way: this is what HOL-TestGen technically is.

[1] to be more specific: *parametric polymorphism*

3 Installation

3.1 Prerequisites

HOL-TestGen is build on top of Isabelle/HOL, version 2008, thus you need a working installation of *Isabelle 2008*, either based on SML/NJ [7] or Poly/ML [5] to use HOL-TestGen. To install Isabelle, follow the instructions on the Isabelle web-site:

```
http://isabelle.in.tum.de/download.html
```

We strongly recommend also to install the generic proof assistant front-end *Proof General* [6].

3.2 Installing HOL-TestGen

3.2.1 Installation from Source

In the following we assume that you have a running Isabelle 2008 environment including the Proof General based front-end. The installation of HOL-TestGen requires the following steps:

1. Unpack the HOL-TestGen distribution, e. g.:

   ```
   tar zxvf hol-testgen-1.4.0.tar.gz
   ```

   This will create a directory `hol-testgen-1.4.0` containing the HOL-TestGen distribution.

2. Check the settings in the configuration file `hol-testgen-1.4.0/make.config`. If you can use the `isatool` tool from Isabelle on the command line, the default settings should work.

3. Change into the `src` directory

   ```
   cd hol-testgen-1.4.0/src
   ```

   and build the HOL-TestGen heap image for Isabelle by calling

   ```
   isatool make
   ```

3.2.2 Using Debian Packages

Installing HOL-TestGen on an Debian GNU/Linux on a i386 architecture should be straight forward. Just add the IsaMorph apt-repostory to the sources of your package manager, e. g.by adding the following lines

```
# IsaMorph repository
deb-src http://projects.brucker.ch/debian/rep stable main
deb     http://projects.brucker.ch/debian/rep stable main
```

to `/etc/apt/sources.list` file. Please replace `stable` by the distribution you are using (we provide packages for all three flavours, i. e., `stable`, `testing` or `unstable`. After that, update your package list, i. e., by executing

```
aptitude update
```

Now install a complete Isabelle setup by executing

```
aptitude install x-symbol proofgeneral-misc isabelle isabelle-thy-hol
```

and HOL-TestGen by executing

```
aptitude install hol-testgen hol-testgen-doc
```

This should give you a running installation of Isabelle (based on Poly/ML), Proof General and last but no least, HOL-TestGen.

3.3 Starting HOL-TestGen

HOL-TestGen can now be started using the `Isabelle` command:

```
Isabelle -L HOL-TestGen
```

As HOL-TestGen provides new top-level commands, the `-L HOL-TestGen` is *mandatory*. After a few seconds you should see a Emacs window similar to the one shown in Figure 3.1.

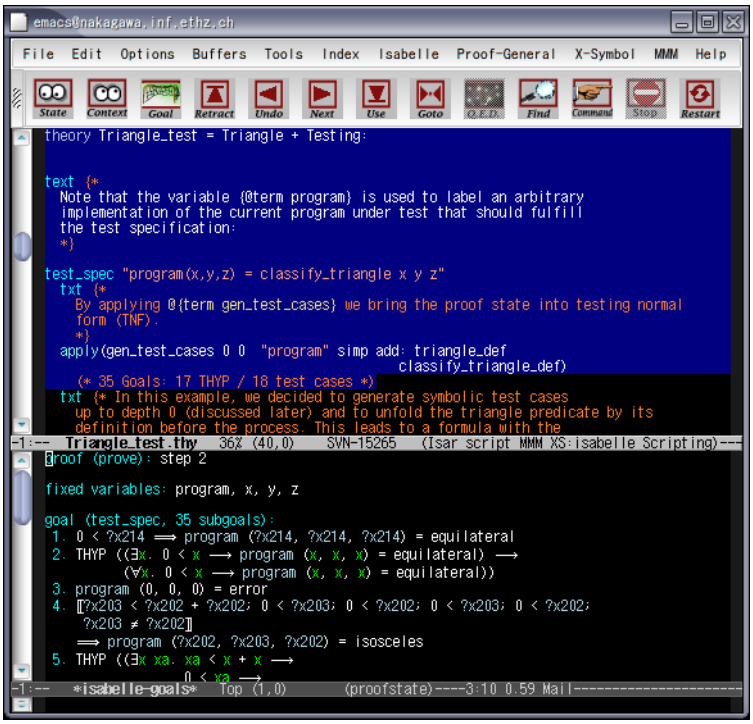


Figure 3.1: A HOL-TestGen session Using the Isar Interface of Isabelle

4 Using HOL-TestGen

4.1 HOL-TestGen: An Overview

HOL-TestGen allows one to automate the interactive development of test cases, refine them to concrete test data, and generate a test script that can be used for test execution and test result verification. The test case generation an test data generation (selection) is done in an Isar-based [27] environment (see Figure 4.1 for details). The Test executable (and the generated test script) can be build with any SML-system.

4.2 Test Case and Test Data Generation

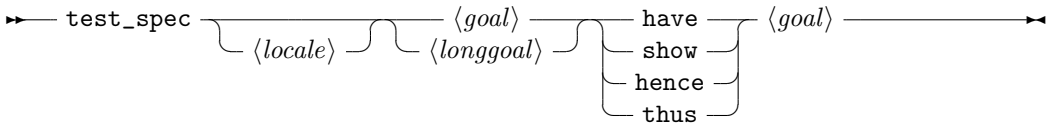
In this section we give a brief overview of HOL-TestGen related extension of the Isar [27] proof language. We also use a presentation similar to the one in the *Isar Reference Manual* [27], e. g. "missing" non-terminals of our syntax diagrams are defined in [27]. We introduce the HOL-TestGen syntax by a (very small) running example: assume we want to test a functions that computes the maximum of two integers.

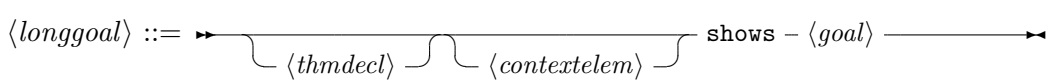
Starting your own theory for testing: For using HOL-TestGen you have to build your Isabelle theories (i. e.test specifications) on top of the theory Testing instead of Main. A sample theory is shown in Table 4.1.

Defining a test specification: Test specifications are defined similar to theorems in Isabelle, e. g.,

```
test_spec "prog a b = max a b"
```

would be the test specification for testing a a simple program computing the maximum value of two integers. The syntax of the keyword `test_spec` : *theory* → *proof*(*prove*) is given by:





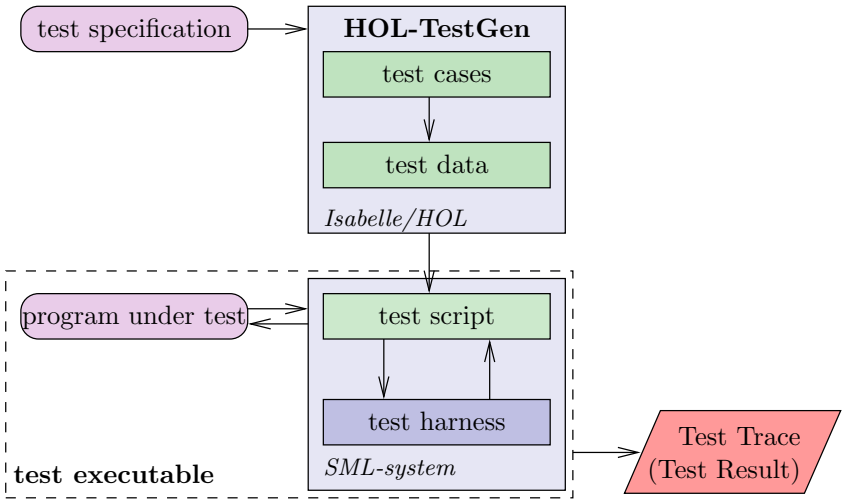


Figure 4.1: Overview of the system architecture of HOL-TestGen

```
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"

  thm max_test.test_data

  gen_test_script "test_max.sml" "max_test" "prog"
                  "myMax.max"
end
```

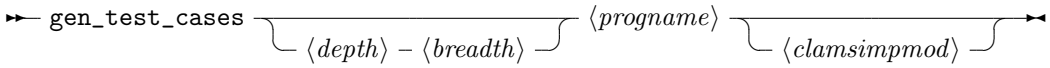
Table 4.1: A simple Testing Theory

Please look into the Isar Reference Manual [27] for the remaining details, e. g.a description of ⟨*contextelem*⟩.

Generating symbolic test cases: Now, abstract test cases for our test specification can (automatically) generated, e. g.by issuing

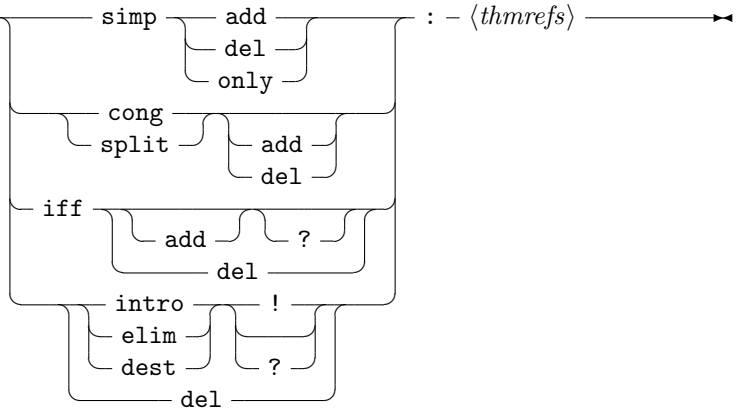
apply(gen_test_cases "prog" simp: max_def)

The `gen_test_cases` : *method* tactic allows a one to control the test case generation in a fine-granular manner:



Where ⟨*depth*⟩ is a natural number describing the depth of the generated test cases and ⟨*breadth*⟩ is a natural number describing their breadth. Roughly speaking, the ⟨*depth*⟩ controls the term size in data separation lemmas in order to establish a regularity hypothesis (see [13] for details), while the ⟨*breadth*⟩ controls the number of variables occurring in the test specification for which regularity hypothesis' were generated. The default for ⟨*depth*⟩ and ⟨*breadth*⟩ is 3 resp. 1. ⟨*progname*⟩ denotes the name of the program under test. Further, one can control the classifier and simplifier sets used internally in the `gen_test_cases` tactic using the optional ⟨*clasimpmod*⟩ option:

⟨*clamsimpmod*⟩ ::=



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

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

store_test_thm "max_test"

for further processing. This is done using the `test_spec` : *proof*(*prove*) → *proof*(*prove*) | *theory* command which also closes the actual "proof state" (or *test state*. Its syntax is given by:

Where ⟨*name*⟩ is a fresh identifier which is later used to refer to this test state. Isabelle/HOL can access the corresponding test theorem using the identifier ⟨*name*⟩.test_thm, e. g.:

```
thm max_test.test_thm
```

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

```
gen_test_data "max_test"
```

The *gen_test_data* : *theory*|*proof* → *theory*|*proof* command takes only one parameter, the name of the test environment for which the test data should be generated:

After the successful execution of this command Isabelle can access the test hypothesis using the identifier ⟨*name*⟩.test_hyps and the test data using the identifier ⟨*name*⟩.test_data

```
thm max_test.test_hyps
thm max_test.test_data
```

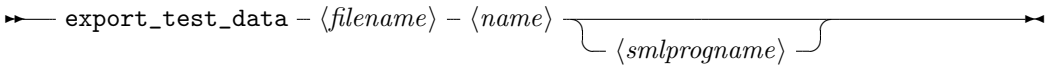
It is important to understand that generating test data is (partly) done by calling the *random solver* which is incomplete. If the random solver is not able to find a solution, it instantiate the term with the constant `RSF` (random solve failure).

Note, that one has a broad variety of configurations options using the **testgen_params** command.

Exporting test data:: After the test data generation, HOL-TestGen is able to export the test data into an external file, e. g.:

```
export_test_data "test_max.dat" "max_test"
```

exports the generated test data into a file `text_max.dat`. The generation of a test data file is done using the *export_test_data* : *theory*|*proof* → *theory*|*proof* command:



Where ⟨*filename*⟩ is the name of the file in which the test data is stored and ⟨*name*⟩ is the name of a collection of test data in the test environment.

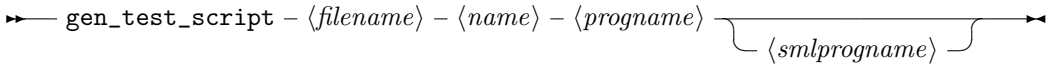
Generating test scripts: After the test data generation, HOL-TestGen is able to generate a test script, e. g.:

```
gen_test_script "test_max.sml" "max_test" "prog"
                "myMax.max"
```

```
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];

  val _ = TestHarness.printList toString testres;
end
```

Table 4.2: Test Script

produces the test script shown in Table 4.2 that can (together with the provided test harness) be used to test real implementations. The generation of test scripts is done using the *generate_test_script* : *theory*|*proof* → *theory*|*proof* command:

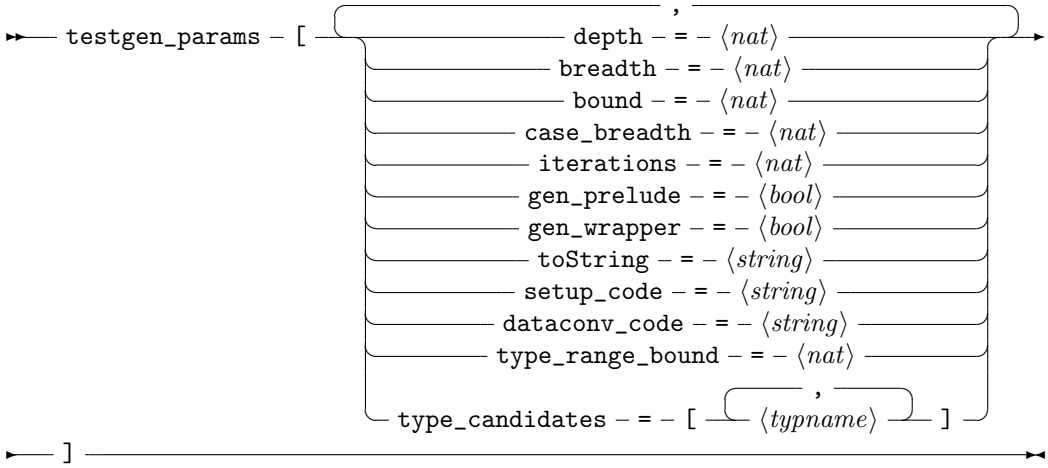


Where ⟨*filename*⟩ is the name of the file in which the test script is stored, and ⟨*name*⟩ is the name of a collection of test data in the test environment, and ⟨*progname*⟩ the name of the program under test. The optional parameter ⟨*smlprogname*⟩ allows for the configuration of different names of the program under test that is used within the test script for calling the implementation.

Configure HOL-TestGen: The overall behavior of test data and test script generation can be configured, e. g.

testgen_params [iterations=15]

using the *testgen_params* : *theory* → *theory* command:



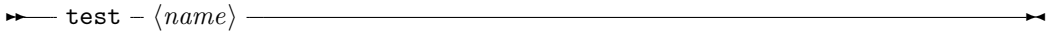
Configuring the test data generation: Further, a attribute *test* : *attribute* is provided, i. e.:

lemma max_abscase [test "maxtest"]:"max 4 7 = 7"

or

declare max_abscase [test "maxtest"]

that can be used for hierarchical test case generation:



4.3 Test Execution and Result Verification

In principle, any SML-system, e. g. [7, 5, 8, 3, 4], should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test

- implementations using the .Net platform (more specific: CLR IL), e. g., written in C# using sml.net [8],
- implementations written in C using, e. g.the foreign language interface of sml/NJ [7] or MLton [4],
- implementations written in Java using mlj [3].

Also, depending on the SML-system, the test execution can be done within an interpreter (it is even possible to execute the test script within HOL-TestGen) or using a compiled test executable. In this section, we will demonstrate the test of SML programs (using SML/NJ or MLton) and ANSI C programs.

```
structure myMax = struct
  fun max x y = if (x < y) then y else x
end
```

Table 4.3: Implementation in SML of max

```
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
===============
```

Table 4.4: Test Trace

4.3.1 Testing an SML-Implementation

Assume we have written a max-function in SML (see Table 4.3 stored in the file `max.sml` and we want to test it using the test script generated by HOL-TestGen. Following Figure 4.1 we have to build a test executable based on our implementation, the generic test harness (`harness.sml`) provided by HOL-TestGen, and the generated test script (`test_max.sml`), shown in Table 4.2.

If we want to run our test interactively in the shell provided by sml/NJ, we just have to issue the following commands:

```
use "harness.sml";
use "max.sml";
use "test_max.sml";
```

After the last command, sml/NJ will automatically execute our test, e. g.you will see a output similar to the one shown in Table 4.4.

If we prefer to use the compilation manager of sml/NJ, or compile our test to a single test executable using MLton, we just write a (simple) file for the compilation manager of sml/NJ (which is understood both, by MLton and sml/NJ) with the following content:

```
Group  is
```

```
int max (int x, int y) {
        if (x < y) {
                return y;
        }else{
                return x;
        }
}
```

Table 4.5: Implementation in ANSI C of max

```
harness.sml
max.sml
test_max.sml

#if(defined(SMLNJ_VERSION))
  $/basis.cm
  $smlnj/compiler/compiler.cm
#else
#endif
```

and store it as `test.cm`. We have two options, we can

- use sml/NJ, e. g.we can start the sml/NJ interpreter and just enter

  ```
  CM.make("test.cm")
  ```

  which will build a test setup and run our test.

- use MLton to compile a single test executable by executing

  ```
  mlton test.cm
  ```

  on the system shell. This will result in a test executable called `test` which can be directly executed.

In both cases, we will get a test output (test trace) similar to the one presented in Table 5.1.

4.3.2 Testing Non-SML Implementations

Suppose we have an ANSI C implementation of max (see Table 4.5) that we want to test using the foreign language interface provided by MLton. First we have to provide import the max method written in C using the `_import` keyword of MLton. Further, we provide a "wrapper" function doing the pairing of the curried arguments:

```
structure myMax = struct
  val cmax    = _import "max": int * int -> int ;
  fun max a b = cmax(a,b);
end
```

We store this file as `max.sml` and write a small configuration file for the compilation manager:

```
Group  is
harness.sml
max.sml
test_max.sml
```

We can compile a test executable by the command

```
mlton -default-ann 'allowFFI true' test.cm max.c
```

on the system shell. Again, we end up with an test executable `test` which can be called directly. Running our test executable will result in trace similar to the one presented in Table 5.1.

5 Examples

Before introducing to the HOL-TestGen showcase ranging from simple to more advanced examples, one general remark: The test data generation uses as final procedure to solve the constraints of test cases a *random solver*. This choice has the advantage that the random process is more faster in general while requiring less interaction as, say, an enumeration based solution principle. However this choice has the feature that two different runs of this document will produce outputs that differs in the details o displayed data. Even worse, in very unlikely cases, the random solver does not find a solution that a previous run could easily produce (in such cases, one should upgrade the `iterations`-variable in the test environment.

5.1 Triangle

theory
  Triangle
imports
  Testing
begin

A prominent example for automatic test case generation is the triangle problem [25]: given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe an equilateral, isosceles, scalene triangle, or no triangle at all. First we define an abstract data type describing the possible results in Isabelle/HOL:

datatype *triangle = equilateral | scalene | isosceles | error*

For clarity (and as an example for specification modularization) we define an auxiliary predicate deciding if the three lengths are describing a triangle:

```
constdefs triangle :: "[int,int,int] => bool"
          "triangle x y z ≡ (0<x ∧ 0<y ∧ 0 < z ∧
                            (z < x+y) ∧ (x < y+z) ∧ (y < x+z))"
```

Now we define the behavior of the triangle program:

```
constdefs
classify_triangle :: "[int,int,int] ⇒ triangle"
"classify_triangle x y z ≡ (if triangle x y z
                           then if x=y
                                then if y=z
                                     then equilateral
                                     else  isosceles
```

```
                                    else if y=z
                                         then isosceles
                                         else if x=z then isosceles
                                else scalene else error)"
end

theory
  Triangle_test
imports
  Triangle
  Testing
begin
```

The test theory `Triangle_test` is used to demonstrate the pragmatics of HOL-TestGen in the standard triangle example; The demonstration elaborates three test plans: standard test generation (including test driver generation), abstract test data based test generation, and abstract test data based test generation reusing partially synthesized abstract test data.

5.1.1 The Standard Workflow

We start with stating a test specification for a program under test: it must behave as in the definition of *classify_triangle* specified.

Note that the variable *program* is used to label an arbitrary implementation of the current program under test that should fulfill the test specification:

```
test_spec "program(x,y,z) = classify_triangle x y z"
```

By applying *gen_test_cases* we bring the proof state into testing normal form (TNF).

```
apply(simp add: classify_triangle_def)
apply(gen_test_cases  "program" simp add: triangle_def
                                          classify_triangle_def)
```

In this example, we decided to generate symbolic test cases and to unfold the triangle predicate by its definition before the process. This leads to a formula with, among others, the following clauses:

```
1. 0 < ?X1X347 ⟹ program (?X1X347, ?X1X347, ?X1X347) = equilateral
2. THYP
    ((∃x. 0 < x ⟶ program (x, x, x) = equilateral) ⟶
     (∀x>0. program (x, x, x) = equilateral))
3. ¬ 0 < ?X1X341 ⟹ program (?X1X341, ?X1X341, ?X1X341) = error
4. THYP
    ((∃x. ¬ 0 < x ⟶ program (x, x, x) = error) ⟶
     (∀x. ¬ 0 < x ⟶ program (x, x, x) = error))
5. ⟦?X2X330 < 2 * ?X1X329; 0 < ?X2X330; 0 < ?X1X329; 0 < ?X2X330;
    0 < ?X1X329; ?X2X330 ≠ ?X1X329⟧
   ⟹ program (?X1X329, ?X2X330, ?X1X329) = isosceles
```

Note that the computed TNFis not minimal, e.g., further simplification and rewriting steps are needed to compute the *minimal set of symbolic test cases*. The following post-generation simplification improves the generated result before "frozen" into a *test theorem*:

apply *(simp_all)*

Now, "freezing" a test theorem technically means storing it into a specific data structure provided by HOL-TestGen, namely a *test environment* that captures all data relevant to a test:

store_test_thm *"triangle_test"*

The resulting test theorem is now bound to a particular name in the Isar environment, such that it can inspected by the usual Isar command `thm`.

thm *"triangle_test.test_thm"*

We compute the concrete *test statements* by instantiating variables by constant terms in the symbolic test cases for "*program*" via a random test procedure:

gen_test_data *"triangle_test"*

thm *"triangle_test.test_hyps"*
thm *"triangle_test.test_data"*

Now we use the generated test data statement lists to automatically generate a test driver, which is controlled by the test harness. The first argument is the external SML-file name into which the test driver is generated, the second argument the name of the test data statement set and the third the name of the (external) program under test:

gen_test_script *"triangle_script.sml" "triangle_test" "program"*

which results in

```
program (7, 7, 7) = equilateral
program (0, 0, 0) = error
program (9, 10, 9) = isosceles
program (10, 0, 10) = error
RSF ⟹ program (6, 10, 6) = error
program (2, 9, 2) = error
program (4, 9, 9) = isosceles
program (8, 2, 2) = error
program (10, 0, 0) = error
program (0, 4, 4) = error
program (8, 8, 7) = isosceles
program (3, 3, 9) = error
program (0, 0, 5) = error
program (4, 4, 0) = error
program (6, 5, 10) = scalene
program (10, 7, 1) = error
program (2, 4, 9) = error
program (5, 8, 0) = error
RSF ⟹ program (4, 8, 2) = error
program (1, 10, 8) = error
```

5.1.2 The Modified Workflow: Using Abstract Testdata

There is a viable alternative for the standard development process above: instead of unfolding triangle and trying to generate ground substitutions satisfying the constraints, one may keep triangle in the test theorem, treating it as a building block for new constraints. Such building blocks will also be called *abstract test cases*.

In the following, we will set up a new version of the test specification, called *triangle2*, and prove the relevant abstract test cases individually before test case generation. These proofs are highly automatic, but the choice of the abstract test data in itself is ingenious, of course. Nevertheless, the computation for establishing if a certain triple is encapsulated in these proofs, deliberating the main test case generation of *triangle2* from them. In fact, these contain 5 arithmetic constraints which represent already a sensible load if given to the random solver.

The abstract test data will be assigned to the subsequent test generation for the test generation *triangle2*. Then the test data generation phase is started for *triangle2* implicitly using the abstract test cases. The association established by this assignment is also stored in the test environment.

The point of having abstract test data is that it can be generated "once and for all" and inserted before the test data selection phase producing a "partial" grounding. It will turn out that the main state explosion is shifted from the test case generation to the test data selection phase.

The "ingenious approach"

```
lemma triangle_abscase1 [test "triangle2"]: "triangle 1 1 1"
  by(auto simp: triangle_def)

lemma triangle_abscase2 [test"triangle2"]:"triangle 1 2 2"
  by(auto simp: triangle_def)

lemma triangle_abscase3 [test"triangle2"]:"triangle 2 1 2"
  by(auto simp: triangle_def)

lemma triangle_abscase4 [test"triangle2"]:"triangle 2 2 1"
  by(auto simp: triangle_def)

lemma triangle_abscase5 [test"triangle2"]:"triangle 3 4 5"
  by(auto simp: triangle_def)

lemma triangle_abscase6 [test"triangle2"]:"¬ triangle -1 1 2"
  by(auto simp: triangle_def)

lemma triangle_abscase7 [test"triangle2"]:"¬ triangle 1 -1 2"
  by(auto simp: triangle_def)
```

```
lemma triangle_abscase8 [test"triangle2"]:"¬ triangle 1 2 -1"
  by(auto simp: triangle_def)
```

Test specification is as shown in the standard case, but the underlying simplification does not use the definition of *triangle_def*. Afterwards we inspect the resulting test theorem.

```
test_spec "prog(x,y,z) = classify_triangle x y z"
  apply(gen_test_cases "prog" simp add: classify_triangle_def)
  store_test_thm "triangle2"

thm "triangle2.test_thm"
```

The test data generation is started and implicitly uses the abstract test data assigned to the test theorem *triangle2*. Again, we inspect the results:

```
gen_test_data "triangle2"



thm "triangle2.test_hyps"
thm "triangle2.test_data"
```

Alternative: Synthesizing Abstract Test Data

In fact, part of the ingenious work of generating abstract test data can be synthesized by using the test case generator itself. This scenario of use proceeds as follows:

1. we set up a the decomposition of `triangle` in an equality to itself; this identity is disguised by introducing a variable `prog` which is stated equivalent to `triangle` in an assumption,
2. the introduction of this assumption is delayed; i.e. the test case generation is performed in a state where this assumption is not visible,
3. after executing test case generation, we fold back `prog` against `triangle`.

```
test_spec abs_triangle :
assumes 1: "prog = triangle"
shows      "triangle x y z = prog x y z"
  apply(gen_test_cases "prog" simp add: triangle_def)
  apply(simp_all add: 1)
store_test_thm "abs_triangle"

thm abs_triangle.test_thm
```

which results in

```
⟦⟦?X2X89 < ?X3X90 + ?X1X88; ?X3X90 < ?X2X89 + ?X1X88;
  ?X1X88 < ?X3X90 + ?X2X89; 0 < ?X1X88; 0 < ?X2X89; 0 < ?X3X90⟧
```

```
⟹ triangle ?X3X90 ?X2X89 ?X1X88;
THYP
 ((∃x xa xb.
     xa < xb + x ⟶ xb < xa + x ⟶ x < xb + xa ⟶ triangle xb xa x) ⟶
  (∀x xa xb.
     xa < xb + x ⟶ xb < xa + x ⟶ x < xb + xa ⟶ triangle xb xa x));
¬ 0 < ?X3X75 ⟹ ¬ triangle ?X3X75 ?X2X74 ?X1X73;
THYP
 ((∃x xa xb. ¬ 0 < xb ⟶ ¬ triangle xb xa x) ⟶
  (∀x xa xb. ¬ 0 < xb ⟶ ¬ triangle xb xa x));
¬ 0 < ?X2X64 ⟹ ¬ triangle ?X3X65 ?X2X64 ?X1X63;
THYP
 ((∃x xa. ¬ 0 < xa ⟶ (∃xb. ¬ triangle xb xa x)) ⟶
  (∀x xa. ¬ 0 < xa ⟶ (∀xb. ¬ triangle xb xa x)));
¬ 0 < ?X1X53 ⟹ ¬ triangle ?X3X55 ?X2X54 ?X1X53;
THYP
 ((∃x. ¬ 0 < x ⟶ (∃xa xb. ¬ triangle xb xa x)) ⟶
  (∀x. ¬ 0 < x ⟶ (∀xa xb. ¬ triangle xb xa x)));
¬ ?X1X43 < ?X3X45 + ?X2X44 ⟹ ¬ triangle ?X3X45 ?X2X44 ?X1X43;
THYP
 ((∃x xa xb. ¬ x < xb + xa ⟶ ¬ triangle xb xa x) ⟶
  (∀x xa xb. ¬ x < xb + xa ⟶ ¬ triangle xb xa x));
¬ ?X3X35 < ?X2X34 + ?X1X33 ⟹ ¬ triangle ?X3X35 ?X2X34 ?X1X33;
THYP
 ((∃x xa xb. ¬ xb < xa + x ⟶ ¬ triangle xb xa x) ⟶
  (∀x xa xb. ¬ xb < xa + x ⟶ ¬ triangle xb xa x));
¬ ?X2X24 < ?X3X25 + ?X1X23 ⟹ ¬ triangle ?X3X25 ?X2X24 ?X1X23;
THYP
 ((∃x xa xb. ¬ xa < xb + x ⟶ ¬ triangle xb xa x) ⟶
  (∀x xa xb. ¬ xa < xb + x ⟶ ¬ triangle xb xa x))⟧
⟹ (triangle x y z = prog x y z)
```

Thus, we constructed test cases for being triangle or not in terms of arithmetic constraints. These are amenable to test data generation by increased random solving, which is controlled by the test environment variable `iterations`:

```
testgen_params[iterations=100]
gen_test_data "abs_triangle"
```

resulting in:

```
prog = triangle
triangle 8 8 10
¬ triangle 0 0 5
¬ triangle 10 0 8
¬ triangle 10 2 0
¬ triangle 0 1 10
¬ triangle 8 2 4
¬ triangle 1 6 5
```

Thus, we achieve solved ground instances for abstract test data. Now, we assign

these synthesized test data to the new future test data generation. Additionally to the synthesized abstract test data, we assign the data for isosceles and equilateral triangles; these can not be revealed from our synthesis since it is based on a subset of the constraints available in the global test case generation.

```
declare abs_triangle.test_data[test"triangle3"]
declare triangle_abscase1[test"triangle3"]
declare triangle_abscase2[test"triangle3"]
declare triangle_abscase3[test"triangle3"]
```

The setup of the testspec is identical as for triangle2; it is essentially a renaming.

```
test_spec "program(x,y,z) = classify_triangle x y z"
  apply(simp add: classify_triangle_def)
  apply(gen_test_cases  "program" simp add: classify_triangle_def)
  store_test_thm "triangle3"
```

The test data generation is started again on the basis on synthesized and selected hand-proven abstract data.

```
testgen_params[iterations=3]
gen_test_data "triangle3"

thm "triangle3.test_hyps"
thm "triangle3.test_data"

end
```

5.2 Lists

```
theory
  List_test
imports
  List
  Testing
begin
```

In this example we present the current main application of HOL-TestGen: generating test data for black box testing of functional programs within a specification based unit test. We use a simple scenario, developing the test theory for testing sorting algorithms over lists.

5.2.1 A Quick Walk Through

In the following we give a first impression how the testing process using HOL-TestGen looks like. For brevity we stick to default parameters an explain possible decision points and parameters where the testing can be improved in the next section.

Writing the Test Specification We start by specifying a primitive recursive predicate describing sorted lists:

```
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)"
```

We will use this HOL predicate for describing our test specification, i.e., the properties our implementation should fulfill:

```
test_spec "is_sorted(PUT (l::('a list)))"
```

where *PUT* is a "placeholder" for our program under test.

Generating test cases Now we can automatically generate *test cases* Using the default setup, we just apply our *gen_test_cases*:

```
test_spec "is_sorted(PUT (l::('a list)))"
  apply(gen_test_cases "PUT")
```

which leads to the test partitioning one would expect:

```
1. is_sorted (PUT [])
2. is_sorted (PUT [?X1X26])
3. THYP ((∃x. is_sorted (PUT [x])) ⟶ (∀x. is_sorted (PUT [x])))
4. is_sorted (PUT [?X2X22, ?X1X21])
5. THYP
    ((∃x xa. is_sorted (PUT [xa, x])) ⟶ (∀x xa. is_sorted (PUT [xa, x])))
6. is_sorted (PUT [?X3X16, ?X2X15, ?X1X14])
7. THYP
    ((∃x xa xb. is_sorted (PUT [xb, xa, x])) ⟶
     (∀x xa xb. is_sorted (PUT [xb, xa, x])))
8. THYP (3 < length l ⟶ is_sorted (PUT l))
```

Now we bind the test theorem to a particular named *test environment*.

```
store_test_thm "test_sorting"
```

Generating test data Now we want to generate concrete test data, i.e., all variables in the test cases must be instantiated with concrete values. This involves a random solver which tries to solve the constraints by randomly choosing values.

```
gen_test_data "test_sorting"
```

Which leads to the following test data:

```
is_sorted (PUT [])
is_sorted (PUT [10])
is_sorted (PUT [6, 6])
is_sorted (PUT [4, 9, 0])
```

Note that by the following statements, the test data, the test hypothesis's' and the test theorem can be inspected interactively.

thm *test_sorting.test_data*
thm *test_sorting.test_hyps*
thm *test_sorting.test_thm*

The generated test data can be exported to an external file:

export_test_data *"list_data.dat" test_sorting*

Test Execution and Result Verification In principle, any SML-system should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test implementations written

- for the.Net platform, e.g., written in C# using sml.net [8],
- in C using, e.g. the foreign language interface of sml/NJ [7] or MLton [4],
- in Java using MLj [3].

Depending on the SML-system, the test execution can be done within an interpreter or using a compiled test executable. Testing implementations written in SML is straightforward, based on automatically generated test scripts. This generation is based on the internal code generator of Isabelle and must be set up accordingly.

consts_code *"op <" ("(_ </ _)")*

The key command of the generation is:

gen_test_script *"list_script.sml" test_sorting PUT "myList.sort"*

which generates the following test harness:

```
(*****************************************************************
 *                          Test-Driver
 *            generated by HOL-TestGen 1.4.0 (build: 8174)
 *****************************************************************)

structure TestDriver : sig end = struct



fun is_sorted [] = true
  | is_sorted (x :: xs) =
    ((case xs of [] => true | (xa :: xb) => ((x < xa) orelse (x = xa))) andalso
      is_sorted xs);



val return = ref ( []:((int list)));
fun eval  x1  =  let val ret = myList.sort  x1  in ((return := ret);ret)end
fun retval () = SOME(!return);
```

```
fun toString  a = (fn l => ("["^(foldr (fn (s1,s2)
                  => (if s2 = "" then s1 else s1^",␣"^s2))
                     "" (map (fn x => Int.toString x) l))^"]"))  a;

val testres = [];

val _ = print ("\nRunning␣Test␣Case␣3:\n")
val pre_3  = [];
val post_3 = fn () => ( is_sorted (eval [4, 9, 0]));
val res_3  = TestHarness.check retval pre_3 post_3;
val testres = testres@[res_3];

val _ = print ("\nRunning␣Test␣Case␣2:\n")
val pre_2  = [];
val post_2 = fn () => ( is_sorted (eval [6, 6]));
val res_2  = TestHarness.check retval pre_2 post_2;
val testres = testres@[res_2];

val _ = print ("\nRunning␣Test␣Case␣1:\n")
val pre_1  = [];
val post_1 = fn () => ( is_sorted (eval [10]));
val res_1  = TestHarness.check retval pre_1 post_1;
val testres = testres@[res_1];

val _ = print ("\nRunning␣Test␣Case␣0:\n")
val pre_0  = [];
val post_0 = fn () => ( is_sorted (eval []));
val res_0  = TestHarness.check retval pre_0 post_0;
val testres = testres@[res_0];

val _ = TestHarness.printList toString testres;

end
```

Further, suppose we have an ANSI C implementation of our sorting method for sorting C arrays that we want to test. Using the foreign language interface provided by the SML compiler MLton we first we have to import the sort method written in C using the `_import` keyword of MLton and further, we provide a "wrapper" doing some datatype conversion, e.g. converting lists to arrays and vice versa:

```
structure myList = struct
  val csort = _import "sort": int array * int -> int array;
  fun toList a = Array.foldl (op ::) [] a;
  fun sort l = toList(csort(Array.fromList(list),length l));
end
```

That's all, now we can build the test executable using MLton and end up with a test executable which can be called directly. Running our test executable will result in the test trace in Tab. 5.1 on the facing page. Even this small set of test vectors is sufficient to exploit an error in your implementation.

```
Test Results:
=============
Test 0 -     SUCCESS, result: []
Test 1 -     SUCCESS, result: [10]
Test 2 -     SUCCESS, result: [72, 42]
Test 3 - *** FAILURE: post-condition false, result: [8, 15, -31]

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
===============
```

Table 5.1: A Sample Test Trace

Improving the Testing Results

Obviously, in reality one would not be satisfied with the test cases generated in the section: for testing sorting algorithms one would expect that the test data somehow represents the set of permutations of the list elements. We have already seen, that the test specification used in the last section "only" enumerates lists up to a specific length without any ordering constraints on their elements. Thus we decide to try a more ''descriptive" test specification that is based on the behavior of an insertion sort algorithm:

```
consts   ins :: "('a::ord) ⇒ 'a list ⇒ 'a list"
primrec "ins x [] = [x]"
        "ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys)))"
consts   sort:: "('a::ord) list ⇒ 'a list"
primrec "sort [] = [] "
        "sort (x#xs) = ins x (sort xs)"
```

Now we state our test specification by requiring that the behavior of the program under test `PUT` is identical to the behavior of our specified sorting algorithm `sort`:

Based on this specification `gen_test_cases` produces test cases representing all permutations of lists up to a fixed length n. Normally, we also want to configure up to which length lists should be generated (we call this the *depth* of test case), e.g. if we decide to generated lists up to length 4. Our standard setup

```
test_spec "sort l = PUT  l"
  apply(gen_test_cases 4 1 "PUT")
  apply(simp_all)
```

which leads to the following test partitioning (excerpt):

```
1. [] = PUT []
2. [?X1X1802] = PUT [?X1X1802]
3. THYP ((∃x. [x] = PUT [x]) ⟶ (∀x. [x] = PUT [x]))
4. ?X2X1797 < ?X1X1796 ⟹ [?X2X1797, ?X1X1796] = PUT [?X2X1797, ?X1X1796]
5. THYP
    ((∃x xa. xa < x ⟶ [xa, x] = PUT [xa, x]) ⟶
     (∀x xa. xa < x ⟶ [xa, x] = PUT [xa, x]))
6. ¬ ?X2X1789 < ?X1X1788 ⟹ [?X1X1788, ?X2X1789] = PUT [?X2X1789, ?X1X1788]
7. THYP
    ((∃x xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x]) ⟶
     (∀x xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x]))
8. ⟦?X2X1778 < ?X1X1777; ?X3X1779 < ?X1X1777; ?X3X1779 < ?X2X1778⟧
   ⟹ [?X3X1779, ?X2X1778, ?X1X1777] = PUT [?X3X1779, ?X2X1778, ?X1X1777]
9. THYP
    ((∃x xa.
        xa < x ⟶
        (∃xb. xb < x ⟶ xb < xa ⟶ [xb, xa, x] = PUT [xb, xa, x])) ⟶
     (∀x xa. xa < x ⟶ (∀xb<x. xb < xa ⟶ [xb, xa, x] = PUT [xb, xa, x])))
10. ⟦¬ ?X2X1764 < ?X1X1763; ?X3X1765 < ?X1X1763; ?X3X1765 < ?X2X1764⟧
    ⟹ [?X3X1765, ?X1X1763, ?X2X1764] = PUT [?X3X1765, ?X2X1764, ?X1X1763]
```

store_test_thm *"test_insertion_sort"*

generates 34 test cases describing all permutations of lists of length $1, 2, 3$ and 4.

Generating concrete test data already takes a remarkable length of time, as it's quite unlikely that the random solver generates values that fulfills these ordering constraints. Therefore we restrict the attempts (*iterations*) the random solver takes for solving a single test case to 10.

testgen_params *[iterations=10]*
gen_test_data *"test_insertion_sort"*

thm *test_insertion_sort.test_data*

which is sadly not sufficient to solve all conditions, e.g. we obtain test cases like:

```
[] = PUT []
[6] = PUT [6]
[0, 3] = PUT [0, 3]
[1, 8] = PUT [8, 1]
[3, 4, 5] = PUT [3, 4, 5]
[3, 7, 8] = PUT [3, 8, 7]
[2, 5, 6] = PUT [5, 6, 2]
[0, 6, 9] = PUT [6, 0, 9]
[5, 8, 9] = PUT [9, 5, 8]
[5, 5, 10] = PUT [10, 5, 5]
RSF ⟹ [10, 10, 5, 7] = PUT [10, 10, 5, 7]
[0, 3, 9, 10] = PUT [0, 3, 10, 9]
RSF ⟹ [5, 7, 9, 0] = PUT [5, 9, 0, 7]
RSF ⟹ [4, 1, 8, 8] = PUT [1, 8, 8, 4]
```

```
RSF ⟹ [9, 6, 1, 6] = PUT [6, 9, 1, 6]
RSF ⟹ [4, 3, 2, 3] = PUT [3, 4, 3, 2]
RSF ⟹ [1, 8, 6, 8] = PUT [6, 1, 8, 8]
RSF ⟹ [3, 0, 5, 1] = PUT [5, 0, 1, 3]
RSF ⟹ [2, 9, 5, 6] = PUT [2, 5, 9, 6]
RSF ⟹ [8, 8, 0, 0] = PUT [8, 0, 8, 0]
RSF ⟹ [0, 1, 5, 9] = PUT [0, 9, 5, 1]
[5, 5, 9, 10] = PUT [5, 10, 9, 5]
RSF ⟹ [6, 7, 6, 0] = PUT [6, 6, 7, 0]
RSF ⟹ [10, 0, 2, 10] = PUT [10, 10, 0, 2]
RSF ⟹ [5, 10, 3, 6] = PUT [6, 5, 3, 10]
RSF ⟹ [10, 8, 9, 6] = PUT [6, 8, 9, 10]
RSF ⟹ [2, 4, 1, 0] = PUT [4, 1, 2, 0]
RSF ⟹ [8, 0, 4, 4] = PUT [0, 4, 8, 4]
[4, 6, 8, 9] = PUT [8, 9, 4, 6]
RSF ⟹ [1, 10, 4, 1] = PUT [4, 1, 10, 1]
[1, 3, 9, 10] = PUT [9, 3, 1, 10]
[2, 2, 6, 9] = PUT [9, 2, 2, 6]
RSF ⟹ [8, 0, 0, 1] = PUT [1, 0, 8, 0]
RSF ⟹ [8, 10, 7, 7] = PUT [7, 7, 10, 8]
```

were *RSF* marks unsolved cases. Analyzing the generated test data reveals, that all cases for lists with length up to (and including) 3 could be solved. From the 24 cases for lists of length 4 only 9 could be solved by the random solver (thus, overall 19 of the 34 cases were solved). To achieve more concrete test cased, we could interactive increase the number of iterations which reveals that we need to set iterations to 100 to find all solutions reliably:r

iterations	5	10	20	25	30	40	50	75	100
solved goals (of 34)	13	19	23	24	25	29	33	33	34

Instead of increasing the number of iterations one could also add other techniques such as

1. deriving new rules that allow for the generation of a simplified test theorem,
2. introducing abstract test cases or
3. supporting the solving process by derived rules.

end

5.3 AVL

theory
  AVL_def
imports

```
  Testing
```
begin

This test theory specifies a quite conceptual algorithm insertion and deletion of AVL Trees. It is essentially a streamlined version of the AFP [1] theory developed by Pusch, Nipkow, Klein and the authors.

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

consts
  height :: "'a tree ⇒ nat"
  is_in  :: "'a ⇒ 'a tree ⇒ bool"
  is_ord :: "('a::order) tree ⇒ bool"
  is_bal :: "'a tree ⇒ bool"

primrec
  "height ET = 0"
  "height (MKT n l r) = 1 + max (height l) (height r)"

primrec
  "is_in k ET = False"
  "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"

primrec
 isord_base: "is_ord ET = True"
 isord_rec:  "is_ord (MKT n l r) = ((∀n'. is_in n' l ⟶ n' < n) ∧
                                    (∀n'. is_in n' r ⟶ n < n') ∧
                                    is_ord l ∧ is_ord r)"

primrec
  "is_bal ET = True"
  "is_bal (MKT n l r) = ((height l = height r ∨
                          height l = 1+height r ∨
                          height r = 1+height l) ∧
                          is_bal l ∧ is_bal r)"
```

We also provide a more efficient variant of `is_in`:

```
consts
  is_in_eff   :: "('a::order) ⇒ 'a tree ⇒ bool"
primrec
 "is_in_eff k ET = False"
 "is_in_eff k (MKT n l r) = (if k = n then True
                             else (if k<n then (is_in_eff k l)
                                   else (is_in_eff k r)))"

datatype bal = Just | Left | Right

constdefs
  bal :: "'a tree ⇒ bal"
  "bal t ≡ case t of ET ⇒ Just
```

```
                | (MKT n l r) ⇒ if height l = height r then Just
                                else if height l < height r then Right
                                     else Left"

consts
  r_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  l_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  lr_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  rl_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"

recdef r_rot "{}"
  "r_rot (n, MKT ln ll lr, r) = MKT ln ll (MKT n lr r)"

recdef l_rot "{}"
  "l_rot(n, l, MKT rn rl rr) = MKT rn (MKT n l rl) rr"

recdef lr_rot "{}"
  "lr_rot(n, MKT ln ll (MKT lrn lrl lrr), r) =
            MKT lrn (MKT ln ll lrl) (MKT n lrr r)"

recdef rl_rot "{}"
  "rl_rot(n, l, MKT rn (MKT rln rll rlr) rr) =
               MKT rln (MKT n l rll) (MKT rn rlr rr)"


constdefs
  l_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "l_bal n l r ≡ if bal l = Right
                 then lr_rot (n, l, r)
                 else r_rot (n, l, r)"

  r_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "r_bal n l r ≡ if bal r = Left
                 then rl_rot (n, l, r)
                 else l_rot (n, l, r)"

consts
  insert :: "'a::order ⇒ 'a tree ⇒ 'a tree"

primrec
insert_base:  "insert x ET = MKT x ET ET"
insert_rec:   "insert x (MKT n l r) =
                   (if x=n
                    then MKT n l r
                    else if x<n
                         then let l' = insert x l
                              in if height l' = 2+height r
                                 then l_bal n l' r
                                 else MKT n l' r
```

```
                        else let r' = insert x r
                             in if height r' = 2+height l
                                then r_bal n l r'
                                else MKT n l r')"
```

delete

```
consts
tmax :: "'a tree ⇒ 'a"
delete :: "'a::order × ('a tree) ⇒ ('a tree)"

end

theory
  AVL_test
imports
  AVL_def
begin
```

This test plan of this theory follows more or less the standard. However, we insert some minor theorems into the test theorem generation in order to ease the task of solving; this both improves speed of the generation and quality of the test.

```
declare insert_base insert_rec [simp del]

lemma size_0[simp]: "(size x = 0) = (x = ET)"
  by(induct "x",auto)

lemma height_0[simp]: "(height x = 0) = (x = ET)"
  by(induct "x",auto)

lemma [simp]: "(max (Suc a) b) ~= 0"
  by(auto simp: max_def)

lemma [simp]: "(max b (Suc a) ) ~= 0"
  by(auto simp: max_def)
```

We adjust the random generator at a fairly restricted level and go for a solving phase.

```
testgen_params [iterations=10]

test_spec "(is_bal t) --> (is_bal (insert  x t))"
  apply(gen_test_cases "insert")
  store_test_thm "foo"
  gen_test_data "foo"


thm foo.test_data

end
```

5.4 RBT

This example is used to generate test data in order to test the sml/NJ library, in particular the implementation underlying standard data-structures like set and map. The test scenario reveals an error in the library (so in software that is really used, see [13] for more details). The used specification of the invariants was developed by Angelika Kimmig.

theory
  RBT_def
imports
  Testing
begin

The implementation of Red-Black trees is mainly based on the following datatype declaration:

```
datatype ml_order = LESS | EQUAL | GREATER

axclass ord_key < type

consts
  compare :: "'a::ord_key ⇒ 'a ⇒ ml_order "

axclass LINORDER < linorder, ord_key
  LINORDER_less    : "((compare x y) = LESS)    = (x < y)"
  LINORDER_equal   : "((compare x y) = EQUAL)   = (x = y)"
  LINORDER_greater : "((compare x y) = GREATER) = (y < x)"



types    'a item = "'a::ord_key"

datatype  color = R | B

datatype 'a tree = E | T color "'a tree" "'a item" "'a tree"
```

In this example we have chosen not only to check if keys are stored or deleted correctly in the trees but also to check if the trees fulfill the balancing invariants. We formalize the red and black invariant by recursive predicates:

```
consts
  isin         :: "'a::LINORDER item ⇒ 'a tree ⇒ bool"
  isord        :: "('a::LINORDER item) tree ⇒ bool"
  redinv       :: "'a tree ⇒ bool"
  blackinv     :: "'a tree ⇒ bool"
  strong_redinv:: "'a tree ⇒ bool"
  max_B_height :: "'a tree ⇒ nat"


primrec
```

```
  isin_empty : "isin x E = False"
  isin_branch: "isin x (T c a y b) = (((compare x y) = EQUAL)
                                     | (isin x a) | (isin x b))"


primrec
  isord_empty : "isord E = True"
  isord_branch: "isord (T c a y b)
                 = (isord a ∧ isord b
                 ∧ (∀ x. isin x a ⟶ ((compare x y) = LESS))
                 ∧ (∀ x. isin x b ⟶ ((compare x y) = GREATER)))"


recdef redinv "measure (%t. (size t))"
  redinv_1:   "redinv E = True"
  redinv_2:   "redinv (T B a y b) = (redinv a ∧ redinv b)"
  redinv_3:   "redinv (T R (T R a x b) y c) = False"
  redinv_4:   "redinv (T R a x (T R b y c)) = False"
  redinv_5:   "redinv (T R a x b) = (redinv a ∧ redinv b)"


recdef strong_redinv "{}"
  Rinv_1:  "strong_redinv E = True"
  Rinv_2:  "strong_redinv (T R a y b) = False"
  Rinv_3:  "strong_redinv (T B a y b) = (redinv a ∧ redinv b)"


recdef max_B_height "measure (%t. (size t))"
  maxB_height_1:  "max_B_height E = 0"
  maxB_height_3:  "max_B_height (T B a y b)
                   = Suc(max (max_B_height a) (max_B_height b))"
  maxB_height_2:  "max_B_height (T R a y b)
                   = (max (max_B_height a) (max_B_height b))"


recdef blackinv "measure (%t. (size t))"
  blackinv_1:  "blackinv E = True"
  blackinv_2:  "blackinv (T color a y b)
                = ((blackinv a) ∧ (blackinv b)
                ∧ ((max_B_height a) = (max_B_height b)))"
end

theory
  RBT_test
imports
  RBT_def
  Testing
begin
```

The test plan is fairly standard and very similar to the AVL example: test spec,

test generation on the basis of some lemmas that allow for exploiting contradictions in constraints, data-generation and test script generation.

Note that without the interactive proof part, the random solving phase is too blind to achieve a test script of suitable quality. Improving it will definitively improve also the quality of the test. In this example, however, we deliberately stopped at the point where the quality was sufficient to produce relevant errors of the program under test.

First, we define certain functions (inspired from the real implementation) that specialize the program to a sufficient degree: instead of generic trees over class *LINORDER*, we will generate test cases over integers.

5.4.1 Test Specification and Test-Case-Generation

```
instance int::ord_key
  by(intro_classes)

instance int::linorder
  by intro_classes

defs compare_def: "compare (x::int) y
                  == (if (x < y) then LESS
                                 else (if (y < x)
                                       then GREATER
                                       else EQUAL))"

instance int::LINORDER
apply intro_classes
apply (simp_all add: compare_def)
done


lemma compare1[simp]:"(compare (x::int) y = EQUAL) = (x=y)"
by(auto simp:compare_def)

lemma compare2[simp]:"(compare (x::int) y = LESS) = (x<y)"
by(auto simp:compare_def)

lemma compare3[simp]:"(compare (x::int) y = GREATER) = (y<x)"
by(auto simp:compare_def)
```

Now we come to the core part of the test generation: specifying the test specification. We will test an arbitrary program (insertion `add`, deletion `delete`) for test data that fulfills the following conditions:

- the trees must respect the invariants, i.e. in particular the red and the black invariant,
- the trees must even respect the strong red invariant - i.e. the top node must be black,

- the program under test gets an additional parameter `y` that is contained in the tree (useful for delete),
- the tree must be ordered (otherwise the implementations will fail).

The analysis of previous test case generation attempts showed, that the following lemmas (altogether trivial to prove) help to rule out many constraints that are unsolvable - this knowledge is both useful for increasing the coverage (not so much failures will occur) as well for efficiency reasons: attempting to random solve unsolvable constraints takes time. Recall that that the number of random solve attempts is controlled by the `iterations` variable in the test environment of this test specification.

```
lemma max_0_0 : "((max (a::nat) b) = 0) = (a = 0 ∧ (b = 0))"
  by(auto simp: max_def)

lemma [simp]: "(max (Suc a) b) ~= 0"
  by(auto simp: max_def)

lemma [simp]: "(max b (Suc a) ) ~= 0"
  by(auto simp: max_def)

lemma size_0[simp]: "(size x = 0) = (x = E)"
  by(induct "x",auto)


test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
           ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 5 1 "prog" simp: compare1 compare2 compare3)
store_test_thm "red-and-black-inv"
```

5.4.2 Test Data Generation

Brute Force

This fairly simple setup generates already 25 subgoals containing 12 test cases, altogether with non-trivial constraints. For achieving our test case, we opt for a "brute force" attempt here:

```
testgen_params [iterations=40]

gen_test_data "red-and-black-inv"

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

Using Abstract Test Cases

```
test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
           ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 3 1 "prog" simp: compare1 compare2 compare3)
store_test_thm "red-and-black-inv2"
```

By inspection of the onstraints of the test theorem, on immediately identifies predicates for which solutions are difficult to find by a random process (a mesure for this difficulty could be the percentage of trees up to depth k, that make this predicate valid. One can easily convince oneself, that this percentage is decreasing).

Repeatedly, ground instances were needed for:

1. `max_B_height ?X = 0`
2. `max_B_height ?Y = max_B_height ?Z`
3. `blackinv ?X`
4. `redinv ?X`

The point is, that enumerating some examples of ground instances for these predicates is fairly easy if one bears its informal definition in mind. For `max_B_height ?X` this is: "maximal number of black nodes on any path from root to leaf". So let's enumerate some trees who contain no black nodes:

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

lemma *maxB_0_2: "max_B_height (T R E (5::int) E) = 0"*
by *auto*

lemma *maxB_0_3: "max_B_height (T R (T R E 2 E) (5::int) E) = 0"*
by *auto*

lemma *maxB_0_4: "max_B_height (T R E (5::int) (T R E 7 E)) = 0"*
by *auto*

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

Note that these ground instances have already been produced with hindsight to the ordering constraints - ground instances must fulfil all the other constraints, otherwise they wouldn't help the solver at all. On the other hand, continuing with this enumeration doesn't help too much since we start to enumerate trees that do not fulfil the red invariant.

A good overview over what is needed gives the set of rules generated from the redinv-definition. We bring this into the form needed for a lemma

thm *redinv.simps*

An Alternative Approach with a little Theorem Proving

which will suffice to generate the critical test data revealing the error in the sml/NJ library.

Alternatively, one might:

1. use abstract test cases for the auxiliary predicates *redinv* and *blackinv*,

2. increase the depth of the test case generation and introduce auxiliary lemmas, that allow for the elimination of unsatisfiable constraints,

3. or applying more brute force.

Of course, one might also apply a combination of these techniques in order to get a more systematic test than the one presented here.

We will describe option 2) briefly in more detail: part of the following lemmas require induction and real theorem proving, but help to refine constraints systematically an to increase

```
lemma aux :"x = x ⟹ x = x"
  by(auto)

lemma height_0:
   "(max_B_height x = 0) =
    (x = E ∨ (∃ a y b. x = T R a y b ∧
                       (max (max_B_height a) (max_B_height b)) = 0))"
  by(induct "x", simp_all,case_tac "color",auto)


lemma max_B_height_dec :
"((max_B_height (T x t1 val t3)) = 0) ⟹ (x = R) "
  by(case_tac "x",auto)
```

This paves the way for the following testing scenario:

```
test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
           ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 3 1 "prog" simp: compare1 compare2 compare3
                                      max_B_height_dec)

apply(simp_all only: height_0, simp_all add: max_0_0)
apply(simp_all only: height_0, simp_all add: max_0_0)
apply(safe)
```

unfortunately, at this point a general hyp_subst_tac would be needed that allows for instantiating meta variables. TestGen provides a local tactic for this (should be integrated as a general Isabelle tactic ...)

```
apply(tactic "ALLGOALS(fn n => TRY(TestGen.var_hyp_subst_tac n))")
apply(simp_all)
store_test_thm "red-and-black-inv3"

testgen_params [iterations=20]

gen_test_data "red-and-black-inv3"

thm "red-and-black-inv3.test_data"
```

The inspection shows now a stream-lined, quite powerful test data set for our problem. Note that the "depth 3" parameter of the test case generation leads to "depth 2" trees, since the constructor `E` is counted. Nevertheless, this test case produces the error regularly (Warning: recall that randomization is involved; in general, this makes the search faster (while requiring less control by the user) than brute force enumeration, but has the prize that in rare cases the random solver does not find the solution at all):

```
blackinv (prog (4, T B E 4 E))
blackinv (prog (3, T B E 3 (T R E 10 E)))
blackinv (prog (3, T B E 1 (T R E 3 E)))
blackinv (prog (7, T B (T R E 4 E) 7 E))
blackinv (prog (6, T B (T R E 6 E) 8 E))
blackinv (prog (3, T B (T R E 1 E) 3 (T R E 7 E)))
blackinv (prog (4, T B (T B E 4 E) 7 (T B E 9 E)))
blackinv (prog (9, T B (T B E 1 E) 3 (T B E 9 E)))
```

When increasing the depth to 5, the test case generation is still feasible - we had runs which took less than two minutes and resulted in 348 test cases.

5.4.3 Configuring the Code Generator

We have to perform the usual setup of the internal Isabelle code generator, which involves providing suitable ground instances of generic functions (in current Isabelle) and the map of the the data structures to the data structures in the environment.

Note that in this setup the mapping to the target program under test is done in the wrapper script, that also maps our abstract trees to more concrete data structures as used in the implementation.

```
testgen_params [setup_code="open IntRedBlackSet;",
                toString="wrapper.toString"]

lemma [code]: "(max (a::nat) b) = (if (a < b) then b else a)"
  by(simp add: max_def)

types_code
  color    ("color")
  ml_order ("order")
  tree     ("_ tree")

consts_code
  "compare" ("Key.compare (_,_)")
  "color.B" ("B")
  "color.R" ("R")
  "tree.E"  ("E")
  "tree.T"  ("(T(_,_,_,_))")
```

Now we can generate a test script (for both test data sets):

```
gen_test_script "rbt_script.sml" "red-and-black-inv" "prog"
```

```
                "wrapper.del"


gen_test_script "rbt2_script.sml" "red-and-black-inv3" "prog"
                "wrapper.del"
```

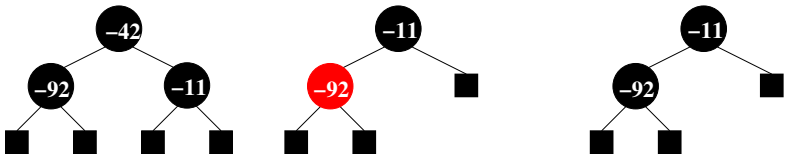
5.4.4 Test Result Verification

Running the test executable (either based on *red-and-black-inv* or on *red-and-black-inv3*) results in an output similar to

```
Test Results:
=============
Test  0 -     SUCCESS, result: E
Test  1 -     SUCCESS, result: T(R,E,67,E)
Test  2 -     SUCCESS, result: T(B,E,~88,E)
Test  3 - **  WARNING: pre-condition false (exception
                                           during post_condition)
Test  4 - **  WARNING: pre-condition false (exception
                                           during post_condition)
Test  5 -     SUCCESS, result: T(R,E,30,E)
Test  6 -     SUCCESS, result: T(B,E,73,E)
Test  7 - **  WARNING: pre-condition false (exception
                                           during post_condition)
Test  8 - **  WARNING: pre-condition false (exception
                                           during post_condition)
Test  9 - *** FAILURE: post-condition false, result:
                       T(B,T(B,E,~92,E),~11,E)
Test 10 -     SUCCESS, result: T(B,E,19,T(R,E,98,E))
Test 11 -     SUCCESS, result: T(B,T(R,E,8,E),16,E)


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
===============
```

The error that is typically found has the: Assuming the red-black tree presented in Fig. 5.1(a), deleting the node with value -49 results in the tree presented in Fig. 5.1(c) which obviously violates the black invariant (the expected result is the balanced tree in Fig. 5.1(b)). Increasing the depth to at least 4 reveals several test cases where unbalanced trees are returned from the SML implementation.cat

end



(a) pre-state (b) correct result (c) result of sml/NJ

Figure 5.1: Test Data for Deleting a Node in a Red-Black Tree

5.5 IMP

theory
hoare_test_theory
imports
VC
Testing
begin

5.5.1 Alternative Formulations of Hoare-Rules

lemma *semi_rev:* "⟦ *|- {Q} d {R}; |- {P} c {Q}* ⟧ ⟹ *|- {P} c; d {R}*"
by *(rule semi, assumption+)*

lemma *conseq_rev:*
"⟦*|- {P} c {Q}; ∀s. P' s ⟶ P s; ∀s. Q s ⟶ Q' s*⟧ ⟹ *|- {P'} c {Q'}*"
by *(rule conseq, auto)*

lemma *conseq_skip:*
" (∀ *s. P s ⟶ Q s)* ⟹ *|- {P} SKIP {Q}*"
by *(rule conseq_rev, rule skip, auto)*

lemma *conseq_ass:*
" (∀ *s. P s ⟶ Q (s[x ↦ E s]))* ⟹ *|- {P} x :== E {Q}*"
by *(rule conseq_rev, rule ass, auto)*

thm *While*

lemma *conseq_while:*
"⟦ *|- {λ s. I s ∧ cond s} s {I};*
∀s. P s ⟶ I s;

```
        ∀s. I s ∧ ¬ cond s ⟶ Q s ⟧
    ⟹  |- {P} WHILE cond DO s {Q}"
  by (rule conseq_rev, rule While, assumption+)
```

5.5.2 Hoare-Rules Using Semantic Equivalences

Soundness and Completeness justify the following equivalence, which allows to conclude the existence of Hoare-proofs from denotational-semantic equivalences.

```
lemma hoare_deriv_eq_valid: "( |- {P} c {Q}) = ( |= {P} c {Q})"
  by(auto intro!: hoare_sound hoare_relative_complete)


lemma hoare_false:
      " |- {λ x. False} c {Q}"
  by(simp add: hoare_deriv_eq_valid  hoare_valid_def)


lemma while_unfold:
      "( |- {P} WHILE b DO c {Q}) =
       ( |- {P}IF b THEN c; WHILE b DO c ELSE SKIP {Q})"
  by(simp only: hoare_deriv_eq_valid  hoare_valid_def
                Denotation.C_While_If[symmetric])


lemma semi_skip1:
      "( |- {P} SKIP ;c {Q}) = ( |- {P} c {Q})"
  by(simp only: hoare_deriv_eq_valid  hoare_valid_def
                Denotation.C.simps Id_O_R R_O_Id)


lemma semi_skip2:
      "( |- {P} c;SKIP {Q}) = ( |- {P} c {Q})"
  by(simp only: hoare_deriv_eq_valid hoare_valid_def
                Denotation.C.simps Id_O_R R_O_Id)


lemma semi_assoc:
      "( |- {P} (c;d) ;e {Q}) = ( |- {P} c; (d;e) {Q})"
  by(simp only: hoare_deriv_eq_valid hoare_valid_def
                Denotation.C.simps Relation.O_assoc)


lemma semi_assoc2:
      "( |- {P} (c;(d;e)) ; f {Q}) = ( |- {P} c;(d;(e;f)) {Q})"
  by(simp only: hoare_deriv_eq_valid hoare_valid_def
                Denotation.C.simps Relation.O_assoc)
```

5.5.3 Unwind and its Correctness

The core of our white box testing function is the following "unwind" function, that "unfolds" while loops and normalizes the resulting program in order to expose it to the operational semantics (i.e. the "natural semantics" *evalc* up to an unwind factor k. Evaluating programs leads to accumulating path-conditions: If a remaining constraint (whose components essentially result from applications of the *If_split* rule), is satisfiable that a path through a program is traceable and results to a certain successor state.

This can be used to test program specifications: Hoare-Triples were checked against for all paths up to a certain depth.

```
consts "@@" :: "[com,com] ⇒ com" (infixr 70)
primrec
ca_skip : "SKIP @@ c = c"
ca_ass  : "(x:== E) @@ c = ((x:== E); c)"
ca_semi : "(c;d) @@ e = (c; d @@ e)"
ca_If   : "(IF b THEN c ELSE d) @@ e =
                     (IF b THEN c @@ e ELSE d @@ e)"
ca_while: "(WHILE b DO c) @@ e = ((WHILE b DO c);e)"


lemma C_skip_cancel1[simp] : "C(SKIP;c) = C(c)"
  by (simp add: Denotation.C.simps Id_O_R R_O_Id)

lemma C_skip_cancel2[simp] : "C(c;SKIP) = C(c)"
  by (simp add: Denotation.C.simps Id_O_R R_O_Id)

lemma C_If_semi[simp] :
      "C((IF x THEN c ELSE d);e) = C(IF x THEN (c;e) ELSE (d;e))"
  by auto

lemma comappend_correct [simp]: "C(c @@ d) = C(c;d)"
  apply(induct "c")
  apply(simp_all only: C_If_semi ca_If)
  apply(simp_all add: Relation.O_assoc)
  done

consts unwind :: "nat × com ⇒ com"
recdef unwind "less_than <*lex*> measure(λ s. size s)"
uw_skip : "unwind(n, SKIP)     = SKIP"
uw_ass  : "unwind(n, a :== E)  = (a :== E)"
uw_If   : "unwind(n, IF b THEN c ELSE d) =
                 IF b THEN unwind(n, c) ELSE unwind(n, d)"
uw_while: "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))"
uw_semi1: "unwind(n, SKIP ; c) = unwind(n, c)"
```

```
uw_semi2: "unwind(n, c ; SKIP) = unwind(n, c)"
uw_semi3: "unwind(n, (IF b THEN c ELSE d) ; e) =
                (IF b THEN (unwind(n,c;e)) ELSE (unwind(n,d;e)))"
uw_semi4: "unwind(n, (c ; d); e) =  (unwind(n, c;d))@@(unwind(n,e))"
uw_semi5: "unwind(n, c ; d) =  (unwind(n, c))@@(unwind(n, d))"
```

```
lemma unwind_correct_aux1 :
  assumes H : "∀x. C (unwind (x, c))   = C c"
  shows       "C(unwind(n,WHILE b DO c)) = C(WHILE b DO c)"
  apply(induct "n")
  apply(simp add: Denotation.C.simps H)
  apply(subst uw_while)
  apply(subst if_P, simp)
  apply(simp only: Denotation.C.simps  comappend_correct)
  apply(rule_tac s = "n" and t = "Suc n - 1" in subst)
  apply arith
  apply(simp only: Denotation.C.simps [symmetric] H)
  apply(simp only: Denotation.C_While_If)
  done
```

```
declare uw_while [simp del]
```

```
lemma unwind_correct_aux2 :
  "C(unwind(n,c;d))= C(unwind(n,c) ; unwind(n, d))"
  apply (induct "c",   simp_all)
  apply (case_tac "d", simp_all)
  apply (case_tac "d", simp_all)
  apply (case_tac "d", simp_all)
  apply (simp_all only: C.simps[symmetric] C_If_semi)
  apply (case_tac "d", simp_all)
  done
```

```
lemma unwind_correct : "C(unwind(n,c)) = C(c)"
  apply (rule_tac x = n in spec)
  apply (induct "c")
  prefer 5
  apply (rule allI)
  apply (rule unwind_correct_aux1, simp, simp_all)
  apply (case_tac "c1",case_tac "c2",
         simp_all add: unwind_correct_aux2)
  done
```

lemma *unwind_hoare:*
```
      "( |- {P} unwind(n,c) {Q} ) = ( |- {P} c {Q})"
```
by(*simp only: unwind_correct hoare_deriv_eq_valid*
hoare_valid_def)

lemma *wp_unwind : "wp (c) (p) = wp(unwind(n,c)) (p) "*
by(*simp add: wp_def unwind_correct)*

lemma *wp_test : "∀σ. P σ ⟶ wp (unwind(k,c)) Q σ*
⟹ |- {P} c {Q}"
apply *(rule Hoare.hoare_conseq1)*
prefer *2*
apply *(rule wp_is_pre)*
apply *(simp add: wp_unwind[symmetric])*
done

ML{*

```
fun thyp_ify n = EVERY[TestGen.mp_fy n,
                       TestGen.all_ify [] n,
                       rtac ((thm"THYP_def") RS (thm"HOL.meta_eq_to_obj_eq") RS (thm"iffD1"))
n]
*}
```

5.5.4 Relation to Operational Semantics

lemma *If_split:*
```
 "⟦ b s ⟹ ⟨c0,s⟩ ⟶c s';
   ¬ b s ⟹ ⟨c1,s⟩ ⟶c s' ⟧
  ⟹ ⟨IF b THEN c0 ELSE c1,s⟩ ⟶c s'"
```
by *(cases "b s", simp_all)*

lemma *If_splitE:*
```
  "⟦   ⟨IF b THEN c ELSE d,s⟩ ⟶c s';
       ⟦ b s; ⟨c,s⟩ ⟶c s' ⟧ ⟹ P;
       ⟦  ¬ b s; ⟨d,s⟩ ⟶c s' ⟧ ⟹ P ⟧ ⟹ P"
```
by(*cases "b s", simp_all)*

lemma *setup_test :*
```
   "( |- {Pre} c {Post}) =
      (∀s t.  ⟨unwind (n, c),s⟩ ⟶c t ⟶ Pre s ⟶ Post t)"
```
apply(*subst unwind_hoare[symmetric])*
apply(*subst hoare_deriv_eq_valid)*
apply(*simp only: hoare_valid_def denotational_is_natural)*
done

consts *STOP :: com*

uninterpreted constant stopping deduction for cases reached for it ...

constdefs *ASSERT :: "[bexp,com] ⇒ com"*
"ASSERT b c ≡ IF b THEN c ELSE STOP"

constdefs *AWHILE :: "[bexp,bexp,com] ⇒ com"*
"AWHILE b invnt c ≡ ASSERT invnt (WHILE b DO (c; ASSERT invnt SKIP))"

end

theory
squareroot_test
imports
hoare_test_theory
begin

5.5.5 The Definition of the Integer-Squareroot Program

constdefs
```
squareroot :: "[loc,loc,loc,loc] => com"
"squareroot tm sum i a ==
 ((  tm   :== (λs. 1));
  (( sum  :== (λs. 1));
   ((i    :== (λs. 0));
     WHILE (λs. (s sum) <= (s a)) DO
       (( i    :== (λs. (s i) + 1));
        ((tm   :== (λs. (s tm) + 2));
         (sum  :== (λs. (s tm) + (s sum)))))))
 )"
```

constdefs
```
pre   :: assn
"pre  ≡  λ x. True"
post  :: "[loc,loc] ⇒ assn"
"post a i ≡   λ s. (s i)*(s i)≤(s a) ∧ s a < (s i + 1)*(s i + 1)"
inv   :: "[loc,loc,loc,loc] ⇒ assn"
"inv i sum tm a ≡ λs.(s i + 1) * (s i + 1) = s sum
                    ∧ s tm = (2 * (s i) + 1)
                    ∧ (s i) * (s i) <= (s a)"
```

5.5.6 Computing Program Paths and their Path-Constraints

lemma *derive_pathconds:*
assumes *no_alias : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum ∧*
sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm ∧
a ≠ i ∧ i ≠ a"
shows *"⟨unwind(3, squareroot tm sum i a), s⟩ ⟶c s'"*

apply *(simp add: squareroot_def uw_while)*
apply *(rule If_split, simp_all add: update_def no_alias)+*

The resulting proof state capturing the test hypothesis as well as the resulting 4 evaluation paths (no entry into loop, 1 pass, 2 passes and 3 passes through the loop) looks as follows:

```
1. Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ s a ⟹
   ⟨WHILE λs. s sum
              ≤ s a DO i :== λs. Suc (s i) ; (tm :== λs.
             Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ),s
   (i := Suc (Suc (Suc 0)),
    tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
    sum :=
      Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
   ⟶c s'
2. ⟦Suc (Suc (Suc (Suc 0))) ≤ s a;
    ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ s a⟧
   ⟹ s' = s
       (i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
        sum := Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))))
3. ⟦Suc 0 ≤ s a; ¬ Suc (Suc (Suc (Suc 0))) ≤ s a⟧
   ⟹ s' = s
       (i := Suc 0, tm := Suc (Suc (Suc 0)), sum := Suc (Suc (Suc (Suc 0))))
4. ¬ Suc 0 ≤ s a ⟹ s' = s(tm := Suc 0, sum := Suc 0, i := 0)
```

oops

Summary: With this approach, one can synthesize paths and their conditions.

5.5.7 Testing Specifications

lemma *wb_specification_test:*
assumes *no_alias : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum ∧*
sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm ∧
a ≠ i ∧ i ≠ a"
shows *"|- {pre} squareroot tm sum i a {post a i}"*

apply *(simp add: squareroot_def pre_def)*
apply *(rule_tac n1 = "3"* **in** *iffD2[OF setup_test])*
apply *(rule allI, rule allI, simp, rule impI)*

apply(*simp_all add: update_def no_alias uw_while*)
apply(*erule If_splitE, simp_all add: update_def no_alias uw_while*)+

We "wrap" the first part into the test hypothesis:

apply(*erule rev_mp*) **back**
apply(*erule THYP_app1_rev*)

The resulting proof state captures the essence of this white box test:

```
1. ⋀s t. THYP
          (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ s a ⟶
           ⟨WHILE λs. s sum
                      ≤ s a DO i :== λs. Suc
(s i) ; (tm :== λs. Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ),s
           (i := Suc (Suc (Suc 0)),
            tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
            sum :=
              Suc (Suc (Suc (Suc (Suc (Suc
  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
           ⟶c t ⟶ post a i t)
2. ⋀s t. ⟦Suc (Suc (Suc (Suc 0))) ≤ s a;
          ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ s a;
          t = s
          (i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
           sum := Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))⟧
         ⟹ post a i
              (s(i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
                 sum :=
                   Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))
3. ⋀s t. ⟦Suc 0 ≤ s a; ¬ Suc (Suc (Suc (Suc 0))) ≤ s a;
          t = s
          (i := Suc 0, tm := Suc (Suc (Suc 0)),
           sum := Suc (Suc (Suc (Suc 0))))⟧
         ⟹ post a i
              (s(i := Suc 0, tm := Suc (Suc (Suc 0)),
                 sum := Suc (Suc (Suc (Suc 0)))))
4. ⋀s t. ⟦¬ Suc 0 ≤ s a; t = s(tm := Suc 0, sum := Suc 0, i := 0)⟧
         ⟹ post a i (s(tm := Suc 0, sum := Suc 0, i := 0))
```

Now testing all paths for compliance to post condition:

apply(*simp_all add: no_alias post_def*)

In this special case—arithmetic constraints—the system can even **verify** these constraints, i.e. the simplifier shows that all postconditions follow from the initial constraints and the computed relation between pre-state and post state.

```
1. ⋀s t. THYP
          (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ s a ⟶
           ⟨WHILE λs. s sum
                      ≤ s a DO i :== λs. Suc
```

```
(s i) ; (tm :== λs. Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ),s
            (i := Suc (Suc (Suc 0)),
             tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))),
             sum :=
               Suc (Suc (Suc (Suc (Suc (Suc
  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
          ⟶c t ⟶ t i * t i ≤ t a ∧ t a < Suc (t i + (t i + t i * t i))
```

To say it loud and clearly: The white box test decomposes the original specification into a test hypothesis for cases with $3^3 = 9 <= sa$ and all other cases (e.g. $2^2 = 4 <= sa \wedge sa < 9$). The latter have been proven automatically.

oops

5.5.8 An Alternative Approach with an On-The-Fly generated Explicit Test-Hyp.

lemma *path_exploration_test:*
assumes *no_alias : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum ∧*
sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm ∧
a ≠ i ∧ i ≠ a"
shows *"|- {pre} squareroot tm sum i a {post a i}"*

We fire the basic white-box scenario:

apply *(rule wp_test [of _ "3"])*

Given the concrete unwinding factor and the concrete program term, standard normalization yields an "Path Exhaustion Theorem" with the explicit test hypothesis:

apply *(auto simp: squareroot_def update_def no_alias uw_while)*
apply *(tactic "thyp_ify 1")*
defer *1*

and we reach the following instantiation of a white-box test-theorem (with explicit test-hypothesis for the uncovered paths):

```
1. ⋀σ. ⟦pre σ; Suc (Suc (Suc (Suc 0))) ≤ σ a;
        ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ σ a⟧
       ⟹ post a i
            (σ(i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc 0)))),
               sum :=
                 Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))
2. ⋀σ. ⟦pre σ; Suc 0 ≤ σ a; ¬ Suc (Suc (Suc 0)) ≤ σ a⟧
       ⟹ post a i
            (σ(i := Suc 0, tm := Suc (Suc 0)),
               sum := Suc (Suc (Suc (Suc 0)))))
3. ⋀σ. ⟦pre σ; ¬ Suc 0 ≤ σ a⟧
       ⟹ post a i (σ(tm := Suc 0, sum := Suc 0, i := 0))
4. THYP
    (∀x xa xb xc xd.
       Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ xd xc ⟶
```

```
         pre xd ⟶
         wp (WHILE λs. s xa
                       ≤ s xc DO xb := λs.
Suc (s xb) ; (x := λs. Suc (Suc (s x)) ; xa := λs. s x + s xa ))
          (post xc xb)
          (xd(xb := Suc (Suc (Suc 0)),
              x := Suc (Suc (Suc (Suc (Suc (Suc 0))))),
              xa :=
                Suc (Suc (Suc (Suc (Suc (Suc
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))))
```

Now we allso perform the "tests" by symbolic execution:

apply(*auto simp: no_alias pre_def post_def*)

which leaves us just with test-hypothesis case; for all paths *not* leading to a remaining while, the program is correct.

```
 1. THYP
     (∀x xa xb xc xd.
         Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ xd xc ⟶
         wp (WHILE λs. s xa
                       ≤ s xc DO xb := λs.
Suc (s xb) ; (x := λs. Suc (Suc (s x)) ; xa := λs. s x + s xa ))
          (λs. s xb * s xb ≤ s xc ∧
               s xc < Suc (s xb + (s xb + s xb * s xb)))
          (xd(xb := Suc (Suc (Suc 0)),
              x := Suc (Suc (Suc (Suc (Suc (Suc 0))))),
              xa :=
                Suc (Suc (Suc (Suc (Suc (Suc
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))))
```

oops

end

5.6 Sequence Testing

In this section, we apply HOL-TestGento different sequence testing scenarios; see [15] for more details.

theory *Sequence_test*
imports
List
Testing
begin

In this theory, we present present a simple reactive system and demonstrate and show how HOL-TestGen can be used for testing such systems.

Our scenario is motivated by the following communcation scenario: A client sends a server a communication request and specifies a port-range X. The server non-deterministically chooses a port Y which is within the specified range. The client sends a sequence of data (abstracted away in our example to a constant `Data`) on the port allocated by the server. The communication is terminated by the client with a `stop` event. Using a CSP-like notation, we can describe such a system as follows:

$$req?X \rightarrow port?Y[Y < X] \\ \rightarrow (rec\ N \bullet send!D, Y \rightarrow ack \rightarrow N \ \Box\ stop \rightarrow ack \rightarrow SKIP)$$

It is necessary for our approach that the protocol strictly alternates client-side and server-side events; thus, we will be able to construct in a test of the server a step-function ioprog (see below) that stimulates the server with an input and records its result. If a protocol does not have alternation in its events, it must be constructed by artificial acknowledge events; it is then a question of their generation in the test-harness, if they were sent anyway or if they correspond to something like "server reacted within timebounds."

The *stimulation sequence* of the system under test results just from the projection of this protocol to the input events:

$$req?X \rightarrow (rec\ N \bullet send!D, Y \rightarrow N \ \Box\ stop \rightarrow SKIP)$$

5.6.1 Basic Technique: Events with explicit variables

We define *abstract traces* containing explicit variables X, Y, ... The whole testcase-generation is done on the basis of the abstract traces. However, some small additional functions *substitute* and *bind* were used to replace them with concrete values during the run of the test-driver, as well as programs that check pre and post conditions on the concrete values occuring in the concrete run.

We specify explicit variables and a joined type containing abstract events (replacing values by explicit variables) as well as their concrete counterparts.

```
datatype vars = X | Y
datatype data = Data

datatype dvars = inp vars | outp vars

types    chan = int

datatype InEvent  = req chan       | reqA vars |
                    send data chan | sendA data vars |
                    stop

datatype OutEvent = port chan      | portA vars|
                    ack
```

```
constdefs lookup :: "['a ⇀ 'b, 'a] ⇒ 'b"
          "lookup env v ≡  the(env v)"
          success:: "'α option ⇒ bool"
          "success x ≡ case x of None => False | Some x => True"

types     event = "InEvent + OutEvent"
```

5.6.2 The infra-structure of the observer: substitute and rebind

The predicate *subsitute* allows for mapping abstract events containing explicit variables to concrete events by substituting the variables by values communicated in the system run. It requires an environment ("substitution") where the concrete values occuring in the system run were assigned to variables.

```
consts   substitute :: "[vars ⇀ chan, InEvent] ⇒ InEvent"
primrec
   "substitute env (req n)    = req n"
   "substitute env (reqA v)   = req(lookup env v)"
   "substitute env (send d n) = send d n"
   "substitute env (sendA d v)= send d (lookup env v)"
   "substitute env stop       = stop"
```

The predicate *rebind* extracts from concrete output events the values and binds them to explicit variables in env. It should never be applied to abract values; therefore, we we can use an underspecified version (arbitrary). The predicate *rebind* only stores ?-occurrences in the protocol into the environment; !-occurences were ignored. Events, that are the same in the abstract as well as the concrete setting are treated as abstract event.

In a way, *rebind* can be viewed as an abstraction of the concrete log produced at runtime.

```
consts   rebind :: "[vars ⇀ chan, OutEvent] ⇒ vars ⇀ chan"
primrec
   "rebind env (port n)   = env(Y↦n)"
   "rebind env ack        = env"
```

5.6.3 Abstract Protocols and Abstract Stimulation Sequences

Now we encode the protocol automaton (abstract version) by a recursive acceptance predicate. One could project the set of stimulation sequences just by filtering out the outEvents occuring in the traces.

We will not pursue this approach since a more constructive presentation of the stimulation sequence set is advisable for testing.

However, we show here how such concepts can be specified.

```
syntax  A :: "nat"
        B :: "nat"
        C :: "nat"
        D :: "nat"
```

```
        E :: "nat"

translations "A" == "0"
             "B" == "Suc A"
             "C" == "Suc B"
             "D" == "Suc C"
             "E" == "Suc D"

consts accept' :: "nat × event list ⇒ bool"
recdef accept' "measure(λ (x,y). length y)"
   "accept'(A,(Inl(reqA X))#S)    = accept'(B,S)"
   "accept'(B,(Inr(portA Y))#S)   = accept'(C,S)"
   "accept'(C,(Inl(sendA d Y))#S) = accept'(D,S)"
   "accept'(D,(Inr(ack))#S)       = accept'(C,S)"
   "accept'(C,(Inl(stop))#S)      = accept'(E,S)"
   "accept'(E,[Inr(ack)])         = True"
   "accept'(x,y)                  = False"

constdefs accept :: "event list ⇒ bool"
   "accept s ≡ accept'(0,s)"
```

We proceed by modeling a subautomaton of the protocol automaton accept.

```
consts stim_trace' :: "nat × InEvent list ⇒ bool"
recdef stim_trace' "measure(λ (x,y). length y)"
   "stim_trace'(A,(reqA X)#S)    = stim_trace'(C,S)"
   "stim_trace'(C,(sendA d Y)#S) = stim_trace'(C,S)"
   "stim_trace'(C,[stop])        = True"
   "stim_trace'(x,y)             = False"

constdefs stim_trace :: "InEvent list ⇒ bool"
   "stim_trace s ≡ stim_trace'(A,s)"
```

5.6.4 The Post-Condition

```
consts
   postcond' :: "((vars ⇀ int) × unit × InEvent × OutEvent) ⇒ bool"

recdef postcond' "{}"
   "postcond' (env, x, req n, port m) = (m <= n)"
   "postcond' (env, x, send z n, ack) = (n = lookup env Y)"
   "postcond' (env, x, stop, ack)     = True"
   "postcond' (env, x, y, z)          = False"


constdefs
   postcond :: "(vars ⇀ int) × unit ⇒ InEvent ⇒ OutEvent ⇒ bool"
  "postcond  x y z ≡ postcond' (fst x, snd x, y, z)"
```

5.6.5 Testing for successful system runs of the server under test

So far, we have not made any assumption on the state σ' of the of our program under test ioprog. It could be a log of the actual system run. However, for simplicity, we give it inly a trivial state in this test specification.

5.6.6 Test-Generation: The Standard Approach

```
declare stim_trace_def[simp]
declare Mfold.simps[simp del]

test_spec "stim_trace trace ⟶
           success(Mfold trace (empty(X↦init),())
                       (observer rebind substitute postcond ioprog))"
  apply(gen_test_cases 4 1 "ioprog")
  store_test_thm "reactive"

testgen_params [iterations=1000]
gen_test_data "reactive"

thm  reactive.test_data

testgen_params [
  gen_wrapper=false,
  setup_code="
    fun fst(x,y) = x
    fun snd(x,y) = y
  ",
  dataconv_code="
    fun sendToServer event Unity = let
      fun toServerData Data = server.Data
      fun toServerVars X    = server.X
        | toServerVars Y    = server.Y
      fun fromServerData server.Data = Data
      fun fromServerVars server.X    = X
        | fromServerVars server.Y    = Y
      fun convert_InEvent (req x)    = server.req x
        | convert_InEvent (reqA x)   = server.reqA (toServerVars x)
        | convert_InEvent (send (x,y))
                                = server.send  (toServerData x,y)
        | convert_InEvent (sendA (x,y))
                                = server.sendA (toServerData x, toServerVars y)
        | convert_InEvent stop      = server.stop
      fun convert_OutEvent (server.port x)  = Some(port x,Unity)
        | convert_OutEvent (server.portA x) = Some(portA (fromServerVars x),Unity)
        | convert_OutEvent server.ack       = Some(ack,Unity)
    in
      convert_OutEvent (server.read (convert_InEvent event))
    end
```

```
"]
```

gen_test_script *"sequence_script.sml" reactive ioprog "sendToServer"*

5.6.7 Test-Generation: Refined Approach involving TP

An analysis of the previous approach shows, that random solving on trace patterns is obviously still quite ineffective. Although path coverage wrt. the input stimulation trace automaton can be achieved with a reasonable high probability, the depth remains limited.

The idea is to produce a better test theorem by more specialized rules, that take the special form of the input stimulation protocol into account.

```
lemma start :
   "stim_trace'(A,x#S) = ((x = reqA X) ∧ stim_trace'(C,S))"
apply(cases "x", simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done

lemma final[simp]:
"(stim_trace' (x, stop # S)) = ((x=C)∧(S=[]))"
apply(case_tac "x=Suc (Suc (A::nat))", simp_all)
apply(cases "S",simp_all)
apply(case_tac "x=Suc (A::nat)", simp_all)
apply(case_tac "x = (A::nat)", simp_all)
apply(subgoal_tac "∃ xa. x = Suc(Suc(Suc xa))",erule exE,simp)
apply(arith)
done

lemma step1 :
   "stim_trace'(C,x#y#S) = ((x=sendA Data Y) ∧stim_trace'(C,y#S))"
apply(cases "x", simp_all)
apply(rule_tac y="data" in  data.exhaust,simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done

lemma step2:
   "stim_trace'(C,[x]) = (x=stop)"
apply(cases "x", simp_all)
apply(rule_tac y="data" in  data.exhaust,simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done
```

The three rules *start*, *step*1 and *step*2 give us a translation of a constraint of the form $\mathrm{stim_t race}'(x, [a, \ldots, b])$ into a simple conjunction of equalities (in general: disjunction and existential quantifier will also occur). Since a formula of this form is an easy game for $fast_tac$ inside *gen_test_cases*, we will get dramatically better test theorems, where the constraints have been resolved already.

We reconfigure the rewriter ...

```
declare start[simp] step1[simp] step2 [simp]


test_spec "stim_trace trace ⟶
           success(Mfold trace (empty(X↦init),())
                    (observer rebind substitute postcond ioprog))"
  apply(gen_test_cases 40 1 "ioprog")
  store_test_thm "reactive2"

testgen_params [iterations=1]
gen_test_data "reactive2"

thm  reactive2.test_data

gen_test_script "sequence_script.sml" reactive2 ioprog "sendToServer"
```

Within the timeframe of 1 Minute, we get trace lengthes of about 40 in the stimulation input protocol, which corresponds to traces of 80 in the standard protocol. The examples shows, that it is not the length of traces that is a limiting factor of our approach. The main problem is the complexity in the stimulation automaton (size, brachning-factors, possible instantiations of parameter input).

end

5.7 HOL-TestGen/FW: A Domain-specific Test Tool for Firewall Policies

As HOL-TestGenis built on the framework of Isabelle with a general plug-in mechanism, HOL-TestGencan be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FWwhich extends HOL-TestGenby:

1. a theory (or library) formalizing networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [26].

HOL-TestGen/FWis part of the HOL-TestGendistribution. It is located in the directory `examples/hol-testgen-fw`; see [15, 12] for more details.

Figure 5.2 shows the overall architecture of HOL-TestGen/FW.

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see [12] and the examples provided in the dirctory `examples/hol-testgen-fw` for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

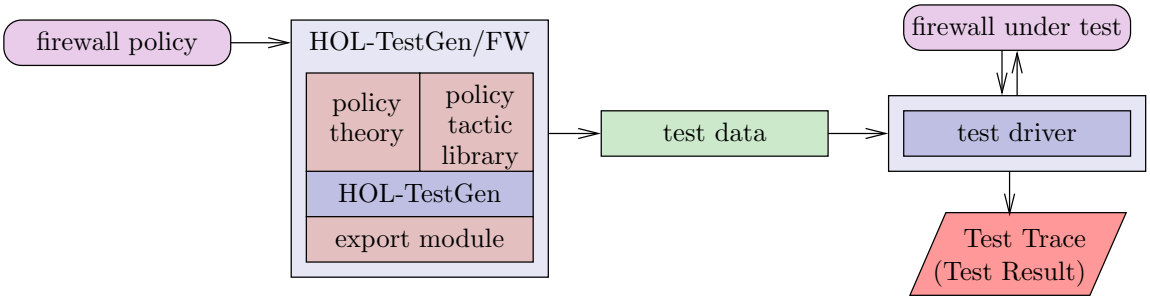


Figure 5.2: The HOL-TestGen/FWarchitecture.

With item 2 we refer to domain-specific processing encapsulated the general HOL-TestGentest-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimized version of the procedure, now tuned for stateless firewall policies. Moreover, there are new control parameters for the simplification.

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i. e., a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [26]. This completes our toolchain which, thus, supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

A Glossary

Abstract test data : In contrast to pure ground terms over constants (like integers $1, 2, 3$, or lists over them, or strings ...) abstract test data contain arbitrary predicate symbols (like *triangle 3 4 5*).

Regression testing: Repeating of tests after addition/bug fixes have been introduced into the code an checking that behavior of unchanged portions has not changed.

Stub: Stubs are "simulated" implementations of functions, they are used do simulate functionality that does not yet exists ore cannot be run in the test environment.

Test case: An abstract test stimuli that test some aspects of the implementation and validates the result.

Test case generation: For each operation of the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test data: One or more representative for a given test case.

Test data generation (Test data selection): For each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test execution: The implementation is run with the selected test input data in order to determine the test output data.

Test executable: An executable program that consists of a test harness, the test script and the program under test. The Test executable executes the test and writes a test trace documenting the events and the outcome of the test.

Test harness: When doing unit testing the program under test is not a runnable program in itself. The *test harness* or *test driver* is a main program that initiates test calls (controlled by the test script), i. e., drives the method under test and constitutes a test executable together with the test script and the program under test.

Test hypothesis : The hypothesis underlying a test that makes a successful test equivalent to the validity of the tested property, the test specification. The current implementation of HOL-TestGen only supports uniformity and regularity hypothesis, which were generated "on-the-fly" according to certain parameters given by the user like *depth* and *breadth*.

Test specification : The property the program under test is required to have.

Test result verification: The pair of input/output data is checked against the specification of the test case.

Test script: The test program containing the control logic that drives the test using the test harness. HOL-TestGen can automatically generate the test script for you based on the generated test data.

Test theorem: The test data together with the test hypothesis will imply the test specification. HOL-TestGen conservatively computes a theorem of this form that relates testing explicitly with verification.

Test trace: Output made by a test executable.

Bibliography

[1] The archive of formal proofs (AFP). URL `http://afp.sourceforge.net/`.

[2] Isabelle. URL `http://isabelle.in.tum.de`.

[3] MLj. URL `http://www.dcs.ed.ac.uk/home/mlj/index.html`.

[4] MLton. URL `http://www.mlton.org/`.

[5] Poly/ML. URL `http://www.polyml.org/`.

[6] Proof General. URL `http://proofgeneral.inf.ed.ac.uk`.

[7] SML of New Jersey. URL `http://www.smlnj.org/`.

[8] sml.net. URL `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

[9] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, May 1986. ISBN 0120585367.

[10] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[11] A. Biere, A. Cimatti, Edmund Clarke, Ofer Strichman, and Y. Zhu. *Bounded Model Checking*. Number 58 in Advances In Computers. 2003.

[12] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Model-based firewall conformance testing. In Kenji Suzuki and Teruo Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, Tokyo, Japan, 2008.

[13] 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, Linz, 2005. ISBN 3-540-25109-X. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-symbolic-2005`.

[14] Achim D. Brucker and Burkhart Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005. ISSN 1433-2779. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-verification-2005`.

[15] 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, Zurich, 2007. ISBN 978-3-540-73769-8.

[16] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[17] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000. ISBN 1-58113-202-6.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[19] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 1972. ISBN 0-12-200550-3.

[20] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specications. In J.C.P. Woodcock and P.G. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284. Springer, April 1993.

[21] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying haskell programs by combining testing and proving. In *Proceedings of the Third International Conference on Quality Software*, page 272. IEEE Computer Society, 2003. ISBN 0-7695-2015-4. URL `http://csdl.computer.org/comp/proceedings/qsic/2003/2015/00/20150272abs.htm`.

[22] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995. ISBN 3-540-59293-8.

[23] Susumu Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. URL `http://www4.in.tum.de/~nipkow/LNCS2283/`.

[25] N. D. North. Automatic test generation for the triangle problem. Technical Report DITC 161/90, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UK, February 1990.

[26] Diana von Bidder. *Specification-based Firewall Testing*. Ph.D. Thesis, ETH Zurich, 2007. ETH Diss. No. 17172. Diana von Bidder's maiden name is Diana Senn.

[27] Markus Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 2004. URL `http://isabelle.in.tum.de/dist/Isabelle2004/doc/isar-ref.pdf`.

[28] Hong Zhu, Patrick A.V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. ISSN 0360-0300.

Index

U