Under consideration for publication in Formal Aspects of Computing

On Theorem Prover-based Testing

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

[1]SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany, e-mail: achim.brucker@sap.com
[2]Université Paris-Sud, LRI, Parc Club, 4, Rue Jaques Monod, 91893 Orsay Cedex, France, e-mail: wolff@lri.fr[1]

Abstract. HOL-TestGen is a specification and test case generation environment extending the interactive theorem prover Isabelle/HOL. As such, HOL-TestGen allows for an integrated workflow supporting interactive theorem proving, test case generation, and test data generation.

The HOL-TestGen method is two-staged: first, the original formula is partitioned into *test cases* by transformation into a normal form called *test theorem*. Second, the test cases are analyzed for ground instances (the *test data*) satisfying the constraints of the test cases. Particular emphasis is put on the control of explicit test-hypotheses which can be proven over concrete programs.

Due to the generality of the underlying framework, our system can be used for black-box unit, sequence, reactive sequence and white-box test scenarios. Although based on particularly clean theoretical foundations, the system can be applied for substantial case-studies.

Keywords: test case generations; domain partitioning; test sequence theorem proving; HOL-TestGen

1. Introduction

Today, essentially two software validation techniques are used: *software verification* and *software testing*. As far as verification methods and model-based testing techniques (i. e., generating test cases for unit or sequence testing based on an abstract specification of the system to be tested) are concerned, the interest among researchers in the mutual fertilization of these fields is growing. From the verification perspective, testing offers:

- experiences on test-adequacy criteria [ZHM97], which can be viewed as *abstraction techniques* reducing infinite models to finite and checkable ones,
- new approaches to generate *counter-examples* and therefore ways to debug specifications early, and
- new application scenarios for verification, since black-box testing can be used as a systematic experimentation method for *reverse engineering specifications* for legacy systems.

From the testing perspective, symbolic verification offers:

- ways to cope with the *state space explosions* inherent to the generation of test cases (also called "partitioning of the input-output relation" in the literature, e. g., [ZHM97]), and

[1] This work was partially supported by the Digiteo Foundation.

- ways to log the implicit *testing-hypothesis* underlying a test, to make them explicit and amenable for further analysis.

The HOL-TestGen system [BW09, BBW08, BW07, BW04, BW05] is designed to explore and exploit these complementary assets. Built on top of a widely-used interactive theorem prover, it provides automatic procedures for *test case generation* and *test data selection* as well as interactive means to perform logical massages of the intermediate results by derived rules.

For the purpose of this introduction, we will introduce these concepts generically and present our concrete instance in the later sections of this paper. A *test case generation* is a procedure that decomposes a *test specification* (*TS*), i. e., a test-property over a program under test *PUT*, into a *test theorem* of the form:

$$\frac{TC_1 \cdots TC_n \qquad H_1 \cdots H_m}{TS}\,, \tag{1}$$

where the TC_i are the *test cases* (partitions of the input/output relation) and where the $H_1, \dots, H_m$ are the explicit *test-hypotheses* underlying this test. Thus, a test theorem has the following meaning: *If the program under test passes the tests with a witness for all TC_i successfully, and if it satisfies all test-hypothesis, it is correct with respect to TS.* A test case generation is called *complete*, if and only if the test specification also implies the conjunction $TC_1 \wedge \cdots \wedge TC_n \wedge H_1 \wedge \cdots \wedge H_m$. As we will see, a test theorem will bridge under this condition the gap between test and verification. Moreover, a test data selection is an automated procedure that converts the test cases TC_i having the form:

$$\exists x_1 \cdots x_{ik}.\; C_{i_1}(x_1, \dots, x_{i_k}) \wedge \cdots \wedge C_{i,p}(x_1, \dots, x_{ik}) \rightarrow P(PUT, x_1, \dots, x_{ik}) \tag{2}$$

into the formula $P(PUT, c_1, \dots, c_{i,k})$, where the *test data* $c_1, \dots, c_{ik}$ are ground terms, PUT is a free variable serving as place-holder for the function under test, and P is a closed expression uniquely composed from *executable operators* (to be defined later). In its essence, test data selection is a constraint solving process that provides a solution for the set of constraints $C_{i1}(x_1, \dots, x_{ik}), \dots, C_{ip}(x_1, \dots, x_{ik})$; for the moment we make no further restriction on the logical language and the syntactic form of these constraints. By a sequence of further technical steps, the *test oracles* $P(PUT, c_1, \dots, c_{ik})$ were compiled to code to be included in a test-driver. For example, we might want to express the desired property "PUT is a sorting algorithm on integer lists" by the test specification

$$PUT(l :: \text{int list}) = \text{sort}(l) \tag{3}$$

where sort has been specified by, for example, an insertion-sort. A test case generation could yield the test cases in the (complete) test theorem:

$$\begin{array}{rl}
\text{sort-testthm}: & PUT(l :: \text{int list}) = \text{sort}(l)\\
1. & [] = PUT[]\\
2. & \exists x.\ [x] = PUT[x]\\
3. & \text{THYP}((\exists x.\ [x] = PUT[x]) \rightarrow (\forall x.[x] = PUT[x]))\\
4. & \exists x\, xa.\ xa < x \rightarrow [xa, x] = PUT[xa, x]\\
5. & \text{THYP}(((\exists x\, xa.\ xa < x \rightarrow [xa, x] = PUT[xa, x]) \rightarrow\\
 & \quad (\forall x\, xa.\ xa < x \rightarrow [xa, x] = PUT[xa, x]))\\
6. & \exists x\, xa.\ x \leq xa \rightarrow [x, xa] = PUT[xa, x]\\
7. & \text{THYP}(((\exists x\, xa.\ x \leq xa \rightarrow [x, xa] = PUT[xa, x]) \rightarrow\\
 & \quad (\forall x\, xa.\ x \leq xa \rightarrow [x, xa] = PUT[xa, x]))\\
\vdots & \qquad\qquad\qquad \vdots\\
20. & \text{THYP}((\forall t.\ |t| < 4 \rightarrow \text{sort}\, t = PUT\ t) \rightarrow (\forall t.\ \text{sort}\, t = PUT\ t)))
\end{array} \tag{4}$$

where the test-hypothesis were syntactically marked by the THYP operator, a constant defined semantically as the identity. The clause 2 (i. e., $\exists x.\ [x] = PUT[x]$) denotes the class of test data for the lists of length one, the clause 4 the class of input lists of length 2 with the first element smaller than the second while the clause 6 represents the class of input lists of length 2 where its the other way round. The concrete test-hypothesis used here is the uniformity-hypothesis (if the test passes for one instance in a class, it will pass always in this class; cf. 3, 5, 7 and the regularity-hypothesis (if the property holds for lists smaller length 4 then it will always hold; cf. 20.). Both types of test-hypothesis going back to [Gau95] are discussed at length in Sec. 3. It is not too hard to see that this test theorem is complete—consider that the listed test cases are conjoined,

that THYP can be omitted, and that lists are an *inductively* defined data structure. Resulting test oracles are, e. g., $[] = PUT([])$, $[3] = PUT([3])$, $[2,3] = PUT([2,3])$, $[2,3] = PUT([3,2])$.

Besides this fresh view on the foundations of testing, our paper provides the following contributions:

1. a theory for the *presentation* of a variety of test methods (e. g., unit-test, sequence-test, reactive sequence-test, white-box-test) within a unifying, symbolic framework,
2. a concrete implementation on Isabelle/HOL that covers the entire workflow from modeling to test-driver generation,
3. a pragmatics for theorem-proving based testing presented by a number of paradigmatic test scenarios contained in a library of examples, and
4. an analysis of our novel concept of *explicit test-hypothesis*, revealing some insight into the relations between tests and proofs.

While HOL-TestGen has been used for case-studies in the domain of network infrastructures [BBKW10] as well as access-control policies [BBKWed]; we will focus in this paper on fairly small, paradigmatic examples. In particular, in-depth empirical case studies as well as advanced symbolic evaluation techniques implemented on HOL-TestGen (e. g., using massive parallelism and SMT techniques) are out of the scope of this paper.

This paper consists of five parts: In the first part (Sec. 2), we introduce the formal basis, i. e., the logical framework Isabelle/HOL which is used as programmable symbolic computation environment. In the following two parts, we show how our framework supports both functional unit testing (Sec. 3 and Sec. 4) and (reactive) sequence testing (Sec. 5 and Sec. 6). The unit testing scenario is accompanied by a case study in testing a red-black tree implementation, which is reused later in a sequence test case study. In the fourth part (Sec. 7 and Sec. 8), we validate explicit test-hypothesis themselves via tests and via verification, and discuss alternative forms of test-hypothesis for other test-scenarios. Finally, we draw conclusions and discuss related work in the last part (Sec. 9) of the paper.

2. Foundations

2.1. Isabelle

Isabelle [NPW02] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL, which we choose as basis for HOL-TestGen and which is introduced in the subsequent section.

Isabelle's inference rules are based on the built-in meta-level implication $_ \Longrightarrow _$ allowing to form constructs like $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}" and which is written in Isabelle as

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \qquad \frac{A_1 \qquad \cdots \qquad A_n}{A_{n+1}} \,. \tag{5}$$

The built-in meta-level quantification $\bigwedge x.\ Px$ captures the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables. Meta-level quantification leads to a generalization of Horn-clauses of the form:

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \,. \tag{6}$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ, using the Isar [Wen02] language, will look as follows in Isabelle:

```
lemma label:   φ
  apply(case_tac)
  apply(simp_all)
done
```
(7)

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting

proof state. Such a proof state is an implicitly conjoint sequence of generalized Horn-clauses (called *subgoals*) $\phi_1, \ldots, \phi_n$ and a *goal* ϕ. Proof states were usually denoted by:

$$\begin{array}{rl} \text{label}: & \phi \\ 1. & \phi_1 \\ & \vdots \\ n. & \phi_n \end{array} \tag{8}$$

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ at any time; this mechanism helps to generate test theorems. Further, Isabelle supports meta-variables (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification.

2.2. Higher-order Logic

Higher-order logic (HOL) [Chu40, And02] is a classical logic based on a simple type system. It provides the usual logical connectives like $_ \wedge _$, $_ \to _$, $\neg _$ as well as the object-logical quantifiers $\forall x.\ P\,x$ and $\exists x.\ P\,x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $_ = _ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger Arithmetic, and via various integration mechanisms, also external provers such as Vampire and the SMT-solver Z3.

Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way: A theory-extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *wellfounded recursive definitions*.

For example, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:[2]

$$\begin{array}{llll} \textbf{types} & \alpha\ \text{set} & = \alpha \Rightarrow \text{bool} & \\ \textbf{definition} & \text{Collect} & :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha\ \text{set} & \text{— set comprehension} \\ \textbf{where} & \text{Collect}\ S & \equiv S & \\ \textbf{definition} & \text{member} & :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} & \text{— membership test} \\ \textbf{where} & \text{member}\ s\ S & \equiv S s & \end{array} \tag{9}$$

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda\, x.\ P$ and the notation $s \in S$ for member $s\ S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe since it works like an abbreviation. The syntactic side-conditions of this axiom are mechanically checked, of course. It is straight-forward to express the usual operations on sets like $_ \cup _, _ \cap _ :: \alpha\ \text{set} \Rightarrow \alpha\ \text{set} \Rightarrow \alpha\ \text{set}$ as conservative extensions, too, while the rules of typed set-theory were derived by proofs from these definitions.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

$$\begin{array}{lll} \textbf{datatype} & \text{option} & = \text{None} \mid \text{Some}\ \alpha \\ \textbf{datatype} & \alpha\ \text{list} & = \text{Nil} \mid \text{Cons}\ a\ l \end{array} \tag{10}$$

Here, [] or $a\#l$ are an alternative syntax for Nil or Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax

[2] To increase readability, we use a slightly simplified presentation.

for $a\#b\#c\#[]$. These (recursive) statements were internally represented in by internal type- and constant definitions. Besides the *constructors* None, Some, [] and Cons, there is the match-operation case x of None $\Rightarrow F$ | Some $a \Rightarrow G\,a$ respectively case x of $[] \Rightarrow F$ | Cons $a\,r \Rightarrow G\,a\,r$. From the internal definitions (not shown here) a number and properties were automatically derived. We show only the case for lists:

$$
\begin{array}{ll}
(\text{case } [] \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = F & \\
(\text{case } b\#t \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = G\ b\ t & \\
[] \neq a\#t & \text{– distinctness} \\
[\![a = [] \rightarrow P; \exists\ x\ t.\ a = x\#t \rightarrow P]\!] \Longrightarrow P & \text{– exhaust} \\
[\![P[]; \forall\ at.\ Pt \rightarrow P(a\#t)]\!] \Longrightarrow Px & \text{– induct}
\end{array}
\qquad (11)
$$

Finally, there is a compiler for primitive and well-founded recursive function definitions. For example, we may define the sort-operation of our running test example by:

$$
\begin{array}{lll}
\textsf{fun} & \text{ins} & :: [\alpha :: \text{linorder}, \alpha\ \text{list}] \Rightarrow \alpha\ \text{list} \\
\textsf{where} & \text{ins}\ x\ [\,] & = [x] \\
 & \text{ins}\ x\ (y\#ys) & = \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins}\ x\ ys)
\end{array}
\qquad (12)
$$

$$
\begin{array}{lll}
\textsf{fun} & \text{sort} & :: (\alpha :: \text{linorder})\ \text{list} \Rightarrow \alpha\ \text{list} \\
\textsf{where} & \text{sort}\ [\,] & = [\,] \\
 & \text{sort}(x\#xs) & = \text{ins}\ x\ (\text{sort}\ xs)
\end{array}
\qquad (13)
$$

The internal (non-recursive) constant definition for these operations is quite involved; however, the logical compiler will finally derive all the equations in the statements above from this definition and make them available for automated simplification.

Thus, Isabelle/HOL also provides a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. In particular, Isabelle manages a set of *executable types and operators*, i. e., types and operators for which a compilation to SML, OCaml or Haskell is possible. Setups for arithmetic types such as int have been done; moreover any datatype and any recursive function were included in this executable set (providing that they only consist of executable operators). Similarly, Isabelle manages a large set of (higher-order) rewrite rules into which recursive function definitions were included. Provided that this rule-set represents a terminating and confluent rewrite-system, the Isabelle simplifier provides also a highly potent decision procedure for many fragments of theories underlying the constraints to be processed when constructing test-theorems.

2.3. The HOL-TestGen Workflow and System Architecture

Using Isabelle as a symbolic computation environment, i. e., as a framework for implementing HOL-TestGen, allows us to profit from the Isabelle infrastructure in many ways. For example, HOL-TestGen inherits from Isabelle a document-centric workflow: the user extends existing library-theories by a new *test theory* modeling a specific application domain, by test specifications, by proofs for rules that support the overall process and by test set-ups, while the system provides essentially editing and a stepwise validation/execution functionality for these documents. Overall, these documents can be seen as formal and technically checked test plan of a program under test. Fig. 1 shows a screenshot of HOL-TestGen. Besides processing these documents interactively, the user can also process them in batch mode, e. g., for integrating the test data generation into an automated build process of the program under test.

The HOL-TestGen workflow is, conceptually, divided into five distinct phases: first, the *Test Specification Phase* in which die program under test is modeled and the test specification is written. Second, the *Test Case Generation Phase* in which the abstract test cases are generated. Third, in the *Test Data Generation Phase* (also called Test Data Selection Phase) we chose (at least) one representative, i. e., a concrete test data that is processable by the program under test. Fourth, during the *Test Execution Phase*, the implementation is run with the selected test. Finally, during the *Test Result Verification Phase*, the behavior of the program under test is checked against the specification of the test case.

Fig. 2 shows a brief overview over the system architecture supporting this workflow: the first three phases (writing the test specification and the generation of test cases and test data) takes place in an environment based on Isabelle/HOL. Thus, the user of HOL-TestGen can profit from most features (e. g., proofing properties over the test specification, transforming the specification into a form that is more suitable

Fig. 1: A HOL-TestGen session: the right window shows the Proof General interface. The upper-right sub-window allows for interactively stepping through a test theory comprising test specifications while the lower-right sub-window shows the corresponding system state. This may be a proof state in a test theorem development, a list of generated test data or a list of test-hypothesis. After test data generation, a test script is generated that drives the test, resulting in a test trace.

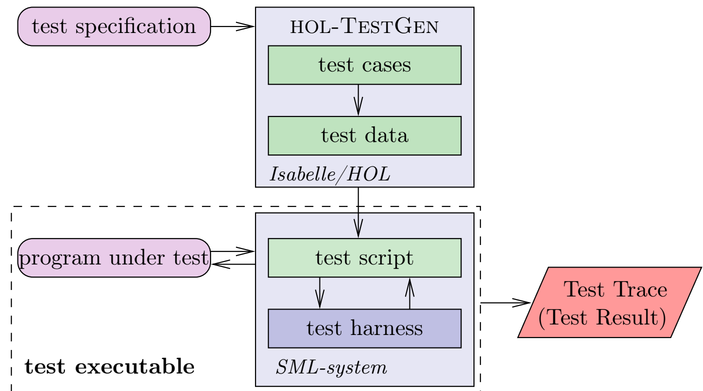


Fig. 2: HOL-TestGen extends the Isabelle-Framework, which itself is based on a SML interpreter. Thus, generated test data can be converted into a test-script to be run inside the latter. The test-script is bound to a test-harness which, if driving the program under test, also generates a statistics which can be included in the overall test document.

for the generation of test cases). After the successful generation of test data, the user can either export a *test script* or a file containing the test data in an XML-like representation. The generated test script is an SML script that, together with a test harness provided by HOL-TestGen, can be executed independently from HOL-TestGen using an arbitrary SML compiler.[3] By exploiting the various foreign language interfaces of the different SML compilers, this allows for an automated setup for testing implementations in programming languages such as Java, C, SML, any language running on the .net environment, or implementations accessible via Web service calls (e. g., based on widely-used standards such as **wsdl!**). Exporting the test data using an XML-like representation allows for using the test data together with domain-specific test drivers, e. g., for testing the compliance of network firewalls.

3. The Approach to Test Case Generation and Test Data Selection

As input of the test case generation phase, the *test specification*, one might expect a special format like $\text{pre}(x) \to \text{post } x\ (PUT(x))$. This rules out trivial instances such as $3 < PUT(x)$ or just $PUT(x)$ (meaning that PUT must evaluate to true for x). We do not impose any other restriction on a specification other than the final test statements being executable, i. e., the result of the process can be compiled into a test program.

Processing a test specification, our test case generation procedure (called gen_test_case_tac) can be separated into the following phases which were organized to the conceptual algorithm shown in Fig. 3. The phases are implemented by tactics that are largely re-configurable.

[3] As the code generator of HOL-TestGen is based on the code-generator framework provided by Isabelle/HOL, we can quite easily generate test scripts in languages such as Scala, F#, Haskell, or OCaml.

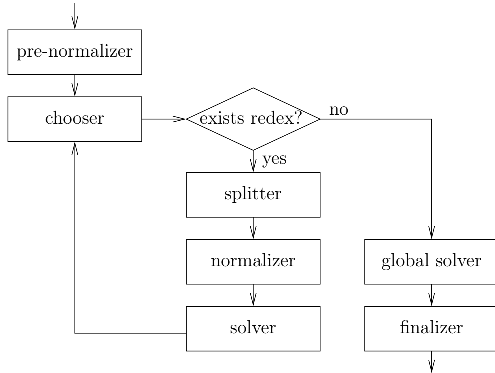


Fig. 3: A high-level description of the algorithm: after a chooser-phase, where subterms were marked for splitting (default: free variables in the test specification), a splitter introduces case-splits of the clauses in a proof state (default: datatype exhaustion theorem, conditionals), while the normalizer brings the list of clauses into TNF . Several solvers attempt to eliminate clauses with unsatisfiable constraints (representing vacuous test cases), and tries to eliminate redundant (subsumed) cases. The finalizer simplifies again the logical structure of the testing theorem by introducing explicit uniformity-hypothesis.

The Pre-normalizer is an initial phase where definitions of the test specification may be unfolded. Its default is just a simplification tactic.

The Chooser selects (splitting) "redexes," i. e., subterms in the current clause lists on which case-splitting rules will be applied. The default are free variables of a type stemming from a datatype definition such as α list, if _ then _ else _ expressions as well as matching expressions case _ of $[] \Rightarrow _ \mid (_\#_) \Rightarrow _$. The chooser also produces heuristically a ranking among these splitting redexes.

The Splitter executes case-splitting rules for the selected redexes. In the default, this includes the generation of datatype exhaustion theorems as discussed in Sec. 3.1.1, or splitting rewrites (see Tab. 1e).

The Normalizer applies the tableaux calculus (see Tab. 1) to split the list of subgoals into *Horn-clause normal form* (HCNF). Finally, by re-ordering the clauses, the calls of the program under test are rearranged such that they occur only in the conclusion, where they must occur at least once. These re-ordered HCNF clauses are called to be in *testing normal form* (TNF), if the conclusion is an executable term.

The Solver attempts to eliminate horn-clauses with unsatisfiable constraints. In the default, this is configured as just a rewriter. A finally applied variant of the solver, which applies a more powerful combination of Isabelle decision procedures, is applied when no more redexes have been found. This final solving attempts also tries to eliminate redundant cases.

The Finalizer introduces for all remaining free variables the uniformity-hypothesis (cf. Sec. 3.1.2).

The gen_test_case_tac procedure performs these steps until no more redexes were found. In the subsequent sections, we discuss two key components of the overall test case generation process, namely two test specific rule-schemata as well as the normalizer constructing the actual test cases. We will briefly sketch the constraint solver used to find concrete instances of a test case, and conclude with a discussion of coverage criteria.

3.1. Test Cases Generation with Explicit Test-hypothesis

We apply two test specific rule schema that start respectively finalize the normalization process. These rule schema introduce certain subformula which can be seen as a *testing-hypothesis* or proof-obligation and encapsulates them via a constant symbol THYP (which is semantically defined as just an identity) from the rest of the test cases. Following the terminology of Gaudel [Gau95], we distinguish *regularity* and *uniformity*-hypothesis. Note, however, that the explicit use of the hypothesis as proof-obligation inside the logic, even inside the test-theorem is specific to our framework. These two kinds of hypothesis are configured as default into our system, but alternative test-hypothesis are discussed in Sec. 8.

3.1.1. Using Regularity-hypothesis in Splitting.

In the following, we address the problem of test case generation for universally quantified (or, equivalently, free variables) ranging over recursive datatypes such as lists or trees. For testing recursive data structures,

the following form of a *regularity-hypothesis* [Gau95] has been suggested:

$$\frac{\begin{matrix}[|x| < k] \\ \vdots \\ P\,x\end{matrix}}{P\,x} \tag{14}$$

This rule formalizes the hypothesis: assuming that a predicate P is true for all data x whose *size* (denoted by $|x|$) is less than a given depth k, P is always true. The original rule can be viewed as a meta-notation: In a rule for a concrete datatype, the premises $|x| < k$ can be expanded to several premises enumerating constructor terms.

Instead of this (deliberately) unsound rule, HOL-TestGen *derives* on-the-fly a special datatype exhaustion theorem; its form depends on the *depth* d and the structure of the datatype of x. For the user-defined value $d = 3$ and for the type α list, we have:

$$\frac{\begin{matrix}[x = [\,]] \\ \vdots \\ P(x)\end{matrix} \qquad \bigwedge a.\ \begin{matrix}[x = [a]] \\ \vdots \\ P(x)\end{matrix} \qquad \bigwedge a\ b.\ \begin{matrix}[x = [a,b]] \\ \vdots \\ P(x)\end{matrix} \qquad \bigwedge a\ b\ c.\ \begin{matrix}[x = [a,b,c]] \\ \vdots \\ P(x)\end{matrix} \qquad \mathrm{THYP}(H)}{P(x)} \tag{15}$$

where the explicit test-hypothesis "regularity" has the form $H = (\forall x.\ |x| < 4 \to P(x)) \to \forall x.\ P\,x$.

In the sequel, we will show the effect of the datatype exhaustion theorem on our running example presented in the introduction. The presentation of the testing theory, in our case the definition of the datatype list and the recursive function definitions ins and sort Fact 12, is already complete. The test specification "the program under test should be a sorting algorithm" is straight-forward:

$$\text{testspec test:} \quad PUT(x) = \mathrm{sort}(x) \tag{16}$$

The chooser will detect as redex the free variable x of type list; the splitter will apply the datatype exhaustion theorem accordingly. The resulting proof state reads as follows:

$$\begin{array}{rl} \text{test :} & PUT(x) = \mathrm{sort}(x) \\ 1. & PUT([]) = \mathrm{sort}([]) \\ 2. & \bigwedge a.\ PUT([a]) = \mathrm{sort}([a]) \\ 3. & \bigwedge a\ b.\ PUT([a,b]) = \mathrm{sort}([a,b]) \\ 4. & \bigwedge a\ b\ c.\ PUT([a,b,c]) = \mathrm{sort}([a,b,c]) \\ 5. & \mathrm{THYP}(\forall x.\ |x| < 4 \to PUT(x) = \mathrm{sort}(x)) \to \forall x.\ PUT(x) = \mathrm{sort}(x)) \end{array} \tag{17}$$

Elementary rewriting by the definitions of sort in Fact 12 and the normalization process described in Sec. 3.2 will turn our test specification into the final test-theorem.

3.1.2. Using Uniformity-hypothesis in the Finalizer.

Uniformity-hypothesis have the form:

$$\mathrm{THYP}(\exists x_1 \ldots x_n.\ P\,x_1, \ldots, x_n \to \forall x_1 \ldots x_n.\ P\,x_1 \ldots x_n) \tag{18}$$

and were used in the finalizer phase of the test-generation procedure. Semantically, this kind of hypothesis expresses the following: whenever a test case is passed successfully for one data of this test case, the program behaves correctly for *all* data of this test case. The derived rule in natural deduction format expressing this kind of test theorem transformation reads as follows:

$$\frac{P\ ?x_1 \ldots ?x_n \qquad \mathrm{THYP}(\exists x_1 \ldots x_n.\ P\,x_1 \ldots x_n \to \forall x_1 \ldots x_n.\ P\,x_1 \ldots x_n)}{\forall x_1 \ldots x_n.\ P\,x_1 \ldots x_n} \tag{19}$$

where the $?x_i$ are just meta variables, i. e., place-holders for arbitrary terms. This rule can also be applied for arbitrary formulae containing free variables since universal quantifiers may be introduced for them.

In contrast to our presentation in Fact 4, we do not use existential quantifiers in the test-theorem to mark test cases; rather, we use meta-variables and meta-implications which can be processed by Isabelle's deduction engine directly.

$$\frac{P\ ?x}{\exists x.\ P\ x} \qquad \frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}$$

(a) Quantifier Introduction Rules

$$\frac{}{t = t} \qquad \frac{}{\text{true}} \qquad \frac{P \quad Q}{P \wedge Q} \qquad \frac{\begin{matrix}[\neg Q]\\ \vdots \\ P\end{matrix}}{P \vee Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \rightarrow Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ \text{false}\end{matrix}}{\neg P} \qquad \frac{\begin{matrix}[P] & [Q]\\ \vdots & \vdots \\ Q & P\end{matrix}}{P = Q}$$

(b) Safe Introduction Rules

$$\frac{\forall x.\ P\ x \quad \begin{matrix}[P\ ?x]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{\forall x.\ P\ x \quad \begin{matrix}[\forall x.\ P\ x,\ P\ ?x]\\ \vdots \\ R\end{matrix}}{R}$$

(c) Unsafe Elimination Rules

$$\frac{\text{false}}{P} \qquad \frac{P \wedge Q \quad \begin{matrix}[P, Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \vee Q \quad \begin{matrix}[P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \rightarrow Q \quad \begin{matrix}[\neg P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{\exists x.\ P\ x \quad \bigwedge x.\ \begin{matrix}[P\ x]\\ \vdots \\ Q\end{matrix}}{Q} \qquad \frac{P = Q \quad \begin{matrix}[P, Q]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[\neg P, \neg Q]\\ \vdots \\ R\end{matrix}}{R}$$

(d) Safe Elimination Rules

$$P(\text{if } C \text{ then } A \text{ else } B) = (C \rightarrow P(A)) \wedge (\neg C \rightarrow P(B))$$
$$P(\text{case } x \text{ of Nil} \Rightarrow F \,|(a\#r) \Rightarrow G\ a\ r) = (x = [] \rightarrow P(F)) \wedge (\exists a\ t.\ x = a\#t \rightarrow P(G\ a\ t))$$

(e) (Splitting)-Rewrites

Tab. 1. The Standard Tableaux Calculus for HOL

3.2. Normal Form Computations

At the heart of the test case generation, i. e., the generation of the testing theorem, lies a normal form computation process similar to the DNF-computation pioneered by Dick and Faivre [DF93]. In contrast to the latter, however, we chose to adopt a Horn-clause normal form (HCNF) used in the usual Isabelle proof states. In a classical logic like HOL, Horn-clauses like: $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$ are logically equivalent to $\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1}$. Therefore, the HCNF can be viewed as a conjunctive normal form (CNF). We will interpret the subgoals of a proof state as test cases, and view the assumptions A_i of each subgoal as *constraints* restricting the valid input of a test case.

In the following, we describe the tableaux, rewriting and testing normal form computations in more detail. In Isabelle/HOL, the automated proof procedures for HOL formulae depend heavily on tableaux calculi [DGHP96] presented as (derived) natural deduction rules. Tab. 1 presents the core tableaux calculus of HOL. With the notable exception of the elimination rule for the universal quantifier (see Tab. 1c), any rule application leads to a logically equivalent proof state: therefore, all rules (except $\forall$ elimination) are called *safe*. When applied bottom up in backwards reasoning (which may introduce meta-variables explicitly marked in Tab. 1), the technique leads in a deterministic manner to a HCNF. Note, however, that test cases are not necessarily minimal: there may be test cases that overlap. In practice, however, this occurs seldom in specifications that are based on distinct constructors of data types.

Coming back to our running example sort, the proof-state shown in Fact 17 is transformed in the nor-

malization phase as follows. Rewriting the definitions of sort yields:

$$\begin{array}{rl} \text{test}: & PUT(x) = \text{sort}(x) \\ 1. & PUT([]) = [] \\ 2. & \bigwedge a.\ PUT([a]) = \text{ins}\ a\ [] \\ 3. & \bigwedge a\,b.\ PUT([a,b]) = \text{ins}\ a\ (\text{ins}\ b\ []) \\ 4. & \bigwedge a\,b\,c.\ PUT([a,b,c]) = \text{ins}\ a\ (\text{ins}\ b\ (\text{ins}\ c\ [])) \\ 5. & \text{THYP}(\forall x.\ |x| < 4 \to PUT(x) = \text{sort}(x)) \to \forall x.\ PUT(x) = \text{sort}(x)) \end{array} \tag{20}$$

and further rewrite steps unfolding ins result in:

$$\begin{array}{rl} \text{test}: & PUT(x) = \text{sort}(x) \\ 1. & PUT([]) = [] \\ 2. & \bigwedge a.\ PUT([a]) = [a] \\ 3. & \bigwedge a\,b.\ PUT([a,b]) = \text{if}\ a \leq b\ \text{then}\ [a,b]\ \text{else}\ [b,a] \\ 4. & \bigwedge a\,b\,c.\ PUT([a,b,c]) = \text{ins}\ a\ (\text{if}\ b \leq c\ \text{then}\ [b,c]\ \text{else}\ [c,b]) \\ 5. & \text{THYP}(\forall x.\ |x| < 4 \to PUT(x) = \text{sort}(x)) \to \forall x.\ PUT(x) = \text{sort}(x)) \end{array} \tag{21}$$

This proof-state is in normal-form, the overall algorithm continues therefore executing the main loop shown in Fig. 3. The chooser picks in this iteration the conditionals in subgoals 3 and 4, while the splitter uses the splitting rewrites together Tab. 1e with the safe introduction rules in Tab. 1b to compute the following successor proof-state (some elementary rewriting on arithmetic is omitted here):

$$\begin{array}{rl} \text{test}: & PUT(x) = \text{sort}(x) \\ 1. & PUT([]) = [] \\ 2. & \bigwedge a.\ PUT([a]) = [a] \\ 3. & \bigwedge a\,b.\ [\![a \leq b]\!] \Longrightarrow PUT([a,b]) = [a,b] \\ 4. & \bigwedge a\,b.\ [\![b < a]\!] \Longrightarrow PUT([a,b]) = [b,a] \\ 6. & \bigwedge a\,b\,c.\ [\![b \leq c]\!] \Longrightarrow PUT([a,b,c]) = \text{ins}\ a\ [b,c] \\ 7. & \bigwedge a\,b\,c.\ [\![c < b]\!] \Longrightarrow PUT([a,b,c]) = \text{ins}\ a\ [c,b] \\ 8. & \text{THYP}(\forall x.\ |x| < 4 \to PUT(x) = \text{sort}(x)) \to \forall x.\ PUT(x) = \text{sort}(x)) \end{array} \tag{22}$$

After a few further iterations (and the finalization phase introducing the clauses representing the used uniformity hypothesis), our algorithm will result in the test-theorem shown in Fact 4.

Our test specifications may contain higher-order constants and all sorts of bounded quantifiers (e. g., over lists; their elimination is part of the rewrite rule set not discussed in detail here). Moreover, the procedure also works for unbounded quantifiers ranging over datatypes (although in the default setup, only universal quantifiers in positive occurrence and existential quantifier in negative occurrence will be selected in the chooser). However, the procedure leaves quantifiers of other types (such as higher-order function types or sets) unchanged and leaves it to suitable (user-programmed) procedures in the constraint solver.

The chooser also performs an internal bookkeeping of the variables introduced in the process; thus, a splitting of meta-quantified variables a, b, c introduced by the datatype exhaustion theorem is avoided to ensure termination. Finally, observe that the number of test cases that the algorithm constructs is finite, but test cases in itself have usually infinitely many witnesses (test data). This is in sharp contrast to all model-checking related approaches that attempt to approximate infinite datatypes early, and usual in an ad-hoc manner.

A final normalization step brings the proof state in HCNF into a particular variant of it. In particular, this final transformation eliminate subgoals like:

$$[\![\neg(PUT\ x = c);\ \neg(PUT\ x = d)]\!] \Longrightarrow A_{n+1}\,, \tag{23}$$

and transform them into the equivalently clause:

$$[\![\neg(A_{n+1})]\!] \Longrightarrow PUT\ x = c \lor PUT\ x = d\,. \tag{24}$$

We call this form of Horn-clauses *testing normal form* (TNF), if after the normalization the conclusions of all horn-clauses are executable. Not all specifications can be converted to TNF. For example, if the specification does not make a suitably strong constraint over program PUT, in particular if PUT does not occur in the specification. In such cases, gen_test_case stops with an exception.

3.3. Test Data Generation by Constraint Solving

The test data generator called gen_test_data implements the test data selection phase. It extracts from a given test theorem the constraints of each test case and starts a constraint resolution phase for the latter. Our constraint solver consists of a chain of solvers, filtering smaller constraints from more complex ones. The first level is represented by auto [Pau99] (an Isabelle standard tactic combining a tableaux-prover with a rewrite engine and a linear arithmetic procedure). Remaining unsolved constraints were passed to the second level, an own symbolic random-solver (a number of random ground instances were substituted for the variables occurring in the constraint which is then passed auto). The next level is the compiling random-solver quick_check [BN04] (an Isabelle standard procedure that compiles all constraints to code and searches solutions by test-and-verify random values). Finally, we extended an integration of external SMT-solvers available in recent versions of Isabelle, in particular as Z3[BTV09], by constructing from its counter-models substitutions and by verifying them inside Isabelle.

The precise order of solvers and the number of repetitions is user-defined and highly reconfigurable. The choice for the default order sketched above is entirely pragmatic—it turned out to be the fastest for the examples we check, and actually changed over the years according to the availabilities and the increasing power of its components.

Unresolved constraints (marked by RSF in our examples) where still represented in test data statements and thus mark possibly inconclusive tests in the test-execution phase.

The test case generation phase can be very costly in some realistic examples; in others, it is the test data generation which is the bottle neck. Massaging a test theorem into a form that permits the solver to solve all constraints is tantamount for using HOL-TestGen effectively. This form of massage, possibly resulting in new, hand-proven lemmas or axiomatically stated facts to be inserted into the test case generation and test data generation procedure is *the* activity that gives HOL-TestGen an interactive flavor, but also makes the system so powerful.

In our example Fact 21, gen_test_data produces the 9 ground instances for the non-trivial test cases in a fraction of a second; in this case, the work is solely done by our symbolic random-solver procedure.

3.4. Test-adequacy and Theoretical Properties

In the following, we discuss the theoretical and practical properties, e. g., the underlying test-adequacy criteria, of our black-box test case generation approach. Obviously, the heart of it is a decomposition of the test specification into a normal form and the construction of test cases for each of its clauses. This is in the tradition of [BGM91] and the work of Dick and Faivre [DF93]. Besides the conceptually minor difference that our basic TNF is essentially a conjunctive normal form (CNF), there is the major difference that we strive to solve a (SMT) problem for each clause (i. e., test case), and that each clause is also normalized with respect to if _ then _ else _ and datatype induced case-statements.

Definition 3.1 ($\mathbf{TNF_{E/d}}$(TS) Normal Test Specifications). The *testing normal form* (TNF) modulo a theory E of depth d from a test specification TS has the following properties:

1. all constraints $C_{i,1}, \ldots, C_{i,k}$ do neither contain if _ then _ else _ nor datatype induced case-statements, i. e., they are *fully splitted*,
2. all datatype-generated free variables in TS were splitted at least d times, i. e.. at least d times an exhaustion rule must have been applied to this variable of its descendants,
3. the oracles have the form $P_1(PUT, c_1, \ldots, c_k) \vee \cdots \vee P_m(PUT, c_1, \ldots, c_k)$, where the P_j are closed and executable,
4. all constraints of all clauses (i. e., test cases) are satisfiable modulo E; i. e., $\exists x_1, \ldots, x_k.\ C_{i,1}(x_1, \ldots, x_k) \wedge \cdots \wedge C_{i,k}(x_1, \ldots, x_k)$ are true, and
5. all test-hypothesis are non-redundant, i. e., $\mathrm{THYP}(X) \neq \mathrm{true}$.

TNF_E is essentially a conjunctive normal form (CNF) since existential quantifiers can be eliminated via meta-variables, and the implications into disjunctions.

Definition 3.2 ($\mathbf{TNF_{E/d}}$-Test-adequacy for TS). A set of test cases for a test specification TS is TNF_E-

adequate, if a $TNF_{E/d}(TS)$ normal form could be computed for TS and the set of test cases contains at least one test case for each clause.

Theorem 3.1. HOL-TestGen approximates $TNF_{E/d}$-adequacy.

Proof sketch. For the case that we have a complete decision procedure for E, for example, for a Noetherian and convergent set of rewrite rules for the entire theory used in the test specification, the proposition follows by construction. The question arises, what happens if solvers fail (are not a decision procedure). In these cases, there are more clauses with undetected unsatisfiable constraints, or satisfiable constraints for which the constraint solver are unable to construct a solution. These cases are explicitly marked in the resulting test-driver and will result in "inconclusive tests," i. e., tests which require further human inspection. □

Theorem 3.2. HOL-TestGen is a correct testing procedure, i. e., if a test theorem of the form

$$\frac{TC_1 \cdots TC_n \qquad H_1 \cdots H_m}{TS}\,, \tag{25}$$

is constructed with all TC_i in $TNF_{E/d}(TS)$, then the implication $TC_1 \wedge \cdots \wedge TC_n \wedge H_1 \wedge \cdots H_m \rightarrow TS$ is logically valid.

Proof sketch. The entire procedure is based on the application of derived rules in HOL. (We assume consistency of HOL and its correct implementation in Isabelle; However, if one has serious doubts into the latter, it is perfectly possible to generate for the entire derivation of the test-theorem a proof object for HOL and check the latter independently from Isabelle). □

Theorem 3.3. HOL-TestGen is a complete testing procedure, i. e., $TS \rightarrow TC_1 \wedge \cdots \wedge TC_n \wedge H_1 \wedge \cdots H_m$ holds.

Proof sketch. The construction of the normal form uses only the "safe" (i. e., logically equivalent) rules of Tab. 1, plus rewrite rules for the user-defined operations. □

Running our sorting example on standard hardware requires less than a second for depth $d = 3$ (10 cases), less than five seconds for depth $d = 4$ (34 cases). For depth $d = 5$ (154 cases) the generation already requires around 15 minutes. At the first glance, this seems to indicate that the HOL-TestGen approach does not scale well. Especially, as random testing tools like QuickCheck [CH00] promise to check similar properties with several thousands of test cases within 15 minutes. We argue, however, that an in-depth analysis of the situation refutes this conclusion: 1. on average, a purely random testing approach needs to check 4 000 000 test cases[4] to hit the 153 cases of the $TNF_{E/5}$, 2. in many application scenarios of model-based testing, a small number of significant tests is crucial to make testing practically feasible, and 3. $TNF_{E/5}$ is indeed what the sorting problem imposes, that the algorithmic structure of the problem motivates a certain structure of the test cases. While the efficiency of QuickCheck can be improved by *manually* providing specialized test case generators, our approach reveals the underlying problem structure *automatically.*

Finally, the global solver also attempts to eliminate redundant test cases. Since this analysis is costly and in general impossible—subsumption of a test case ϕ in a test case ψ boils down to decide $\psi \rightarrow \phi$—we have to live with the fact that test cases are *not* partitions and we will have more test data in practice than needed in a minimal set of test cases in which the classes of solutions are strictly disjoint. Our procedure is well-behaved for medium-size examples shown throughout this paper. The effect of generating redundant test cases can be annoying in very large examples in our experience.

4. Case Study: Unit-testing Red-black Trees

In this section, we show the standard application scenario of HOL-TestGen: the generation of test data for black box testing of side-effect free programs. As mentioned earlier, unit-test specifications have the following scheme:

$$\text{testspec}: \quad \text{pre}(x) \Rightarrow \text{post}\,(x)\;\; PUT(x) \tag{26}$$

[4] For lists of length n, we generate $n!$ test cases (every permutation). A purely random testing-bases approach that generates lists of length n for integers up to k needs to generate $\binom{k}{n}$ test cases for ensuring the inclusion of all $n!$ permutations.

where pre and post refer to pre-and postconditions over input variables and results, and $PUT(x)$ represents logically the result of the program under test. We will instantiate this scheme and present a test development by running through the different phases. In particular, we will emphasize the interactive aspects of the test-plan development. As a target for testing, we chose the red-black implementation of the sml/NJ (`http://www.smlnj.org`) library. We will also show *how* errors are detected and how test data can be generated that explores the program under test to a satisfactory degree.

4.1. The Test Specification

Red-black trees store the balancing information in one additional bit per node, which is called the "color of a node." This is either red or black. A valid (balanced) red-black tree must fulfill the following three invariants:

Red Invariant: each red node has a black parent.
Strong Red Invariant: the root is black *and* the red invariant holds.
Black Invariant: each path from the root to a leaf has the same number of black nodes.

An invariant can be represented as recursive predicate; as a prerequisite, we first specify the data structure tree: We start by specifying the datatype for red-black trees:

$$\begin{array}{lll} \text{types} & \alpha\ \text{item} & = \alpha :: \text{order} \\ \text{datatype} & \alpha\ \text{color} & = \text{R} \mid \text{B} \\ \text{datatype} & \alpha\ \text{tree} & = \text{E} \mid \text{T color } (\alpha\ \text{tree})\ (\alpha\ \text{item})\ (\alpha\ \text{tree}) \end{array} \tag{27}$$

Here, $\alpha :: \text{order}$ requires that the type α is a member of the type class order. Thus, α tree can store items of any type on which an ordering is defined. In the following definition, we present the recursive predicate for the red invariant:

$$\begin{array}{lll} \text{definition} & \text{redinv} & :: \alpha\ \text{tree} \Rightarrow \text{bool} \\ \text{where} & & \\ & \text{redinv E} & = \text{true} \\ \mid & \text{redinv } (\text{T B } a\, y\, b) & = (\text{redinv } a \wedge \text{redinv } b) \\ \mid & \text{redinv } (\text{T R } (\text{T R } a\, x\, b)\, y\, c) & = \text{false} \\ \mid & \text{redinv } (\text{T R } a\, x\ (\text{T R } b\, y\, c)) & = \text{false} \\ \mid & \text{redinv } (\text{T R } a\, x\, b) & = (\text{redinv } a \wedge \text{redinv } b) \end{array} \tag{28}$$

Similarly, we are formalizing the strong red invariant and the black invariant:

$$\begin{array}{lll} \text{definition} & \text{strong_redinv} & :: \alpha\ \text{tree} \Rightarrow \text{bool} \\ \text{where} & & \\ & \text{strong_redinv E} & = \text{true} \\ \mid & \text{strong_redinv } (\text{T R } a\, y\, b) & = \text{false} \\ \mid & \text{strong_redinv } (\text{T B } a\, y\, b) & = (\text{redinv } a \wedge \text{redinv } b) \end{array} \tag{29}$$

$$\begin{array}{lll} \text{definition} & \text{blackinv} & :: \alpha\ \text{tree} \Rightarrow \text{bool} \\ \text{where} & & \\ & \text{blackinv E} & = \text{true} \\ \mid & \text{blackinv } (\text{T } c\, a\, y\, b) & = ((\text{blackinv } a) \wedge (\text{blackinv } b) \\ & & \quad \wedge ((\text{max_B_height } a) = (\text{max_B_height } b))) \end{array} \tag{30}$$

Where max_B_height is a predicate determining the maximum path length of black nodes:

$$\begin{array}{lll} \text{definition} & \text{max_B_height} & :: \alpha\ \text{tree} \Rightarrow \text{nat} \\ \text{where} & \text{max_B_height E} & = 0 \\ \mid & \text{max_B_height } (\text{T B } a\, y\, b) & = \text{Suc}(\max(\text{max_B_height } a)(\text{max_B_height } b)) \\ \mid & \text{max_B_height } (\text{T R } a\, y\, b) & = (max(\text{max_B_height } a)(\text{max_B_height } b)) \end{array} \tag{31}$$

Moreover, we define a test for element hood:

$$\begin{array}{lll} \text{fun} & \text{isin} & :: (\alpha :: \text{order})\ \text{item} \Rightarrow \alpha\ \text{tree} \Rightarrow \text{bool} \\ \text{where} & \text{isin } x \text{ E} & = \text{false} \\ \mid & \text{isin } x\ (\text{T } c\, a\, y\, b) & = (x = y) \vee (\text{isin } x a) \vee (\text{isin } x b)) \end{array} \tag{32}$$

Assume we want to test that insertion or deletion (summarized by the place-holder PUT) fulfill the black invariant. Hence, we are searching for test data fulfilling the premise of the following test specification:

$$\text{testspec test:} \quad \text{isord}\, t \wedge \text{isin}\, y\, t \wedge \text{strong_redinv}\, t \wedge \text{blackinv}\, t \longrightarrow \text{blackinv}\big(PUT(y,t)\big) \tag{33}$$

where strong_redinv and blackinv are predicates formalizing the strong red invariant and the black invariant. Moreover, isord is a recursive predicate that is true if and only if the red-black tree t is ordered.

4.2. Brute Force

4.2.1. Test Case Generation

Now we can automatically generate *test cases* in a model checking-like approach by applying the method gen_test_cases. This method generates data-structures (here: trees) up to a certain depth and performs case splitting over all possible cases; remaining constraints are simplified. The default depth-parameter of the method is set to three. Finally, the resulting test theorem is stored in a *test environment*. For the program under test $ioprog(y,t)$, the complete test script looks as follows:

$$\begin{array}{l}\text{testspec test:} \quad \text{isord}\, t \wedge \text{isin}\, y\, t \wedge \text{strong_redinv}\, t \wedge \text{blackinv}\, t \longrightarrow \text{blackinv}\big(\mathit{ioprog}(y,t)\big)\\ \quad \text{apply}(\text{gen_test_cases}\ \mathit{ioprog})\\ \text{store_test_thm red_and_black_inv}\end{array} \tag{34}$$

This fairly simple setup generates, in less than a second, already 25 subgoals containing 12 test cases, altogether with non-trivial constraints, among them:

$$1.\ [\![x_1 = x_2]\!] \Longrightarrow \text{blackinv}\big(\mathit{ioprog}(x_1, \text{T B E}\, x_2\ \text{E})\big) \tag{35}$$

$$\begin{array}{l}2.\ [\![x_1 = x_6; \text{blackinv}\, x_4;\ \text{redinv}(\text{T}\, x_5\, x_4\, x_3\, x_2);\ \text{isord}\, x_2;\ \text{isord}\, x_4;\\ \quad \text{max_B_height}(\text{T}\, x_5\, x_4\, x_3\, x_2) = 0;\ \text{blackinv}\, x_2;\ \text{max_B_height}\, x_4 = \text{max_B_height}\, x_2;\\ \quad \forall x.\ (x = x_3 \longrightarrow x_6 < x) \wedge (\text{isin}\, x\, x_4 \longrightarrow x_6 < x) \wedge (\text{isin}\, x\, x_2 \longrightarrow x_6 < x);\\ \quad \forall x.\ \text{isin}\, x\, x_2 \longrightarrow x_3 < x;\ \forall x.\ \text{isin}\, x\, x_4 \longrightarrow x < x_3\\ \ \ ]\!] \Longrightarrow \text{blackinv}(\mathit{ioprog}(x_1, \text{T}\, B\, \text{E}\, x_6\ (\text{T}\, x_5\, x_4\, x_3\, x_2)))\end{array} \tag{36}$$

Here, the first test case (Fact 35) tests that deleting the root note of a red-black tree of height one preserves the black invariant.The second test case (Fact 36) tests the same property enforcing a valid (i. e., the tree is sorted and both the strong red invariant and the black invariant holds) red-black of height two.

An example for a generated uniformity test-hypothesis is:

$$\begin{array}{l}\text{THYP}\Big(\Big(\exists x\ xa.\ x = xa \longrightarrow \text{blackinv}\big(\mathit{ioprog}(x, \text{T B E}\, xa\, \text{E})\big)\Big)\\ \qquad\qquad \longrightarrow \Big(\forall x\ xa.\ x = xa \longrightarrow \text{blackinv}\big(\mathit{ioprog}(x, \text{T B E}\, xa\ \text{E})\big)\Big)\Big)\end{array} \tag{37}$$

This instance of a uniformity test-hypothesis describes that if the program under tests is correct for an arbitrary red-black tree of height one, it is correct for all red-black trees of height one.

4.2.2. Test Data Generation

Generating concrete test data already takes notable time (i. e., a few minutes), as it's quite unlikely that the random solver generates values that fulfill these ordering constraints. Therefore we restrict the attempts (*iterations*) the random solver takes for solving a single test case to 40:

$$\begin{array}{l}\text{testgen_params [iterations=40]}\\ \text{gen_test_data red_and_black_inv}\end{array} \tag{38}$$

which unfortunately fails to solve all constraints, e. g., we obtain test cases like

$$\text{RSF} \longrightarrow \text{blackinv}\ (\mathit{ioprog}\ (100, \text{T B E}\ 7\, \text{E})) \tag{39}$$

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

Summary:
 Number successful tests cases: 7 of 12 (ca. 58%)
 Number of warnings:            4 of 12 (ca. 33%)
 Number of errors:              0 of 12 (ca. 0%)
 Number of failures:            1 of 12 (ca. 8%)
 Number of fatal errors:        0 of 12 (ca. 0%)
```

Tab. 2. Running the test executable for red_and_black_inv already reveals an error in the sml/NJ library (see Test 9). Moreover, we see successful test cases and warnings caused by unresolved cased (where the random solver returns RSF as precondition).

$$\text{RSF} \longrightarrow \text{blackinv}\,(\textit{ioprog}\,(83, \text{T B}\,(\text{T B}\,(\text{T B E} -8\ \text{E})\ 57\,(\text{T R E}\ 13\ \text{E}))\ -62\ \text{E})) \tag{40}$$

$$\text{blackinv}\,(\textit{ioprog}\,(-91, \text{T B}\,(\text{T R E} -91\ \text{E})\ 5\ \text{E})) \tag{41}$$

$$\text{RSF} \longrightarrow \text{blackinv}\,(\textit{ioprog}\,(-33, \text{T B}\,(\text{T R E} -2\ \text{E})\ 37\ \text{E})) \tag{42}$$

where RSF is a pretty-printing trick for all formula unequal true, i. e., constraints that could not be resolved. Thus, very few of these test cases will lead to conclusive tests. To compute more conclusive test data, we can increase the number of iterations to more than 150 to find reliably a sufficiently conclusive set of test data which takes notably more time, i. e. about 40 seconds, and about 300 seconds for 600.

4.2.3. Test Script Generation

Now we generate the test script for *ioprog*, i. e., a test script that allows for testing an implementation of the delete functions of a red-black tree data structure. During this step, we often need to map the abstract name for the tested program, e. g., *ioprog*, to the name of the implementation on the programming level. In the following, we assume that the delete operation of red-black trees is implemented by the SML function `delete`:

$$\text{gen_test_script}\ \texttt{rbt_script.sml}\ \text{red_and_black_inv}\ \textit{ioprog}\ \texttt{delete} \tag{43}$$

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 implementations, e. g., Java, or C, is supported. Depending on the SML-system, the test execution is run within an interpreter or using a compiled test executable. Testing implementations written in SML is straight-forward.

4.2.4. Test Result Verification

Running the test executable for red_and_black_inv results in an output similar to Tab. 2, showing successful test cases, failures (i. e., the implementation violates the postcondition) and warning caused by test cases for which the constraint could not be solved during the data generation phase (i. e., where the random solver returns RSF as precondition). In the latter case, the *ioprog* is still executed (and throws in our example an exception). Already in this highly automatic set-up, we were able to produce the reported error in the sml/NJ library [BW04] (for an illustration, see Fig. 4). However, based on the test depth tree (which represents the limit of the standard approach if we restrict ourselves to a time investment of 5 minutes for the overall run) we cannot have trees with more than three nodes on the level of the test case generation. Of course, random solving increases the depth of the trees sporadically, as can be seen from the test result, but in an

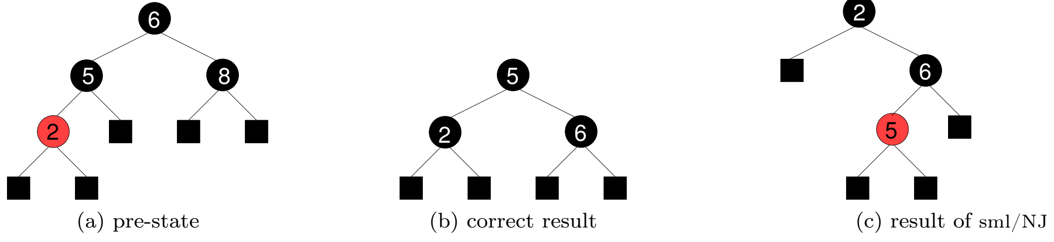


Fig. 4. This test case for deleting a node in the red-black tree reveals an error in the sml/NJ library: the black invariant does not hold after the deletion.

unsystematic way. Thus, the program under test has obviously not been tested satisfactorily, and we need means to treat test data sets with higher depth.

4.3. Using a Little Theorem Proving

The question arises how this problematic aspect of ingeniously added abstract test data can be overcome and be systematized for our example. One answer is a characterization theorem of redinv:

$$
\begin{aligned}
\mathrm{redinv}\, x = \big((x = \mathrm{E})& \\
&\vee (\exists a\, y\, b.\; x = \mathrm{T\,B}\, a\, y\, b \wedge \mathrm{redinv}\, a \wedge \mathrm{redinv}\, b) \\
&\vee (\exists y.\; x = \mathrm{T\,R\,E}\, y\, \mathrm{E}) \\
&\vee (\exists y\; am\; an\; ao.\; x = \mathrm{T\,R\,E}\, y\, (\mathrm{T\,B}\, am\; an\; ao) \wedge \mathrm{redinv}(\mathrm{T\,B}\, am\; an\; ao)) \\
&\vee (\exists ae\; af\; ag\; y.\; x = (\mathrm{T\,R}\, (\mathrm{T\,B}\;\, ae\; af\;\, ag)\, y\, \mathrm{E}) \wedge \mathrm{redinv}(\mathrm{T\,B}\;\, ae\; af\;\, ag)) \\
&\vee\; (\exists ae\; af\;\, ag\; y bg\; bh\; bi.\; x = (\mathrm{T\,R}\, (\mathrm{T\,B}\;\, ae\; af\;\, ag)\, y\, (\mathrm{T\,B}\;\, bg\; bh\; bi)) \\
&\qquad\quad \wedge (\mathrm{redinv}(\mathrm{T\,B}\;\, ae\; af\;\, ag) \wedge \mathrm{redinv}(\mathrm{T\,B}\;\, bg\; bh\; bi))))\big)
\end{aligned}
\tag{44}
$$

The precise form of this lemma can be inferred when inspecting the rule set generated by Isabelle from the definition of redinv. The proof is a routine induction proof which nevertheless needs knowledge about theorem proving in general and Isabelle in particular. This lemma helps to improve the form of the test theorem. To be a bit more precise, we insert after the test case generation a sequence of Isar-methods that resolve in any constraint of the form redinv x the above lemma, recomputes the TNF and repeats this process once. The resulting test of depth five can be constructed fairly fast; the adequacy $TNF_{E/5}(TS)$ represents a pragmatically satisfactory test of our program.

4.4. Summing Up

In our experience, increasing the number of iterations also increases the time needed for test data generation substantially. On the other side, this underpins the usual criticism with respect to random testing: deeply nested (either in the sense of data or execution paths) program structures cannot be tested seriously using pure random tests; guidance generated by test cases is crucially needed.

Further, our results show that highly automated approaches yield useful “first shots” but heavily profit from more or less ingenious user interaction. A trade-off must be made here between the time needed to run a test (including generation), the quality of the test and the time and experience needed in advanced techniques such as theorem proving.

5. Principles of Test Sequence Generation in HOL-TestGen

So far, we considered programs under test of the form $PUT :: \iota \Rightarrow\; o$ where ι is the type of the input and o the type for the output. In the sequel, we will consider the case where some form of state is involved; we will use the type variables σ, σ', σ'', ... for expressions denoting state.

We will center our presentation around the concept of an "i/o stepping function" having the type:

$$ioprog :: \iota \Rightarrow \sigma \Rightarrow (o \times \sigma)\,\text{option} \quad \text{or equivalently} \quad \iota \Rightarrow (o, \sigma)\ \text{Mon}_{\text{SE}} \tag{45}$$

where Mon_{SE} stands for *state-exception-monad*.[5] Here, an *ioprog* may either fail for a given state σ and input ι, or produce an output and a successor state. We will discuss operators that transform stepping functions to others, and we will also view programs under test PUT as stepping functions. We can easily specialize our scenario by setting ι, o or σ to the unit-type with the only, trivial constant denoted "()." Thus, depending on the concrete test-scenario, we admit PUT's to export their internal state, export it partially, or hide it completely in the case $\sigma = \text{unit}$. This hiding of internal state may be convenient or simply mandatory in a black-box scenario, however, there is a trade-off to hide not too much. Otherwise, we violate the fundamental *testability-hypothesis* (we follow the terminology established in [BGM91]) that must be imposed on a test scenario with hidden state to make test results reproduceable:

- there must be a means to initialize the internal state of PUT,
- PUT must be a *function* in the modeled state (this is reflected by its type $\iota \Rightarrow (o, \sigma)\ \text{Mon}_{\text{SE}}$ in our model), and
- within a test, there are no other state-transitions of the system than those resulting from calls of the PUT.

Note that the second condition excludes non-determinism in the implementation-side (independent of the internal, invisible state) and imposes that their are no implicit side-channels into our system like clocks or communications to other processes that can influence the resulting determinism in the system. We refrain from a formalization of testability-hypothesis here, while concentrating on formal, explicit *testing*-hypothesis.

Stepping functions may be combined via the core operators:

$$\text{unit}_{\text{SE}} :: o \Rightarrow (o, \sigma)\ \text{Mon}_{\text{SE}} \tag{46}$$

$$\text{bind}_{\text{SE}} :: (o, \sigma)\ \text{Mon}_{\text{SE}} \times (o \Rightarrow (o', \sigma)\ \text{Mon}_{\text{SE}}) \Rightarrow (o', \sigma)\ \text{Mon}_{\text{SE}} \tag{47}$$

to which we give the more intuitive notation $\text{return}(e)$ for $\text{unit}_{\text{SE}}(e)$ and $x \leftarrow f_1;\ f_2$ for $\text{bind}_{\text{SE}}\ f_1\ f_2$. They are defined as follows, giving the combination of computations a strict semantics for reporting failures:

$$\begin{array}{lll} \text{return}(e) & \equiv \lambda\,\sigma.\ \text{Some}(e, \sigma) & \\ x \leftarrow f_1;\ f_2 & \equiv \lambda\ \sigma.\ \text{case}\ f_1\ \sigma\,\text{of} & \text{None} \Rightarrow \text{None} \\ & & \mid \text{Some}(o, \sigma') \Rightarrow f_2(o)(\sigma') \end{array} \tag{48}$$

Over state-exception monads, the notion of *test-sequences* over stepping functions can be captured by:

$$o_1 \leftarrow f_1(i_1);\ o_2 \leftarrow \lambda\,_.\ f_2(i_2);\ \ldots;\ \lambda\,_.\ o_n \leftarrow f_n(i_n);\ \text{return}\ (post\ o_1\ \ldots\ o_n) \tag{49}$$

where *post* is a predicate on the outputs reported from each individual i/o step and where the individual steps ignore the output of the previous step. This is specific to test sequences; in *reactive test-sequences* discussed in the sequel we will relax this restriction.

The operators above enjoy the usual algebraic laws of monads, which we derive from the definitions:

$$(x \leftarrow \text{return}\ a;\ k) = k \tag{50}$$

$$(x \leftarrow m;\ \text{return}\ x) = m \tag{51}$$

$$(y \leftarrow (x\ \leftarrow m;\ k);\ h) = (x \leftarrow m;\ (y \leftarrow k;\ h)) \tag{52}$$

As we will see, these basic rules will also play a role in the symbolic evaluation of test-sequences.

5.1. Using SE-monads as Infrastructure for Sequence Test

To express test-sequences also on the object-level and to make them amenable to formal reasoning, we represent them as *lists of input* and generalize the bind-operator of the state-exception monad accordingly.

[5] Since HOL is a purely functional logical language, it cannot come as a surprise for the reader familiar with languages such as Haskell that we will apply monad-techniques to model state and state-transitions. For this presentation, however, we will only need a modest and slightly simplified subset of the machinery discussed elsewhere [Mog91, Wad95].

The approach is straight-forward, but comes with a price: we have to apply a technique called *interface encapsulation* that wraps up all input and output data into an own type. Assume that we have a typed interface to a module with the operations $\mathrm{op}_1, \mathrm{op}_2, \ldots, \mathrm{op}_n$ with the inputs $\iota_1, \iota_2, \ldots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input-type:

$$\text{datatype} \;\; \mathrm{in} = \mathrm{op}_1 :: \iota_1 \mid \cdots \mid \mathrm{op}_n :: \iota_n \tag{53}$$

$$\text{datatype} \;\; \mathrm{out} = \mathrm{res}_1 :: \; o_1 \mid \cdots \mid \mathrm{res}_n :: \; o_n \tag{54}$$

Obviously, we loose some type-safety in this approach; we have to express that input op_k and output res_k correspond in the sequences; however, this can be achieved by adding some tests on the test-driver side. The sequence-bind operator called mbind has type: $\iota\,\mathrm{list} \Rightarrow \sigma \Rightarrow ((\iota \Rightarrow (o,\sigma)\,\mathrm{Mon}_{\mathrm{SE}}) \Rightarrow (o\,\mathrm{list} \times \sigma)\,\mathrm{option})$ and is defined by the recursive equations:

$$\begin{array}{lll} \multicolumn{3}{l}{\ldots\text{where}} \\ \;\mathrm{mbind}\,[\,]\;\;\sigma\;\; \mathit{ioprog} & = & \mathrm{Some}([\,],\sigma) \\ \mid \mathrm{mbind}\,(a\#H)\;\;\sigma\;\;\mathit{ioprog} & = & \mathrm{case}\;\;\mathit{ioprog}(a,\sigma)\,\mathrm{of}\,\mathrm{None} \Rightarrow \mathrm{None} \\ & & \quad \mid \mathrm{Some}(\mathit{out},\sigma') \Rightarrow \;\; \mathrm{case\;mbind}\;\;H\;\sigma'\;\mathit{ioprog}\,\mathrm{of} \\ & & \qquad\qquad\qquad\qquad\quad \mathrm{None} \Rightarrow \mathrm{Some}([\mathit{out}],\sigma') \\ & & \qquad\qquad\qquad\qquad\quad \mid \mathrm{Some}(\mathit{outs},\sigma'') \Rightarrow \mathrm{Some}(\mathit{out}\#\mathit{outs},\sigma'') \end{array} \tag{55}$$

This notion of test-sequence allows the i/o-stepping function (and the special case of a program under test) to stop execution *within* the sequence; however, such premature terminations are characterized by an output lists which are shorter than their input list. Thus, if an interface to of a component (represented by the functions $f_1, \ldots, f_n$) is standardized via interface encapsulation, test-sequences as introduced in Fact 49 can be represented in the following pattern:

$$\mathit{os} \leftarrow \mathrm{mbind}\,\mathit{PUT}\;\iota s;\;\mathrm{return}\;(\mathit{post}(\mathit{os})) \tag{56}$$

So far, we have been concerned with the issue of *modeling* test-sequences in HOL-TestGen; in the sequel, we will address the question how to *generate* them. A pivotal step towards this goal is the core notion of a *valid test-sequence*

$$\begin{array}{ll} \text{definition} & _ \models _ :: \sigma \Rightarrow (\mathit{bool},\sigma)\mathit{Mon}_{\mathrm{SE}} \\ \text{where} & \sigma_0 \models m \equiv (m\;\sigma) \neq \mathrm{None} \wedge \mathrm{fst}(\mathrm{the}\;(m\;\sigma_0)), \end{array} \tag{57}$$

i. e. a test sequence is valid, if its evaluation based on an initial state is not interrupted by an exception and its final value is true (“the” strips off the Some which must exist due to the first condition and “fst” selects the first element of a tuple). Note that this definition and the definition of the monadic operator as well as mbind are altogether executable for Isabelle, i. e., can be used in code generation.

Valid test sequences, considered as a tuple, enjoy a number of (formally derived) properties, that make them amenable to symbolic computation:

$$(\sigma \models (\mathrm{return}\,P)) = P \tag{58}$$

$$\neg(\sigma_0 \models (\lambda\,\sigma.\;\mathrm{None})) \tag{59}$$

$$\mathit{ioprog}\;\;\sigma = \mathrm{None} \Longrightarrow \neg(\sigma \models ((s \leftarrow \mathit{ioprog}; M\;\;s))) \tag{60}$$

$$M\;\;\sigma = \mathrm{Some}(f\;\;\sigma,\sigma) \Longrightarrow (\sigma \models M) = f\sigma \tag{61}$$

$$(\sigma \models (\lambda\,\sigma.\;\mathrm{Some}(f\sigma,\sigma))) = (f\sigma) \tag{62}$$

$$\mathit{ioprog}\;\;\sigma = \mathrm{Some}(b,\sigma') \Longrightarrow (\sigma \models ((s \leftarrow \mathit{ioprog}; Ms))) = (\sigma' \models (Mb)) \tag{63}$$

$$\sigma \models (s \leftarrow \mathrm{mbind}[]\;\mathit{ioprog}; Ms)) = (\sigma \models (M[])) \tag{64}$$

$$\mathit{ioprog}\;\;a\;\sigma = \mathrm{None} \Longrightarrow (\sigma \models (s \leftarrow \mathrm{mbind}(a\#S)\;\mathit{ioprog}; Ms)) = (\sigma \models (M[])) \tag{65}$$

$$\sigma \models ((s \leftarrow A; Ms)) \Longrightarrow \exists v\sigma'.\;A\;\;\sigma = \mathrm{Some}(v,\sigma') \wedge \sigma' \models (M\;\;v) \tag{66}$$

$$\begin{array}{l} \mathit{ioprog}\;\;a\;\sigma = \mathrm{Some}(b,\sigma') \Longrightarrow \\ \qquad (\sigma \models (s \leftarrow \mathrm{mbind}(a\#S)\;\mathit{ioprog}; Ms)) = (\sigma' \models (s \leftarrow \mathrm{mbind}\,S\;\;\mathit{ioprog}; M(b\#s))) \end{array} \tag{67}$$

Not all rules are immediately usable by the Isabelle rewriter, in some cases second-order variables (like *ioprog*, M) have to be instantiated by concrete constants stemming from definitions in a concrete model in order to make them applicable.

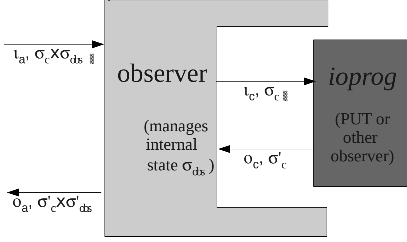


Fig. 5: An observer from HOL-TESTGEN library: Managing an own state σ_{obs}, it takes *abstract* input ι_a, transforms it to *concrete* input ι_c based on own observations in σ_{obs}, transfers the concrete input to the internal i/o stepping function, which can be a PUT or another i/o stepping function, observes the resulting concrete output o_c and logs it in its state σ_{obs}. Seen from the outside, the observer represents an i/o stepping function.

Reviewing the pattern of unit test specifications shown in Fact 26, we recognize that it also holds the key for test sequence generation. A natural way to represent sets of input sequences, i. e.a *language* of inputs, is by automata or labeled transition systems. We only need to represent them as (mutual) recursive acceptance predicates accept on input lists, i. e., *traces*, and the test case generation will attempt to explore the input lists used as stimulation of our PUT. Since branching in an automaton will be represented by a disjunction, which will lead to a case-split in the test case generation, the test cases of the $TNF_{E/d}$ will correspond to all paths through the automaton up to length d. The (executable) mbind-combinator takes care of the serialization of the repetitive execution of PUT. Summing up, test specifications for sequence tests over *ioprog* will have the following scheme:

$$\text{accept } \iota s \to \sigma_0 \models (os \leftarrow (\text{mbind } \iota s\, PUT);\ \text{result } (post\ \iota s\ os)) \tag{68}$$

After HOL-TESTGEN derived a concrete input trace ιs, a test driver running the test sequence can be generated since the monad-operators as well as mbind and $_ \models _$ are all executable. In particular, mbind logs the output of the system; the condition *post* can depend on the input sequence ιs and the output log os of the system trace. Therefore, instead of a "hand-coded" accept predicates representing an automata, we can use again a valid sequence on modeled operations. Therefore, a variant of this sequence test-specification scheme looks as follows:

$$\begin{aligned} & \sigma_0 \models (os \leftarrow (\text{mbind } \iota s\, SYS);\ \text{result } (os = X)) \\ \to\ & \sigma_0 \models (os \leftarrow (\text{mbind } \iota s\, PUT);\ \text{result } (post\ \iota s\ os\ X)) \end{aligned} \tag{69}$$

This scheme exploits the double nature of the "$\sigma \models M$"- pair, it can be interpreted for symbolic computation (at least if the operations for SYS are formally described in the model, i. e.part of the test theory, and instantiated forms of the symbolic evaluation rules for Fact 58–67 are available during the test case generation process) *as well* as a program, i. e. a test-driver over PUT: Case-splitting over ιs and symbolic evaluation of the specification will reduce a test-specification to test cases of the form:

$$\phi(X) \to \quad \sigma_0 \models (os \leftarrow (\text{mbind } \iota s\, PUT);\ \text{result } (post\ \iota s\ os\ X)) \tag{70}$$

where the constraint resolution in the test data generation phase will have to find a solution for $\phi(X)$.

5.2. An Infrastructure for Reactive Sequence Test

This concept is also powerful enough to cover situations where the program under test produces output that changes the input of later runs of *ioprog*, i. e., in situations where the test-driver and the external program under test represent a communicating system. Technically, we push our approach to synthesize test drivers of increasing complexity a little further.

The key element for the instantiation of the scheme of Fact 68 lies in the generic definition of an adapter function that builds another stepping function from an i/o stepping function (possibly the PUT). This type of function is colloquially called a wrapper or technically a monad transformer, since it builds a type of form $(\iota_c \Rightarrow (o_c, \sigma)\ \text{MON}_{\text{SE}}) \Rightarrow (\iota_a \Rightarrow (o_c, \sigma_{\text{obs}} \times \sigma)\ \text{MON}_{\text{SE}})$. Such an adapter function can transform input and output from and to a given PUT, manage an own state σ_{obs} and check that output and state satisfy given postconditions. In case of a violation of the latter it reports a failure. Fig. 5 illustrates the overall scheme of an observer. The HOL-TESTGEN library provides a collection of observer functions; for our example discussed in Sec. 6.2, we will present the variant observer$_2$. It implements the scenario where the guards (or: post conditions) are in fact executable and *abstract input traces* can be mapped to *concrete input traces* solely by information that has been communicated between PUT and observer previously in the communication.

The library also provides versions of observer functions where the postcondition is non-executable (this boils down to calling a constraint solver at run-time, see [BPZ09]) or where no dependency between input and output exists and where a pre-computation of input-sequences via constraint solving is possible during the test data generation phase of HOL-TestGen.

The formal presentation of observer_2 needs the following prerequisites: the function $rebind :: \sigma_{\text{obs}} \Rightarrow o_c \Rightarrow \sigma_{\text{obs}}$ that abstracts from a concrete output of type o_c information and inserts it into the observer state σ_{obs}, the function $subst :: \sigma_{\text{obs}} \Rightarrow \iota_a \Rightarrow \iota_c$ that substitutes the collected information form the observer state back into abstract input of type ι_a to gain concrete input of type ι_c, and the postcondition $post :: \sigma_{\text{obs}} \Rightarrow \sigma \Rightarrow \iota_c \Rightarrow o_c \Rightarrow \text{bool}$ which we allow to depend on both states, the observer state and the exported state of the given *ioprog*. Wiring everything together, we get the following definition:

$$
\begin{aligned}
&\text{constdefs}\quad \text{observer}_2 :: [\sigma_{obs} \Rightarrow o_c \Rightarrow \sigma_{obs}, \sigma_{obs} \Rightarrow \iota_a \Rightarrow \iota_c, \sigma_{obs} \Rightarrow \sigma \Rightarrow \iota_c \Rightarrow o_c \Rightarrow bool] \\
&\qquad\qquad \Rightarrow (\iota_c \Rightarrow (o_c, \sigma)\ \text{MON}_{\text{SE}}) \Rightarrow (\iota_a \Rightarrow (o_c, \sigma_{obs} \times \sigma)\ \text{MON}_{\text{SE}}) \\
&\qquad \text{observer}_2\ \ rebind\ subst\ post\ ioprog \equiv \lambda\, in_a\,.\ \lambda(\sigma_{obs}, \sigma).\ \text{let}\ in_c = subst\ \sigma_{obs}\ in_a\ \text{in} \\
&\qquad\quad \text{case}\ ioprog\ \ in_c\ \ \sigma\ \text{of}\ \ \text{None} \Rightarrow \text{None} \\
&\qquad\qquad \mid \text{Some}(out_c, \sigma') \Rightarrow \text{let}\ \sigma'_{obs} = rebind\ \sigma_{obs}\ out_c\ \text{in} \\
&\qquad\qquad\qquad \text{if}\ post\ \sigma'_{obs}\ \sigma'\ \ in_c\ out_c \\
&\qquad\qquad\qquad \text{then}\ \text{Some}(out_c, (\sigma'_{obs}, \sigma')) \\
&\qquad\qquad\qquad \text{else}\ \text{None}
\end{aligned}
\tag{71}
$$

Rephrasing the scheme of sequence tests shown in Fact 68, we get for reactive sequence tests of the observer_2 kind (*post* deterministic, only dependent on post-state, and executable) the following new scheme:

$$
\begin{aligned}
&\text{testspec sequence:} \\
&\quad \text{accept}\ \iota s \rightarrow \sigma_0 \models (os \leftarrow (\text{mbind}\ \iota s\ (\text{observer}_2\ rebind\ subst\ post\ PUT); \text{result}\ (post\ \iota s\ os))
\end{aligned}
\tag{72}
$$

In contradiction to folk wisdom, it follows from our presentation that reactive sequence tests are an *instance* of sequence testing, which in itself is an *instance* of unit-testing schemes, at least from the perspective of a powerful modeling language such as HOL in which states, state-sequences, and computations (monads) are altogether just data of higher types. This also holds for other, less constrained variants of observer.

5.3. Test-Adequacy and Theoretical Properties

As shown in Sec. 6.2, the possibility to represent computational structures like state-exception monads inside HOL paves the way to the apparently absurd result that reactive sequence tests are a special case of unit test scenarios. Our technique crucially depends of the possibility to postpone the concrete binding of explicit variables occurring in protocols and the checking of postconditions at runtime of the test. This technique is enabled by our approach to synthesize the concrete code for test-drivers after the test data generation phase.

For the test depth $d = 4$ of the test case-generation procedure we reach transition coverage in the stimulation protocol automaton and therefore implicitly on the protocol automaton shown in Fig. 6. In general, transition coverage of a finite automaton can be simulated by $TNF_{E/d}$ for a reasonably large d. Reasonably large means here the maximal length of the longest path in an all-transition covering path set, which can be constructed for any given automaton.

6. Case-studies: Sequence-testing Red-black Trees and Protocol Tests

6.1. Example: Red-Black Trees as Sequence Test

We will turn our red-black tree example from Sec. 4 into a sequence test, i. e., we will turn the tree itself into a component of the state to which we do not have direct access during the test execution. Not using the internal state of a program under test does *not* exclude that this state is part of the model and that specification-related reasoning can make symbolic evaluations over it in order to compute possible results of the program under test. We will consider two variants of sequence tests: tests against a *reference implementation* and tests against an *abstract implementation*.

6.1.1. *Tests against a* Reference Implementation

The idea of this scenario is to use an off-the-shelf implementation and test the system under test against it. In such a scenario we approximate the intended implementation by the other one, which results in a fine-grained partitioning of the input-output relation. Under the testing assumption that the *PUT* will be algorithmically similar, it can be speculated that test-set will have a high fault-detection capacity.

As reference implementation for this scenario, we chose a functional version in SML which is part of the sml/NJ library. A transcription into Isabelle/HOL is straight-forward and is omitted here for reasons of space[6]. It provides the operations isin and insert, for which we define wrappers into the world of $\mathrm{MON}_{\mathrm{SE}}$ as follows:

$$\begin{array}{lll} \text{definition} & \text{insert}' & :: \alpha :: \text{linorder} \to (\text{unit}, \alpha \text{ tree})\ \mathrm{MON}_{\mathrm{SE}} \\ \text{where} & \text{insert}'\, a & = (\lambda\, \sigma.\ \text{Some}((), \text{insert}\ a\, \sigma)) \end{array} \tag{73}$$

$$\begin{array}{lll} \text{definition} & \text{isin}' & :: \alpha :: \text{linorder} \to (\text{bool}, \alpha \text{ tree})\ \mathrm{MON}_{\mathrm{SE}} \\ \text{where} & \text{isin}'\, a & = (\lambda\, \sigma.\ \text{Some}(\text{isin}\ a\, \sigma, \sigma)) \end{array} \tag{74}$$

Note that insert′ is a typical operation just storing information in the state, while isin′ is a typical query operation on the state. From both definitions we derive instances of the symbolic evaluation rule set Fact 58–67 which we add to the background theory E of the test case generation process.

It does not come as a surprise that we chose the scheme Fact 69 for our test specification. We are interested in a tests expressing *values that should be in the state after an input sequence <iota>s should also be in the state of PUT*:

$$\begin{array}{ll} \text{testspec}: & (E \models (_ \leftarrow \text{mbind}\, \iota s\ \text{insert}'; out \leftarrow \text{isin}'(A :: \text{int}); \text{return}(out))) \\ \longrightarrow & (E \models (_ \leftarrow \text{mbind}\, \iota s\ \mathit{INSERT}; out \leftarrow \mathit{ISIN}\ A; \text{return}(out)))\text{''} \end{array} \tag{75}$$

Test case generation configuration completely trivial since more case-splittings concerning inner data structures of our implementation model have to be considered and since we have actually two different entries to the system under test:

$$\text{apply}(\text{gen_test_cases}\ \mathit{ISIN}\ \mathit{INSERT}\quad \text{split: ml_order.split}\ \ \text{ml_order.split_asm}) \tag{76}$$

The resulting test cases without test-hypothesis look as follows:

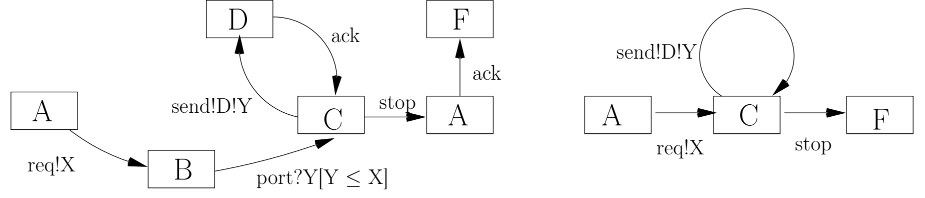
$$\begin{array}{rl}
1. & E \models (_ \leftarrow \text{mbind}\, [?x_1]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_1) \\
2. & \text{THYP}(\ldots) \\
3. & ?x_2 < ?x_3 \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_3, ?x_2]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_3) \\
4. & \text{THYP}(\ldots) \\
5. & ?x_4 < ?x_5 \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_5, ?x_4]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_4) \\
6. & \text{THYP}(\ldots) \\
7. & E \models (_ \leftarrow \text{mbind}\, [?x_6, ?x_6]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_6) \\
8. & \text{THYP}(\ldots) \\
9. & ?x_7 < ?x_8 \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_7, ?x_8]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_7) \\
10. & \text{THYP}(\ldots) \\
11. & ?x_9 < ?x_{10} \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_9, ?x_{10}]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_{10}) \\
12. & \text{THYP}(\ldots) \\
13. & ?x_{10} < ?x_{11} \wedge ?x_{11} < ?x_{12} \wedge \text{isin} ?x_{10}\ \text{undefined} \\
 & \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_{12}, ?x_{11}, ?x_{10}]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_{10}) \\
14. & \text{THYP}(\ldots) \\
15. & ?x_{14} < ?x_{13} \Longrightarrow E \models (_ \leftarrow \text{mbind}\, [?x_{13}, ?x_{13}, ?x_{14}]\ \mathit{INSERT}; \mathit{ISIN}\ ?x_{13}) \\
\vdots & \qquad\qquad\vdots
\end{array} \tag{77}$$

These test cases check for insertions in all possible intervals induced by the ordering relation.

Just for completeness, consider the "opposite" test-specification *values that not should be in the state after*

[6] To the anonymous referees: It can be found in Appendix A.

Fig. 6: An abstract protocol automaton (containing variables and constraints over them) and the resulting stimulation sequence automaton.



an input sequence ιs should also not be in the state of PUT:

$$\begin{aligned} \text{testspec}: \quad & (E \models (_ \leftarrow \text{mbind}\,\iota s\ \text{insert}'; \mathit{out} \leftarrow \text{isin}'(A :: \text{int}); \text{return}(\neg \mathit{out}))) \\ \longrightarrow \quad & (E \models (_ \leftarrow \text{mbind}\,\iota s\ \mathit{INSERT}; \mathit{out} \leftarrow \mathit{ISIN}\ A; \text{return}(\neg \mathit{out})))" \end{aligned} \tag{78}$$

As in LTL-like formalisms, "opposite" test-specifications are not just their negations.

6.1.2. *Tests Against an* Abstract Implementation

The previous scenario required to include a reference implementation into our test specification. This can be error-prone in itself, and it can lead to complex constraint systems that are out of the reach of constraint-solving technologies.

Therefore it can be desirable to use an *abstract* model in our test specification of the program to test and to define a test in terms of the latter. For the case of insert and delete operations in red-black trees, this is remarkably straight-forward:

$$\text{types} \quad \alpha\ \text{state}_{\text{abs}} = \alpha\ \text{set} \tag{79}$$

$$\begin{aligned} \text{fun} \quad & \text{I}_{\text{abs}} && :: \alpha\ \text{events} \Rightarrow (\text{unit}, \alpha\ \text{state}_{\text{abs}})\,\text{MON}_{\text{SE}} \\ \text{where} \quad & \text{I}_{\text{abs}}\,(\text{insert}\ a) && = (\lambda\,\sigma.\ \text{Some}((), \{a\} \cup \sigma)) \\ & \text{I}_{\text{abs}}\,(\text{delete}\ a) && = (\lambda\,\sigma.\ \text{Some}((), \sigma - \{a\})) \end{aligned} \tag{80}$$

$$\begin{aligned} \text{fun} \quad & \text{isin}'_{\text{abs}} && :: \alpha \Rightarrow (\text{bool}, \alpha\ \text{state}_{\text{abs}})\,\text{MON}_{\text{SE}} \\ \text{where} \quad & \text{isin}'_{\text{abs}}\ a && = (\lambda\,\sigma.\ \text{Some}(a \in \sigma, \sigma)) \end{aligned} \tag{81}$$

Thus, we assume implicitly a data-abstraction relation between abstract and concrete state, and view our test-specification as a form of data-refinement testing:

$$\begin{aligned} \text{testspec}: \quad & (\{\} \models (_ \leftarrow \text{mbind}\,\iota s\ \text{I}_{\text{abs}}; \mathit{out} \leftarrow \text{isin}'_{\text{abs}}\ A; \text{return}(\mathit{out}))) \\ \longrightarrow \quad & (E \models (_ \leftarrow \text{mbind}\,\iota s\ \mathit{PUT_OP}; \mathit{out} \leftarrow \mathit{ISIN}\ A; \text{return}(\mathit{out})))" \end{aligned} \tag{82}$$

The case for the "opposite" test specification is analogously and omitted.

The generated test cases are much coarser than the ones generated in Sec. 6.1.1—not enumeration of insertion-possibilities in intervals is the common ground, rather the existence in a set (without any consideration of order). So, there is a trade-off between the advantage to have a small and easy to understand specification and the necessity of a finer partitioning (leading, for example, to a higher code-coverage of the program under test).

6.2. An Example of Reactive Sequence Test

As an example of a reactive system, we assume a client/server situation where the client sends a server a communication request and specifies a "port-range" X (for simplicity, just an upper bound). The server non-deterministically chooses a port Y which is within the specified range. The client sends a sequence of data (abstracted away in our example to just one constant Data) on the port allocated by the server. The communication is terminated by the client with a stop event. Fig. 6 shows the abstract protocol (containing variables and constraints over them) and its sub-protocol containing just the input stimulation sequence.

The key idea to test a server implementing this protocol is to distinguish abstract and concrete events; abstract events may contain explicit variables X and Y which were replaced at runtime of the test by concrete values, say 100 and 25. Thus, the test case generation process can be executed over abstract events which not only reduces drastically the size of the proof state, but makes *reactivity* of the protocol possible:

reactions (like the port number given by the server) may depend on inputs (like the range of the port number) exchanged earlier in the communication.

In the following, we describe the necessary infra-structure of our model in HOL-TestGen. We define the explicit variables occurring in this protocol:

$$\text{datatype} \quad \text{vars} \quad = X \mid Y \tag{83}$$

and specify abstract and concrete input and output events:

$$\begin{array}{lll} \text{types} & \mathit{chan} & = \mathit{int} \\ \text{datatype} & \text{InEvent}_{conc} & = \text{req vars} \mid \text{send data vars} \mid \text{stop} \\ \text{datatype} & \text{InEvent}_{abs} & = \text{reqA vars} \mid \text{sendA data vars} \mid \text{stop} \\ \text{datatype} & \text{OutEvent}_{conc} & = \text{port chan} \mid \text{ack} \\ \text{datatype} & \text{OutEvent}_{abs} & = \text{portA vars} \mid \text{ack} \end{array} \tag{84}$$

where data is just the unit type.

Now we have to fix the observer state σ_{obs}. We will set it to an environment, i. e., a partial map vars $\rightharpoonup$ chan which is just a synonym for vars $\Rightarrow$ chan option. The definitions of subst and rebind are now straight-forward:

$$\begin{array}{lll} \text{fun} \ldots \text{where} & \text{subst}\ \mathit{env}\ (\text{reqA}\ \ v) & = \text{req}(\text{lookup}\ \mathit{env}\ \ v) \\ & \text{subst}\ \mathit{env}\ (\text{sendA}\ \ d\, v) & = \text{send}\ d(\text{lookup}\ \mathit{env}\ \ v) \\ & \text{subst}\ \mathit{env}\ \ \text{stop} & = \text{stop} \end{array} \tag{85}$$

$$\begin{array}{lll} \text{fun} \ldots \text{where} & \text{rebind}\ \mathit{env}(\text{port}\ n) & = \mathit{env}(Y \mapsto n) \\ & \text{rebind}\ \mathit{env}\ \text{ack} & = \mathit{env} \end{array} \tag{86}$$

It remains to define the postcondition. For the technical reason that Isabelle can establish termination, it comes in two parts. The postcondition encodes the protocol constraint for the transition between state B and C in Fig. 6. Moreover, we ensure that resent values agree indeed with the environment:

$$\begin{array}{lll} \text{fun...where} & \text{post}'\ (\mathit{env}, _, \text{req}\ \ n, \text{port}\ m) & = (n \leq m) \\ & \text{post}'\ (\mathit{env}, _, \text{send}\ \ z\ n, \text{ack}) & = (n = \text{lookup}\ \mathit{env}\ Y) \\ & \text{post}'\ (\mathit{env}, _, \text{stop}, \text{ack}) & = \text{true} \\ & \text{post}'\ (\mathit{env}, _, _, _) & = \text{false} \end{array} \tag{87}$$

$$\begin{array}{lll} \text{fun} & \text{post} & :: (\text{vars} \rightharpoonup \text{int}) \times \text{unit} \Rightarrow \text{InEvent}_{conc} \Rightarrow \text{OutEvent}_{conc} \Rightarrow \text{bool} \\ \text{where} & \text{post}\ x\ y\ z & \equiv \text{post}'(\text{fst}\ x, \text{snd}\ x, y, z) \end{array} \tag{88}$$

Our post does not use the internal state of the program under test. The automaton for the set of stimulation traces results from a direct translation of the right diagram in Fig. 6:

$$\begin{array}{lll} \text{fun} \ldots \text{where} & \text{stimTrace}'\ (A, (\text{reqA}\ \ X)\#S) & = \text{stimTrace}'(C, S) \\ & \text{stimTrace}'\ (C, (\text{sendA}\ \ d\, Y)\#S) & = \text{stimTrace}'(C, S) \\ & \text{stimTrace}'\ (C, [\text{stop}]) & = \text{true} \\ & \text{stimTrace}'\ (x, y) & = \text{false} \end{array} \tag{89}$$

$$\begin{array}{lll} \text{definition} & \text{stimTrace} & :: \text{InEvent}_{abs}\ \text{list} \Rightarrow \text{bool} \\ \text{where} & \text{stimTrace}\ s & \equiv \text{stimTrace}'(A, s) \end{array} \tag{90}$$

Finally, we state the test specification for the reactive sequence test of our example. It is an instance of the sequence test pattern (see Fact 72):

$$\begin{array}{lll} \text{testspec}: & \text{stimTrace}\ \iota s \longrightarrow ((X \mapsto \mathit{init}), C) & \models (\text{mbind}\ \iota s\ (\text{observer}_2\ \text{rebind}\ \text{subst}\ \text{post}\ \ PUT); \\ & & \quad \text{result}\ (\text{length}\ \iota s = \text{length}\ \ os)) \end{array} \tag{91}$$

In our concrete example no use whatsoever has been made of the internal state of the ioprog; the variable C is therefore polymorphic and can be instantiated by an arbitrary type (e. g., unit and its only constant ()). Applying our test case generation and test data generation procedures for $d = 40$ takes less than a second, while the generation of the test script containing the abstract input sequences plus the test program run over them ranges up to five seconds here; this test program also contains the compiled versions of, e. g., observer,

subst, rebind. A sample of the generated test cases reads as follows:

$$
\begin{array}{lll}
1. & ([X \mapsto ?x_2], ()) & \models (os \leftarrow \text{mbind}[\text{reqA } X, \text{stop}](\text{observer}_2 \text{ rebind subst post } PUT); \\
& & \quad \text{result } 2 = \text{length } os) \\
2. & \quad \dots & \\
3. & ([X \mapsto ?x_3], ()) & \models (os \leftarrow \text{mbind}[\text{reqA } X, \text{sendA Data } Y, \text{stop}](\text{observer}_2 \text{ rebind subst post } PUT); \\
& & \quad \text{result } 3 = \text{length } os) \\
4. & \quad \dots & \\
5. & ([X \mapsto ?x_4], ()) & \models (os \leftarrow \text{mbind}[\text{reqA } X, \text{sendA Data } Y, \text{sendA Data } Y, \text{stop}] \\
& & \qquad (\text{observer}_2 \text{ rebind subst post } PUT); \\
& & \quad \text{result } 4 = \text{length } os) \\
\vdots & \quad \vdots &
\end{array}
\tag{92}
$$

In the sample above, we omitted the test-hypothesis for space reasons.

7. Explicit Test-hypothesis Validated

We have seen in the previous sections, that uniformity and regularity-hypotheses are an amazingly flexible means for the systematic weakening of specifications. We have seen that regularity-hypotheses and its induced specification-based coverage criterion $TNF_{E/d}$ subsumes other coverage criteria (for sufficiently large d). In this section, we will explore explicit test-hypothesis in another direction: How to validate uniformity and regularity test-hypotheses? We will address the this question by a standard example, the insertion-sort algorithm, for testing and verifying test-hypothesis in a (post-hoc) white-box setting. This verification will shed some light on the role of tests and proofs.

7.1. Testing Test-hypothesis

Following the standard HOL-TestGen workflow, we start by specifying the program under test:

$$
\begin{array}{lll}
\text{fun} & \text{is_sorted} & :: (\alpha :: \text{order}) \text{ list} \Rightarrow \text{bool} \\
\text{where} & \text{is_sorted } [\,] & = \text{true} \\
& \text{is_sorted}(x\#xs) & = \text{case } xs \text{ of } [\,] \Rightarrow \text{true} \\
& & \qquad\qquad | \ (y\#ys) \Rightarrow (((x < y) \lor (x = y)) \land \text{is_sorted } xs)
\end{array}
\tag{93}
$$

and fire the test case generation for the test specification with the (implicit) default values $d = 3$ (depth of test case generation):

$$
\begin{array}{l}
\text{testspec sorting:} \quad \text{is_sorted } ioprog(l :: (\alpha :: \text{order}) \text{ list}) \\
\quad \text{apply}(\text{gen_test_cases } ioprog)
\end{array}
\tag{94}
$$

This results in the test theorem, containing both test cases and test-hypothesis:

1. $\text{is_sorted}(ioprog\ [\,])$
2. $\text{is_sorted}(ioprog\ [?x_1])$
3. $\text{THYP}((\exists x.\ \text{is_sorted}(ioprog\ [x])) \longrightarrow (\forall x.\ \text{is_sorted}(ioprog\ [x])))$
4. $\text{is_sorted}(ioprog\ [?x_2, ?x_3])$
5. $\text{THYP}((\exists x\ xa.\ \text{is_sorted}(ioprog\ [xa, x])) \longrightarrow (\forall x\ xa.\ \text{is_sorted}(ioprog\ [xa, x])))$
6. $\text{is_sorted}(ioprog\ [?x_4, ?x_5, ?x_6])$
7. $\text{THYP}((\exists x\ xa\ xb.\ \text{is_sorted}(ioprog\ [xb, xa, x])) \longrightarrow (\forall x\ xa\ xb.\ \text{is_sorted}(ioprog\ [xb, xa, x])))$
8. $\text{THYP}(3 < |l| \longrightarrow \text{is_sorted}(ioprog\ l))$

Since all test cases are unconstrained, the test data selection phase picks just random integer values for the meta-variables $?x_1, \dots, ?x_6$.

Re-feeding explicit test-hypothesis into the testing process is easy in principle: just remove the THYP operator—which protects the formula inside from further decomposition in the test case generation—and generate another test-theorem from it. Fig. 7 illustrates the case of a refinement of a uniformity-hypothesis.

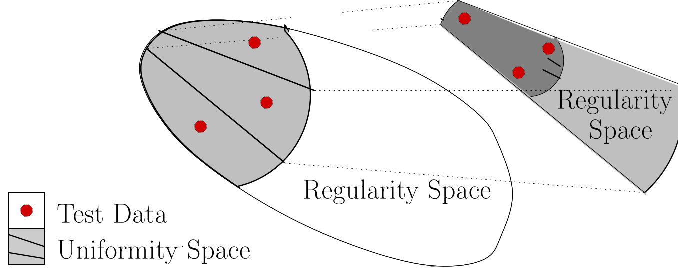


Fig. 7: Refining test data spaces, i. e., test cases, by testing-hypothesis: specific partition of the uniformity space, results in both more fine-grained uniformity spaces (and thus test data) and a new partition of the regularity space.

Refining a specific uniformity-hypothesis, i. e., a specific partition of the uniformity space, results in both more fine-grained uniformity spaces (and thus test data) and a new partition of the regularity space.

While the approach leads to the construction of more test cases and therefore more distinct test data in principle, the present example will not work for the current HOL-TestGen version since gen_test_cases treats test-classes induced by basic types like integer as atomic. A list of integer of length two is therefore not further separated. The approach will work, however, in our Red-black tree example.

7.2. Proving Test-hypothesis

As a pre-requisite, we have to give a program for our test: this reflects the change to the white-box testing paradigm. As in all white-box tests, we make the meta-assumption that the program under test is *faithfully* modeled inside our tool—and thus amenable to analysis based on this presentation.

The proof of a uniform-hypothesis (where ioprog is now instantiated with sort, i. e., our definitions shown in Fact 12) is straight-forward and automatic:

$$\begin{array}{l}\text{lemma uniformity_verified:}\\ \quad \text{THYP}((\exists x\, xa.\ \text{is_sorted}(\text{sort}[xa, x])) \longrightarrow (\forall x\, xa.\ \text{is_sorted}(\text{sort}[xa, x])))\end{array} \tag{95}$$

We reduce the test-hypothesis to the core and get:

$$1.\quad \bigwedge x\, xa.\ \text{is_sorted}(\text{sort}\ [xa, x]) \tag{96}$$

Unfolding sort yields:

$$1.\quad \bigwedge x\, xa.\ \text{is_sorted}\ (\text{ins}\ xa\ (\text{ins}\, x\ [\,])) \tag{97}$$

and after unfolding of ins we get:

$$1. \bigwedge x\, xa.\ \text{is_sorted}(\text{if}\ xa < x\ \text{then}[xa, x]\ \text{else}[x, xa]) \tag{98}$$

Case-splitting results in:

$$\begin{array}{ll}1. & \bigwedge x\, xa.\ xa < x \Longrightarrow \text{is_sorted}\ [xa, x]\\ 2. & \bigwedge x\, xa.\ \neg xa < x \Longrightarrow \text{is_sorted}\ [x, xa]\end{array} \tag{99}$$

Evaluation of is_sorted yields:

$$\begin{array}{ll}1. & \bigwedge x\, xa.\ xa < x \Longrightarrow \text{case}\ [x]\ \text{of}\ [\,] \Rightarrow \text{true}\\ & \qquad\qquad\qquad\qquad \mid y\#ys \Rightarrow (xa < y \lor xa = y)\\ & \qquad\qquad\qquad\qquad\quad \land (\,\text{case}[\,]\ \text{of}\ [\,] \Rightarrow \text{true}\\ & \qquad\qquad\qquad\qquad\qquad\qquad \mid y\#ys \Rightarrow x < y \lor x = y) \land \text{true}\\ 2. & \bigwedge x\, xa.\ \neg xa < x \Longrightarrow \text{case}\ [xa]\ \text{of}\ [\,] \Rightarrow \text{true}\\ & \qquad\qquad\qquad\qquad \mid y\#ys \Rightarrow (x < y \lor x = y)\\ & \qquad\qquad\qquad\qquad\quad \land (\,\text{case}[\,]\ \text{of}\ [\,] \Rightarrow \text{true}\\ & \qquad\qquad\qquad\qquad\qquad\qquad \mid y\#ys \Rightarrow xa < y \lor xa = y) \land \text{true}\end{array} \tag{100}$$

which can be reduced to:

$$1.\quad \bigwedge x\, xa.\ \neg xa < x \Longrightarrow x < xa \lor x = xa \tag{101}$$

which results by arithmetic reasoning to true.

The proof reveals that the test is in itself irrelevant (the existential part is just discarded) for the proof of uniformity. The three uniformity test-hypothesis together can be combined to

$$\text{lemma separation_for_sort:} \quad \forall (l :: \text{int})\ \text{list}.\ |l| \leq 3 \rightarrow \text{is_sorted}(\text{sort}\, l) \tag{102}$$

which states that the depth parameter of the data-separation theorem is in fact *exhausted* by the uniformity statements; this result is independent from the definition of sort and could be generated by HOL-TestGen together with the data-separation theorem.

Altogether, we can now *verify* the regularity-hypothesis. Without explaining the Isabelle proof commands, we show the full straight-forward induction proof:

$$\text{lemma regularity_verified:} \quad \text{THYP}(3 < |l| \rightarrow \text{is_sorted}(\text{sort}\, l)) \tag{103}$$

```
proof−
    have anchor: ⋀ a l. |(l :: int list)| = 3 ⟹ is_sorted(ins a (sort l))
      by(auto intro!: separation_for_sort[THEN spec;THEN mp] is_sorted_invariant_ins)
    have step: ⋀ a l. is_sorted(sort(l :: int list)) ⟹ is_sorted (ins a (sort l))      (104)
      by(erule is_sorted_invariant_ins))
  show ?thesis
  apply(simp only: THYP_def)
```

which results in

$$1. \quad 3 < |l| \Longrightarrow \text{is_sorted}(\text{sort}\, l) \tag{105}$$

We continue the proof by induction l:

$$\text{apply(induct } l\text{; auto)} \tag{106}$$

resulting in:

$$\begin{aligned} &1. \quad \textstyle\bigwedge a\, l.\ [\![2 < |l|; \neg 3 < |l|]\!] \Longrightarrow \text{is_sorted}(\text{ins}\, a(\text{sort}\, l)) \\ &2. \quad \textstyle\bigwedge a\, l.\ [\![2 < |l|; \text{is_sorted}(\text{sort}\, l)]\!] \Longrightarrow \text{is_sorted}(\text{ins}\, a(\text{sort}\, l)) \end{aligned} \tag{107}$$

finally, we insert the anchor step and conclude our proof:

```
  apply(subgoal_tac length l = 3)
  apply(auto elim!: anchor step)                                                     (108)
done
```

Overall, this script follows the structure that can be expected in an informal proof sketch. Here, the lemma is_sorted_invariant_ins is just the invariant over the “inner loop” of the sorting algorithm:

$$\text{lemma is_sorted_invariant_ins[rule_format]:} \quad \text{is_sorted}\, l \rightarrow \text{is_sorted}(\text{ins}\, a\, l) \tag{109}$$

which is just as well established by a straight-forward induction.

To complete the comparison, we briefly show the *direct proof* of the test specification:

```
lemma testspec_proven:        is_sorted(sortl)
  apply(induct l, simp_all)
  apply(erule is_sorted_invariant_ins)                                               (110)
done
```

To sum up, the good news is that testing test-hypotheses can indeed be used to approximate verification—our methodology is therefore complete in this sense. The bad news is, that our example offers no hope for the desire to use tests to simplify *proofs*. We believe that our example proof stands here for a wide class of similar problems: It can be expected that uniformity will *always* be established independently from a test, and regularity will boil down to an induction, where uniformity clauses are indeed relevant for establishing the anchor, but contribute nothing to the step. Moreover, the problem of “guessing the right invariants”—in our case: is_sorted_invariant_ins—remains.

8. Alternative Formats of Explicit Test-hypothesis

Our concept of explicit test-hypothesis is not restricted to uniformity and regularity, although these are the only two formats supported by HOL-TestGen as default. In this section, we will discuss alternatives to show the expressive power of the concept and its overall significance for the field of testing as a whole and its potential to explain a wide area of testing techniques within one framework. In particular, we will discuss error-based domain analysis, tests of concurrent systems, and white-box test scenarios of imperative programs. These alternative rules can be added to the splitter process or, respectively, to the finalizer process of HOL-TestGen (cf. Fig. 3).

8.1. Generalized Uniformity for Error-based Domain Analysis

A straight-forward extension of the uniformity-hypotheses as discussed in Sec. 3.1.2 is "Generalized Uniformity": in each test-class $C = C_1 \cup \cdots \cup C_k$ represented by the (disjoint) partitions $C_1, \ldots, C_k$, we require that for each partition there is a test case to be tested. Obviously, C and k will depend on the type τ since it is bound by the cardinality of the carrier set of the type, i. e., the set $\{x :: \tau \mid \text{true}\}$. *Generalized Uniformity* has the following form:

$$\frac{\begin{array}{c}[e = a_1, a_1 \in C_1]\\ \vdots \\ P\,e\end{array} \quad \cdots \quad \begin{array}{c}[e = a_k, a_k \in C_k]\\ \vdots \\ P\,e\end{array} \quad \text{THYP}(H)}{P\;(e :: \tau)} \tag{111}$$

where $H \equiv \exists a_1 \in C_1, \ldots, a_k \in C_k.\ |\{a_1, \ldots, a_k\}| = k \wedge P\,a_1 \wedge \cdots \wedge P\,a_k \rightarrow \forall x.\ P\,x$; the purpose of the additional constraints is to ensure that $?x_i$ are pairwise disjoint and are a legal instance of C.

Generalized Uniformity (together with specific tactic control of the application of this rule) can be used to implement *error-based testing* methods. The idea of these approaches is that programs are checked on certain error-prone points [Fos80, MS04] A prominent example are machine arithmetic types such as two's complement integers as implemented in Java for example. In such a machine arithmetic, two constants MaxInt and MinInt exist with MaxInt $+1 =$ MinInt, MinInt $-1 =$ MaxInt and $-$ MinInt $=$ MinInt. On this basis, a test class $d = 5$, $\tau = \text{JavaInt}$ can be defined by

$$C = \{\text{MinInt}\} \cup \{x \mid \text{MinInt} < x < 0\} \cup \{0\} \cup \{y \mid 0 < y < \text{MaxInt}\} \cup \{\text{MaxInt}\}\,, \tag{112}$$

characterizing that in a legal partition must be the boundary cases MinInt, MaxInt and 0 as well as a positive and a negative number.

With such a form of uniformity-hypothesis, finding a counter-example for the following specification of the Java library implementation of the abs function is straight-forward. Our test specification is:

$$0 \leq \text{abs}((x :: \text{JavaInt}) - 2) \tag{113}$$

Applying the above generalized uniformity rule for the substitution $[P := \text{abs}]$ leads to constraints of the form: $x - 2 = \text{MinInt}$, $x - 2 = 0$, $x - 2 = \text{MaxInt}$, $x - 2 = a_2 \wedge \text{MinInt} < a_2 \wedge a_2 < 0$, and $x - 2 = a_4 \wedge 0 < a_4 \wedge a_4 < \text{MaxInt}$. Their resolution leads directly to the counter-example $x = \text{MinInt} + 2$. In contrast, finding this error by brute-force random testing is extremely unlikely: chances are against 2^{32}.

The number of possible substitutions for applying this rule may be too large in practice and must be constrained, for example, to just the expressions occurring as argument term of our function under test (i. e., the *input vector*) in the chooser component of HOL-TestGen (cf. Fig. 3).

8.2. Independence-hypothesis for Concurrent Systems

A typical example of a test-hypothesis occurring in sequence test is a process-independence-hypothesis. The idea is that two processes are independent (do not share global state or exchange signals), if and only if we can test two traces t_1 and t_2 locally instead of testing *all* their interleavings; hypotheses of this form dramatically simplify the number of test cases; and composition of independent processes can be tested

component-wise. The form of the separation theorem and the resulting test hypothesis in CSP [Ros98] looks as follows:

$$\frac{\begin{array}{ccc}[(?x_1 \in \text{Traces}(a_1))] & [?x_2 \in \text{Traces}(a_2))] & \\ \vdots & \vdots & \\ P\ ?x_1 & P\ ?x_2 & \text{THYP}(H)\end{array}}{t \in \text{Traces}(a_1 \mid\mid\mid a_2) \rightarrow P\ (t :: \alpha\ list)} \tag{114}$$

where $H \equiv (\forall xy.\ x \in \text{Traces}(a_1) \wedge y \in \text{Traces}(a_2) \wedge P\ x \wedge P\ y) \rightarrow (\forall x.\ x \in \text{Traces}((a_1 \mid\mid\mid a_2)) \rightarrow P\ x)$ and where $_ \mid\mid\mid _$ is the CSP interleaving operator on traces. Together with rules of the form:

$$x \in \text{Traces}(\texttt{SKIP}) = x = [\,] \tag{115}$$

$$x \in \text{Traces}(a \rightarrow P) = x = [\,] \vee \exists x'.\ x = a\#x' \wedge x' \in \text{Traces}(P) \tag{116}$$

$$x \in \text{Traces}(P \,\Box\, Q) = x \in \text{Traces}(P) \vee x \in \text{Traces}(Q) \tag{117}$$

$$x \in \text{Traces}(\mu X.\ Q(X)) = x \in \text{Traces}(Q(\mu X.\ Q(X))) \tag{118}$$

derived from the operational semantics of CSP (see [TW97] for a formal, Isabelle-based framework to perform such derivations to be included in the test-theory), we can later synthesize the concrete instance of traces $?x$ by generating more and more constraints to be handled by HOL-TestGen. Of course, the rule for the recursion operator $\mu X.\ Q(X)$ has to be applied with care: one way handle it is analogously to an unwinding-hypothesis as discussed in the next section; furthermore, rules for the synchronized communication have to be added which is straight-forward.

The introduction of partial orders over process-expressions can also be viewed as test-hypothesis, which cut away certain branches early. We refrain from a more detailed presentation due to space restrictions.

8.3. Unwinding-hypothesis for White-Box Testing

Analogously to test-adequacy criteria, test-hypotheses may be specification-based or program-based. In the sequel, we will discuss the important class of program-based test-hypothesis and how to formalize them in HOL-TestGen; for this purpose, we will have to represent a particular programming language inside a rich test-theory.

The Isabelle distribution comes already with various such theories describing languages. For the sake of this presentation, we chose the simplest one, an imperative core language called IMP, which is intended as formalization of a textbook on programming language semantics [Win93, Nip98]. IMP provides a particularly clean and complete collection of several semantics (natural semantics, transition semantics, denotational semantics, axiomatic semantics), proofs of their relations (e. g., denotational is equivalent to natural) and proofs of crucial meta-properties (axiomatic semantics is sound and relative complete).

8.3.1. Syntax of IMP

The basic concepts of IMP are *values* val (just natural numbers, for example), and *states* state = loc ⇒ val. *Boolean expressions* bexp and *atomic expressions* (aexp) are represented as functions from state to val or bool. Thus, IMP has in fact no syntax of its own for atomic and Boolean expressions; rather, it uses HOL-terms of the appropriate type at this place (this technique is also called a shallow embedding). In contrast, the syntax of IMP commands of type com is defined by a conventional abstract syntax represented as datatype:

```
datatype com  = SKIP
               | := loc aexp
               | Semi com com        (_ ; _ [60, 60] 10)
               | Cond bexp com com   (IF _ THEN _ ELSE _ 60)
               | While bexp com      (WHILE _ DO _ 60)          (119)
```

where the text in the parenthesis are just pragmas for the powerful Isabelle syntax engine to allows to use an intuitive notation.

8.3.2. Semantics of IMP

One of the operational semantics of IMP is a relation of triples evalc :: (com × state × state) set. A statement of the form: $(\mathrm{cm}, \sigma, \sigma') \in \mathrm{evalc}$ will be written as $\langle \mathrm{cm}, \sigma\rangle \xrightarrow[c]{} \sigma'$. The relation evalc is inductively defined:

$$\begin{array}{l}
\textbf{inductive}\ \mathrm{evalc}\ \textbf{intros}\\
\quad \langle \mathtt{SKIP}, \sigma\rangle \xrightarrow[c]{} \sigma\\
\quad \langle x :== a, \sigma\rangle \xrightarrow[c]{} \sigma(x := a\,\sigma)\\
\quad [\![\langle c_0, \sigma\rangle \xrightarrow[c]{} \sigma_1; \langle c_1, \sigma_1\rangle \xrightarrow[c]{} \sigma_2]\!] \Longrightarrow \langle c_0 \mathbin{;} c_1, \sigma\rangle \xrightarrow[c]{} \sigma_2\\
\quad [\![b\,\sigma; \langle c_0, \sigma\rangle \xrightarrow[c]{} \sigma_1]\!] \Longrightarrow \langle \mathtt{IF}\ b\ \mathtt{THEN}\ c_0\ \mathtt{ELSE}\ c_1, \sigma\rangle \xrightarrow[c]{} \sigma_1\\
\quad [\![\neg b\,\sigma; \langle c_1, \sigma\rangle \xrightarrow[c]{} \sigma_1]\!] \Longrightarrow \langle \mathtt{IF}\ b\ \mathtt{THEN}\ c_0\ \mathtt{ELSE}\ c_1, \sigma\rangle \xrightarrow[c]{} \sigma_1\\
\quad [\![\neg b\,\sigma]\!] \Longrightarrow \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c, \sigma\rangle \xrightarrow[c]{} \sigma\\
\quad [\![b\,\sigma; \langle c, \sigma\rangle \xrightarrow[c]{} \sigma_1; \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c, \sigma_1\rangle \xrightarrow[c]{} \sigma_2]\!] \Longrightarrow \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c, \sigma\rangle \xrightarrow[c]{} \sigma_2
\end{array} \tag{120}$$

The usual notation for the memory update operator $\sigma(x := v)$ is defined by $\lambda\, y.\ \text{if}\ y = x\ \text{then}\ v\ \text{else}\ \sigma\, y$. We will write $\sigma(x := v, y := u)$ for $\sigma(x := v)(y := u)$. This definition gives rise to a very simply theory called the *memory model*:

$$\begin{array}{l}
\sigma(x := v)(x) = v\\
x \neq y \Longrightarrow \sigma(y := u)(x) = \sigma(x).
\end{array} \tag{121}$$

From the rules Fact 120, an alternative rule set is derived that turns the Isabelle rewriter directly into a symbolic evaluation engine eventually collecting updates over the symbolic initial state σ:

$$\begin{array}{l}
\langle \mathtt{SKIP}, \sigma\rangle \xrightarrow[c]{} \sigma' = (\sigma' = \sigma)\\
\langle x :== a, \sigma\rangle \xrightarrow[c]{} \sigma' = (\sigma' = \sigma(x := a\,\sigma))\\
\langle x :== a \mathbin{;} S, \sigma\rangle \xrightarrow[c]{} \sigma' = (\langle S, \sigma(x := a\,\sigma)\rangle \xrightarrow[c]{} \sigma')\\
\langle \mathtt{IF}\ b\ \mathtt{THEN}\ c\ \mathtt{ELSE}\ d, \sigma\rangle \xrightarrow[c]{} \sigma' = (b\,\sigma \wedge \langle c, \sigma\rangle \xrightarrow[c]{} \sigma' \vee \neg b\,\sigma \wedge \langle d, \sigma\rangle \xrightarrow[c]{} \sigma')\\
b\,\sigma \Longrightarrow \langle \mathtt{WHILE}\ b\ \mathtt{DO}\ c, \sigma\rangle \xrightarrow[c]{} \sigma' = (\sigma' = \sigma)
\end{array} \tag{122}$$

Due to space limitations, we omit the definition of the denotational semantics reflecting the partial correctness $C :: \mathrm{com} \Rightarrow (\mathrm{state} \times \mathrm{state})\ \mathrm{set}$ (see [BW08] for details). It is linked to the operational semantics via the theorem

$$((\sigma, \sigma') \in C\ c) = \langle c, \sigma\rangle \xrightarrow[c]{} \sigma'\ . \tag{123}$$

In the denotational semantics, program transformations relevant for the next subsection are easily shown:

$$\begin{array}{l}
C(\mathtt{SKIP} \mathbin{;} c) = C(c) \qquad C(c \mathbin{;} \mathtt{SKIP}) = C(c) \qquad C((c \mathbin{;} d) \mathbin{;} e) = C(c \mathbin{;} (d \mathbin{;} e))\\
C((\mathtt{IF}\ b\ \mathtt{THEN}\ c\ \mathtt{ELSE}\ d); e) = C(\mathtt{IF}\ b\ \mathtt{THEN}\ c \mathbin{;} e\ \mathtt{ELSE}\ d \mathbin{;} e)\\
C(\mathtt{WHILE}\ b\ \mathtt{DO}\ c) = C(\mathtt{IF}\ b\ \mathtt{THEN}\ c \mathbin{;} \mathtt{WHILE}\ b\ \mathtt{DO}\ c\ \mathtt{ELSE}\ \mathtt{SKIP})
\end{array} \tag{124}$$

On the level of the denotational semantics, the usual notion of "valid Hoare triple" is defined as:

$$\models \{P\}\ c\ \{Q\} \equiv \forall \sigma\ \sigma'.\ (\sigma, \sigma') \in C(c) \longrightarrow P\,\sigma \longrightarrow Q\,\sigma' \tag{125}$$

where P, Q are *assertions*, i. e., functions from state to bool. From Fact 123 and Fact 125 we easily derive the following rule:

$$\frac{\bigwedge \sigma, \sigma'.\ \begin{array}{c}[P\,\sigma, \langle c, \sigma\rangle \xrightarrow[c]{} \sigma']\\ \vdots\\ Q\,\sigma'\end{array}}{\models \{P\}\ c\ \{Q\}} \tag{126}$$

which will turn out to be the key for program-based testing of IMP.

8.3.3. Unwind Theorems for IMP Programs

To perform white box tests in the style of Pathfinder [VPK04], Spec Explorer [GTV04] or Pex [dHT08], it is necessary to make the program paths explicit in the program representation and amenable to the rules of the operational semantics. Therefore, a pre-processing step —called unwinding—is necessary that normalizes the

program and unfolds all WHILE-loops up to a certain limit, which is called the *unwind-factor* k. This principle can also be applied in a language with procedure calls; however, a re-use of existing test cases from previously tested procedures—as used, e. g., in Pex—is advisable to bound the blow-up of symbolic states. Additionally, our pre-processing step will transform the program into a certain normal form to be efficiently processed by the symbolic evaluation rules shown in Fact 122. In particular, left associative sequential compositions as well as conditionals sequenced with other statements must be avoided since our rule-set does not simplify them. We define two recursive functions on com-terms that perform both these normalizations as well as the unwinding up to k. We will not program this function outside the logic as tactic, i. e., a control program in SML, but inside HOL, such that we can also prove its correctness with respect to the IMP semantics:

$$\begin{array}{ll} \text{primrec} & _@@_ :: [\text{com}, \text{com}] \Rightarrow \text{com} \\ \text{where} & \texttt{SKIP}\,@@c = c \\ & (x :== E)@@c = ((x :== E)\,;\,c) \\ & (c\,;\,d)@@e = (c\,;\,d@@e) \\ & (\texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d)@@e = (\texttt{IF}\ b\ \texttt{THEN}\ c@@e\ \texttt{ELSE}\ d@@e) \\ & (\texttt{WHILE}\ b\ \texttt{DO}\ c)@@e = ((\texttt{WHILE}\ b\ \texttt{DO}\ c)\,;\,e) \end{array} \tag{127}$$

$$\begin{array}{ll} \text{fun} & \text{unwind} :: \text{nat} \times \text{com} \Rightarrow \text{com} \\ \text{where} & \text{unwind}(n, \texttt{SKIP}) = \texttt{SKIP} \\ & \text{unwind}(n, a :== E) = (a :== E) \\ & \text{unwind}(n, \texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d) = \texttt{IF}\ b\ \texttt{THEN}\ \text{unwind}(n, c)\ \texttt{ELSE}\ \text{unwind}(n, d) \\ & \text{unwind}(n, \texttt{WHILE}\ b\ \texttt{DO}\ c) = \ \text{if}\ 0 < n \\ & \qquad \text{then}\ \texttt{IF}\ b \\ & \qquad\quad \texttt{THEN}\ \text{unwind}(n, c)@@\,\text{unwind}(n-1, \texttt{WHILE}\ b\ \texttt{DO}\ c) \\ & \qquad\quad \texttt{ELSE}\ \texttt{SKIP} \\ & \qquad \text{else}\ \texttt{WHILE}\ b\ \texttt{DO}\ \text{unwind}(0, c) \\ & \text{unwind}(n, \texttt{SKIP}\,;\,c) = \text{unwind}(n, c) \\ & \text{unwind}(n, c\,;\,\texttt{SKIP}) = \text{unwind}(n, c) \\ & \text{unwind}(n, (\texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ d)\,;\,e) = (\texttt{IF}\ b\ \texttt{THEN}\ (\text{unwind}(n, c\,;\,e))\ \texttt{ELSE}\ (\text{unwind}(n, d\,;\,e))) \\ & \text{unwind}(n, (c\,;\,d)\,;\,e) = (\text{unwind}(n, c\,;\,d))@@(\text{unwind}(n, e)) \\ & \text{unwind}(n, c\,;\,d) = (\text{unwind}(n, c))@@(\text{unwind}(n, d)) \end{array} \tag{128}$$

The primitive recursive auxiliary function $c@@d$ appends a command d to the last command in c that is reachable from the root via sequential composition modes. The more tricky unwind function unfolds WHILE-loops as long as the unwind factor is positive and performs the program normal form computation along the program equivalences as discussed in Fact 124.

Isabelle will adopts a “first fit” pattern matching strategy (similar to SML) when processing the recdef construct. This means that in overlapping cases, the first match is taken into account with higher priority—this is reflected on the level of the rewrite rule set generated from this definition. Thus, the last equation in the recursive definition is a catch-all rule for sequential composition.

Lemma 1 (Termination:). Both functions terminate.

Proof. In the case of _@@_ this is trivial due to machine checked primitive recursion; in case of unwind termination is established by providing a well-founded ordering. In this case, the lexicographic composition of the standard ordering $_ < _$ and the standard term ordering suffices since all inner calls in this recursive definition use smaller arguments with respect to this ordering. This proof is done fully automatically. □

Lemma 2 (Correctness:). $C(c@@d) = C(c; d)$ and $C(\text{unwind}(n, c)) = C(c)$

Proof. For _@@_, a straight-forward induction suffices. As for unwind, the proof is non-trivial, but routine (generalization over n, induction over c, intricate case splitting, application of semantic equivalences of Fact 124). □

8.3.4. Constructing the Unwinding-hypothesis

In the following, we will show that for a given program c and a given unwind-factor k, we can derive on-the-fly an equivalent for the data-separation lemma Fact 15 that can be used for the generation of test cases, which correspond to paths through our program.

As example, we chose a little program that computes the square-root of a natural number. It is defined in Isabelle/IMP syntax as follows:

$$\begin{array}{lll} \text{definition} & \text{squareroot} & :: [\text{loc},\text{loc},\text{loc},\text{loc}] \Rightarrow \text{com} \\ \text{where} & \text{squareroot}\ tm\ sum\ i\ a \equiv & tm :== \lambda\, s.\ 1\,;\\ & & sum :== \lambda\, s.\ 1\,;\\ & & i :== \lambda\, s.\ 0\,;\\ & & \texttt{WHILE}\ \lambda\, s.\ (s\ sum) \leq (s\ a)\ \texttt{DO}\\ & & \quad i :== \lambda\, s.\ (s\ i) + 1\,;\\ & & \quad tm :== \lambda\, s.\ (s\ tm) + 2\,;\\ & & \quad sum :== \lambda\, s.\ (s\ tm) + (s\ \ sum) \end{array} \tag{129}$$

where the locations (references) represent the input variables. The shallow embedding of the expressions has the consequence that program variable accesses must be represented as explicit application of the state s (at this program point) to a location representing this variable. Hence, we implicitly require a pre-parser that makes these bindings of program variables explicit.

Putting this symbolic computation together, we can now formulate our test specification, which states that our IMP-program squareroot conforms to its specification, i. e. its pair of pre and post-conditions:

$$\begin{array}{l} \text{testspec program_based_test:}\\ \text{assumes no_alias:}\ \ : i \neq a \wedge i \neq \text{sum} \wedge i \neq tm \ldots\\ \text{shows}\ \ \models \{\lambda\, \sigma.\ true\}\, \text{squareroot}\ tm\ sum\ i\ a \{\lambda\, \sigma. (\sigma i)^2 \leq (\sigma a) \wedge \sigma a < (\sigma i + 1)^2\} \end{array} \tag{130}$$

where the technical side-condition no_alias specifies that the locations tm, sum, i, and a are pairwise disjoint, i. e., they are not alias of each other. We will introduce the abbreviations `pre` and `post` for the corresponding expressions in the Hoare-Triple above.

In the sequel, we present a hand-simulation reflecting the essential steps that the test case generation procedure yields. Elementary simplification of `pre` and application of Fact 126, Fact 123 and the correctness theorem Lemma 2 (by instantiating the unwind-factor k to 3) reduces the above formula to:

$$i \neq a \wedge i \neq \text{sum} \wedge i \neq tm \ldots \Longrightarrow \langle \text{unwind}(3, \text{squareroot}\ tm\ sum\ i\ a), \sigma \rangle \xrightarrow[c]{} \sigma' \Longrightarrow \texttt{post}\ \sigma' \tag{131}$$

After unfolding the definition of squareroot, the loop unwinding (along Fact 127 and Fact 128) yields a program term which is symbolically executed via Fact 122 with the help of the memory rules Fact 121 and the facts `no_alias`.

The resulting proof-state consists of the following goals:

$$\begin{array}{ll} 1. & [\![9 \leq \sigma\ a; \langle \texttt{WHILE}\ \lambda \sigma.\ \sigma\ sum \leq \sigma\ a\ \texttt{DO}\ i :== \lambda \sigma.\ \text{Suc}(\sigma\ i)\,;\\ & \quad (tm :== \lambda \sigma.\ \text{Suc}(\text{Suc}(\sigma\ tm))\,; sum :== \lambda \sigma.\ \sigma\ tm + \sigma\ sum),\\ & \quad \sigma(i := 3, tm := 7, sum := 16) \rangle \xrightarrow[c]{} \sigma']\!] \Longrightarrow \texttt{post}\ \sigma'\\ 2. & [\![4 \leq \sigma\ a; 8 < \sigma\ a; \sigma' = \sigma\ (i := 2, tm := 5, sum := 9)]\!] \Longrightarrow \texttt{post}\ \sigma'\\ 3. & [\![1 \leq \sigma\ a; \sigma\ a\ < 4; \sigma' = \sigma\ (i := 1, tm := 3, sum := 4)]\!] \Longrightarrow \texttt{post}\ \sigma'\\ 4. & [\![\sigma\ a = 0; \sigma' = \sigma\ (tm := 1, sum := 1, i := 0)]\!] \Longrightarrow \texttt{post}\ \sigma' \end{array} \tag{132}$$

The resulting proof state enumerates the possible symbolic states σ', explained in terms of a series of updates θ from σ, up to certain depth including their path conditions. The path to the remaining, not fully unfolded loop represents the class of the paths not yet explored.

The symbolic computation described above can be summarized by a rule, that can be computed on-the-fly from the redex $\models \{P\} cmd \{Q\}$ and the unfold-factor k. Analogously to the standard datatype exhaustion theorem Fact 15, we call this rule the *unwinding-hypothesis theorem* (for factor $k = 3$ and program squareroot tm sum $i\ a$). In our case, it has the following form:

$$\frac{\begin{array}{c}[P\ \sigma, path_1\ \sigma]\\ \vdots\\ \bigwedge \sigma.\quad Q(\theta_1\ \sigma)\end{array} \quad \cdots \quad \begin{array}{c}[P\ \sigma, path_m\ \sigma]\\ \vdots\\ \bigwedge \sigma.\quad Q(\theta_m\ \sigma)\end{array} \qquad \text{THYP}(\forall \sigma.\ P\ \sigma \longrightarrow path_{m+1}\ \sigma \longrightarrow Q(\theta_{m+1})\ \sigma)}{\models \{P\}\ c\ \{Q\}} \tag{133}$$

where $path_i$ (with $i \in \{1, \ldots, m+1\}$ are the *path conditions* (in our example: $\sigma\ a = 0$, $1 \leq \sigma\ a \wedge \sigma\ a < 4$ and $4 \leq \sigma\ a \wedge 8 < \sigma\ a$) and where the $\theta_i\ \sigma$ (with $i \in \{1, \ldots, m\}$) are the *simple symbolic states*, i. e., those

that were simply constructed via update from the initial state (in our example: $\sigma(tm := 1, sum := 1, i := 0)$, $\sigma(i := 1, tm := 3, sum := 4)$ and $\sigma(i := 2, tm := 5, sum := 9)$). The $\theta_i\,\sigma$ (here only: $i = m + 1$) that are not simple, i. e., do not exclusively consist of updates, were collected in the test-hypothesis. In our example, this leads to the test-hypothesis H:

$$\begin{aligned}\mathrm{THYP}(\forall\sigma.\ 9 \leq \sigma a \longrightarrow \langle & \texttt{WHILE}\ \lambda\,\sigma.\ \sigma\ sum \leq \sigma\ a\ \texttt{DO} \\ & i :== \lambda\,\sigma.\ \mathrm{Suc}(\sigma\ i)\ ; \\ & tm :== \lambda\,\sigma.\ \mathrm{Suc}(Suc(\sigma\ tm))\ ; \\ & sum :== \lambda\,\sigma.\ \sigma\ tm + \sigma sum, \\ & \sigma(i := 3, tm := 7, sum := 16)\rangle \underset{\mathrm{c}}{\rightarrow} \sigma' \longrightarrow post\ a\ i\ \sigma')\end{aligned} \tag{134}$$

A test-hypothesis of this form—representing a variant of a regularity-hypothesis over the program structure—formalizes precisely what is left, due to the fact that our symbolic evaluation rule set Fact 122 cannot handle `WHILE` loops. Note the computing time for unwind-factor 10 based on this simplistic implementation remains under a second, including pretty-printing. Further techniques (like *normalization by evaluation* (NBE) [AHN08], which applies a combination of compilation and optimized term-rewriting) offer enough potential for scaling up to large, realistic program examples; however, we consider a setup and a detailed experimental evaluation along this line as out of the scope of this paper.

8.3.5. White-box Test-scenarios

HOL-TestGen can be configured to apply splitting rules generated on-the-fly like data separation lemmas Fact 15 or unwinding theorems like Fact 133 to arbitrary parts of a test specification. Thus, for test specifications like Fact 130, we can apply our test case generation procedure **gen_test_cases** and get as test cases all path conditions generated along the *k-path all-instruction* path coverage criteria (see [ZHM97] for a comprehensive overview of various path-based coverage criteria); this coincides with *all-instruction* for $k = 1$; weaker criteria can be implemented if more paths were tagged as explicit test-hypothesis during the unfold-process or if less aggressively unwinding strategies were applied in the unwind-operator.

A proof state resulting from the application of an unfolding-hypothesis theorem can be used in two fundamentally different ways for test:

1. as *automated partial verification method*, or
2. as *white box testing method*.

An automated partial verification method results if the non-THYP clauses of the above unwinding-hypothesis theorem are fed into an automated theorem prover, attempting to prove the postcondition for this test case. In many cases (as our example), the prover will simply be able to resolve them (without using the uniformity-hypothesis), thus leaving the test-hypothesis as a marker for what had *not* been done in this approximate proof: the induction, or alternatively: the invariant construction involving generalization and instantiation, which would establish the postcondition for *all* legal inputs. Still, such a partial verification method has due to its high degree of automation a high practical value: it leverages the *early* debugging of specifications, i. e., preconditions and postconditions.

A white box testing method results, if we apply the uniformity-hypothesis for each path which allows to reduce the problem to a concrete ground state that satisfies the precondition and path condition, and test if the postcondition can be evaluated to true on the resulting updated state. After constraint-solving, this approach does not depend on automated theorem proving any longer, simply on computing.

These two scenarios show in which ways an end-user may profit from having a *spectrum* between full-blown verification and systematic test. Here, the trade-off is between using more and more explicit test-hypothesis while increasing the degree of proof automation.

9. Conclusion

We have presented the theory and pragmatics of HOL-TestGen, a specification and test case generation environment extending the interactive theorem prover Isabelle/HOL. HOL-TestGen allows for a workflow integrating both automated deduction techniques as well as interactive theorem proving for test case generation. Particular emphasis is put on the novel concept of explicit test-hypotheses which establish a formal

link between test and proof. Our implementation is based entirely on derived rules and tactical programs controlling them; thus, to the best of our knowledge, HOL-TestGen is the only *formally verified* test-system.

9.1. Related Work

In 2002, a report by the US National Institute of Standards and Technology estimates that software failures cost the US economy between $ 20 and $ 60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost [GK02]. Despite this significant economic importance, which is also conformed by industrial applications ([GKM+08]), hand-written and unsystematic tests are still predominant in the industrial practice, while as the number of test data generators based on a user-defined models is quite low. In the following, we distinguish roughly two classes of approaches: *input enumeration* methods and *symbolic* methods.

Into the former class, we count QuickCheck [CH00], a typical random-based test generator. Input is enumerated via a pseudo-random number generator, which is a (usually long-cycle) enumeration function. The original implementation for Haskell inspired many random-based test data generators for various different programming languages. They usually provide a test execution and test result verification environment. In practice, the random tests were improved by hand-programmed test data generators. As already discussed in Sec. 3.4 and Sec. 8.1, random test can be ineffective in many cases, in particular, if preconditions rule out most of randomly generated data.

A further candidate is Spec Explorer [VCG+08], which is a major model-based testing tool for software components specified in Spec# or AsmL, The system unwinds the resulting finite state machine to produce behavioral tests that cover all explored transitions. A binding mechanism allows users to associate actions of the model with methods of an implementation written in any .net language. Spec Explorer extracts the specifications from a central document containing informal and formal parts; HOL-TestGen shares the document-centric approach with Spec Explorer. Spec Explorer [VCG+08] has been used in impressive industrial-strength case studies [GKM+08].

Based on labeled i/o-transition systems, TorX [TB03] is a test generator for test sequence generation that is centered around i/o-conformance. Compared with HOL-TestGen, these tools are developed *specifically* for automata-oriented models (such labeled transition systems can be generated from various input languages such as SDL or LOTOS). Non-determinism is managed via on-line test generation and execution using ad-hoc test cases test selection criteria. TorX has been used in medium-size case studies.

In the following, we focus on the class of symbolic approaches which are conceptually closer HOL-TestGen, be it based on a formal specification, or based additionally on a program.

The Loft-System and its successor Gatel [MB05] are perhaps the closest to our approach: as test theories, algebraic specifications restricted to first-order Horn-clauses were used. However, this rules out inherent second-order constructions like "inductively defined sets" as we used in the operational semantics for white-box tests and severely limits the possibility to derive essential rules inside the system. Moreover, in contrast to HOL-TestGen, whose redex-search strategy is (data)type-centric, Gatel uses a rule-centric approach for superpositions of test theory axioms into the test-theorem; this turned out to be far more difficult to control and to extend than our procedure. And, last-but-not-least, Gatel generates no explicit test-hypotheses and uses therefore unsound versions of regularity-hypotheses (cf. Fact 15).

Moreover, there are commercial test tools that are especially designed for the use in industrial applications, for example, LEIRIOS[7] Test Generator [JL07] or ConformiQ Qtronic [Hui07]. Both tools support a high-level system specification, written in B and UML/OCL (LEIRIOS Test Generator) or annotated state-sequence charts (ConformiQ Qtronic) and apply symbolic constraint solving techniques for generating test cases that cover all symbolic execution paths of the specification. In addition, these tools contain datatype theories describing "interesting input values" (boundary cases) to ensure that each parameter of each method is executed at least once for every boundary case.

TGV [JJ05] is oriented like TorX [TB03] towards labeled i/o-transition systems and centered around i/o-conformance; in contrast to HOL-TestGen, it is therefore specialized towards a specific testing technique. In contrast to TorX it uses truly symbolical techniques for state representation; however, the limitations of the used constraint solving techniques are quite painful. Non-determinism is again managed via on-line test

[7] The product is now called *Smartesting Design Center.*

generation and execution using ad-hoc test cases test selection criteria. Moreover, TGV can be integrated into commercial simulation environments for LOTOS specifications.

For program-based tests, there are two test data generators that apply symbolic techniques: Korat [BKM02]and Java Pathfinder [VHB+03]. Korat [BKM02] generates from preconditions and a bound on the number of nodes of data-structures (thus similar to 2^k in our regularity-hypothesis for binary trees), an input partitioning by a combination of symbolic execution and (simple) constraint solving; since data-structures have to be encoded as object-graphs within a state, the system has to cope with many isomorphic representations of data-structures. Unlike traditional debuggers, Java Pathfinder reports the entire execution path that leads to a defect. In practice, the search space exploration has to be instrumented by “hints” resembling preconceived generators in random testing procedures to find data satisfying the preconditions effectively.

The idea of integrating a symbolic state deeply inside the execution environment, i. e., inside a Java virtual machine (JVM) as suggested in JPF-SE [APV07], substantially improved the approach and inspired systems such as Pex [TdH08]; the latter is model-based testing tools for the .net framework in general and programs written in C# in particular. Recent versions support statically linked procedures like operating system calls to be replaced by stubs and other features necessary for a practically useful tool.

9.2. Achievements

HOL-TestGen *is a unifying framework* for a wide range of testing-techniques. In this paper, we have demonstrated its practical use for unit tests, sequence tests, reactive sequence tests, and provided proofs of technology for error-based domain analysis tests, concurrent system tests as well as white-box tests. Therefore, our system and methodology represents a basis for theoretical analysis of test-technique combinations as well as a practical testbed for their application. For example, proving lemmas and using them for the “logical massage” of policies in test specifications paves the way for eliminating redundant test cases by computing a semantically equivalent, but “simpler” policy with respect to time and space consumption [BBKW10].

HOL-TestGen *leverages explicit test-hypothesis* and suggests the concept as an alternative to traditional implicit *test adequacy criteria* [ZHM97]: instead of telling *when we have tested enough* we concentrate on *what remains to prove*. Besides the traditional partition coverage test-adequacy which is equivalent to our unit-test scenarios, regularity-hypothesis can be used to simulate all-transition-coverage of automata (if d is chosen to be the length of the longest path), and unwinding-hypothesis (for $k = 1$) simulates path-coverage over the program (and when adding additional weakening test-hypothesis, also condition coverage). In contrast to classical test-adequacy criteria, however, explicit test-hypothesis can be viewed as proof-obligations to be proven later. Although we do not see any evidence that this simplifies the proof, tests can be used as fast checks of the correspondence between programs and specifications. Thus, testing can make the overall modeling- and verification effort more effective.

HOL-TestGen *puts emphasis on the spectrum between test and proof* rather than the idea that there is a sharp contrast between them, as is implicitly expressed by the famous verdict of Dijkstra [DDH72, p. 6]: “Program testing can be used to show the presence of bugs, but never to show their absence!” In our setting, a specification-based test is an approximation to verification, which can be completed with the verification of the test-hypothesis.

And finally, systematic tests have complementary assets to verification: if legacy code or unknown system implementations are given, testing allows for systematically *experimenting* with them, allows to *validate* the modeling assumptions made. This may be inherently necessary for legacy-components as well as assumptions over the environment of a system. Recently, this potential to reverse-engineer legacy code by model-based testing has been acknowledged as crucial in industrial applications [GKM+08].

9.3. HOL-TestGen-Experiences with Large-scale Applications

There are many large-scale applications of Isabelle/HOL in various areas including pure mathematics (e. g., proving important parts of Hale’s proof of the Kepler Conjecture [LMR08]), programming language semantics (e. g., formalizing Java [vO01]), or verification of real-world systems (e. g., verifying the L4 operating system micro-kernel [Kle09]) showing both the applicability of HOL as a flexible formalism for expressing large and complicated specifications and Isabelle/HOL as framework of formally proving properties over such

specifications. By using Isabelle/HOL as a basis for HOL-TestGen, we transfer, on a technical level and a conceptual level, the benefits that allowed the use of IsabelleHOL in those applications to the testing area.

While we emphasize in this paper the wide-spread applicability of the approach and therefore concentrated on small, paradigmatic test specifications, we successfully used HOL-TestGen to large-scale applications such as compliance testing of network firewalls or generating test cases for the infrastructure of the National Health Service (**nhs!**). In all these applications, we made the experience that combining theorem proving techniques and testing techniques can improve the overall quality of the generated test cases and test data.

In the case of testing firewall conformance, we started by modeling stateless packet filters (stateless firewalls) and their security policy in HOL (see [BBW08] for details). Based on this specification, we generated test cases for testing that a real firewall implements a specific security policy (using a specialized test setup for the test result verification phase). As packet filters do not need to keep track of an internal protocol state for their decision to accept or deny a package, this application is a instance of the unit-testing scenario. Furthermore, we developed this application of HOL-TestGen into an instantiation of our sequence-testing scenario allowing for modeling and testing stateful application level firewalls (see [BW07]) for details. Generating test cases, in a naïve way, for real-world firewall policies can take more than 24 hours of computing time. By developing a formally verified policy normalization (see [BBKW10]), we were able to reduce the required computing time to less than 24 seconds. Moreover, as this normalization also resulted in a decrease of the size of the test data sets, the resulting test execution takes also significantly less time. Especially this experiences emphasizes the technical advantages of using a generic theorem prover, together with a powerful and flexible specification language, as the basis for a specification-based test case generation tool.

In case of the National Health Service, we developed a modular policy modeling framework in HOL which we could instantiate with the key security mechanisms of applications and services of the NPfIT. NPfIT, the National Programme for IT, is a very large-scale development project aiming to modernize the IT infrastructure of the National Health Service (**nhs!**) in England. Consisting of heterogeneous and distributed applications, it is an ideal target for model-based testing techniques of a large system exhibiting critical security features. We modeled the four information governance principles, comprising a role-based access control model, as well as policy rules governing the concepts of patient consent, sealed envelopes and legitimate relationship. For this application, we developed two kind of test specifications: the first class of test specifications ensure certain quality criteria (e. g., that a policy is always defined) and the second class of test specifications ensures the policy conformance of the applications under test (see [BBKWed] for details). For this application, we ported our test harness to the .net platform which enables us to use the features of the .net environment for testing Web service-based applications.

9.4. Future Work

The ultimate goal of future improvements is to extend the realm of feasible state-spaces for HOL-TestGen system substantially as well as to increase the degree of automation. We suggest a combination of four techniques to achieve this goal:

1. more refined theories that relate high-level test specification goals such as $I \sqsubseteq_T S$ or i/o-conformance I ioco S to concrete, monad-based test-driver implementations as shown in Sec. 5.1,
2. integration of more high-level support for process representations, in particular for concurrent models (a possible starting point could be the integration of HOL-CSP [TW97]),
3. more automated support for the overall process to scale up to larger test models (this involves deep integration of new parallel computation facilities, new evaluation mechanisms such as NBE [AHN08] which are available in most recent versions of Isabelle),
4. more automated *use* of the knowledge contained in explicit test-hypothesis; for example, automated provers could establish that one test-set is actually a test refinement of another since the hypothesis' are strictly weaker,
5. derived rules from the datatype theories and recursively defined function definitions over them that help to detect unsatisfiable test cases early, i. e., ways to automate the reasoning behind Sec. 4.3, and
6. ways to combine our approach based on discrete uniformity (assuring the construction of one ore more tests in each test case) with randomized uniformity (where only the *randomly* uniform distribution of

tests along all test cases is guaranteed); approaches such as [GDG+08] are another promising line to scale up to larger test models.

Acknowledgments: Lukas Brügger, Ana Cavalcanti, Abdou Feliachi, and Marie-Claude Gaudel made valuable comments on earlier versions of this paper.

References

[AHN08] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 39–54, Heidelberg, August 2008. Springer-Verlag.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002.

[APV07] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138, Heidelberg, 2007. Springer-Verlag.

[BBKW10] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. Verified firewall policy transformations for test-case generation. In *Third International Conference on Software Testing, Verification, and Validation (ICST)*, pages 345–354. 2010.

[BBKWed] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. An approach to modular and testable security models of real-world health-care applications. submitted.

[BBW08] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Model-based firewall conformance testing. In Kenji Suzuki and Teruo Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[BN04] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM)*, pages 230–239, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[BPZ09] Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *TestCom/FATES*, volume 5826 of *Lecture Notes in Computer Science*, pages 16–32, Heidelberg, 2009. Springer-Verlag.

[BTV09] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 307–321, Heidelberg, 2009. Springer-Verlag.

[BW04] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, 2004.

[BW05] Achim D. Brucker and Burkhart Wolff. Interactive testing using HOL-TestGen. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Testing of Software*, number 3997 in Lecture Notes in Computer Science. Springer-Verlag, 2005.

[BW07] Achim D. Brucker and Burkhart Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In Bertrand Meyer and Yuri Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[BW08] Achim D. Brucker and Burkhart Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249, 2008.

[BW09] Achim D. Brucker and Burkhart Wolff. HOL-TestGen: An interactive test-case generation framework. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY USA, 2000. ACM Press.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[cis08] Securing cyberspace for the 44th presidency. Technical report, Center for Strategic and International Studies (CSIS), December 2008.

[DDH72] 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, 3rd edition, 1972.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, Heidelberg, April 1993. Springer-Verlag.

[DGHP96] Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods.* Kluwer, Dordrecht, 1996.

[dHT08] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with Pex. In Bernhard Beckert and

Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181, Heidelberg, 2008. Springer-Verlag.

[Fos80] K.A. Foster. Error sensitive test cases analysis (estca). *Software Engineering, IEEE Transactions on*, SE-6(3):258–264, May 1980.

[Gau95] Marie Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, TAPSOFT *'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 82–96. Springer-Verlag, Heidelberg, 1995.

[GDG+08] Marie-Claude Gaudel, Alain Denise, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased random exploration of models. *Electronic Notes in Theoretical Computer Science*, 220(1):3–14, 2008.

[GK02] M.P. Gallaher and B.M. Kropp. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-03, National Institute of Standards & Technology, May 2002.

[GKM+08] Wolfgang Grieskamp, Nicolas Kicillof, Dave MacDonald, Alok Nandan, Keith Stobie, and Fred L. Wurden. Model-based quality assurance of windows protocol documentation. In *Software Testing, Verification, and Validation (ICST)*, volume 0, pages 502–506, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[GTV04] Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, 2004.

[Hui07] Antti Huima. Implementing conformiq qtronic. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, Heidelberg, 2007. Springer-Verlag.

[JJ05] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JL07] Eddie Jaffuel and Bruno Legeard. Leirios test generator: Automated test generation from b models. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 277–280, Heidelberg, 2007. Springer-Verlag.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[LMR08] Christoph Lange, Sean McLaughlin, and Florian Rabe. Flyspeck in a semantic Wiki. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *SemWiki*, volume 360 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[MB05] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in GATeL. *Electronic Notes in Theoretical Computer Science*, 111:93–111, 2005.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

[Ros98] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[TB03] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex—white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153, Heidelberg, 2008. Springer-Verlag.

[TW97] Haykal Tej and Burkhart Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76, Heidelberg, 2008. Springer-Verlag.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52, Heidelberg, 1995. Springer-Verlag.

[Wen02] Markus M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, February 2002.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

[ZHM97] Hong Zhu, Patrick A.V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

A. Case-study: Sequence Testing Red-Black Trees

Here is an implementation of the reb-black tree insertion used in the example, based on a transcription of the red-black tree implementation[8] provided by sml/NJ into Isabelle/HOL.

```
datatype ml_order = LESS | EQUAL | GREATER

class LINORDER = linorder +
  fixes compare :: "'a ⇒ 'a ⇒ ml_order"
  assumes LINORDER_less : "((compare x y) = LESS) = (x < y)"
    and LINORDER_equal : "((compare x y) = EQUAL) = (x = y)"
    and LINORDER_greater : "((compare x y) = GREATER) = (y < x)"

datatype color = R | B
datatype 'a tree = E | T color "'a tree" "'a" "'a tree"

(* insert *)
fun
  ins :: "'a::LINORDER ×'a tree ⇒'a tree"
where
  ins_empty : "ins (x, E) = T R E x E"
| ins_branch : "ins (x, (T color a y b)) =
  (case (compare x y) of
    LESS ⇒(case a of
            E ⇒ (T B (ins (x, a)) y b)
          |(T m c z d) ⇒ (case m of
                           R ⇒ (case (compare x z) of
                                 LESS ⇒(case (ins (x, c))of
                                          E ⇒ (T B (T R E z d) y b)
                                         |(T m e w f) ⇒
                                        (case m of
                                          R ⇒ (T R (T B e w f) z (T B d y b))
                                        | B ⇒ (T B (T R (T B e w f) z d) y b)))
                                |EQUAL ⇒(T color (T R c x d) y b)
                                |GREATER ⇒(case (ins (x, d)) of
                                            E ⇒ (T B (T R c z E) y b)
                                           |(T m e w f) ⇒ (case m of
                                              R ⇒ (T R (T B c z e) w (T B f y b))
                                             |B ⇒ (T B (T R c z (T B e w f)) y b))
                                           )
                                )
                         | B ⇒ (T B (ins (x, a)) y b))
          )
  | EQUAL ⇒(T color a x b)
  | GREATER ⇒(case b of
               E ⇒ (T B a y (ins (x, b)))
             |(T m c z d) ⇒ (case m of
                              R ⇒(case (compare x z) of
                                   LESS ⇒(case (ins (x, c)) of
                                           E ⇒ (T B a y (T R E z d))
                                          |(T m e w f) ⇒ (case m of
                                              R ⇒ (T R (T B a y e) w (T B f z d))
                                             |B ⇒ (T B a y (T R (T B e w f) z d)))
```

[8] Provided in the file `int-redblack-set.sml`, which itself is part of the archive `http://smlnj.cs.uchicago.edu/dist/working/110.53/smlnj-lib.tgz`.

```
                                          )
                                    | EQUAL ⇒(T color a y (T R c x d))
                                    | GREATER ⇒(case (ins (x, d)) of
                                                 E ⇒ (T B a y (T R c z E))
                                                |(T m e w f) ⇒ (case m of
                                                    R ⇒ (T R (T B a y c) z (T B e w f))
                                                   |B ⇒ (T B a y (T R c z (T B e w f))))
                                                   )
                               )
                          |B ⇒ (T B a y (ins (x, b)))))
  )"

definition insert :: "['a::LINORDER, 'a tree] ⇒'a tree"
where "insert a t = (case ins (a,t) of
                       E ⇒ E
                     | T R (T R l' e' r') e r ⇒ T B (T R l' e' r') e r
                     | T R E e r              ⇒ T R E e r (* id *)
                     | T R l e (T R l' e' r') ⇒ T R l e (T R l' e' r')
                     | T R l e E              ⇒ T R l e E (* id *))"

declare insert_def[simp] (* this definition is a computational rule *)
```