

HOL-TestGen 1.9.1

User Guide

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

Achim D. Brucker	Lukas Brügger	Abderrahmane Feliachi
Chantal Keller	Matthias P. Krieger	Delphine Longuet
Yakoub Nemouchi	Frederic Tuong	Burkhart Wolff

August 31, 2017

Department of Computer Science
The University of Sheffield
S14DP Sheffield
UK

Laboratoire en Recherche en Informatique (LRI)
Université Paris-Sud 11
91405 Orsay Cedex
France

Copyright © 2003–2012 ETH Zurich, Switzerland
Copyright © 2007–2015 Achim D. Brucker, Germany
Copyright © 2008–2016 University Paris-Sud, France
Copyright © 2016 The University of Sheffield, UK

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Note:

This manual describes HOL-TestGen version 1.9.1 (r13171). The manual of version 1.8.0 is also available as technical report number 1586 from the Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France.

Contents

1. Introduction	5
2. Preliminary Notes on Isabelle/HOL	7
2.1. Higher-order logic — HOL	7
2.2. Isabelle	7
3. Installation	9
3.1. Prerequisites	9
3.2. Installing HOL-TestGen	9
3.3. Starting HOL-TestGen	9
4. Using HOL-TestGen	11
4.1. HOL-TestGen: An Overview	11
4.2. Test Case and Test Data Generation	11
4.3. Test Execution and Result Verification	17
4.3.1. Testing an SML-Implementation	17
4.3.2. Testing Non-SML Implementations	19
4.4. Profiling Test Generation	19
5. Examples	21
5.1. List	21
6. Testing List Properties	29
6.1. Bank	41
7. A Simple Deterministic Bank Model	51
8. A Simple Non-Deterministic Bank Model	61
8.1. MyKeOS	64
9. The MyKeOS Case Study	65
10. The MyKeOS “Traditional” Data-sequence enumeration approach	71
A. Glossary	87

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 *Dijkstra’s verdict* [13, 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 [3, 12].

Systematic Testing: the discussion over *test adequacy criteria* [26], 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 [16, 14].

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 modelling errors early and to increase the overall productivity [2, 17, 11].

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 [7]).

Following standard terminology [26], 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 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 [8] has been inspired by [15], 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 [11], the work presented in [15] 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 [2]. 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 exploits these predicates and other specification data in order to produce adequate data, combining automatic data splitting, automatic constraint solving, and manual deduction.

As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis is valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification, see [6, 9] for details.

2. Preliminary Notes on Isabelle/HOL

2.1. Higher-order logic — HOL

Higher-order logic (HOL) [10, 1] is a classical logic with equality enriched by total polymorphic¹ 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 [21, 18] is a *generic* theorem prover. New object logics 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 chose 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.

¹to be more specific: *parametric polymorphism*

3. Installation

3.1. Prerequisites

HOL-TestGen is built on top of Isabelle/HOL, version 2016-1, thus you need a working installation of *Isabelle 2016-1*. To install Isabelle, follow the instructions on the Isabelle web-site:

```
http://isabelle.in.tum.de/website-Isabelle2016-1/index.html
```

3.2. Installing HOL-TestGen

In the following we assume that you have a running Isabelle 2016-1 environment. The installation of HOL-TestGen requires the following steps:

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

```
tar zxvf hol-testgen-1.9.1.tar.gz
```

This will create a directory `hol-testgen-1.9.1` containing the HOL-TestGen distribution.

```
cd hol-testgen-1.9.1
```

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

```
isabelle build -d . -b HOL-TestGen
```

3.3. Starting HOL-TestGen

HOL-TestGen can now be started using the `isabelle` command:¹

```
isabelle jedit -d . -l HOL-TestGen "examples/unit/List/List_test.thy"
```

After a few seconds you should see an jEdit window similar to the one shown in Figure 3.1.

Alternatively, the example can be run in batch mode, e. g.,

```
isabelle build -d . HOL-TestGen-List
```

¹Note that the `isabelle` command must be provided by Isabelle 2016-1.

The screenshot shows the Isabelle/jEdit interface. The main editor window, titled "List_test.thy (modified)", contains the following code:

```

test_spec "sort l = PUT l"
apply(gen_test_cases 4 1 "PUT")
txt {
  which leads after 2 seconds to the following test partitioning (excerpt):
  @{subgoals [display, goals_limit=10]}
}
store_test_thm "is_sorting_algorithm"

thm is_sorting_algorithm.test_thm

declare [[testgen_iterations=100]]
gen_test_data "is_sorting_algorithm"

thm is_sorting_algorithm.test_data
text{* We obtain test cases like:

```

The output window at the bottom displays the generated test cases, each preceded by a `PUT` command. The test cases are:

```

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

```

The status bar at the bottom indicates the session is running on Isabelle2019.1.0, with 287.18 (10196/12319) MB of memory used.

Figure 3.1.: A HOL-TestGen session Using the jEdit 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 and test data generation (selection) is done in an Isar-based [25] environment (see Figure 4.1 for details). The test executable (and the generated test script) can be built with any SML-system.

4.2. Test Case and Test Data Generation

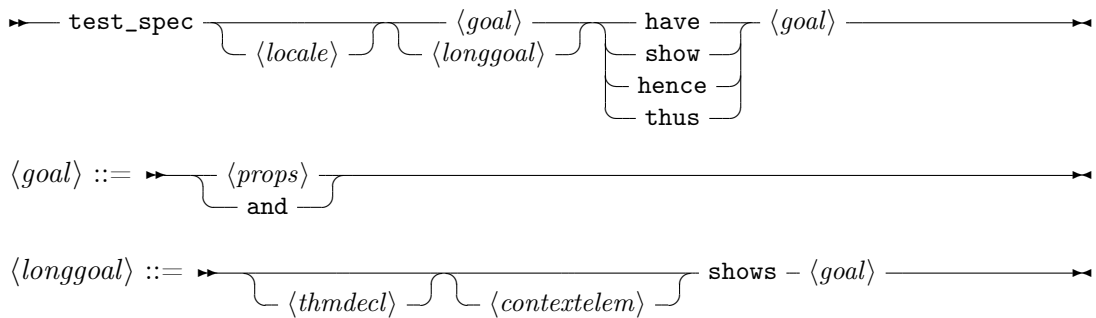
In this section we give a brief overview of HOL-TestGen related extension of the Isar [25] proof language. We use a presentation similar to the one in the *Isar Reference Manual* [25], e.g. “missing” non-terminals of our syntax diagrams are defined in [25]. We introduce the HOL-TestGen syntax by a (very small) running example: assume we want to test a function 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 simple program computing the maximum value of two integers. The syntax of the keyword **test_spec** : *theory* → *proof*(*prove*) is given by:



Please look into the Isar Reference Manual [25] for the remaining details, e.g. a description of *<contextelem>*.

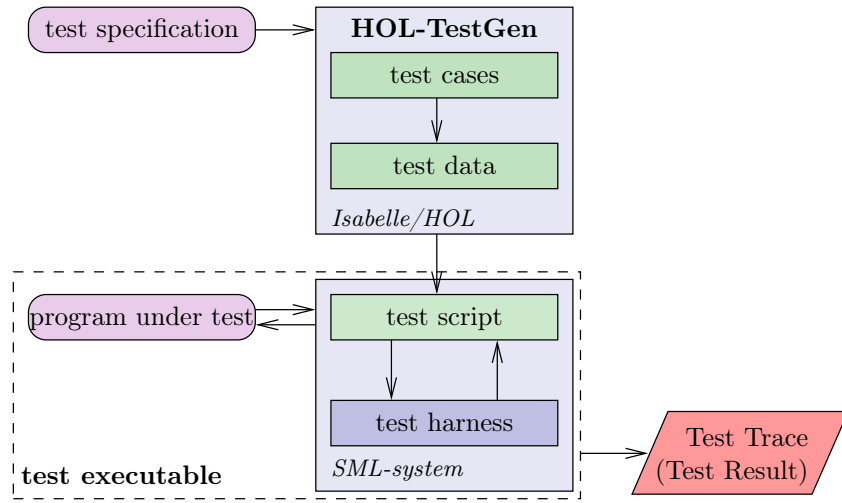


Figure 4.1.: Overview of the system architecture of HOL-TestGen

```

theory max_test
imports Testing
begin

test_spec "prog a b = max a b"
  apply(gen_test_cases "prog" simp: max_def)
  mk_test_suite "max_test"

gen_test_data "max_test"

print_conc_tests max_test

generate_test_script "max_test"
thm max_test.test_script

text {* Testing an SML implementation: *}
export_code max_test.test_script in SML module_name TestScript file "impl/sml/max_test_script.sml"

text {* Finally, we export the raw test data in an XML-like format: *}
export_test_data "impl/data/max_data.dat" max_test

end

```

Table 4.1.: A simple Testing Theory

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

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

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

► `gen_test_cases` $\underbrace{\hspace{10em}}_{\langle depth \rangle - \langle breadth \rangle}$ $\langle proname \rangle$ $\underbrace{\hspace{10em}}_{\langle clasimpmod \rangle}$ ►

where $\langle depth \rangle$ is a natural number describing the depth of the generated test cases and $\langle breadth \rangle$ is a natural number describing their breadth. Roughly speaking, the $\langle depth \rangle$ controls the term size in data separation lemmas in order to establish a regularity hypothesis (see [6] for details), while the $\langle breadth \rangle$ controls the number of variables occurring in the test specification for which regularity hypotheses are generated. The default for $\langle depth \rangle$ and $\langle breadth \rangle$ is 3 resp. 1. $\langle proname \rangle$ 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 $\langle clasimpmod \rangle$ option:

$\langle clasimpmod \rangle ::=$ ► $\underbrace{\hspace{10em}}_{\text{simp}} \underbrace{\hspace{10em}}_{\text{cong}} \underbrace{\hspace{10em}}_{\text{iff}} \underbrace{\hspace{10em}}_{\text{intro}} : - \langle thmrefs \rangle$ ►

$\underbrace{\hspace{10em}}_{\text{del}} \underbrace{\hspace{10em}}_{\text{only}}$
 $\underbrace{\hspace{10em}}_{\text{split}} \underbrace{\hspace{10em}}_{\text{add}} \underbrace{\hspace{10em}}_{\text{del}}$
 $\underbrace{\hspace{10em}}_{\text{add}} \underbrace{\hspace{10em}}_{\text{del}} \underbrace{\hspace{10em}}_{\text{?}}$
 $\underbrace{\hspace{10em}}_{\text{!}} \underbrace{\hspace{10em}}_{\text{?}}$
 $\underbrace{\hspace{10em}}_{\text{del}}$

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

Creating a test suite: HOL-TestGen provides a kind of container, called *test-suites*, which store all relevant logical and configuration information related to a particular test-scenario. Test-suites were initially created after generating the test cases (and test hypotheses); you should store your result of the derivation, usually the test-theorem which is the output of the test-generation phase, in a test suite by:

```
mk_test_suite "max_test"
```

for further processing. This is done using the `mk_test_suite` : $proof(prove) \rightarrow proof(prove) \mid theory$ command which also closes the actual “proof state” (or *test state*). Its syntax is given by:

► `mk_test_suite` - $\langle name \rangle$ ►

where $\langle name \rangle$ is a fresh identifier which is later used to refer to this test state. This name is even used at the very end of the test driver generation phase, when test-executions are performed (externally to HOL-TestGen in a shell). Isabelle/HOL can access the corresponding test theorem using the identifier $\langle name \rangle.test_thm$, 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 suite for which the test data should be generated:

►— `gen_test_data - <name>` —————►

After the successful execution of this command Isabelle can access the test hypotheses using the command `print_thyps <name>` and the test data using the command `print_conc_tests <name>`

```
print_thyps max_test
print_conc_tests max_test
```

In our concrete example, we get the output:

```
THYP (( $\exists x\ xa.\ x \leq xa \wedge \text{prog } x\ xa = xa$ )  $\longrightarrow$  ( $\forall x\ xa.\ x \leq xa \longrightarrow \text{prog } x\ xa = xa$ ))
THYP (( $\exists x\ xa.\ \neg x \leq xa \wedge \text{prog } x\ xa = x$ )  $\longrightarrow$  ( $\forall x\ xa.\ \neg x \leq xa \longrightarrow \text{prog } x\ xa = x$ ))
```

as well as :

```
prog -9 -3 = -3
prog -5 -8 = -5
```

By default, generating test data is done by calling the *random solver*. This is fine for such a simple example, but as explained in the introduction, this is far incomplete when the involved data-structures become more complex. To handle them, HOL-TestGen also comes with a more advanced data generator based on *SMT solvers* (using their integration in Isabelle, see e. g. [4]).

To turn on SMT-based data generation, use the following option:

```
declare [[testgen_SMT]]
```

(which is thus set to `false` by default). It is also recommended to turn off the random solver:

```
declare [[ testgen_iterations=0]]
```

In order for the SMT solver to know about constant definitions and properties, one needs to feed it with these definitions and lemmas. For instance, if the test case involves some inductive function `foo`, you can provide its definition to the solver using:

```
declare foo.simps [testgen_smt_facts]
```

as well as related properties (if needed).

A complete description of the configuration options can be found below.

Note that the SMT solver which is used is Z3, which is restricted to non-commercial use in Isabelle. Hence you can use the SMT backend only for academic purposes. To make this clear, you need to define (in your operating system) the following environment variable:

OLD_Z3_NON_COMMERCIAL=yes

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 `test_max.dat`. The generation of a test data file is done using the `export_test_data : theory|proof → theory|proof` command:

→ `export_test_data - <filename> - <name>` $\xrightarrow{\langle smlprogname \rangle}$ →

where $\langle filename \rangle$ is the name of the file in which the test data is stored and $\langle name \rangle$ 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"
```

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

→ `gen_test_script - <filename> - <name> - <progname>` $\xrightarrow{\langle smlprogname \rangle}$ →

where $\langle filename \rangle$ is the name of the file in which the test script is stored, and $\langle name \rangle$ is the name of a collection of test data in the test environment, and $\langle progname \rangle$ the name of the program under test. The optional parameter $\langle smlprogname \rangle$ allows for the configuration of different names of the program under test that is used within the test script for calling the implementation.

Alternatively, the code-generator can be configured to generate test-driver code in other programming languages, see below.

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

```
declare [[ testgen_ iterations =15]]
```

The parameters (all prefixed with `testgen_`) have the following meaning:

depth:	Test-case generation depth. Default: 3.
breadth:	Test-case generation breadth. Default: 1.
bound:	Global bound for data statements. Default: 200.
case_breadth:	Number of test data per case, weakening uniformity. Default: 1.
iterations:	Number of attempts during random solving phase. Default: 25. Set to 0 to turn off the random solver.
gen_prelude:	Generate datatype specific prelude. Default: true.

```

structure TestDriver : sig end = struct
  val return = ref ~63;
3  fun eval x2 x1 = let
      val ret = myMax.max x2 x1
      in
        ((return := ret);ret)
      end
8  fun retval () = SOME(!return);
  fun toString a = Int.toString a;
  val testres = [];

  val pre_0 = [];
13  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 = [];
18  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;
23 end

```

Table 4.2.: Test Script

gen_wrapper:	Generate wrapper/logging-facility (increases verbosity of the generated test script). Default: true.
SMT:	If set to “true” external SMT solvers (e.g., Z3) are used during test-case generation. Default: false.
smt_facts:	Add a theorem to the SMT-based data generator basis.
toString:	Type-specific SML-function for converting literals into strings (e.g., <code>Int.toString</code>), used for generating verbose output while executing the generated test script. Default: "".
setup_code:	Customized setup/initialization code (copied verbatim to generated test script). Default: "".
dataconv_code:	Customized code for converting datatypes (copied verbatim to generated test script). Default: "".
type_range_bound:	Bound for choosing type instantiation (effectively used elements type grounding list). Default: 1.
type_candidates:	List of types that are used, during test script generation, for instantiating type variables (e.g., α list). The ordering of the types determines their likelihood of being used for instantiating a polymorphic type. Default: [int, unit, bool, int set, int list]


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

Table 4.3.: Implementation in SML of max

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

```
lemma max_abscae [test "maxtest"]:"max 4 7 = 7"
```

or

```
declare max_abscase [test "maxtest"]
```

that can be used for hierarchical test case generation:

► test - $\langle name \rangle$ _____ ◄

4.3. Test Execution and Result Verification

In principle, any SML-system, e. g. [24, 22, 23, 19, 20], 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 [23],
- implementations written in C using, e.g. the foreign language interface of sml/NJ [24] or MLton [20],
- implementations written in Java using mlj [19].

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.

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:

```

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

```

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

```

After the last command, sml/NJ will automatically execute our test and 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
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: 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

```

1  int max (int x, int y) {
2      if (x < y) {
3          return y;
4      }else{
5          return x;
6      }
7  }

```

Table 4.5.: Implementation in ANSI C of max

```
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 4.4.

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 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 4.4.

4.4. Profiling Test Generation

HOL-TestGen includes support for profiling the test procedure. By default, profiling is turned off. Profiling can be turned on by issuing the command

➡— profiling_on —————➡

Profiling can be turned off again with the command

►— `profiling_off` —►

When profiling is turned on, the time consumed by `gen_test_cases` and `gen_test_data` is recorded and associated with the test theorem. The profiling results can be printed by

►— `print_clocks` —►

A LaTeX version of the profiling results can be written to a file with the command

►— `write_clocks - <filename>` —►

Users can also record the runtime of their own code. A time measurement can be started by issuing

►— `start_clock - <name>` —►

where `<name>` is a name for identifying the time measured. The time measurement is completed by

►— `stop_clock - <name>` —►

where `<name>` has to be the name used for the preceding `start_clock`. If the names do not match, the profiling results are marked as erroneous. If several measurements are performed using the same name, the times measured are added. The command

►— `next_clock` —►

proceeds to a new time measurement using a variant of the last name used.

These profiling instructions can be nested, which causes the names used to be combined to a path. The `Clocks` structure provides the tactic analogues `start_clock_tac`, `stop_clock_tac` and `next_clock_tac` to these commands. The profiling features available to the user are independent of HOL-TestGen's profiling flag controlled by `profiling_on` and `profiling_off`.

5. Examples

5.1. List

Implementation of integer numbers by target-language integers

```
theory Code-Target-Int
imports ../GCD
begin

code-datatype int-of-integer

declare [[code drop: integer-of-int]]

context
includes integer.lifting
begin

lemma [code]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

lemma [code]:
  Int.Pos = int-of-integer  $\circ$  integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code]:
  Int.Neg = int-of-integer  $\circ$  uminus  $\circ$  integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code-abbrev]:
  int-of-integer (numeral k) = Int.Pos k
  by transfer simp

lemma [code-abbrev]:
  int-of-integer ( $-$  numeral k) = Int.Neg k
  by transfer simp

lemma [code, symmetric, code-post]:
  0 = int-of-integer 0
  by transfer simp

lemma [code, symmetric, code-post]:
  1 = int-of-integer 1
  by transfer simp
```

```

lemma [code-post]:
   $\text{int-of-integer } (-\ 1) = -\ 1$ 
by simp

lemma [code]:
   $k + l = \text{int-of-integer } (\text{of-int } k + \text{of-int } l)$ 
by transfer simp

lemma [code]:
   $- k = \text{int-of-integer } (-\ \text{of-int } k)$ 
by transfer simp

lemma [code]:
   $k - l = \text{int-of-integer } (\text{of-int } k - \text{of-int } l)$ 
by transfer simp

lemma [code]:
   $\text{Int.dup } k = \text{int-of-integer } (\text{Code-Numeral.dup } (\text{of-int } k))$ 
by transfer simp

declare [[code drop: Int.sub]]

lemma [code]:
   $k * l = \text{int-of-integer } (\text{of-int } k * \text{of-int } l)$ 
by simp

lemma [code]:
   $k \text{ div } l = \text{int-of-integer } (\text{of-int } k \text{ div } \text{of-int } l)$ 
by simp

lemma [code]:
   $k \text{ mod } l = \text{int-of-integer } (\text{of-int } k \text{ mod } \text{of-int } l)$ 
by simp

lemma [code]:
   $\text{divmod } m\ n = \text{map-prod int-of-integer int-of-integer } (\text{divmod } m\ n)$ 
unfolding prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv
by transfer simp

lemma [code]:
   $\text{HOL.equal } k\ l = \text{HOL.equal } (\text{of-int } k :: \text{integer})\ (\text{of-int } l)$ 
by transfer (simp add: equal)

lemma [code]:
   $k \leq l \iff (\text{of-int } k :: \text{integer}) \leq \text{of-int } l$ 
by transfer rule

lemma [code]:
   $k < l \iff (\text{of-int } k :: \text{integer}) < \text{of-int } l$ 
by transfer rule

declare [[code drop: gcd :: int  $\Rightarrow$  - lcm :: int  $\Rightarrow$  -]]

```

```

lemma gcd-int-of-integer [code]:
  gcd (int-of-integer x) (int-of-integer y) = int-of-integer (gcd x y)
by transfer rule

lemma lcm-int-of-integer [code]:
  lcm (int-of-integer x) (int-of-integer y) = int-of-integer (lcm x y)
by transfer rule

end

lemma (in ring-1) of-int-code-if:
  of-int k = (if k = 0 then 0
    else if k < 0 then - of-int (- k)
    else let
      l = 2 * of-int (k div 2);
      j = k mod 2
      in if j = 0 then l else l + 1)
proof -
  from div-mult-mod-eq have *: of-int k = of-int (k div 2 * 2 + k mod 2) by simp
  show ?thesis
  by (simp add: Let-def of-int-add [symmetric]) (simp add: * mult.commute)
qed

declare of-int-code-if [code]

lemma [code]:
  nat = nat-of-integer ∘ of-int
  including integer.lifting by transfer (simp add: fun-eq-iff)

code-identifier
  code-module Code-Target-Int  $\rightarrow$ 
    (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

```

Avoidance of pattern matching on natural numbers

```

theory Code-Abstract-Nat
imports Main
begin

```

When natural numbers are implemented in another than the conventional inductive $0/Suc$ representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

Case analysis Case analysis on natural numbers is rephrased using a conditional expression:

```

lemma [code, code-unfold]:
  case-nat = (λf g n. if n = 0 then f else g (n - 1))
  by (auto simp add: fun-eq-iff dest!: gr0-implies-Suc)

```

Preprocessors The term $Suc\ n$ is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
assumes $\bigwedge n. f\ (Suc\ n) \equiv h\ n$
assumes $f\ 0 \equiv g$
shows $f\ n \equiv if\ n = 0\ then\ g\ else\ h\ (n - 1)$
by (*rule eq-reflection*) (*cases n, insert assms, simp-all*)

The rule above is built into a preprocessor that is plugged into the code generator.

```

setup (
  let

    val Suc-if-eq = Thm.incr-indexes 1 @ {thm Suc-if-eq};

    fun remove-suc ctxt thms =
      let
        val vname = singleton (Name.variant-list (map fst
          (fold (Term.add-var-names o Thm.full-prop-of) thms [])) n;
        val cv = Thm.ctrm-of ctxt (Var ((vname, 0), HOLogic.natT));
        val lhs-of = snd o Thm.dest-comb o fst o Thm.dest-comb o Thm.cprop-of;
        val rhs-of = snd o Thm.dest-comb o Thm.cprop-of;
        fun find-vars ct = (case Thm.term-of ct of
          (Const (@{const-name Suc}, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
        | - $ - =>
          let val (ct1, ct2) = Thm.dest-comb ct
          in
            map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
            map (apfst (Thm.apply ct1)) (find-vars ct2)
          end
        | - => []);
        val eqs = maps
          (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
        fun mk-thms (thm, (ct, cv')) =
          let
            val thm' =
              Thm.implies-elim
                (Conv.fconv-rule (Thm.beta-conversion true)
                  (Thm.instantiate'
                    [SOME (Thm.ctyp-of-ctrm ct)] [SOME (Thm.lambda cv ct),
                     SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
                    Suc-if-eq)) (Thm.forall-intr cv' thm)
          in
            case map-filter (fn thm'' =>
              SOME (thm'', singleton
                (Variable.trade (K (fn [thm'''] => [thm''' RS thm']))
                  (Variable.declare-thm thm'' ctxt)) thm''))
              of handle THM - => NONE) thms of
              [] => NONE
            | thmps =>

```



```

      let val (thms1, thms2) = split-list thmps
      in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
    end
  in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = @{const-name Suc});
  in
    if forall (can dest) thms andalso exists (contains-suc o dest) thms
    then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
    else NONE
  end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end;
)

end

```

Implementation of natural numbers by target-language integers

```

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

```

```

Implementation for nat context
includes natural.lifting integer.lifting
begin

```

```

lift-definition Nat :: integer  $\Rightarrow$  nat
  is nat
  .

```

```

lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
  by (transfer, simp)+

```

```

lemma [code-abbrev]:
  integer-of-nat = of-nat
  by transfer rule

```

```

lemma [code-unfold]:
  Int.nat (int-of-integer k) = nat-of-integer k

```

by *transfer rule*

lemma [*code abstype*]:
 $\text{Code-Target-Nat.Nat } (\text{integer-of-nat } n) = n$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (\text{nat-of-integer } k) = \max 0 k$
by *transfer auto*

lemma [*code-abbrev*]:
 $\text{nat-of-integer } (\text{numeral } k) = \text{nat-of-num } k$
by *transfer (simp add: nat-of-num-numeral)*

lemma [*code abstract*]:
 $\text{integer-of-nat } (\text{nat-of-num } n) = \text{integer-of-num } n$
by *transfer (simp add: nat-of-num-numeral)*

lemma [*code abstract*]:
 $\text{integer-of-nat } 0 = 0$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } 1 = 1$
by *transfer simp*

lemma [*code*]:
 $\text{Suc } n = n + 1$
by *simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m + n) = \text{of-nat } m + \text{of-nat } n$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m - n) = \max 0 (\text{of-nat } m - \text{of-nat } n)$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m * n) = \text{of-nat } m * \text{of-nat } n$
by *transfer (simp add: of-nat-mult)*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m \text{ div } n) = \text{of-nat } m \text{ div } \text{of-nat } n$
by *transfer (simp add: zdiv-int)*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m \text{ mod } n) = \text{of-nat } m \text{ mod } \text{of-nat } n$
by *transfer (simp add: zmod-int)*

lemma [*code*]:
 $\text{Divides.divmod-nat } m n = (m \text{ div } n, m \text{ mod } n)$

```

by (fact divmod-nat-div-mod)

lemma [code]:
  divmod m n = map-prod nat-of-integer nat-of-integer (divmod m n)
by (simp only: prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv)
    (transfer, simp-all only: nat-div-distrib nat-mod-distrib
      zero-le-numeral nat-numeral)

lemma [code]:
  HOL.equal m n = HOL.equal (of-nat m :: integer) (of-nat n)
by transfer (simp add: equal)

lemma [code]:
  m ≤ n ↔ (of-nat m :: integer) ≤ of-nat n
by simp

lemma [code]:
  m < n ↔ (of-nat m :: integer) < of-nat n
by simp

lemma num-of-nat-code [code]:
  num-of-nat = num-of-integer ∘ of-nat
by transfer (simp add: fun-eq-iff)

end

lemma (in semiring-1) of-nat-code-if:
  of-nat n = (if n = 0 then 0
    else let
      (m, q) = Divides.divmod-nat n 2;
      m' = 2 * of-nat m
      in if q = 0 then m' else m' + 1)

proof -
  from div-mult-mod-eq have *: of-nat n = of-nat (n div 2 * 2 + n mod 2) by simp
  show ?thesis
  by (simp add: Let-def divmod-nat-div-mod of-nat-add [symmetric])
    (simp add: * mult.commute of-nat-mult add.commute)
qed

declare of-nat-code-if [code]

definition int-of-nat :: nat ⇒ int where
  [code-abbrev]: int-of-nat = of-nat

lemma [code]:
  int-of-nat n = int-of-integer (of-nat n)
by (simp add: int-of-nat-def)

lemma [code abstract]:
  integer-of-nat (nat k) = max 0 (integer-of-int k)
including integer.lifting by transfer auto

```

```

lemma term-of-nat-code [code]:
  — Use nat-of-integer in term reconstruction instead of Code-Target-Nat.Nat such that recon-
  structed terms can be fed back to the code generator
  term-of-class.term-of n =
    Code-Evaluation.App
      (Code-Evaluation.Const (STR "Code-Numeral.nat-of-integer")
        (typerep.Typerep (STR "fun")
          [typerep.Typerep (STR "Code-Numeral.integer") [],
            typerep.Typerep (STR "Nat.nat") []]))
        (term-of-class.term-of (integer-of-nat n))
  by (simp add: term-of-anything)

```

```

lemma nat-of-integer-code-post [code-post]:
  nat-of-integer 0 = 0
  nat-of-integer 1 = 1
  nat-of-integer (numeral k) = numeral k
  including integer.lifting by (transfer, simp) +

```

```

code-identifier
  code-module Code-Target-Nat  $\hookrightarrow$ 
    (SML) Arith and (OCaml) Arith and (Haskell) Arith

```

end

Implementation of natural and integer numbers by target-language integers

```

theory Code-Target-Numeral
imports Code-Target-Int Code-Target-Nat
begin

end

```

6. Testing List Properties

This is a reference show-case for HOL-TestGen providing three test-scenarios that were treated from A to Z. This includes:

1. The modeling phase ("building the test-theory") comprising definitions and theorems representing the "background theory" of a particular model to test.
2. The test-specification, the formal statement from which the tests were derived.
3. The abstract test generation phase which basically cuts the input-output relation of the program under test into partitions represented by constraint systems. (since the constraint systems can be unsatisfiable, abstract test cases can be vacuous).
4. The test selection phase that attempts to find concrete test-cases, i. e. ground instances of abstract test cases.
5. The test driver generation phase converts the concrete test-cases into a program that executes these tests; it is linked to a test-harness allowing to track the test evaluation and the program or system under test.
6. The test execution phase (which is currently done outside HOL-TestGen via makefiles).

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, develop test specifications (elsewhere called test targets or test goals), and explore the different possibilities.

```
theory   List-test
imports
  List
  ~~ /src/HOL/Library/Code-Target-Numeral
  Code-Integer-Fsharp
  Testing
begin
```

A Test-theory as a whole starts with the import of its main components, among them the HOL-TestGen environment grouped together in the *Testing*. The theories *Code-Target-Numeral* and *Code-Integer-Fsharp* are required to support the test driver generation process.

A First Model and a Quick Walk Through

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

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

```
primrec is-sorted:: int list  $\Rightarrow$  bool
  where is-sorted [] = True |
        is-sorted (x#xs) = (case xs of
                              []  $\Rightarrow$  True
                              | y#ys  $\Rightarrow$   $x \leq y \wedge$  is-sorted xs)
```

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

```
test-spec is-sorted(PUT l)
oops
```

where *PUT* is a “placeholder” for our program under test. For the sake of the presentation, we drop the test attempt here.

However, for the code-generation necessary to generate a test-driver and actually *run* the test of an external program, the *program under test* or **PUT** for short, it is sensible to represent the latter as an **uninterpreted constant**; the code-generation will later on be configured such that the place-holder in the test-driver code is actually linked to the real, external program which is a black box from the point of view of this model (the testing procedure needs actually only executable code).

```
consts PUT :: 'a list  $\Rightarrow$  'a list
```

Note that the choice of the name is arbitrary.

Generating Abstract 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)
  apply(gen-test-cases 3 1 PUT)
```

which leads to the test partitioning one would expect:

1. *is-sorted* (PUT [])
2. *THYP* (*is-sorted* (PUT []) \longrightarrow *is-sorted* (PUT []))
3. *is-sorted* (PUT [??X8X44])
4. *THYP* (($\exists x$. *is-sorted* (PUT [x])) \longrightarrow ($\forall x$. *is-sorted* (PUT [x])))
5. *is-sorted* (PUT [??X6X38, ??X5X37])
6. *THYP*

$$((\exists x\ xa. \textit{is-sorted} \text{ (PUT [xa, x])}) \longrightarrow (\forall x\ xa. \textit{is-sorted} \text{ (PUT [xa, x])}))$$
7. *is-sorted* (PUT [??X3X30, ??X2X29, ??X1X28])
8. *THYP*

$$((\exists x\ xa\ xb. \textit{is-sorted} \text{ (PUT [xb, xa, x])}) \longrightarrow (\forall x\ xa\ xb. \textit{is-sorted} \text{ (PUT [xb, xa, x])}))$$
9. *THYP* ($3 < \textit{length } l \longrightarrow \textit{is-sorted} \text{ (PUT } l)$)

. Now we bind the test theorem to a particular named *test suite*, a kind of container into which all relevant data is stored and under which a group of tests can be referred to during test execution.

```
mk-test-suite is-sorted-result
```

The current test theorem contains holes, that correspond to the concrete data of the test that have not been generated yet

thm *is-sorted-result.test-thm*

Generating Concrete Test-cases 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.

thm *is-sorted-result.test-thm*

gen-test-data *is-sorted-result*

thm *is-sorted-result.test-thm-inst*

Which leads to the following test data: $\backslash\backslash$ *is-sorted* (*PUT* []) *is-sorted* (*PUT* [10]) *is-sorted* (*PUT* [3, 10]) *is-sorted* (*PUT* [- 8, - 3, - 3])

Note that the underlying test hypothesis remain: $\backslash\backslash$ *THYP* (*is-sorted* (*PUT* [])) \longrightarrow *is-sorted* (*PUT* [])) *THYP* (($\exists x$. *is-sorted* (*PUT* [x])) \longrightarrow ($\forall x$. *is-sorted* (*PUT* [x])))) *THYP* (($\exists x$ *xa*. *is-sorted* (*PUT* [xa, x])) \longrightarrow ($\forall x$ *xa*. *is-sorted* (*PUT* [xa, x])))) *THYP* (($\exists x$ *xa xb*. *is-sorted* (*PUT* [xb, xa, x])) \longrightarrow ($\forall x$ *xa xb*. *is-sorted* (*PUT* [xb, xa, x])))) *THYP* (3 < *length l* \longrightarrow *is-sorted* (*PUT l*))

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

print-conc-tests *is-sorted-result*

print-abs-tests *is-sorted-result*

print-thyps *is-sorted-result*

print-upos *is-sorted-result*

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

export-test-data *impl/data/test-data.data is-sorted-result*

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 [23],
- in C using, e.g. the foreign language interface of sml/NJ [24] or MLton [20],
- in Java using MLj [19].

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.

The the following, we show the general generation of test-scripts (part of the finally generated test-driver) in different languages; finally, we will concentrate on the test-generation scenario for C.

code-printing

```
constant PUT => (Fsharp) ((List.map (fun x -> Int'-of'-integer x)) (myList.sort (List.map  
(fun x -> integer'-of'-int x) ((-)) )))  
      and (SML) ((map (fn x => Int'-of'-integer x)) o myList.sort o (map (fn x =>  
integer'-of'-int x)))  
      and (Scala) ((myList.sort ((-).map {x => integer'-of'-int(x)})).map {x =>  
int'-of'-integer(x)}))
```

generate-test-script *is-sorted-result*

thm *is-sorted-result.test-script*

Testing an SML implementation:

```
export-code is-sorted-result.test-script in SML  
module-name TestScript file impl/sml/is-sorted-test-script.sml
```

We use the SML test script also for testing an implementation written in C:

```
export-code is-sorted-result.test-script in SML  
module-name TestScript file impl/c/is-sorted-test-script.sml
```

Testing an F# implementation:

```
export-code is-sorted-result.test-script in Fsharp  
module-name TestScript file impl/fsharp/is-sorted-test-script.fs
```

We use the F# test script also for testing an implementation written in C#:

```
export-code is-sorted-result.test-script in Fsharp  
module-name TestScript file impl/csharp/is-sorted-test-script.fs
```

Testing a Scala implementation:

```
export-code is-sorted-result.test-script in Scala  
module-name TestScript file impl/scala/is-sorted-test-script.scala
```

We use the Scala script also for testing an implementation written in Java:

```
export-code is-sorted-result.test-script in Scala  
module-name TestScript file impl/java/is-sorted-test-script.scala
```

Finally, we export the raw test data in an XML-like format:

```
export-test-data impl/data/is-sorted-test-data.dat is-sorted-result
```

which generates the following test harness:

In the following, we assume an ANSI C implementation of our sorting method for sorting C arrays that we want to test. (In our example setup, it is contained in the file `impl/c/sort.c`.) Using the foreign language interface provided by the SML compiler MLton we first have to import the sort method written in C using the `_import` keyword of MLton and further, we provide a “wrapper” doing some data-type conversion, e.g. converting lists to arrays and vice versa:

```
structure myList = struct
```

```
val csort = _import "sort": int array * int -> int array;  
fun ArrayToList a = Array.foldl (op ::) [] a;
```



```
>make
mlton -default-ann 'allowFFI true' is_sorted_test.mlb sort.c
./is_sorted_test
```

Test Results:

=====

```
Test 0 -      SUCCESS
Test 1 -      SUCCESS
Test 2 -      SUCCESS
Test 3 -      SUCCESS
Test 4 -      SUCCESS
Test 5 -      SUCCESS
Test 6 -      SUCCESS
```

Summary:

```
Number successful tests cases: 7 of 7 (ca. 100%)
Number of warnings:           0 of 7 (ca. 0%)
Number of errors:             0 of 7 (ca. 0%)
Number of failures:           0 of 7 (ca. 0%)
Number of fatal errors:       0 of 7 (ca. 0%)
```

Overall result: success

=====

Table 6.1.: A Sample Test Trace: The ascending property tested.

```
fun sort_list list = ArrayToList (csort(Array.fromList(list),(length list)));

fun sort list = map IntInf.fromInt (sort_list (map IntInf.toInt list))

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. In `impl/c`, the process of:

1. compiling the generated `impl/c/is_sorted_test_script.sml`, the test harness (`harness.sml`), a main routine `impl/c/List.sml`) and containing a wrapper into an SML structure `myList` as well as the SML-to-C code-stub `sort`,
2. compiling the C test-driver and linking it to the program under test `impl/c/sort.c`, and
3. executing the test is captured in a `impl/c/Makefile`. So: executes the test and displays a test-statistic as shown in Table 6.1.

A Refined Model and Improved Test-Results

Obviously, in reality one would not be satisfied with the test cases generated in the previous 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. What is missing, is a test that input and output sequence are in fact *permutations* of each other. We could state for example :

```

fun   del-member :: 'a ⇒ 'a list ⇒ 'a list option
where del-member x [] = None
      | del-member x (y # S) = (if x = y then Some S
                               else case del-member x S of
                                    None ⇒ None
                                    | Some S' ⇒ Some(y # S'))

fun   is-permutation :: 'a list ⇒ 'a list ⇒ bool
where is-permutation [] [] = True
      | is-permutation (a#S)(a'#S') = (if a = a' then is-permutation S S'
                                         else case del-member a S' of
                                              None ⇒ False
                                              | Some S'' ⇒ is-permutation S (a'#S''))
      | is-permutation _ _ = False

fun is-perm :: 'a list ⇒ 'a list ⇒ bool
where is-perm [] [] = True
      | is-perm [] T = False
      | is-perm (a#S) T = (if length T = length S + 1
                           then is-perm S (remove1 a T)
                           else False)

value is-perm [1,2,3::int] [3,1,2]

```

A test for permutation, that not is hopelessly non-constructive like "the existence of a bijection on the indexes [0 .. n-1], that is pairwise mapped to the list" or the like, is obviously quite complex; the apparent "mathematical specification" is not always the easiest. We convince ourselves that the predicate *is-permutation* indeed captures our intuition by animations of the definition:

```

value is-permutation [1,2,3] [3,2,1::nat]
value ¬ is-permutation [1,2,3] [3,1::nat]
value ¬ is-permutation [2,3] [3,2,1::nat]
value ¬ is-permutation [1,2,1,3] [3,2,1::nat]
value is-permutation [2,1,3] [1::nat,3,2]

value is-perm [1,2,3] [3,2,1::nat]
value ¬ is-perm [1,2,3] [3,1::nat]
value ¬ is-perm [2,3] [3,2,1::nat]
value ¬ is-perm [1,2,1,3] [3,2,1::nat]
value is-perm [2,1,3] [1::nat,3,2]

```

... which are all executable and thus were compiled and all evaluated to true.

Based on these concepts, a test-specification is straight-forward and easy:

```

declare [[goals-limit=5]]
apply(gen-test-cases 5 1 PUT)
mk-test-suite ascending-permutation-test

```

A quick inspection of the test theorem reveals that there are in fact no relevant constraints to solve, so test-data selection is easy:

```

declare [[testgen-iterations=100]]
gen-test-data      ascending-permutation-test

```

```

print-conc-tests    ascending-permutation-test
print-conc-tests (6)ascending-permutation-test
print-thyps        ascending-permutation-test
print-thyps        (0)ascending-permutation-test

```

Again, we convert this into test-scripts that can be compiled to a test-driver.

```

generate-test-script ascending-permutation-test
thm                  ascending-permutation-test.test-script

```

We use the SML implementation also for testing an implementation written in C:

```

export-code          ascending-permutation-test.test-script in SML
module-name TestScript file impl/c/ascending-permutation-test-script.sml

```

Try make `run_ascending_permutation` in directory `impl/c` to compile and execute the generated test-driver.

A Test-Specification based on a Comparison with a Reference Implementation

We might opt for an alternative modeling approach: Thus we decide to try a more “descriptive” test specification that is based on the behavior of an insertion sort algorithm:

```

fun   ins :: ('a::linorder) ⇒ 'a list ⇒ 'a list
where ins x [] = [x]
      | ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys)))
fun   sort :: ('a::linorder) list ⇒ 'a list
where 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 the test case), e.g. we decide to generate lists up to length $(1::'a) + (1::'a) + (1::'a)$. Our standard setup:

```

declare [[goals-limit=100]]
test-spec sort l = PUT l
apply(gen-test-cases PUT)
mk-test-suite is-sorting-algorithm0

```

generates $(1::'a) + (1::'a) + ((1::'a) + (1::'a)) + ((1::'a) + (1::'a) + ((1::'a) + (1::'a))) + (1::'a)$ test cases describing all permutations of lists of length $1::'a, (1::'a) + (1::'a)$ and

$(1::'a) + (1::'a) + (1::'a)$. "Permutation" means here that not only test cases (i.e. I/O-partitions) are generated for lists of length $0::'a$, $1::'a$, $(1::'a) + (1::'a)$ and $(1::'a) + (1::'a) + (1::'a)$; the partitioning is actually finer: for two-elementary lists, for example, the case of a list with the first element larger or equal and the dual case are distinguished. The entire test-theorem looks as follows:

$\llbracket \square = PUT \square \rrbracket; THYP (\square = PUT \square \longrightarrow \square = PUT \square); [\text{??X31X190}] = PUT [\text{??X31X190}];$
 $THYP ((\exists x. [x] = PUT [x]) \longrightarrow (\forall x. [x] = PUT [x])); PO (\text{??X29X182} < \text{??X28X181});$
 $[\text{??X29X182}, \text{??X28X181}] = PUT [\text{??X29X182}, \text{??X28X181}]; THYP ((\exists x xa. xa < x \wedge$
 $[xa, x] = PUT [xa, x]) \longrightarrow (\forall x xa. xa < x \longrightarrow [xa, x] = PUT [xa, x])); PO (\neg \text{??X26X171}$
 $< \text{??X25X170}); [\text{??X25X170}, \text{??X26X171}] = PUT [\text{??X26X171}, \text{??X25X170}]; THYP ((\exists x$
 $xa. \neg xa < x \wedge [x, xa] = PUT [x, xa]) \longrightarrow (\forall x xa. \neg xa < x \longrightarrow [x, xa] = PUT [x, xa])); PO$
 $((\text{??X22X157} < \text{??X21X156} \wedge \text{??X23X158} < \text{??X21X156}) \wedge \text{??X23X158} < \text{??X22X157});$
 $[\text{??X23X158}, \text{??X22X157}, \text{??X21X156}] = PUT [\text{??X23X158}, \text{??X22X157}, \text{??X21X156}];$
 $THYP ((\exists x xa xb. xa < x \wedge xb < x \wedge xb < xa \wedge [xb, xa, x] = PUT [xb, xa, x]) \longrightarrow (\forall x xa$
 $xb. xa < x \longrightarrow xb < x \longrightarrow xb < xa \longrightarrow [xb, xa, x] = PUT [xb, xa, x])); PO ((\neg \text{??X18X140}$
 $< \text{??X17X139} \wedge \text{??X19X141} < \text{??X17X139}) \wedge \text{??X19X141} < \text{??X18X140}); [\text{??X19X141},$
 $\text{??X17X139}, \text{??X18X140}] = PUT [\text{??X19X141}, \text{??X18X140}, \text{??X17X139}]; THYP ((\exists x xa$
 $xb. \neg xa < x \wedge \neg xb < x \wedge \neg xb < xa \wedge [xb, x, xa] = PUT [xb, xa, x]) \longrightarrow (\forall x xa xb. \neg xa$
 $< x \longrightarrow \neg xb < x \longrightarrow \neg xb < xa \longrightarrow [xb, x, xa] = PUT [xb, xa, x])); PO ((\neg \text{??X14X123} <$
 $\text{??X13X122} \wedge \neg \text{??X15X124} < \text{??X13X122}) \wedge \text{??X15X124} < \text{??X14X123}); [\text{??X13X122},$
 $\text{??X15X124}, \text{??X14X123}] = PUT [\text{??X15X124}, \text{??X14X123}, \text{??X13X122}]; THYP ((\exists x xa$
 $xb. \neg xa < x \wedge \neg xb < x \wedge \neg xb < xa \wedge [x, xb, xa] = PUT [xb, xa, x]) \longrightarrow (\forall x xa xb. \neg$
 $xa < x \longrightarrow \neg xb < x \longrightarrow \neg xb < xa \longrightarrow [x, xb, xa] = PUT [xb, xa, x])); PO ((\text{??X10X106}$
 $< \text{??X9X105} \wedge \text{??X11X107} < \text{??X9X105}) \wedge \neg \text{??X11X107} < \text{??X10X106}); [\text{??X10X106},$
 $\text{??X11X107}, \text{??X9X105}] = PUT [\text{??X11X107}, \text{??X10X106}, \text{??X9X105}]; THYP ((\exists x xa$
 $xb. xa < x \wedge xb < x \wedge \neg xb < xa \wedge [xa, xb, x] = PUT [xb, xa, x]) \longrightarrow (\forall x xa xb. xa$
 $< x \longrightarrow xb < x \longrightarrow \neg xb < xa \longrightarrow [xa, xb, x] = PUT [xb, xa, x])); PO ((\text{??X6X89} <$
 $\text{??X5X88} \wedge \neg \text{??X7X90} < \text{??X5X88}) \wedge \neg \text{??X7X90} < \text{??X6X89}); [\text{??X6X89}, \text{??X5X88},$
 $\text{??X7X90}] = PUT [\text{??X7X90}, \text{??X6X89}, \text{??X5X88}]; THYP ((\exists x xa xb. xa < x \wedge \neg xb$
 $< x \wedge \neg xb < xa \wedge [xa, x, xb] = PUT [xb, xa, x]) \longrightarrow (\forall x xa xb. xa < x \longrightarrow \neg xb <$
 $x \longrightarrow \neg xb < xa \longrightarrow [xa, x, xb] = PUT [xb, xa, x])); PO ((\neg \text{??X2X72} < \text{??X1X71} \wedge$
 $\neg \text{??X3X73} < \text{??X1X71}) \wedge \neg \text{??X3X73} < \text{??X2X72}); [\text{??X1X71}, \text{??X2X72}, \text{??X3X73}] =$
 $PUT [\text{??X3X73}, \text{??X2X72}, \text{??X1X71}]; THYP ((\exists x xa xb. \neg xa < x \wedge \neg xb < x \wedge \neg xb <$
 $xa \wedge [x, xa, xb] = PUT [xb, xa, x]) \longrightarrow (\forall x xa xb. \neg xa < x \longrightarrow \neg xb < x \longrightarrow \neg xb <$
 $xa \longrightarrow [x, xa, xb] = PUT [xb, xa, x])); THYP (3 < length l \longrightarrow List-test.sort l = PUT l)$
 $\implies (List-test.sort l = PUT l)$

A more ambitious setting is:

test-spec $sort\ l = PUT\ l$

apply(*gen-test-cases* 5 1 *PUT*)

which leads after 2 seconds to the following test partitioning (excerpt):

1. $\square = PUT \square$
2. $THYP (\square = PUT \square \longrightarrow \square = PUT \square)$
3. $[\text{??X871X8318}] = PUT [\text{??X871X8318}]$
4. $THYP ((\exists x. [x] = PUT [x]) \longrightarrow (\forall x. [x] = PUT [x]))$

5. *PO* ($??X869X8310 < ??X868X8309$)
6. $[??X869X8310, ??X868X8309] = PUT [??X869X8310, ??X868X8309]$
7. *THYP*
 $((\exists x xa. xa < x \wedge [xa, x] = PUT [xa, x]) \longrightarrow$
 $(\forall x xa. xa < x \longrightarrow [xa, x] = PUT [xa, x]))$
8. *PO* ($\neg ??X866X8299 < ??X865X8298$)
9. $[??X865X8298, ??X866X8299] = PUT [??X866X8299, ??X865X8298]$
10. *THYP*
 $((\exists x xa. \neg xa < x \wedge [x, xa] = PUT [xa, x]) \longrightarrow$
 $(\forall x xa. \neg xa < x \longrightarrow [x, xa] = PUT [xa, x]))$
A total of 461 subgoals...

mk-test-suite *permutation-test*

thm *permutation-test.test-thm*

In this scenario, 39 test cases are generated describing all permutations of lists of length 1, 2, 3 and 4. "Permutation" means here that not only test cases (i.e. I/O-partitions) are generated for lists of length 0, 1, 2, 3, 4; the partitioning is actually finer: for two-elementary lists, take one case for the lists with the first element larger or equal.

The case for all lists of depth 5 is feasible, however, it will already take 8 minutes. The resulting constraints for the test cases are complex and require more intensive effort in resolving.

There are several options for the test-data selection. One can either use the (very old) random solver or the more modern smt interface. (One day, we would also have a nitpick-interface to constraint solving via bitblasting sub-models of the constraints to SAT.) The random solver, however, finds only 67 instances out of 150 abstract test cases, while smt instantiates all of them:

Test theorem (gen_test_data) 'permutation_test': 67 test cases in 2.951 seconds

```
declare [[testgen-iterations=0]]
declare [[testgen-SMT]]
gen-test-data      permutation-test

print-conc-tests   permutation-test
print-thyps       permutation-test
```

```
generate-test-script permutation-test
thm                  permutation-test.test-script
```

We use the SML implementation also for testing an implementation written in C:

```
export-code          permutation-test.test-script in SML
module-name TestScript file impl/c/permutation-test-script.sml
```

We obtain test cases like: $\backslash \backslash [] = PUT [] [- 3] = PUT [- 3] [- 1, 0] = PUT [- 1, 0]$
 $[0, 0] = PUT [0, 0] [- 2, - 1, 0] = PUT [- 2, - 1, 0] [0, 1, 1] = PUT [0, 1, 1] [0, 0,$
 $1] = PUT [0, 1, 0] [- 1, - 1, 0] = PUT [- 1, - 1, 0] [- 1, 0, 0] = PUT [0, - 1, 0]$

$$\begin{aligned}
& [0, 0, 0] = PUT [0, 0, 0] [-3, -2, -1, 0] = PUT [-3, -2, -1, 0] [-1, 0, 1, 1] \\
& = PUT [-1, 0, 1, 1] [0, 1, 1, 2] = PUT [0, 1, 2, 1] [0, 0, 1, 2] = PUT [0, 1, 2, 0] [-2, \\
& -2, -1, 0] = PUT [-2, -2, -1, 0] [0, 0, 1, 1] = PUT [0, 0, 1, 1] [0, 1, 1, 2] \\
& = PUT [1, 0, 2, 1] [0, 0, 0, 1] = PUT [0, 0, 1, 0] [-2, -1, -1, 0] = PUT [-2, -1, \\
& -1, 0] [-2, -1, 0, 0] = PUT [-2, 0, -1, 0] [0, 1, 1, 1] = PUT [0, 1, 1, 1] [0, \\
& 0, 1, 1] = PUT [0, 1, 1, 0] [-2, -1, -1, 0] = PUT [-1, -2, -1, 0] [-2, -1, 0, \\
& 0] = PUT [0, -2, -1, 0] [0, 1, 1, 1] = PUT [1, 0, 1, 1] [0, 0, 1, 1] = PUT [1, 0, 1, \\
& 0] [-2, -2, -1, 0] = PUT [-2, -1, -2, 0] [-1, -1, 0, 0] = PUT [-1, 0, -1, \\
& 0] [-1, 0, 0, 1] = PUT [0, 1, -1, 0] [0, 0, 0, 1] = PUT [0, 1, 0, 0] [-1, -1, -1, \\
& 0] = PUT [-1, -1, -1, 0] [-1, -1, 0, 0] = PUT [0, -1, -1, 0] [-1, 0, 0, 0] = \\
& PUT [0, 0, -1, 0] [0, 0, 0, 0] = PUT [0, 0, 0, 0] [-4, -3, -2, -1, 0] = PUT [-4, \\
& -3, -2, -1, 0] [-2, -1, 0, 1, 1] = PUT [-2, -1, 0, 1, 1] [-1, 0, 1, 1, 2] = \\
& PUT [-1, 0, 1, 2, 1] [0, 1, 1, 2, 3] = PUT [0, 1, 2, 3, 1] [0, 0, 1, 2, 3] = PUT [0, \\
& 1, 2, 3, 0] [-3, -3, -2, -1, 0] = PUT [-3, -3, -2, -1, 0] [-1, -1, 0, 1, 1] \\
& = PUT [-1, -1, 0, 1, 1] [0, 0, 1, 1, 2] = PUT [0, 0, 1, 2, 1] [0, 1, 1, 2, 3] = PUT \\
& [1, 0, 2, 3, 1] [0, 0, 0, 1, 2] = PUT [0, 0, 1, 2, 0] [-3, -2, -2, -1, 0] = PUT [-3, \\
& -2, -2, -1, 0] [-1, 0, 0, 1, 1] = PUT [-1, 0, 0, 1, 1] [-1, 0, 1, 1, 2] = PUT \\
& [-1, 1, 0, 2, 1] [0, 1, 1, 1, 2] = PUT [0, 1, 1, 2, 1] [0, 0, 1, 1, 2] = PUT [0, 1, 1, \\
& 2, 0] [-3, -2, -2, -1, 0] = PUT [-2, -3, -2, -1, 0] [-1, 0, 0, 1, 1] = PUT \\
& [0, -1, 0, 1, 1] [-1, 0, 1, 1, 2] = PUT [1, -1, 0, 2, 1] [0, 1, 1, 1, 2] = PUT [1, 0, \\
& 1, 2, 1] [0, 0, 1, 1, 2] = PUT [1, 0, 1, 2, 0] [-3, -3, -2, -1, 0] = PUT [-3, -2, \\
& -3, -1, 0] [0, 0, 1, 2, 2] = PUT [0, 1, 0, 2, 2] [0, 0, 1, 1, 2] = PUT [0, 1, 0, 2, \\
& 1] [0, 1, 1, 2, 3] = PUT [1, 2, 0, 3, 1] [0, 0, 0, 1, 2] = PUT [0, 1, 0, 2, 0] [-2, -2, \\
& -2, -1, 0] = PUT [-2, -2, -2, -1, 0] [0, 0, 0, 1, 1] = PUT [0, 0, 0, 1, 1] [0, \\
& 0, 1, 1, 2] = PUT [1, 0, 0, 2, 1] [0, 1, 1, 1, 2] = PUT [1, 1, 0, 2, 1] [0, 0, 0, 0, 1] = \\
& PUT [0, 0, 0, 1, 0] [-3, -2, -1, -1, 0] = PUT [-3, -2, -1, -1, 0] [-3, -2, \\
& -1, 0, 0] = PUT [-3, -2, 0, -1, 0] [-1, 0, 1, 1, 1] = PUT [-1, 0, 1, 1, 1] [0, \\
& 1, 1, 2, 2] = PUT [0, 1, 2, 2, 1] [0, 0, 1, 2, 2] = PUT [0, 1, 2, 2, 0] [-2, -2, -1, \\
& -1, 0] = PUT [-2, -2, -1, -1, 0] [-2, -2, -1, 0, 0] = PUT [-2, -2, 0, -1, \\
& 0] [0, 0, 1, 1, 1] = PUT [0, 0, 1, 1, 1] [0, 1, 1, 2, 2] = PUT [1, 0, 2, 2, 1] [0, 0, \\
& 0, 1, 1] = PUT [0, 0, 1, 1, 0] [-3, -2, -1, -1, 0] = PUT [-3, -1, -2, -1, 0] \\
& [-3, -2, -1, 0, 0] = PUT [-3, 0, -2, -1, 0] [-1, 0, 1, 1, 1] = PUT [-1, 1, \\
& 0, 1, 1] [0, 1, 1, 2, 2] = PUT [0, 2, 1, 2, 1] [0, 0, 1, 2, 2] = PUT [0, 2, 1, 2, 0] [-2, \\
& -2, -1, -1, 0] = PUT [-2, -1, -2, -1, 0] [-2, -2, -1, 0, 0] = PUT [-2, \\
& 0, -2, -1, 0] [0, 0, 1, 1, 1] = PUT [0, 1, 0, 1, 1] [0, 1, 1, 2, 2] = PUT [1, 2, 0, \\
& 2, 1] [0, 0, 0, 1, 1] = PUT [0, 1, 0, 1, 0] [-3, -2, -2, -1, 0] = PUT [-3, -2, \\
& -1, -2, 0] [-2, -1, -1, 0, 0] = PUT [-2, -1, 0, -1, 0] [-2, -1, 0, 0, 1] = \\
& PUT [-2, 0, 1, -1, 0] [0, 1, 1, 1, 2] = PUT [0, 1, 2, 1, 1] [0, 0, 1, 1, 2] = PUT [0, \\
& 1, 2, 1, 0] [-2, -1, -1, -1, 0] = PUT [-2, -1, -1, -1, 0] [-2, -1, -1, 0, \\
& 0] = PUT [-2, 0, -1, -1, 0] [-2, -1, 0, 0, 0] = PUT [-2, 0, 0, -1, 0] [0, 1, \\
& 1, 1, 1] = PUT [0, 1, 1, 1, 1] [0, 0, 1, 1, 1] = PUT [0, 1, 1, 1, 0] [-3, -2, -1, -1, \\
& 0] = PUT [-1, -3, -2, -1, 0] [-3, -2, -1, 0, 0] = PUT [0, -3, -2, -1, \\
& 0] [-1, 0, 1, 1, 1] = PUT [1, -1, 0, 1, 1] [0, 1, 1, 2, 2] = PUT [2, 0, 1, 2, 1] [0, 0, \\
& 1, 2, 2] = PUT [2, 0, 1, 2, 0] [-2, -2, -1, -1, 0] = PUT [-1, -2, -2, -1, 0] \\
& [-2, -2, -1, 0, 0] = PUT [0, -2, -2, -1, 0] [0, 0, 1, 1, 1] = PUT [1, 0, 0, 1, \\
& 1] [0, 1, 1, 2, 2] = PUT [2, 1, 0, 2, 1] [0, 0, 0, 1, 1] = PUT [1, 0, 0, 1, 0] [-3, -2,
\end{aligned}$$

$-2, -1, 0] = PUT [-2, -3, -1, -2, 0] [-2, -1, -1, 0, 0] = PUT [-1, -2, 0, -1, 0] [-2, -1, 0, 0, 1] = PUT [0, -2, 1, -1, 0] [0, 1, 1, 1, 2] = PUT [1, 0, 2, 1, 1] [0, 0, 1, 1, 2] = PUT [1, 0, 2, 1, 0] [-2, -1, -1, -1, 0] = PUT [-1, -2, -1, -1, 0] [-2, -1, -1, 0, 0] = PUT [0, -2, -1, -1, 0] [-2, -1, 0, 0, 0] = PUT [0, -2, 0, -1, 0] [0, 1, 1, 1, 1] = PUT [1, 0, 1, 1, 1] [0, 0, 1, 1, 1] = PUT [1, 0, 1, 1, 0] [-3, -2, -2, -1, 0] = PUT [-2, -1, -3, -2, 0] [-2, -1, -1, 0, 0] = PUT [-1, 0, -2, -1, 0] [-2, -1, 0, 0, 1] = PUT [0, 1, -2, -1, 0] [0, 1, 1, 1, 2] = PUT [1, 2, 0, 1, 1] [-1, -1, 0, 0, 1] = PUT [0, 1, -1, 0, -1] [-2, -1, -1, -1, 0] = PUT [-1, -1, -2, -1, 0] [-2, -1, -1, 0, 0] = PUT [0, -1, -2, -1, 0] [-2, -1, 0, 0, 0] = PUT [0, 0, -2, -1, 0] [0, 1, 1, 1, 1] = PUT [1, 1, 0, 1, 1] [0, 0, 1, 1, 1] = PUT [1, 1, 0, 1, 0] [-3, -3, -2, -1, 0] = PUT [-3, -2, -1, -3, 0] [-2, -2, -1, 0, 0] = PUT [-2, -1, 0, -2, 0] [-1, -1, 0, 0, 1] = PUT [-1, 0, 1, -1, 0] [-1, 0, 0, 1, 2] = PUT [0, 1, 2, -1, 0] [0, 0, 0, 1, 2] = PUT [0, 1, 2, 0, 0] [-2, -2, -2, -1, 0] = PUT [-2, -2, -1, -2, 0] [-1, -1, -1, 0, 0] = PUT [-1, -1, 0, -1, 0] [-1, -1, 0, 0, 1] = PUT [0, -1, 1, -1, 0] [-1, 0, 0, 0, 1] = PUT [0, 0, 1, -1, 0] [0, 0, 0, 0, 1] = PUT [0, 0, 1, 0, 0] [-2, -2, -1, -1, 0] = PUT [-2, -1, -1, -2, 0] [-2, -2, -1, 0, 0] = PUT [-2, 0, -1, -2, 0] [-1, -1, 0, 0, 0] = PUT [-1, 0, 0, -1, 0] [-1, 0, 0, 1, 1] = PUT [0, 1, 1, -1, 0] [0, 0, 0, 1, 1] = PUT [0, 1, 1, 0, 0] [-2, -2, -1, -1, 0] = PUT [-1, -2, -1, -2, 0] [-2, -2, -1, 0, 0] = PUT [0, -2, -1, -2, 0] [-1, -1, 0, 0, 0] = PUT [0, -1, 0, -1, 0] [-1, 0, 0, 1, 1] = PUT [1, 0, 1, -1, 0] [0, 0, 0, 1, 1] = PUT [1, 0, 1, 0, 0] [-2, -2, -2, -1, 0] = PUT [-2, -1, -2, -2, 0] [-1, -1, -1, 0, 0] = PUT [-1, 0, -1, -1, 0] [-1, -1, 0, 0, 1] = PUT [0, 1, -1, -1, 0] [-1, 0, 0, 0, 1] = PUT [0, 1, 0, -1, 0] [0, 0, 0, 0, 1] = PUT [0, 1, 0, 0, 0] [-1, -1, -1, -1, 0] = PUT [-1, -1, -1, -1, 0] [-1, -1, -1, 0, 0] = PUT [0, -1, -1, -1, 0] [-1, -1, 0, 0, 0] = PUT [0, 0, -1, -1, 0] [-1, 0, 0, 0, 0] = PUT [0, 0, 0, -1, 0] [0, 0, 0, 0, 0] = PUT [0, 0, 0, 0, 0].$

If we scale down to only 10 iterations, this is not sufficient to solve all conditions, i.e. we obtain many test cases with unresolved constraints where *RSF* marks unsolved cases. In these cases, it is unclear if the test partition is empty. 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 better results, we could interactively increase the number of iterations which reveals that we need to set iterations to 100 to find all solutions reliably.

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.

```
> make run_permutation_test
mlton -default-ann 'allowFFI true' permutation_test.mlb sort.c
./permutation_test
```

Test Results:

=====

```
Test 0 -      SUCCESS
Test 1 -      SUCCESS
Test 2 - *** FAILURE: post-condition false
Test 3 - *** FAILURE: post-condition false
Test 4 - *** FAILURE: post-condition false
Test 5 - *** FAILURE: post-condition false
Test 6 - *** FAILURE: post-condition false
Test 7 - *** FAILURE: post-condition false
Test 8 - *** FAILURE: post-condition false
Test 9 - *** FAILURE: post-condition false
Test 10 - *** FAILURE: post-condition false
Test 11 - *** FAILURE: post-condition false
Test 12 - *** FAILURE: post-condition false
Test 13 - *** FAILURE: post-condition false
Test 14 - *** FAILURE: post-condition false
Test 15 - *** FAILURE: post-condition false
Test 16 - *** FAILURE: post-condition false
Test 17 - *** FAILURE: post-condition false
Test 18 - *** FAILURE: post-condition false
Test 19 - *** FAILURE: post-condition false
Test 20 - *** FAILURE: post-condition false
Test 21 - *** FAILURE: post-condition false
Test 22 -      SUCCESS
Test 23 -      SUCCESS
Test 24 - *** FAILURE: post-condition false
Test 25 - *** FAILURE: post-condition false
Test 26 - *** FAILURE: post-condition false
Test 27 - *** FAILURE: post-condition false
Test 28 - *** FAILURE: post-condition false
Test 29 - *** FAILURE: post-condition false
Test 30 - *** FAILURE: post-condition false
Test 31 - *** FAILURE: post-condition false
Test 32 - *** FAILURE: post-condition false
```

Summary:

```
Number successful tests cases: 4 of 33 (ca. 12%)
Number of warnings:           0 of 33 (ca. 0%)
Number of errors:             0 of 33 (ca. 0%)
Number of failures:           29 of 33 (ca. 87%)
Number of fatal errors:       0 of 33 (ca. 0%)
```

Overall result: failed

=====

Table 6.2.: A Sample Test Trace for the Permutation Test Scenario

Running the test (in the current setup: `make run_permutation_test`) against our sample C-program under `impl/c` yields the following result:

Summary A comparison of the three scenarios reveals that albeit a reasonable degree of automation in the test generation process, the essence of model-based test case generation remains an *interactive process* that is worth to be documented in a formal test-plan with respect to various aspects: the concrete modeling that is chosen, the precise formulation of the test-specifications (or: test-goals), the configuration and instrumentation of the test-data selection process, the test-driver synthesis and execution. This process can be complemented by proofs establishing equivalences allowing to convert initial test-specifications into more executable ones, or more 'symbolically evaluatable' ones, or that help to reduce the complexity of the constraint- resolution in the test-data selection process.

But the most important aspect remains: what is a good testing model ? Besides the possibility that the test specification simply does not test what the tester had in mind, the test theory and test-specification have a crucial importance on the quality of the generated test data that seems to be impossible to capture automatically.

Non-Inherent Higher-order Testing

HOL-TestGen can use test specifications that contain higher-order operators — although we would not claim that the test case generation is actually higher-order (there are no enumeration schemes for the function space, so function variables are untreated by the test case generation procedure so far).

Just for fun, we reformulate the problem of finding the maximal number in a list as a higher-order problem:

```
test-spec foldr max l (0::int) = PUT2 l
apply(gen-test-cases PUT2 simp:max-def)
mk-test-suite maximal-number
```

```
declare [[testgen-iterations = 200]]
gen-test-data maximal-number
```

```
print-conc-tests (0) maximal-number
```

```
end
```

6.1. Bank

Implementation of integer numbers by target-language integers

```
theory Code-Target-Int
imports ../GCD
begin
```

```

code-datatype int-of-integer

declare [[code drop: integer-of-int]]

context
includes integer.lifting
begin

lemma [code]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

lemma [code]:
  Int.Pos = int-of-integer ∘ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code]:
  Int.Neg = int-of-integer ∘ uminus ∘ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code-abbrev]:
  int-of-integer (numeral k) = Int.Pos k
  by transfer simp

lemma [code-abbrev]:
  int-of-integer (− numeral k) = Int.Neg k
  by transfer simp

lemma [code, symmetric, code-post]:
  0 = int-of-integer 0
  by transfer simp

lemma [code, symmetric, code-post]:
  1 = int-of-integer 1
  by transfer simp

lemma [code-post]:
  int-of-integer (− 1) = − 1
  by simp

lemma [code]:
  k + l = int-of-integer (of-int k + of-int l)
  by transfer simp

lemma [code]:
  − k = int-of-integer (− of-int k)
  by transfer simp

lemma [code]:
  k − l = int-of-integer (of-int k − of-int l)
  by transfer simp

```

```

lemma [code]:
  Int.dup k = int-of-integer (Code-Numeral.dup (of-int k))
  by transfer simp

declare [[code drop: Int.sub]]

lemma [code]:
  k * l = int-of-integer (of-int k * of-int l)
  by simp

lemma [code]:
  k div l = int-of-integer (of-int k div of-int l)
  by simp

lemma [code]:
  k mod l = int-of-integer (of-int k mod of-int l)
  by simp

lemma [code]:
  divmod m n = map-prod int-of-integer int-of-integer (divmod m n)
  unfolding prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv
  by transfer simp

lemma [code]:
  HOL.equal k l = HOL.equal (of-int k :: integer) (of-int l)
  by transfer (simp add: equal)

lemma [code]:
  k ≤ l ⟷ (of-int k :: integer) ≤ of-int l
  by transfer rule

lemma [code]:
  k < l ⟷ (of-int k :: integer) < of-int l
  by transfer rule

declare [[code drop: gcd :: int ⇒ - lcm :: int ⇒ -]]

lemma gcd-int-of-integer [code]:
  gcd (int-of-integer x) (int-of-integer y) = int-of-integer (gcd x y)
  by transfer rule

lemma lcm-int-of-integer [code]:
  lcm (int-of-integer x) (int-of-integer y) = int-of-integer (lcm x y)
  by transfer rule

end

lemma (in ring-1) of-int-code-if:
  of-int k = (if k = 0 then 0
    else if k < 0 then - of-int (- k)
    else let
      l = 2 * of-int (k div 2);

```

```

      j = k mod 2
    in if j = 0 then l else l + 1)
proof –
  from div-mult-mod-eq have *: of-int k = of-int (k div 2 * 2 + k mod 2) by simp
  show ?thesis
  by (simp add: Let-def of-int-add [symmetric]) (simp add: * mult.commute)
qed

declare of-int-code-if [code]

lemma [code]:
  nat = nat-of-integer ∘ of-int
  including integer.lifting by transfer (simp add: fun-eq-iff)

code-identifier
code-module Code-Target-Int →
  (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

```

Avoidance of pattern matching on natural numbers

```

theory Code-Abstract-Nat
imports Main
begin

```

When natural numbers are implemented in another than the conventional inductive $0/Suc$ representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

Case analysis Case analysis on natural numbers is rephrased using a conditional expression:

```

lemma [code, code-unfold]:
  case-nat = ( $\lambda f g n. \text{if } n = 0 \text{ then } f \text{ else } g (n - 1)$ )
  by (auto simp add: fun-eq-iff dest!: gr0-implies-Suc)

```

Preprocessors The term $Suc\ n$ is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

```

lemma Suc-if-eq:
  assumes  $\bigwedge n. f (Suc\ n) \equiv h\ n$ 
  assumes  $f\ 0 \equiv g$ 
  shows  $f\ n \equiv \text{if } n = 0 \text{ then } g \text{ else } h\ (n - 1)$ 
  by (rule eq-reflection) (cases n, insert assms, simp-all)

```

The rule above is built into a preprocessor that is plugged into the code generator.

```

setup (
  let

```

```

val Suc-if-eq = Thm.incr-indexes 1 @ {thm Suc-if-eq};

fun remove-suc ctxt thms =
  let
    val vname = singleton (Name.variant-list (map fst
      (fold (Term.add-var-names o Thm.full-prop-of) thms []))) n;
    val cv = Thm.ctrm-of ctxt (Var ((vname, 0), HOLogic.natT));
    val lhs-of = snd o Thm.dest-comb o fst o Thm.dest-comb o Thm.cprop-of;
    val rhs-of = snd o Thm.dest-comb o Thm.cprop-of;
    fun find-vars ct = (case Thm.term-of ct of
      (Const (@{const-name Suc}, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
    | - $ - =>
      let val (ct1, ct2) = Thm.dest-comb ct
      in
        map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
        map (apfst (Thm.apply ct1)) (find-vars ct2)
      end
    | - => []);
    val eqs = maps
      (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
    fun mk-thms (thm, (ct, cv')) =
      let
        val thm' =
          Thm.implies-elim
            (Conv.fconv-rule (Thm.beta-conversion true)
              (Thm.instantiate'
                [SOME (Thm.ctyp-of-ctrm ct)] [SOME (Thm.lambda cv ct),
                  SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv]
                Suc-if-eq)) (Thm.forall-intr cv' thm)
        in
          case map-filter (fn thm'' =>
            SOME (thm'', singleton
              (Variable.trade (K (fn [thm'''] => [thm''' RS thm']))
                (Variable.declare-thm thm'' ctxt)) thm''))
            of handle THM - => NONE) thms of
            [] => NONE
          | thmps =>
              let val (thms1, thms2) = split-list thmps
              in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
        end
      in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = @{const-name Suc});
    in
      if forall (can dest) thms andalso exists (contains-suc o dest) thms
      then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
      else NONE
    end;
  end;

```

```
val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;
```

```
in
```

```
  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)
```

```
end;
```

```
)
```

```
end
```

Implementation of natural numbers by target-language integers

```
theory Code-Target-Nat
```

```
imports Code-Abstract-Nat
```

```
begin
```

Implementation for *nat* context

```
includes natural.lifting integer.lifting
```

```
begin
```

lift-definition *Nat* :: *integer* \Rightarrow *nat*

```
is nat
```

```
.
```

lemma [*code-post*]:

```
Nat 0 = 0
```

```
Nat 1 = 1
```

```
Nat (numeral k) = numeral k
```

```
by (transfer, simp)+
```

lemma [*code-abbrev*]:

```
integer-of-nat = of-nat
```

```
by transfer rule
```

lemma [*code-unfold*]:

```
Int.nat (int-of-integer k) = nat-of-integer k
```

```
by transfer rule
```

lemma [*code abstype*]:

```
Code-Target-Nat.Nat (integer-of-nat n) = n
```

```
by transfer simp
```

lemma [*code abstract*]:

```
integer-of-nat (nat-of-integer k) = max 0 k
```

```
by transfer auto
```

lemma [*code-abbrev*]:

```
nat-of-integer (numeral k) = nat-of-num k
```

```
by transfer (simp add: nat-of-num-numeral)
```

lemma [*code abstract*]:
 $\text{integer-of-nat } (\text{nat-of-num } n) = \text{integer-of-num } n$
by *transfer (simp add: nat-of-num-numeral)*

lemma [*code abstract*]:
 $\text{integer-of-nat } 0 = 0$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } 1 = 1$
by *transfer simp*

lemma [*code*]:
 $\text{Suc } n = n + 1$
by *simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m + n) = \text{of-nat } m + \text{of-nat } n$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m - n) = \max 0 (\text{of-nat } m - \text{of-nat } n)$
by *transfer simp*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m * n) = \text{of-nat } m * \text{of-nat } n$
by *transfer (simp add: of-nat-mult)*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m \text{ div } n) = \text{of-nat } m \text{ div } \text{of-nat } n$
by *transfer (simp add: zdiv-int)*

lemma [*code abstract*]:
 $\text{integer-of-nat } (m \text{ mod } n) = \text{of-nat } m \text{ mod } \text{of-nat } n$
by *transfer (simp add: zmod-int)*

lemma [*code*]:
 $\text{Divides.divmod-nat } m \ n = (m \text{ div } n, m \text{ mod } n)$
by *(fact divmod-nat-div-mod)*

lemma [*code*]:
 $\text{divmod } m \ n = \text{map-prod nat-of-integer nat-of-integer } (\text{divmod } m \ n)$
by *(simp only: prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv)*
(transfer, simp-all only: nat-div-distrib nat-mod-distrib zero-le-numeral nat-numeral)

lemma [*code*]:
 $\text{HOL.equal } m \ n = \text{HOL.equal } (\text{of-nat } m :: \text{integer}) (\text{of-nat } n)$
by *transfer (simp add: equal)*

lemma [*code*]:
 $m \leq n \iff (\text{of-nat } m :: \text{integer}) \leq \text{of-nat } n$

```

by simp

lemma [code]:
  m < n  $\longleftrightarrow$  (of-nat m :: integer) < of-nat n
by simp

lemma num-of-nat-code [code]:
  num-of-nat = num-of-integer  $\circ$  of-nat
by transfer (simp add: fun-eq-iff)

end

lemma (in semiring-1) of-nat-code-if:
  of-nat n = (if n = 0 then 0
    else let
      (m, q) = Divides.divmod-nat n 2;
      m' = 2 * of-nat m
      in if q = 0 then m' else m' + 1)
proof -
  from div-mult-mod-eq have *: of-nat n = of-nat (n div 2 * 2 + n mod 2) by simp
  show ?thesis
  by (simp add: Let-def divmod-nat-div-mod of-nat-add [symmetric])
    (simp add: * mult.commute of-nat-mult add.commute)
qed

declare of-nat-code-if [code]

definition int-of-nat :: nat  $\Rightarrow$  int where
  [code-abbrev]: int-of-nat = of-nat

lemma [code]:
  int-of-nat n = int-of-integer (of-nat n)
by (simp add: int-of-nat-def)

lemma [code abstract]:
  integer-of-nat (nat k) = max 0 (integer-of-int k)
including integer.lifting by transfer auto

lemma term-of-nat-code [code]:
  — Use nat-of-integer in term reconstruction instead of Code-Target-Nat.Nat such that reconstructed terms can be fed back to the code generator
  term-of-class.term-of n =
    Code-Evaluation.App
      (Code-Evaluation.Const (STR "Code-Numeral.nat-of-integer"))
      (typerep.Typerep (STR "fun")
        [typerep.Typerep (STR "Code-Numeral.integer" []),
          typerep.Typerep (STR "Nat.nat" [])])
      (term-of-class.term-of (integer-of-nat n))
  by (simp add: term-of-anything)

lemma nat-of-integer-code-post [code-post]:
  nat-of-integer 0 = 0

```



```

    nat-of-integer 1 = 1
    nat-of-integer (numeral k) = numeral k
    including integer.lifting by (transfer, simp)+

code-identifier
  code-module Code-Target-Nat  $\hookrightarrow$ 
    (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

Implementation of natural and integer numbers by target-language integers

theory Code-Target-Numeral
imports Code-Target-Int Code-Target-Nat
begin

end

```


7. A Simple Deterministic Bank Model

```
theory
  Bank
imports
  ~~/src/HOL/Library/Code-Target-Numeral
  Testing
begin
```

The Bank Example: Test of a Distributed Transaction Machine

```
declare [[testgen-profiling]]
```

The intent of this little example is to model deposit, check and withdraw operations of a little Bank model in pre-postcondition style, formalize them in a setup for HOL-TestGen test sequence generation and to generate elementary test cases for it. The test scenarios will be restricted to strict sequence checking; this excludes aspects of account creation which will give the entire model a protocol character (a create-operation would create an account number, and then all later operations are just referring to this number; thus there would be a dependence between system output and input as in reactive sequence test scenarios.).

Moreover, in this scenario, we assume that the system under test is deterministic.

The theory of Proof-based Sequence Test Methodology can be found in [9].

The state of our bank is just modeled by a map from client/account information to the balance.

```
type-synonym client = string
```

```
type-synonym account-no = int
```

```
type-synonym data-base = (client × account-no) → int
```

Operation definitions: Concept A standard, JML or OCL or VCC like interface specification might look like:

```
Init: forall (c,no) : dom(data_base). data_base(c,no)>=0
```

```
op deposit (c : client, no : account_no, amount:nat) : unit
pre (c,no) : dom(data_base)
post data_base'=data_base[(c,no) := data_base(c,no) + amount]
```

```
op balance (c : client, no : account_no) : int
pre (c,no) : dom(data_base)
post data_base'=data_base and result = data_base(c,no)
```

```

op withdraw(c : client, no : account_no, amount:nat) : unit
pre  (c,no) : dom(data_base) and data_base(c,no) >= amount
post data_base'=data_base[(c,no) := data_base(c,no) - amount]

```

Operation definitions: The model as ESFM Interface normalization turns this interface into the following input type:

```

datatype in-c = deposit client account-no nat
              | withdraw client account-no nat
              | balance client account-no

```

```

typ Bank.in-c

```

```

datatype out-c = depositO | balanceO nat | withdrawO

```

```

fun precondition :: data-base ⇒ in-c ⇒ bool
where precondition σ (deposit c no m) = ((c,no) ∈ dom σ)
  | precondition σ (balance c no) = ((c,no) ∈ dom σ)
  | precondition σ (withdraw c no m) = ((c,no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)))

```

```

fun postcondition :: in-c ⇒ data-base ⇒ (out-c × data-base) set
where postcondition (deposit c no m) σ =
  { (n,σ'). (n = depositO ∧ σ'=σ((c,no)↦ the(σ(c,no)) + int m)) }
  | postcondition (balance c no) σ =
  { (n,σ'). (σ=σ' ∧ (∃ x. balanceO x = n ∧ x = nat(the(σ(c,no)))) ) }
  | postcondition (withdraw c no m) σ =
  { (n,σ'). (n = withdrawO ∧ σ'=σ((c,no)↦ the(σ(c,no)) - int m)) }

```

```

definition init :: data-base ⇒ bool
where init σ ≡ ∀ x ∈ dom σ. the(σ x) ≥ 0

```

Constructing an Abstract Program Using the Operators `impl` and `strong_impl`, we can synthesize an abstract program right away from the specification, i.e. the pair of pre- and postcondition defined above. Since this program is even deterministic, we will derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

```

lemma precondition-postcondition-implementable:
  implementable precondition postcondition
apply(auto simp: implementable-def)
apply(case-tac ι, simp-all)
done

```

Based on this input-output specification, we construct the system model as the canonical completion of the (functional) specification consisting of pre- and post-conditions. *Canonical completion* means that the step function explicitly fails (returns *None*) if the precondition fails; this makes it possible to treat sequential execution failures in a uniform way. The system *SYS* can be seen as the step function in an input-output automata or, alternatively, a kind of Mealy machine over symbolic states, or, as an extended finite state machine.

definition $SYS :: in-c \Rightarrow (out-c, data-base) MON_{SE}$
where $SYS = (strong-impl\ precond\ postcond)$

The combinator *strong-impl* turns the pre-post pair in a suitable step functions with the aforementioned characteristics for failing pre-conditions.

Prerequisites

Proving Symbolic Execution Rules for the Abstractly Program The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec):

lemma $Eps-split-eq' : (SOME\ (x', y').\ x' = x \wedge y' = y) = (SOME\ (x', y').\ x = x' \wedge y = y')$
by (*rule arg-cong[of - - Eps], auto*)

deposit

interpretation $deposit : efsm-det$
 $precond\ postcond\ SYS\ (deposit\ c\ no\ m)\ \lambda\sigma.\ depositO$
 $\lambda\sigma.\ \sigma((c, no) \mapsto (the(\sigma(c, no)) + int\ m))\ \lambda\sigma.\ ((c, no) \in dom\ \sigma)$
by *unfold-locales (auto simp: SYS-def Eps-split-eq')*

find-theorems *name:deposit*

withdraw

interpretation $withdraw : efsm-det$
 $precond\ postcond\ SYS\ (withdraw\ c\ no\ m)\ \lambda\sigma.\ withdrawO$
 $\lambda\sigma.\ \sigma((c, no) \mapsto (the(\sigma(c, no)) - int\ m))\ \lambda\sigma.\ ((c, no) \in dom\ \sigma) \wedge (int\ m) \leq the(\sigma(c, no))$
by *unfold-locales (auto simp: SYS-def Eps-split-eq')*

balance

interpretation $balance : efsm-det$
 $precond\ postcond\ SYS\ (balance\ c\ no)\ \lambda\sigma.\ (balanceO\ (nat(the(\sigma(c, no))))$
 $\lambda\sigma.\ \sigma\ \lambda\sigma.\ ((c, no) \in dom\ \sigma)$
by *unfold-locales (auto simp: SYS-def Eps-split-eq')*

Now we close the theory of symbolic execution by *excluding* elementary rewrite steps on $mbind_{FailSave}$, i.e. the rules $mbind_{FailSave}\ []\ ?iostep\ ?\sigma = Some\ ([],\ ?\sigma)\ mbind_{FailSave}\ (?a\ \#?\ S)\ ?iostep\ ?\sigma = (case\ ?iostep\ ?a\ ?\sigma\ of\ None \Rightarrow Some\ ([],\ ?\sigma) \mid Some\ (out,\ \sigma') \Rightarrow case\ mbind_{FailSave}\ ?S\ ?iostep\ \sigma'\ of\ None \Rightarrow Some\ ([out],\ \sigma') \mid Some\ (outs,\ \sigma'') \Rightarrow Some\ (out\ \#outs,\ \sigma''))$

declare $mbind.simps(1)\ [simp\ del]$
 $mbind.simps(2)\ [simp\ del]$

Here comes an interesting detail revealing the power of the approach: The generated sequences still respect the preconditions imposed by the specification - in this case, where we are talking about a client for which a defined account exists and for which we will never produce traces in which we withdraw more money than available on it.

Restricting the Test-Space by Test Purposes We introduce a constraint on the input sequence, in order to limit the test-space a little and eliminate logically possible, but

irrelevant test-sequences for a specific test-purpose. In this case, we narrow down on test-sequences concerning a specific client c with a specific bank-account number no .

We make the (in this case implicit, but as constraint explicitly stated) test hypothesis, that the *SUT* is correct if it behaves correct for a single client. This boils down to the assumption that they are implemented as atomic transactions and interleaved processing does not interfere with a single thread.

```

fun test-purpose :: [client, account-no, in-c list]  $\Rightarrow$  bool
where
  test-purpose c no [balance c' no'] = (c=c'  $\wedge$  no=no')
| test-purpose c no ((deposit c' no' m)#R) = (c=c'  $\wedge$  no=no'  $\wedge$  test-purpose c no R)
| test-purpose c no ((withdraw c' no' m)#R) = (c=c'  $\wedge$  no=no'  $\wedge$  test-purpose c no R)
| test-purpose c no - = False

```

```

lemma [simp] : test-purpose c no [a] = (a = balance c no)
by(cases a, auto)

```

```

lemma [simp] : R  $\neq [] \implies$  test-purpose c no (a#R) =
  ((( $\exists m. a = (\text{deposit } c \text{ no } m)$ )  $\vee$  ( $\exists m. a = (\text{withdraw } c \text{ no } m)$ )))
   $\wedge$  test-purpose c no R

```

```

apply(simp add: List.neq-Nil-conv, elim exE,simp)
by(cases a, auto)

```

The TestGen Setup The default configuration of `gen_test_cases` does *not* descend into sub-type expressions of type constructors (since this is not always desirable, the choice for the default had been for "non-descent"). This case is relevant here since *in-c list* has just this structure but we need ways to explore the input sequence type further. Thus, we need configure, for all test cases, and derivation descendants of the relisting clauses during splitting, again splitting for all parameters of input type *in-c*:

Preparation: Miscellaneous We construct test-sequences for a concrete client (implicitly assuming that interleaving actions with other clients will not influence the system behaviour. In order to prevent HOL-TestGen to perform case-splits over names, i. e., list of characters—we define it as constant.

```

definition c0 :: string where c0 = "meyer"

```

```

consts PUT :: (in-c  $\Rightarrow$  (out-c, 'σ) MONSE)

```

```

lemma HH : (A  $\wedge$  (A  $\longrightarrow$  B)) = (A  $\wedge$  B) by auto

```

Small, rewriting based Scenarios including standard code-generation

Exists in two formats : General Fail-Safe Tests (which allows for scenarios with normal *and* exceptional behaviour; and Fail-Stop Tests, which generates Tests only for normal behaviour and correspond to inclusion test refinement.

In the following, we discuss a test-scenario with failsave error semantics; i. e. in each test-case, a sequence may be chosen (by the test data selection) where the client has sev-

eral accounts. In other words, tests were generated for both *standard* and *exceptional behaviour*. The splitting technique is general exploration of the type *in-c list*.

test-spec *test-balance*:

```
(c0,no) ∈ dom σ0 ⇒
init σ0 ⇒
test-purpose c0 no S ⇒
σ0 ⊨ (s ← mbindFailSave S SYS; return (s = x)) ⇒

σ0 ⊨ (s ← mbindFailSave S PUT; return (s = x))
```

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking *x* from being case-split (which complicates the process).

```
apply(erule rev-mp)+
apply(rule-tac x=x in spec[OF allI])
```

Starting the test generation process.

```
apply(gen-test-cases 5 1 PUT)
```

```
apply(simp-all add: init-def HH split: HOL.if-split-asm)
```

```
mk-test-suite bank-simpleSNXB
thm bank-simpleSNXB.test-thm
```

And now the Fail-Stop scenario — this corresponds exactly to inclusion tests for normal-behaviour tests: any transition in the model is only possible iff the pre-conditions of the transitions in the model were respected.

```
declare Monads.mbind'-bind [simp del]
```

test-spec *test-balance2*:

```
(c0,no) ∈ dom σ0 ⇒
init σ0 ⇒
test-purpose c0 no S ⇒
σ0 ⊨ (s ← mbindFailStop S SYS; return (s = x)) ⇒
σ0 ⊨ (s ← mbindFailStop S PUT; return (s = x))
```

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking *x* from being case-split (which complicates the process).

```
apply(erule rev-mp)+
apply(rule-tac x=x in spec[OF allI])
```

Starting the test generation process - variant without uniformity generation.

```
using[[no-uniformity]]
apply(gen-test-cases 4 1 PUT)
```

So lets go for a more non-destructive approach:

```
using[[goals-limit=20]]
apply(simp-all add: init-def HH split: HOL.if-split-asm)
```

```
using[[no-uniformity=false]]
apply(tactic TestGen.ALLCASES(TestGen.uniformityI-tac @{context} [PUT]))
mk-test-suite bank-simpleNB
thm bank-simpleNB.test-thm
```

Test-Data Generation

Configuration

```
declare [[testgen-iterations=0]]
```

```
declare [[testgen-SMT]]
```

```
declare c0-def [testgen-smt-facts]  
declare mem-Collect-eq [testgen-smt-facts]  
declare Collect-mem-eq [testgen-smt-facts]  
declare dom-def [testgen-smt-facts]  
declare Option.option.sel [testgen-smt-facts]
```

Test Data Selection for the Normal and Exceptional Behaviour Test Scenario

```
gen-test-data bank-simpleSNXB  
thm bank-simpleSNXB.test-thm  
thm bank-simpleSNXB.test-thm-inst  
print-conc-tests bank-simpleSNXB
```

Test Data Selection for the Normal Behaviour Test Scenario

```
declare [[testgen-iterations=0]]  
declare [[testgen-SMT]]  
gen-test-data bank-simpleNB  
  
print-conc-tests bank-simpleNB  
thm bank-simpleNB.test-thm-inst
```

Generating the Test-Driver for an SML and C implementation

The generation of the test-driver is non-trivial in this exercise since it is essentially two-staged: Firstly, we chose to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. Recall that a test-driver consists of four components:

- `../../../../../../../../harness/sml/main.sml` the global controller (a fixed element in the library),
- `../../../../../../../../harness/sml/main.sml` a statistic evaluation library (a fixed element in the library),
- `bank_simple_test_script.sml` the test-script that corresponds merely one-to-one to the generated test-data (generated)
- `bank_adapter.sml` a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign-language interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code

under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing 'bit-string-representation-behind-the-scene' enabling relatively efficient symbolic computation. Both representations can be compiled "natively" to data types in the SML level. By an appropriate configuration, the code-generator can map "int" of HOL to three different implementations: the SML standard library `Int.int`, the native-C interfaced by `Int32.int`, and the `IntInf.int` from the multi-precision library `gmp` underneath the `polymml-compiler`.

We do a three-step compilation of data-representations model-to-model, model-to-SML, SML-to-C.

A basic preparatory step for the initializing the test-environment to enable code-generation is:

```
generate-test-script bank-simpleSNXB
thm                  bank-simpleSNXB.test-script
```

```
generate-test-script bank-simpleNB
thm                  bank-simpleNB.test-script
```

In the following, we describe the interface of the SML-program under test, which is in our scenario an *adapter* to the C code under test. This is the heart of the model-to-SML translation. The the SML-level stubs for the program under test are declared as follows:

```
consts    balance-stub :: string  $\Rightarrow$  int  $\Rightarrow$  (int, 'σ)MONSE
code-printing
  constant balance-stub => (SML) (fn n => fn i => fn s => (case (BankAdapter.balance n
(integer'-of'-int i) s) of (SOME(i',s')) => SOME(Int'-of'-integer i',s'))))

consts    deposit-stub :: string  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  (unit, 'σ)MONSE
code-printing
  constant deposit-stub => (SML) (fn s => fn i => fn j => BankAdapter.deposit s
(integer'-of'-int i) (integer'-of'-int j))

consts    withdraw-stub:: string  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  (unit, 'σ)MONSE
code-printing
  constant withdraw-stub => (SML) (fn s => fn i => fn j => BankAdapter.withdraw s
(integer'-of'-int i) (integer'-of'-int j))
```

Note that this translation step prepares already the data-adaption; the type `nat` is seen as an predicative constraint on integer (which is actually not tested). On the model-to-model level, we provide a global step function that distributes to individual interface functions via stubs (mapped via the code generation to SML ...). This translation also represents uniformly `nat` by `int`'s.

```
fun    stepAdapter :: (in-c  $\Rightarrow$  (out-c, 'σ)MONSE)
where
  stepAdapter(balance name no) =
    (x  $\leftarrow$  balance-stub name no; return(balanceO (Int.nat x)))
  | stepAdapter(deposit name no amount) =
    (-  $\leftarrow$  deposit-stub name no (int amount); return(depositO))
  | stepAdapter(withdraw name no amount)=
```

$(- \leftarrow \text{withdraw-stub name no (int amount)}; \text{return}(\text{withdrawO}))$

The *stepAdapter* function links the HOL-world and establishes the logical link to HOL stubs which were mapped by the code-generator to adapter functions in SML (which call internally to C-code inside `bank_adapter.sml` via a foreign language interface)

... We configure the code-generator to identify the PUT with the generated SML code implicitly defined by the above *stepAdapter* definition.

code-printing

constant *PUT* => (SML) *stepAdapter*

And there we go and generate the `bank_simple_test_script.sml`:

export-code *stepAdapter bank-simpleSNXB.test-script* **in** SML
module-name *TestScript* **file** *impl/c/bank-simpleSNXB-test-script.sml*

export-code *stepAdapter bank-simpleNB.test-script* **in** SML
module-name *TestScript* **file** *impl/c/bank-simpleNB-test-script.sml*

More advanced Test-Case Generation Scenarios

Exploring a bit the limits ...

Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec *test-balance*:

assumes *account-def* : $(c_0, no) \in \text{dom } \sigma_0$

and *accounts-pos* : *init* σ_0

and *test-purpose* : *test-purpose* c_0 *no* *S*

and *sym-exec-spec* :

$\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSave}} S \text{ SYS}; \text{return } (s = x))$

shows $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSave}} S \text{ PUT}; \text{return } (s = x))$

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking *x* from being case-split (which complicates the process).

apply(*insert account-def test-purpose sym-exec-spec*)

apply(*tactic TestGen.mp-fy @{context} 1, rule-tac x=x in spec[OF allI]*)

Starting the test generation process.

apply(*gen-test-cases 5 1 PUT*)

Symbolic Execution:

apply(*simp-all add: HH split: HOL.if-split-asm*)

mk-test-suite *bank-large*

gen-test-data *bank-large*

print-conc-tests *bank-large*

Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec *test-balance*:

assumes *account-def* : $(c_0, no) \in \text{dom } \sigma_0$

and *accounts-pos* : *init* σ_0

and *test-purpose* : *test-purpose* c_0 *no* *S*

and *sym-exec-spec* :
 $\sigma_0 \models (s \leftarrow mbind_{FailStop} S SYS; \text{return } (s = x))$
shows $\sigma_0 \models (s \leftarrow mbind_{FailStop} S PUT; \text{return } (s = x))$

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-split (which complicates the process).

apply(*insert account-def test-purpose sym-exec-spec*)
apply(*tactic TestGen.mp-fy* @{*context*} 1, *rule-tac x=x* **in** *spec[OF allI]*)

Starting the test generation process.

apply(*gen-test-cases 3 1 PUT*)

Symbolic Execution:

apply(*simp-all add: HH split: HOL.if-split-asm*)

mk-test-suite *bank-large'*

gen-test-data *bank-large'*

print-conc-tests *bank-large'*

And now, to compare, elimination based procedures ...

declare *deposit.exec-mbindFSave-If*[*simp del*]
declare *balance.exec-mbindFSave-If*[*simp del*]
declare *withdraw.exec-mbindFSave-If*[*simp del*]
declare *deposit.exec-mbindFStop* [*simp del*]
declare *balance.exec-mbindFStop*[*simp del*]
declare *withdraw.exec-mbindFStop*[*simp del*]

thm *deposit.exec-mbindFSave-E withdraw.exec-mbindFSave-E balance.exec-mbindFSave-E*

test-spec *test-balance*:
assumes *account-defined*: $(c_0, no) \in \text{dom } \sigma_0$
and *accounts-pos* : *init* σ_0
and *test-purpose* : *test-purpose* c_0 *no* S
and *sym-exec-spec* :
 $\sigma_0 \models (s \leftarrow mbind_{FailStop} S SYS; \text{return } (s = x))$
shows $\sigma_0 \models (s \leftarrow mbind_{FailStop} S PUT; \text{return } (s = x))$
apply(*insert account-defined test-purpose sym-exec-spec*)
apply(*tactic TestGen.mp-fy* @{*context*} 1, *rule-tac x=x* **in** *spec[OF allI]*)
using [[*no-uniformity*]]
apply(*gen-test-cases*
3 1 PUT)

apply(*tactic* *ALLGOALS*(*TestGen.REPEAT'*(*ematch-tac* @{*context*} [@{*thm* *balance.exec-mbindFStop-E*},

@{thm withdraw.exec-mbindFStop-E},
@{thm deposit.exec-mbindFStop-E},
@{thm valid-mbind'-mt}

)))))
apply(*simp-all*)

```

using[[no-uniformity=false]]
apply(tactic TestGen.ALLCASES(TestGen.uniformityI-tac @{context} [PUT]))
mk-test-suite bank-large-very

```

Yet another technique: "deep" symbolic execution rules involving knowledge from the model domain. Here: input alphabet must be case-split over deposit, withdraw and balance. This avoids that `gen_test_cases` has to do deep splitting.

```

theorem hulk :
assumes redex :  $\sigma \models (s \leftarrow (\text{mbind}_{\text{FailStop}} (a \# S) \text{SYS}); \text{return } (P \ s))$ 
and case-deposit :  $\bigwedge c \text{ no } m. a = \text{deposit } c \text{ no } m \implies (c, \text{no}) \in \text{dom } \sigma \implies$ 
 $\sigma((c, \text{no}) \mapsto \text{the } (\sigma(c, \text{no})) + \text{int } m) \models$ 
 $(s \leftarrow \text{mbind}_{\text{FailStop}} S \text{SYS}; \text{return } P (\text{depositO } \# s)) \implies$ 
 $Q$ 
and case-withdraw :  $\bigwedge c \text{ no } m. a = \text{withdraw } c \text{ no } m \implies (c, \text{no}) \in \text{dom } \sigma \implies$ 
 $\text{int } m \leq \text{the } (\sigma(c, \text{no})) \implies$ 
 $\sigma((c, \text{no}) \mapsto \text{the } (\sigma(c, \text{no})) - \text{int } m) \models$ 
 $(s \leftarrow \text{mbind}_{\text{FailStop}} S \text{SYS}; \text{return } P (\text{withdrawO } \# s)) \implies$ 
 $Q$ 
and case-balance :  $\bigwedge c \text{ no}. (c, \text{no}) \in \text{dom } \sigma \implies$ 
 $\sigma \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{SYS};$ 
 $\text{return } P (\text{balanceO } (\text{nat } (\text{the } (\sigma(c, \text{no})))) \# s)) \implies$ 
 $Q$ 
shows  $Q$ 
proof(cases a) print-cases
  case (deposit c no m) assume hyp :  $a = \text{deposit } c \text{ no } m$  show  $Q$ 
    using hyp redex
    apply(simp only: deposit.exec-mbindFStop)
    apply(rule case-deposit, auto)
    done

  next
    case (withdraw c no m) assume hyp :  $a = \text{withdraw } c \text{ no } m$  show  $Q$ 
      using hyp redex
      apply(simp only: withdraw.exec-mbindFStop)
      apply(rule case-withdraw, auto)
      done

  next
    case (balance c no) assume hyp :  $a = \text{balance } c \text{ no}$  show  $Q$ 
      using hyp redex
      apply(simp only: balance.exec-mbindFStop)
      apply(rule case-balance, auto)
      done
qed

```

Experimental Space

```

declare[[testgen-trace]]
ML⟨⟨ prune-params-tac; Drule.triv-forall-equality ⟩⟩

```

end

8. A Simple Non-Deterministic Bank Model

```
theory
  NonDetBank
imports
  Testing
begin
```

```
declare [[testgen-profiling]]
```

This testing scenario is a modification of the Bank example. The purpose is to explore specifications which are nondeterministic, but at least σ -deterministic, i.e. from the observable output, the internal state can be constructed (which paves the way for symbolic executions based on the specification).

The state of our bank is just modeled by a map from client/account information to the balance.

```
type-synonym client = string
```

```
type-synonym account-no = int
```

```
type-synonym register = (client  $\times$  account-no)  $\rightarrow$  int
```

Operation definitions We use a similar setting as for the Bank example — with one minor modification: the withdraw operation gets a non-deterministic behaviour: it may withdraw any amount between 1 and the demanded amount.

```
op deposit (c : client, no : account_no, amount:nat) : unit
pre (c,no) : dom(register)
post register'=register[(c,no) := register(c,no) + amount]

op balance (c : client, no : account_no) : int
pre (c,no) : dom(register)
post register'=register and result = register(c,no)

op withdraw(c : client, no : account_no, amount:nat) : nat
pre (c,no) : dom(register) and register(c,no) >= amount
post result <= amount and
  register'=register[(c,no) := register(c,no) - result]
```

Interface normalization turns this interface into the following input type:

```
datatype in-c = deposit client account-no nat
| withdraw client account-no nat
```

| *balance client account-no*

datatype *out-c* = *depositO* | *balanceO nat* | *withdrawO nat*

fun *precond* :: *register* \Rightarrow *in-c* \Rightarrow *bool*
where *precond* σ (*deposit c no m*) = $((c, no) \in \text{dom } \sigma)$
| *precond* σ (*balance c no*) = $((c, no) \in \text{dom } \sigma)$
| *precond* σ (*withdraw c no m*) = $((c, no) \in \text{dom } \sigma \wedge (\text{int } m) \leq \text{the}(\sigma(c, no)))$

fun *postcond* :: *in-c* \Rightarrow *register* \Rightarrow (*out-c* \times *register*) *set*
where *postcond* (*deposit c no m*) σ =
 $(\{ (n, \sigma'). (n = \text{depositO} \wedge \sigma' = \sigma((c, no) \mapsto \text{the}(\sigma(c, no)) + \text{int } m)) \})$
| *postcond* (*balance c no*) σ =
 $(\{ (n, \sigma'). (\sigma = \sigma' \wedge (\exists x. \text{balanceO } x = n \wedge x = \text{nat}(\text{the}(\sigma(c, no)))) \})$
| *postcond* (*withdraw c no m*) σ =
 $(\{ (n, \sigma'). (\exists x \leq m. n = \text{withdrawO } x \wedge \sigma' = \sigma((c, no) \mapsto \text{the}(\sigma(c, no)) - \text{int } x)) \})$

Proving Symbolic Execution Rules for the Abstractly Constructed Program Using the Operators **impl** and **strong_impl**, we can synthesize an abstract program right away from the specification, i.e. the pair of pre and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

definition *implementable* :: [$\sigma \Rightarrow \iota \Rightarrow \text{bool}, \iota \Rightarrow (\iota, \sigma) \text{MON}_{SB} \Rightarrow \text{bool}$]
where *implementable pre post* = $(\forall \sigma \iota. \text{pre } \sigma \iota \longrightarrow (\exists \text{out } \sigma'. (\text{out}, \sigma') \in \text{post } \iota \sigma))$

lemma *precond-postcond-implementable*:
implementable precond postcond
apply(*auto simp: implementable-def*)
apply(*case-tac* ι , *simp-all*)
apply *auto*
done

The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec)

lemma *impl-1*:
strong-impl precond postcond (deposit c no m) =
 $(\lambda \sigma. \text{if } (c, no) \in \text{dom } \sigma$
 $\text{then Some}(\text{depositO}, \sigma((c, no) \mapsto \text{the } (\sigma(c, no)) + \text{int } m))$
 $\text{else None})$
by(*rule ext, auto simp: strong-impl-def*)

lemma *valid-both-spec1* [*simp*]:
 $(\sigma \models (s \leftarrow \text{mbind } ((\text{deposit } c \text{ no } m) \# S) (\text{strong-impl precond postcond});$
 $\text{return } (P s))) =$
 $(\text{if } (c, no) \in \text{dom } \sigma$
 $\text{then } (\sigma((c, no) \mapsto \text{the } (\sigma(c, no)) + \text{int } m)) \models (s \leftarrow \text{mbind } S (\text{strong-impl precond postcond});$

```

      return (P (depositO#s)))
    else (σ ⊨ (return (P [])))
  by(auto simp: exec-mbindFSave impl-1)

```

lemma impl-2:
strong-impl precondition postcondition (balance c no) =
(λσ. if (c, no) ∈ dom σ
then Some(balanceO(nat(the (σ (c, no)))))σ
else None)
 by(rule ext, auto simp: strong-impl-def)

lemma valid-both-spec2 [simp]:
 (σ ⊨ (s ← mbind ((balance c no)#S) (strong-impl precondition postcondition);
 return (P s))) =
 (if (c, no) ∈ dom σ
 then (σ ⊨ (s ← mbind S (strong-impl precondition postcondition);
 return (P (balanceO(nat(the (σ (c, no)))))#s)))
 else (σ ⊨ (return (P []))))
 by(auto simp: exec-mbindFSave impl-2)

So far, no problem; however, so far, everything was deterministic. The following key-theorem does not hold:

lemma impl-3:
strong-impl precondition postcondition (withdraw c no m) =
(λσ. if (c, no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)) ∧ x ≤ m
then Some(withdrawO x,σ((c, no) ↦ the (σ (c, no)) - int x))
else None)

oops

This also breaks our deterministic approach to compute the sequence beforehand and to run the test of PUT against this sequence.

However, we can give an acceptance predicate (an automaton) for correct behaviour of our PUT:

```

fun   accept :: (in-c list × out-c list × int) ⇒ bool
where accept((deposit c no n)#S, depositO#S', m) = accept (S, S', m + (int n))
      | accept((withdraw c no n)#S, (withdrawO k)#S', m) = (k ≤ n ∧ accept (S, S', m - (int k)))
      | accept([balance c no], [balanceO n], m) = (int n = m)
      | accept(a, b, c) = False

```

This format has the advantage

TODO: Work out foundation. accept works on an abstract state (just one single balance of a user), while PUT works on the (invisible) concrete state. A data-refinement is involved, and it has to be established why it is correct.

Test Specifications **fun** test-purpose :: [client, account-no, in-c list] ⇒ bool
where
 test-purpose c no [] = False
 | test-purpose c no (a#R) = (case R of
 [] ⇒ a = balance c no

$$\begin{aligned} | a' \# R' \Rightarrow & (((\exists m. a = \text{deposit } c \text{ no } m) \vee \\ & (\exists m. a = \text{withdraw } c \text{ no } m)) \wedge \\ & \text{test-purpose } c \text{ no } R)) \end{aligned}$$

```

test-spec test-balance:
assumes account-defined:  $(c, no) \in \text{dom } \sigma_0$ 
and test-purpose : test-purpose  $c \text{ no } \iota s$ 
shows  $\sigma_0 \models (os \leftarrow \text{mbind } \iota s \text{ PUT}; \text{return } (\text{accept}(\iota s, os, \text{the}(\sigma_0 (c, no))))))$ 
apply(insert account-defined test-purpose)
apply(gen-test-cases PUT split: HOL.if-split-asm)

```

```

mk-test-suite nbank

```

```

declare [[testgen-iterations=0]]
gen-test-data nbank

```

```

end

```

8.1. MyKeOS

9. The MyKeOS Case Study

```
theory MyKeOS
imports
  Testing
begin
```

This example is drawn from the operating system testing domain; it is a rough abstraction of PiKeOS and explains the underlying techniques of this particular case study on a small example. The full paper can be found under [5].

This is a fun-operating system — closely following the Bank example — intended to explain the principles of symbolic execution used in our PikeOS study.

Moreover, in this scenario, we assume that the system under test is deterministic.

The state of a thread (belonging to a task, i.e. a Unix/PosiX like “process” just modeled by a map from task-id/thread-id information to the number of a resource (a communication channel descriptor, for example) that was allocated to a thread.

```
type-synonym task-id = int
```

```
type-synonym thread-id = int
```

```
type-synonym thread-local-var-tab = thread-id  $\rightarrow$  int
```

Operation definitions A standard, JML or OCL or VCC like interface specification might look like:

```
Init: forall (c,no) : dom(data_base::thread_local_var_tab). data_base(c,no)>=0
```

```
op alloc (c : task_id, no : thread_id, amount:nat) : unit
pre (c,no) : dom(data_base)
post data_base'=data_base[(c,no) := data_base(c,no) + amount]
```

```
op release(c : task_id, no : thread_id, amount:nat) : unit
pre (c,no) : dom(data_base) and data_base(c,no) >= amount
post data_base'=data_base[(c,no) := data_base(c,no) - amount]
```

```
op status (c : task_id, no : thread_id) : int
pre (c,no) : dom(data_base)
post data_base'=data_base and result = data_base(c,no)
```

Interface normalization turns this interface into the following input type:

```
datatype in-c = send   thread-id thread-id nat
                | rec    thread-id thread-id nat
                | status thread-id
```

```
datatype infc = sendfc thread-id thread-id nat
                | recfc   thread-id thread-id nat
                | allocfc thread-id nat
                | releasefc thread-id nat
                | statfc   thread-id
```

```
typ MyKeOS.in-c
```

```
datatype out-c = send-ok | rec-ok | status-ok nat
```

```
datatype outfc = send-okfc | rec-okfc | alloc-okfc | release-okfc | stat-okfc nat
```

```
fun   precond :: thread-local-var-tab  $\Rightarrow$  in-c  $\Rightarrow$  bool
```

```
where precond  $\sigma$  (send tid tid' m) = (tid  $\in$  dom  $\sigma$   $\wedge$  tid'  $\in$  dom  $\sigma$   $\wedge$  (int m)  $\leq$  the( $\sigma$  tid))
      | precond  $\sigma$  (rec tid tid' m) = (tid  $\in$  dom  $\sigma$   $\wedge$  tid'  $\in$  dom  $\sigma$ )
      | precond  $\sigma$  (status tid) = (tid  $\in$  dom  $\sigma$ )
```

```
fun   precondfc :: thread-local-var-tab  $\Rightarrow$  infc  $\Rightarrow$  bool
```

```
where precondfc  $\sigma$  (sendfc tid tid' m) = (tid  $\in$  dom  $\sigma$   $\wedge$  tid'  $\in$  dom  $\sigma$   $\wedge$  (int m)  $\leq$  the( $\sigma$  tid))
      | precondfc  $\sigma$  (recfc tid tid' m) = (tid  $\in$  dom  $\sigma$   $\wedge$  tid'  $\in$  dom  $\sigma$ )
      | precondfc  $\sigma$  (allocfc tid m) = (tid  $\in$  dom  $\sigma$ )
      | precondfc  $\sigma$  (releasefc tid m) = (tid  $\in$  dom  $\sigma$ )
      | precondfc  $\sigma$  (statfc tid) = (tid  $\in$  dom  $\sigma$ )
```

```
fun   postcond :: in-c  $\Rightarrow$  thread-local-var-tab  $\Rightarrow$  (out-c  $\times$  thread-local-var-tab) set
```

```
where postcond (send tid tid' m)  $\sigma$  =
      { (n,  $\sigma'$ ). n = send-ok  $\wedge$   $\sigma' = \sigma$  (tid  $\mapsto$  the( $\sigma$  tid) - int m)}
  | postcond (rec tid tid' m)  $\sigma$  =
      { (n,  $\sigma'$ ). (n = rec-ok  $\wedge$   $\sigma' = \sigma$  (tid'  $\mapsto$  the( $\sigma$  tid') + int m))}
  | postcond (status tid)  $\sigma$  =
      { (n,  $\sigma'$ ). ( $\sigma = \sigma' \wedge (\exists x. \text{status-ok } x = n \wedge x = \text{nat}(\text{the}(\sigma \text{ tid})))$ )}
```

```
fun   postcondfc :: infc  $\Rightarrow$  thread-local-var-tab  $\Rightarrow$  (outfc  $\times$  thread-local-var-tab) set
```

```
where postcondfc (sendfc tid tid' m)  $\sigma$  =
      { (n,  $\sigma'$ ). n = send-okfc  $\wedge$   $\sigma' = \sigma$  (tid  $\mapsto$  the( $\sigma$  tid) - int m)}
  | postcondfc (recfc tid tid' m)  $\sigma$  =
      { (n,  $\sigma'$ ). (n = rec-okfc  $\wedge$   $\sigma' = \sigma$  (tid'  $\mapsto$  the( $\sigma$  tid') + int m))}
  | postcondfc (statfc tid)  $\sigma$  =
      { (n,  $\sigma'$ ). ( $\sigma = \sigma' \wedge (\exists x. \text{stat-ok}_{fc} \ x = n \wedge x = \text{nat}(\text{the}(\sigma \text{ tid})))$ )}
```

Constructing an Abstract Program Using the Operators `impl` and `strong_impl`, we can synthesize an abstract program right away from the specification, i.e. the pair of pre-

and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

```

lemma precond-postcond-implementable:
  implementable precond postcond
apply(auto simp: implementable-def)
apply(case-tac ι, simp-all)
done

```

Based on this machinery, it is now possible to construct the system model as the canonical completion of the (functional) specification consisting of pre- and post-conditions

definition $SYS = (strong-impl\ precond\ postcond)$

```

lemma SYS-is-strong-impl : is-strong-impl precond postcond SYS
by(simp add: SYS-def is-strong-impl)

```

```

thm SYS-is-strong-impl[simplified is-strong-impl-def, THEN spec, of (send tid tid' m), simplified]

```

Proving Symbolic Execution Rules for the Abstractly Program The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec):

```

lemma Eps-split-eq' :  $(SOME\ (x', y').\ x' = x \wedge y' = y) = (SOME\ (x', y').\ x = x' \wedge y = y')$ 
by(rule arg-cong[of - - Eps], auto)

```

```

interpretation send : efsm-det
  precond postcond SYS (send tid tid' m) λ-. send-ok
   $\lambda\ \sigma.\ \sigma(tid \mapsto (the(\sigma\ tid) - int\ m))\ \lambda\ \sigma.\ (tid \in dom\ \sigma \wedge tid' \in dom\ \sigma \wedge (int\ m) \leq the(\sigma\ tid))$ 
by unfold-locales (auto simp: SYS-def Eps-split-eq')

```

```

interpretation receive : efsm-det
  precond postcond SYS (rec tid tid' m) λ-. rec-ok
   $\lambda\ \sigma.\ \sigma(tid' \mapsto (the(\sigma\ tid') + int\ m))$ 
   $\lambda\ \sigma.\ (tid \in dom\ \sigma \wedge tid' \in dom\ \sigma)$ 
by unfold-locales (auto simp: SYS-def Eps-split-eq')

```

```

interpretation status : efsm-det
  precond postcond SYS (status tid)
   $\lambda\ \sigma.\ (status-ok\ (nat(the(\sigma\ tid))))$ 
   $\lambda\ \sigma.\ \sigma\ \lambda\ \sigma.\ (tid \in dom\ \sigma)$ 
by unfold-locales (auto simp: SYS-def Eps-split-eq')

```

Setup Now we close the theory of symbolic execution by *exluding* elementary rewrite steps on $mbind_{FailSave}$, i.e. the rules $mbind_{FailSave}\ []\ ?iostep\ ?\sigma = Some\ ([],\ ?\sigma)\ mbind_{FailSave}\ (?a\ \# ?S)\ ?iostep\ ?\sigma = (case\ ?iostep\ ?a\ ?\sigma\ of\ None \Rightarrow Some\ ([],\ ?\sigma) \mid Some\ (out,\ \sigma') \Rightarrow case\ mbind_{FailSave}\ ?S\ ?iostep\ \sigma'\ of\ None \Rightarrow Some\ ([out],\ \sigma') \mid Some\ (outs,\ \sigma'') \Rightarrow Some\ (out\ \# outs,\ \sigma''))$

```

declare mbind.simps(1) [simp del]
  mbind.simps(2) [simp del]

```

Here comes an interesting detail revealing the power of the approach: The generated sequences still respect the preconditions imposed by the specification - in this case, where we are talking about a `task_id` for which a defined account exists and for which we will never produce traces in which we release more money than available on it.

Restricting the Test-Space by Test Purposes We introduce a constraint on the input sequence, in order to limit the test-space a little and eliminate logically possible, but irrelevant test-sequences for a specific test-purpose. In this case, we narrow down on test-sequences concerning a specific `task_id` c with a specific bank-account number no .

We make the (in this case implicit, but as constraint explicitly stated) test hypothesis, that the *SUT* is correct if it behaves correct for a single `task_id`. This boils down to the assumption that they are implemented as atomic transactions and interleaved processing does not interfere with a single thread.

term *List.member* $y\ x$

fun *test-purpose* :: [*task-id list*, *in-c list*] \Rightarrow *bool*

where

test-purpose $R\ [status\ tid] = (tid \in set\ R)$
 $| test-purpose\ R\ ((send\ tid\ tid'\ m) \# S) = (tid \in set\ R \wedge tid' \in set\ R \wedge test-purpose\ R\ S)$
 $| test-purpose\ R\ ((rec\ tid\ tid'\ m) \# S) = (tid \in set\ R \wedge tid' \in set\ R \wedge test-purpose\ R\ S)$
 $| test-purpose\ - = False$

lemma [*simp*] : *test-purpose* [] $a = False$ **by** (*induct* a , *simp-all*, *case-tac* $a1$,
simp-all, *case-tac* $a2$, *simp-all*)

lemma [*simp*] : *test-purpose* $r\ [] = False$ **by** *simp*

lemma [*simp*] : *test-purpose* ($tid \# R$) [a] $= ((a = status\ tid) \vee test-purpose\ R\ [a])$

proof (*induct* R)

case *Nil* **show** ?*case* **by** (*cases* a , *simp-all*)

next

case (*Cons* $a'\ R'$) **then show** ?*case* **by** (*cases* a , *simp-all*)

qed

lemma [*rule-format, simp, dest*] : *test-purpose* $R\ (S@[a]) \longrightarrow (\exists\ tid \in set\ R. (a = status\ tid))$

proof (*induct* S *arbitrary:* R)

case *Nil* **then show** ?*case* **apply** (*auto*)

by (*metis* *in-c.exhaust* *test-purpose.simps(1)* *test-purpose.simps(2)*
test-purpose.simps(3) *test-purpose.simps(4)*)

next

case (*Cons* $a\ list$) **show** ?*case* **apply** (*insert* *Cons.hyps*)

by (*metis* *append-Cons* *hd-Cons-tl* *in-c.exhaust* *snoc-eq-iff-butlast*
test-purpose.simps(2) *test-purpose.simps(3)* *test-purpose.simps(5)*)

qed

schematic-goal *sdf* : $(\exists\ tid \in set\ [1,2,3]. (a = status\ tid)) = ?X$

apply (*rule trans*)

apply *simp*
oops

lemma [*simp*] :
 $test_purpose\ H\ (a\#R) = (((\exists m. \exists tid \in set\ H. \exists tid' \in set\ H. a = send\ tid\ tid'\ m) \vee$
 $(\exists m. \exists tid \in set\ H. \exists tid' \in set\ H. a = rec\ tid\ tid'\ m))$
 $\wedge test_purpose\ H\ R) \vee$
 $(R = [] \wedge (\exists tid \in set\ H. a = status\ tid)))$
proof(*induct R arbitrary: a*)
case *Nil* **then show** ?*case* **by**(*simp,case-tac a, simp-all*)
next
case (*Cons a list aa*) **then show** ?*case* **by**(*simp,case-tac aa, simp-all*)
qed

The following scenario checks send-operations to itself

lemma *exhaust1* [*simp*] :
 $test_purpose\ [tid]\ (a\#R) = (((\exists m. a = send\ tid\ tid\ m) \vee$
 $(\exists m. a = rec\ tid\ tid\ m))$
 $\wedge test_purpose\ [tid]\ R) \vee$
 $(R = [] \wedge (a = status\ tid)))$
by *simp*
lemma *exhaust2* [*simp*] :
 $test_purpose\ [tid,tid']\ (a\#R) = (((\exists m. a = send\ tid\ tid\ m) \vee (\exists m. a = send\ tid'\ tid'\ m) \vee$
 $(\exists m. a = send\ tid\ tid'\ m) \vee (\exists m. a = send\ tid'\ tid\ m) \vee$
 $(\exists m. a = rec\ tid\ tid'\ m) \vee (\exists m. a = rec\ tid'\ tid\ m) \vee$
 $(\exists m. a = rec\ tid\ tid\ m) \vee (\exists m. a = rec\ tid'\ tid'\ m))$
 $\wedge test_purpose\ [tid,tid']\ R) \vee$
 $(R = [] \wedge ((a = status\ tid) \vee (a = status\ tid'))))$
by *auto*

Well-formed traces

fun *compair* :: *in-c* \Rightarrow *in-c* \Rightarrow *bool*
where $compair\ (send\ x\ y\ z)\ (rec\ x'\ y'\ z') = ((x=x') \wedge (y=y') \wedge (z = z'))$
 $| compair\ -\ - = False$

definition *wf-trace* :: *in-c list* \Rightarrow *bool*
where $wf_trace\ t = (\exists f. \forall n \in \{0..<length\ t\}. f\ n \in \{0..<length\ t\} \wedge$
 $f\ (f\ n) = n \wedge (*\ derivable\ ?\ *)$
 $(case\ (t!n)\ of$
 $(send\ -\ -\ -) \Rightarrow$
 $(compair\ (t!n)\ (t!(f\ n)) \wedge f\ n > n)$
 $|(rec\ -\ -\ -) \Rightarrow$
 $(compair\ (t!n)\ (t!(f\ n)) \wedge f\ n < n)$
 $|(status\ -) \Rightarrow (f\ n) = n)$
 $)$

Misc

consts $PUT :: in-c \Rightarrow 'σ \Rightarrow (out-c \times 'σ) \text{ option}$

end

10. The MyKeOS “Traditional” Data-sequence enumeration approach

```
theory MyKeOS-test
imports MyKeOS
```

```
begin
```

The purpose of these test-scenarios is to apply the brute-force data-exploration approach to a little operation system example. It is conceptually very close to the Bank-example, essentially a renaming. However, the present "data-exploration" based approach is an interesting intermediate step to the subsequently shown scenarios based on:

1. exploration of the interleaving space
2. optimized exploration of the interleaving space, including theory for partial-order reduction.

```
declare [[testgen-profiling]]
```

The TestGen Setup The default configuration of `gen_test_cases` does *not* descend into sub-type expressions of type constructors (since this is not always desirable, the choice for the default had been for "non-descent"). This case is relevant here since *in-c list* has just this structure but we need ways to explore the input sequence type further. Thus, we need to configure, for all test cases, and derivation descendants of the relisting clauses during splitting, again splitting for all parameters of input type *in-c*:

```
set-pre-safe-tac⟨⟨
  (fn ctxt => TestGen.ALLCASES(
    TestGen.CLOSURE (
      TestGen.case-tac-tyt ctxt [MyKeOS.in-c])))
  ⟩⟩
```

The Scenario We construct test-sequences for a concrete `task_id` (implicitly assuming that interleaving actions with other `task_id`'s will not influence the system behaviour. In order to prevent HOL-TestGen to perform case-splits over names — i.e. list of characters — we define it as constant.

```
definition tid0 :: task-id where tid0 = 0
definition tid1 :: task-id where tid1 = 1
```

```
declare [[goals-limit = 500]]
```

Making my own test-data generation — temporarily lemma *HH* : $(A \wedge (A \longrightarrow B))$
 $= (A \wedge B)$ by *auto*

Some Experiments with nitpick as Testdata Selection Machine. Exists in two formats : General Fail-Safe Tests (which allows for scenarios with normal *and* exceptional behaviour; and Fail-Stop Tests, which generates Tests only for normal behaviour and correspond to inclusion test refinement.

lemma *H*: $((((X586X11506, X587X11507) \in \text{dom } X588X11508 \longrightarrow$
 $[\text{status-ok } (\text{nat } (\text{the } (X588X11508 (X586X11506, X587X11507))))] = X590X11510 \wedge$
 $X588X11508 (X586X11506, X587X11507) = \text{Some } X589X11509) \wedge$
 $((X586X11506, X587X11507) \notin \text{dom } X588X11508 \longrightarrow$
 $[] = X590X11510 \wedge X588X11508 (X586X11506, X587X11507) = \text{Some } X589X11509)))$
nitpick[*satisfy, debug*]
oops

lemma *H*: $((X586X11506, X587X11507) \in \text{dom}$
 $([(X586X11506, X587X11507) \mapsto X589X11509]) \longrightarrow$
 $[\text{status-ok } (\text{nat } (\text{the } ($
 $((X586X11506, X587X11507) \mapsto X589X11509)] (X586X11506, X587X11507))))] =$
 $X590X11510 \wedge$
 $((X586X11506, X587X11507) \mapsto X589X11509)] (X586X11506, X587X11507) = \text{Some}$
 $X589X11509) \wedge$
 $((X586X11506, X587X11507) \notin \text{dom}$
 $([(X586X11506, X587X11507) \mapsto X589X11509]) \longrightarrow$
 $[] = X590X11510 \wedge ((X586X11506, X587X11507) \mapsto X589X11509)] (X586X11506,$
 $X587X11507) = \text{Some } X589X11509))$
nitpick[*satisfy, debug, timeout=500*]
oops

In the following, we discuss a test-scenario with error-abort semantics; i.e. in each test-case, a sequence may be chosen (by the test data selection) where the `task_id` has several accounts. . .

test-spec *test-status*:
assumes *account-def* : $no \in \text{dom } \sigma_0 \wedge no' \in \text{dom } \sigma_0$
and *test-purpose* : *test-purpose* [*no, no'*] *S*
and *sym-exec-spec* : $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSafe}} S \text{ SYS}; \text{return } (s = x))$
shows $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSafe}} S \text{ PUT}; \text{return } (s = x))$

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking *x* from being case-split (which complicates the process).

apply(*rule rev-mp*[*OF sym-exec-spec*])
apply(*rule rev-mp*[*OF account-def*])
apply(*rule rev-mp*[*OF test-purpose*])
apply(*rule-tac* *x=x* **in** *spec*[*OF allI*])

Starting the test generation process.

apply(*gen-test-cases* 3 1 *PUT*)

apply(*simp-all* *add: HH split: HOL.if-split-asm*)

mk-test-suite *mykeos-simpleSNXB*

And now the Fail-Stop scenario — this corresponds exactly to inclusion test.

```
declare Monads.mbind'-bind [simp del]
test-spec test-status2:
assumes system-def :  $tid \in \text{dom } \sigma_0 \wedge tid' \in \text{dom } \sigma_0$ 
and test-purpose : test-purpose [tid, tid'] S
and sym-exec-spec :
 $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{ SYS}; \text{return } (s = x))$ 
shows  $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{ PUT}; \text{return } (s = x))$ 
```

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking *x* from being case-split (which complicates the process).

```
apply(rule rev-mp[OF sym-exec-spec])
apply(rule rev-mp[OF system-def])
apply(rule rev-mp[OF test-purpose])
apply(rule-tac x=x in spec[OF allI])
```

Starting the test generation process.

```
using[[no-uniformity]]
apply(gen-test-cases 3 1 PUT)
```

So lets go for a more non-destructive approach:

```
apply simp-all
```

```
using[[no-uniformity=false]]
apply(tactic TestGen.ALLCASES(TestGen.uniformityI-tac @{context} [PUT]))
```

mk-test-suite *mykeos-simpleNB*

```
ML⟨⟨ Isar-Cmd.done-proof; Proof.global-done-proof ⟩⟩
```

```
Test-Data Generation ML⟨⟨ Thm.close-derivation
⟩⟩
```

```
declare [[testgen-iterations=0]]
declare [[testgen-SMT]]
```

```
declare tid0-def [testgen-smt-facts]
declare tid1-def [testgen-smt-facts]
```

```
declare mem-Collect-eq [testgen-smt-facts]
declare Collect-mem-eq [testgen-smt-facts]
declare dom-def [testgen-smt-facts]
declare Option.option.sel [testgen-smt-facts]
```

gen-test-data *mykeos-simpleSNXB*

print-conc-tests *mykeos-simpleSNXB*

```
gen-test-data mykeos-simpleNB
print-conc-tests mykeos-simpleNB
```

Generating the Test-Driver for an SML and C implementation The generation of the test-driver is non-trivial in this exercise since it is essentially two-staged: Firstly, we chose to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. Recall that a test-driver consists of four components:

- `../../../../../../harness/sml/main.sml` the global controller (a fixed element in the library),
- `../../../../../../harness/sml/main.sml` a statistic evaluation library (a fixed element in the library),
- `bank_simple_test_script.sml` the test-script that corresponds merely one-to-one to the generated test-data (generated)
- `bank_adapter.sml` a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign-language interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing 'bit-string-representation-behind-the-scene' enabling relatively efficient symbolic computation. Both representations can be compiled "natively" to data types in the SML level. By an appropriate configuration, the code-generator can map "int" of HOL to three different implementations: the SML standard library `Int.int`, the native-C interfaced by `Int32.int`, and the `IntInf.int` from the multi-precision library `gmp` underneath the `polymml-compiler`.

We do a three-step compilation of data-representations model-to-model, model-to-SML, SML-to-C.

A basic preparatory step for the initializing the test-environment to enable code-generation is:

```
generate-test-script mykeos-simpleNB
thm mykeos-simpleNB.test-script
generate-test-script mykeos-simpleSNXB
```

More advanced Test-Case Generation Scenarios Exploring a bit the limits ...

Rewriting based approach of symbolic execution ... FailSave Scenario

```
test-spec test-status:
```

```

assumes account-def :  $(no) \in \text{dom } \sigma_0 \wedge (no') \in \text{dom } \sigma_0$ 
and    test-purpose : test-purpose  $[no, no'] S$ 
and    sym-exec-spec :
       $\sigma_0 \models (s \leftarrow \text{mbind}_{FailSave} S \text{ SYS}; \text{return } (s = x))$ 
shows  $\sigma_0 \models (s \leftarrow \text{mbind}_{FailSave} S \text{ PUT}; \text{return } (s = x))$ 

```

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-split (which complicates the process).

```

apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy @{\context} 1, rule-tac x=x in spec[OF allI])

```

Starting the test generation process.

```

apply(gen-test-cases 3 1 PUT)

```

Symbolic Execution:

```

apply(simp-all add: HH split: HOL.if-split-asm)
mk-test-suite mykeos-large

```

```

gen-test-data mykeos-large
print-conc-tests mykeos-large

```

Rewriting based approach of symbolic execution ... FailSave Scenario

```

test-spec test-status:
assumes account-def :  $(no) \in \text{dom } \sigma_0 \wedge (no') \in \text{dom } \sigma_0$ 
and    test-purpose : test-purpose  $[(no), (no')] S$ 
and    sym-exec-spec :
       $\sigma_0 \models (s \leftarrow \text{mbind}_{FailStop} S \text{ SYS}; \text{return } (s = x))$ 
shows  $\sigma_0 \models (s \leftarrow \text{mbind}_{FailStop} S \text{ PUT}; \text{return } (s = x))$ 

```

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-split (which complicates the process).

```

apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy @{\context} 1, rule-tac x=x in spec[OF allI])

```

Starting the test generation process.

```

using  $[[no\text{-}uniformity]]$ 
apply(gen-test-cases 3 1 PUT)

```

Symbolic Execution:

```

apply(simp-all add: HH split: HOL.if-split-asm)

```

```

apply(auto)
mk-test-suite mykeos-large2

```

```

gen-test-data mykeos-large2
print-conc-tests mykeos-large2

```

And now, to compare, elimination based procedures ...

```

declare
  status.exec-mbindFSave-If [simp del]

  status.exec-mbindFStop    [simp del]

```

```

ML⟨⟨ open Tactical; ⟩⟩
test-spec test-status:
assumes account-defined: (no) ∈ dom σ0 ∧ (no′) ∈ dom σ0
and test-purpose : test-purpose [(no),(no′)] S
and sym-exec-spec :
    σ0 ⊨ (s ← mbindFailStop S SYS; return (s = x))
shows σ0 ⊨ (s ← mbindFailStop S PUT; return (s = x))
apply(insert account-defined test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy @{context} 1, rule-tac x=x in spec[OF allI])
apply(tactic asm-full-simp-tac @{context} 1)
using [[no-uniformity]]
apply(gen-test-cases 3 1 PUT )

```

```

oops
end

```

Implementation of integer numbers by target-language integers

```

theory Code-Target-Int
imports ../GCD
begin

code-datatype int-of-integer

declare [[code drop: integer-of-int]]

context
includes integer.lifting
begin

lemma [code]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

lemma [code]:
  Int.Pos = int-of-integer ∘ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code]:
  Int.Neg = int-of-integer ∘ uminus ∘ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code-abbrev]:
  int-of-integer (numeral k) = Int.Pos k
  by transfer simp

lemma [code-abbrev]:

```

```

  int-of-integer (− numeral k) = Int.Neg k
  by transfer simp

lemma [code, symmetric, code-post]:
  0 = int-of-integer 0
  by transfer simp

lemma [code, symmetric, code-post]:
  1 = int-of-integer 1
  by transfer simp

lemma [code-post]:
  int-of-integer (− 1) = − 1
  by simp

lemma [code]:
  k + l = int-of-integer (of-int k + of-int l)
  by transfer simp

lemma [code]:
  − k = int-of-integer (− of-int k)
  by transfer simp

lemma [code]:
  k − l = int-of-integer (of-int k − of-int l)
  by transfer simp

lemma [code]:
  Int.dup k = int-of-integer (Code-Numeral.dup (of-int k))
  by transfer simp

declare [[code drop: Int.sub]]

lemma [code]:
  k * l = int-of-integer (of-int k * of-int l)
  by simp

lemma [code]:
  k div l = int-of-integer (of-int k div of-int l)
  by simp

lemma [code]:
  k mod l = int-of-integer (of-int k mod of-int l)
  by simp

lemma [code]:
  divmod m n = map-prod int-of-integer int-of-integer (divmod m n)
  unfolding prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv
  by transfer simp

lemma [code]:
  HOL.equal k l = HOL.equal (of-int k :: integer) (of-int l)

```

```

    by transfer (simp add: equal)

lemma [code]:
   $k \leq l \iff (of\_int\ k :: integer) \leq of\_int\ l$ 
  by transfer rule

lemma [code]:
   $k < l \iff (of\_int\ k :: integer) < of\_int\ l$ 
  by transfer rule

declare [[code drop: gcd :: int  $\Rightarrow$  - lcm :: int  $\Rightarrow$  -]]

lemma gcd-int-of-integer [code]:
  gcd (int-of-integer x) (int-of-integer y) = int-of-integer (gcd x y)
  by transfer rule

lemma lcm-int-of-integer [code]:
  lcm (int-of-integer x) (int-of-integer y) = int-of-integer (lcm x y)
  by transfer rule

end

lemma (in ring-1) of-int-code-if:
  of-int k = (if k = 0 then 0
    else if k < 0 then - of-int (- k)
    else let
      l = 2 * of-int (k div 2);
      j = k mod 2
      in if j = 0 then l else l + 1)
  proof -
    from div-mult-mod-eq have *: of-int k = of-int (k div 2 * 2 + k mod 2) by simp
    show ?thesis
    by (simp add: Let-def of-int-add [symmetric]) (simp add: * mult.commute)
  qed

declare of-int-code-if [code]

lemma [code]:
  nat = nat-of-integer  $\circ$  of-int
  including integer.lifting by transfer (simp add: fun-eq-iff)

code-identifier
code-module Code-Target-Int  $\hookrightarrow$ 
  (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

```

Avoidance of pattern matching on natural numbers

```

theory Code-Abstract-Nat
imports Main
begin

```

When natural numbers are implemented in another than the conventional inductive $0/Suc$ representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

Case analysis Case analysis on natural numbers is rephrased using a conditional expression:

```
lemma [code, code-unfold]:
  case-nat = ( $\lambda f g n. \text{if } n = 0 \text{ then } f \text{ else } g (n - 1)$ )
  by (auto simp add: fun-eq-iff dest!: gr0-implies-Suc)
```

Preprocessors The term $Suc\ n$ is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

```
lemma Suc-if-eq:
  assumes  $\bigwedge n. f (Suc\ n) \equiv h\ n$ 
  assumes  $f\ 0 \equiv g$ 
  shows  $f\ n \equiv \text{if } n = 0 \text{ then } g \text{ else } h (n - 1)$ 
  by (rule eq-reflection) (cases n, insert assms, simp-all)
```

The rule above is built into a preprocessor that is plugged into the code generator.

```
setup (
  let

  val Suc-if-eq = Thm.incr-indexes 1 @ {thm Suc-if-eq};

  fun remove-suc ctxt thms =
    let
      val vname = singleton (Name.variant-list (map fst
        (fold (Term.add-var-names o Thm.full-prop-of) thms [])) n;
      val cv = Thm.ctrm-of ctxt (Var ((vname, 0), HOLogic.natT));
      val lhs-of = snd o Thm.dest-comb o fst o Thm.dest-comb o Thm.cprop-of;
      val rhs-of = snd o Thm.dest-comb o Thm.cprop-of;
      fun find-vars ct = (case Thm.term-of ct of
        (Const (@{const-name Suc}, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
      | - $ - =>
        let val (ct1, ct2) = Thm.dest-comb ct
        in
          map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
          map (apfst (Thm.apply ct1)) (find-vars ct2)
        end
      | - => []);
      val eqs = maps
        (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
      fun mk-thms (thm, (ct, cv')) =
        let
          val thm' =
            Thm.implies-elim
              (Conv.fconv-rule (Thm.beta-conversion true)
```

```

      (Thm.instantiate'
        [SOME (Thm.ctyp-of-cterm ct)] [SOME (Thm.lambda cv ct),
          SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
        Suc-if-eq) (Thm.forall-intr cv' thm)
    in
      case map-filter (fn thm'' =>
        SOME (thm'', singleton
          (Variable.trade (K (fn [thm'''] => [thm''' RS thm']))
            (Variable.declare-thm thm'' ctxt)) thm''))
        handle THM - => NONE) thms of
      [] => NONE
    | thmps =>
      let val (thms1, thms2) = split-list thmps
      in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
    end
  in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = @{const-name Suc});
  in
    if forall (can dest) thms andalso exists (contains-suc o dest) thms
    then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
    else NONE
  end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end;
)

```

end

Implementation of natural numbers by target-language integers

```

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

```

```

Implementation for nat context
includes natural.lifting integer.lifting
begin

```

```

lift-definition Nat :: integer  $\Rightarrow$  nat
is nat
.

```



```

lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
  by (transfer, simp)+

lemma [code-abbrev]:
  integer-of-nat = of-nat
  by transfer rule

lemma [code-unfold]:
  Int.nat (int-of-integer k) = nat-of-integer k
  by transfer rule

lemma [code abstype]:
  Code-Target-Nat.Nat (integer-of-nat n) = n
  by transfer simp

lemma [code abstract]:
  integer-of-nat (nat-of-integer k) = max 0 k
  by transfer auto

lemma [code-abbrev]:
  nat-of-integer (numeral k) = nat-of-num k
  by transfer (simp add: nat-of-num-numeral)

lemma [code abstract]:
  integer-of-nat (nat-of-num n) = integer-of-num n
  by transfer (simp add: nat-of-num-numeral)

lemma [code abstract]:
  integer-of-nat 0 = 0
  by transfer simp

lemma [code abstract]:
  integer-of-nat 1 = 1
  by transfer simp

lemma [code]:
  Suc n = n + 1
  by simp

lemma [code abstract]:
  integer-of-nat (m + n) = of-nat m + of-nat n
  by transfer simp

lemma [code abstract]:
  integer-of-nat (m - n) = max 0 (of-nat m - of-nat n)
  by transfer simp

lemma [code abstract]:
  integer-of-nat (m * n) = of-nat m * of-nat n

```

```

    by transfer (simp add: of-nat-mult)

lemma [code abstract]:
  integer-of-nat (m div n) = of-nat m div of-nat n
  by transfer (simp add: zdiv-int)

lemma [code abstract]:
  integer-of-nat (m mod n) = of-nat m mod of-nat n
  by transfer (simp add: zmod-int)

lemma [code]:
  Divides.divmod-nat m n = (m div n, m mod n)
  by (fact divmod-nat-div-mod)

lemma [code]:
  divmod m n = map-prod nat-of-integer nat-of-integer (divmod m n)
  by (simp only: prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv)
    (transfer, simp-all only: nat-div-distrib nat-mod-distrib
      zero-le-numeral nat-numeral)

lemma [code]:
  HOL.equal m n = HOL.equal (of-nat m :: integer) (of-nat n)
  by transfer (simp add: equal)

lemma [code]:
  m ≤ n ⟷ (of-nat m :: integer) ≤ of-nat n
  by simp

lemma [code]:
  m < n ⟷ (of-nat m :: integer) < of-nat n
  by simp

lemma num-of-nat-code [code]:
  num-of-nat = num-of-integer ∘ of-nat
  by transfer (simp add: fun-eq-iff)

end

lemma (in semiring-1) of-nat-code-if:
  of-nat n = (if n = 0 then 0
    else let
      (m, q) = Divides.divmod-nat n 2;
      m' = 2 * of-nat m
    in if q = 0 then m' else m' + 1)
proof -
  from div-mult-mod-eq have *: of-nat n = of-nat (n div 2 * 2 + n mod 2) by simp
  show ?thesis
  by (simp add: Let-def divmod-nat-div-mod of-nat-add [symmetric])
    (simp add: * mult.commute of-nat-mult add.commute)
qed

declare of-nat-code-if [code]

```

definition *int-of-nat* :: *nat* \Rightarrow *int* **where**
 [code-abbrev]: *int-of-nat* = *of-nat*

lemma [code]:
int-of-nat *n* = *int-of-integer* (*of-nat* *n*)
by (*simp add: int-of-nat-def*)

lemma [code abstract]:
integer-of-nat (*nat* *k*) = *max* 0 (*integer-of-int* *k*)
including *integer.lifting* **by** *transfer auto*

lemma *term-of-nat-code* [code]:
 — Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of *n* =
Code-Evaluation.App
 (*Code-Evaluation.Const* (*STR* "Code-Numeral.nat-of-integer")
 (*typerep.Typerep* (*STR* "fun")
 [*typerep.Typerep* (*STR* "Code-Numeral.integer") []],
typerep.Typerep (*STR* "Nat.nat") []]))
 (*term-of-class.term-of* (*integer-of-nat* *n*))
by (*simp add: term-of-anything*)

lemma *nat-of-integer-code-post* [code-post]:
nat-of-integer 0 = 0
nat-of-integer 1 = 1
nat-of-integer (*numeral* *k*) = *numeral* *k*
including *integer.lifting* **by** (*transfer, simp*)**+**

code-identifier
code-module *Code-Target-Nat* \hookrightarrow
 (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Int* *Code-Target-Nat*
begin

end

theory *MyKeOS-test-conc*
imports *MyKeOS*
 ~~/src/HOL/Library/Code-Target-Numeral
Code-gdb-script
begin

declare [[*testgen-profiling*]]

Interleaving

The purpose of this example is to model system calls that consists of a number of (internal) atomic actions; the global behavior is presented by the interleaving of the actions

definition $SEND\ tid\ tid'\ m = [send\ tid\ tid'\ m, status\ tid]$

definition $REC\ tid\ tid'\ m = [rec\ tid\ tid'\ m, status\ tid]$

value $interleave\ (SEND\ 5\ 0\ m)\ (REC\ 0\ 5\ m)$

In the following, we do a predicate abstraction on the interleave language, leading to an automaton represented as a set of rewrites ...

fun $Interleave :: in-c\ list \Rightarrow nat \times nat \Rightarrow int \Rightarrow nat \Rightarrow nat \Rightarrow bool$ (**infixl** $\bowtie 100$)

where $S \bowtie (a, b) = (\lambda\ tid\ m\ m'. (S \in interleave\ (drop\ a\ (SEND\ tid\ 5\ m))$
 $(drop\ b\ (REC\ 5\ tid\ m'))))$

lemma $init-Interleave : (S \bowtie (0, 0))\ tid\ m\ m' = (S \in interleave\ (SEND\ tid\ 5\ m)\ (REC\ 5\ tid\ m'))$
by $simp$

value $interleave\ (SEND\ 0\ 5\ m)\ (REC\ 5\ 0\ m')$

find-theorems $name:Interleave$

lemma $ref-mt\ [simp]: \neg((\square \bowtie (0, 0))\ tid\ m\ m')$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-0-0\ [simp]: \neg(((status\ a) \# R) \bowtie (0, 0))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-1-0\ [simp]: a \neq tid \implies \neg(((status\ a) \# R) \bowtie (1, 0))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-0-1\ [simp]: a \neq 5 \implies \neg(((status\ a) \# R) \bowtie (0, 1))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-1-1\ [simp]: a \neq 5 \implies a \neq tid \implies \neg(((status\ a) \# R) \bowtie (1, 1))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-3-1\ [simp]: a \neq 5 \implies \neg(((status\ a) \# R) \bowtie (3, 1))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

lemma $ref-1-3\ [simp]: a \neq tid \implies \neg(((status\ a) \# R) \bowtie (1, 3))\ tid\ m\ m'$
by $(simp\ add:REC-def\ SEND-def)$

value $((a \# R) \bowtie (0, 0))\ tid\ m\ m'$

lemma $trans-0-0\ [simp]: (((a \# R) \bowtie (0, 0))\ tid\ m\ m') =$
 $((a = send\ tid\ 5\ m \wedge (R \bowtie (1, 0))\ tid\ m\ m') \vee$

```

      (a = rec 5 tid m' ∧ (R ⋈ (0, 1)) tid m m')
apply (simp add: SEND-def REC-def)
by auto

lemma trans-1-0 [simp]: (((a # R) ⋈ (1, 0)) tid m m') =
      ((a = rec 5 tid m' ∧ (R ⋈ (1, 1)) tid m m') ∨
       (a = status tid ∧ (R ⋈ (2, 0)) tid m m'))
apply (simp add: SEND-def REC-def)
by auto

lemma trans-2-0 [simp]: (((a # R) ⋈ (2, 0)) tid m m') =
      ( (a = rec 5 tid m' ∧ R = [status 5]))
by (simp add: SEND-def REC-def)

lemma trans-2-1 [simp]: (((a # R) ⋈ (2, 1)) tid m m') = (a = status 5 ∧ R = [])
by (simp add: SEND-def REC-def)

value interleave (drop 0 (SEND tid m m'))(drop 0 (REC tid m'' m'''))

  TestData Hack:

lemma PO-norm0 [simp]: PO True by (simp add: PO-def)

  The following scenario is meant to describe the symbolic execution step by step.

declare Monads.mbind'-bind [simp del]

find-theorems mbind_FailStop []

lemma example-symbolic-execution-simulation :
  assumes H: S = [send tid 1 m, rec tid 0 m', rec tid 1 m'', status tid ]
  assumes SE:  $\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{ SYS}; \text{return } (x = s))$ 
  shows P
apply (insert SE H)
apply (hypsubst)
apply (tactic ematch-tac @ {context} [ @ {thm status.exec-mbindFStop-E},
      @ {thm receive.exec-mbindFStop-E},
      @ {thm send.exec-mbindFStop-E} ] 1)
apply (tactic ematch-tac @ {context} [ @ {thm status.exec-mbindFStop-E},
      @ {thm receive.exec-mbindFStop-E},
      @ {thm send.exec-mbindFStop-E} ] 1)
apply (tactic ematch-tac @ {context} [ @ {thm status.exec-mbindFStop-E},
      @ {thm receive.exec-mbindFStop-E},
      @ {thm send.exec-mbindFStop-E} ] 1)
apply (tactic ematch-tac @ {context} [ @ {thm status.exec-mbindFStop-E},
      @ {thm receive.exec-mbindFStop-E},
      @ {thm send.exec-mbindFStop-E} ] 1)
apply (tactic ematch-tac @ {context} [ @ {thm status.exec-mbindFStop-E},
      @ {thm receive.exec-mbindFStop-E},
      @ {thm send.exec-mbindFStop-E},
      @ {thm valid-mbind'-mtE} ] 1)

apply simp
oops

```

end

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 and checking that behavior of unchanged portions has not changed.

Stub: Stubs are “simulated” implementations of functions, they are used to simulate functionality that does not yet exist or cannot be run in the test environment.

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

Test case generation: For each operation 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 hypotheses, which are 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] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, May 1986.
- [2] S. Berghofer and T. 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.
- [3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. Zelkowitz, editor, *Advances In Computers*, volume 58. Academic Press, 2003.
- [4] S. Böhme and T. Weber. Fast lcf-style proof reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.
- [5] A. D. Brucker, O. Havle, Y. Nemouchi, and B. Wolff. Testing the IPC protocol for a real-time operating system. In *Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, pages 40–60, 2015.
- [6] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, Linz, 2005.
- [7] A. D. Brucker and B. Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005.
- [8] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in *Lecture Notes in Computer Science*, pages 417–420. Springer-Verlag, Heidelberg, 2009.
- [9] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012.
- [10] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [11] K. Claessen and J. 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.

- [12] P. Cousot and R. 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.
- [13] 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.
- [14] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284. Springer, Apr. 1993.
- [15] P. Dybjer, Q. Haiyan, and M. 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.
- [16] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.
- [17] S. Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.
- [18] The Isabelle Home Page, 2016.
- [19] MLj.
- [20] MLton, 2016.
- [21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [22] Poly/ML – the poly/ml implementation of standard ml., 2016.
- [23] sml.net.
- [24] SML of New Jersey.
- [25] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 2004.
- [26] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

Index

- abstract test data, 87
- breadth, 87
 - $\langle \textit{breadth} \rangle$, 13
 - $\langle \textit{clasimpmod} \rangle$, 13
- data separation lemma, 13
- depth, 87
 - $\langle \textit{depth} \rangle$, 13
- `export_test_data` (command), 15
- `gen_test_cases` (method), 13
- `gen_test_data` (command), 14
- `generate_test_script` (command), 15
- higher-order logic, *see* HOL
- HOL, 7
- Isabelle, 6, 7, 9
- `Main` (theory), 11
- `mk_test_suite` (command), 13
 - $\langle \textit{name} \rangle$, 13
- program under test, 13, 15
- random solver, 14
- regression testing, 87
- regularity hypothesis, 13
- SML, 7
- software
 - testing, 5
 - validation, 5
 - verification, 5
- Standard ML, *see* SML
- stub, 87
- test, 6
 - `test` (attribute), 17
 - test specification, 11
 - test theorem, 13
 - test case, 11
 - test data generation, 11
 - test executable, 11
 - test specification, 6
 - test case, 5, 6, 87
 - test case generation, 5, 11, 13, 17, 87
 - test data, 5, 11, 14, 87
 - test data generation, 5, 87
 - test data selection, *see* test data generation
 - test driver, *see* test harness
 - test executable, 17–19, 87
 - test execution, 6, 11, 17, 87
 - test harness, 15, 87
 - test hypothesis, 6, 87
 - test procedure, 5
 - test result verification, 11
 - test result verification, 6, 88
 - test script, 11, 15–17, 88
 - test specification, 13, 87
 - test theorem, 88
 - test theory, 12
 - test trace, 18, 19, 88
 - `test_spec` (command), 11
 - `Testing` (theory), 11
- unit test
 - specification-based, 5