Symbolic Test Case Generation for Primitive Recursive Functions

Achim D. Brucker and Burkhart Wolff

Information Security, ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland.
{brucker, bwolff}@inf.ethz.ch

Abstract We present a method for the automatic generation of test cases for HOL formulae containing primitive recursive predicates. These test cases can be used for the animation of specifications as well as for black-box testing of external programs.
Our method is two-staged: first, the original formula is partitioned into test cases by transformation into a Horn-clause normal form (HCNF). Second, the test cases are analyzed for instances with constant terms satisfying the premises of the clauses. Particular emphasis is put on the control of test hypotheses and test hierarchies to avoid intractability.
We applied our method to several examples, including AVL-trees and the red-black tree implementation in the standard library from SML/NJ.
Keywords: symbolic test case generations, black box testing, theorem proving, Isabelle/HOL

1 Introduction

Today, essentially two software validation techniques are used: *software verification* and *software testing*. Whereas verification is rarely used in "large-scale" software development, testing is widely used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Dijkstra's verdict* [11, p.6]:

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

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

- *Abstraction Techniques:* model-checking raised interest in techniques to abstract infinite models to finite ones. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [5, 9].
- *Systematic Testing:* the discussion over *test adequacy criteria* [21], i.e., criteria answering the question "when did we test enough to meet a given test hypothesis", led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [12, 13].

– *Specification Animation:* constructing counter-examples has raised interest also in the theorem proving community, when combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [14].

The first two areas are motivated by the question "are we building the program right?", the latter is focused on the question "are we specifying the right program?". While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows that it simply does not apply to important situations in practice. In particular, if a formal model of the environment of a software system (e.g., based on, amongst other things, the operating system, middleware or external libraries) must be reverse-engineered, testing — in the sense of "experimenting" — is without alternative (see [7]).

Following standard terminology [21], our approach is a *specification-based unit test*. A test procedure for such an approach can be divided into:

– *Test Case Generation:* for each operation, the pre/post-condition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.
– *Test Data Selection:* for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.
– *Test Execution:* the implementation is run with the selected test input data in order to determine the test output data.
– *Test Result Verification:* the pair of input/output data is checked against the specification of the test case.

As an example for a specification-based unit-test approach, QuickCheck [8] has attracted interest in various research communities. QuickCheck performs random tests, potentially improved by hand-programmed test data generators, and provides a simple test execution and test result verification environment for programs written in Haskell.

However, it is well-known that random test can be ineffective in many cases[1]; in particular, if complex preconditions of programs like "the input tree must be balanced" or "the input must be a well-formed abstract syntax tree" rule out most of randomly generated data. In our approach, we will exploit the specification of pre- and postconditions of a program — the *test specification* — in a preprocessing step, the *test case generation*. Our implementation `TestGen` of a test case generator is built on top of the theorem prover Isabelle/HOL [17]. Isabelle is programmed to execute the underlying symbolic computations in an automatic, but logically safe way. Based on the resulting *test cases*, a random test based data selection procedure can be controlled in a problem-oriented way and achieve a significantly better test coverage. As a particular feature, the automated deduction-based process can log the test hypothesis underlying the test.

[1] Consider `abs(x-2) >= 0` where `abs` from the Haskell Integer library computes the absolute value. Here it is very unlikely that QuickCheck finds the problem...

Provided that the test hypotheses are valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification.

We proceed as follows: we will introduce our implementation built on top of the theorem prover Isabelle by a tiny, but classical example [12] (Sec. 2). This demonstration serves as a means to motivate concepts like *test specification*, *testing normal form*, *test cases*, *test statements*. In Sec. 3, we will discuss the test case generation in more detail. In Sec. 4, we will discuss a technique for controlling the *state explosion* by generating *abstract test cases*. Finally, we apply our technique to a number of non-trivial examples (Sec. 5) involving recursive data types and recursive predicates and functions over them.

2 Symbolic Test Case Generation: A Guided Tour

Our test case generator `TestGen` is integrated into the specification and theorem proving environment Isabelle/HOL. As a specification language, HOL offers data types, recursive function definitions and fairly rich libraries with theories of, e.g., arithmetics; it is often viewed as a "functional programming language with logical quantifiers". As a theorem proving environment, Isabelle is based on a relatively small proof engine (based on higher-order resolution) providing a *proof state* that can be transformed via elementary *tactics* into logically equivalent ones, until a final proof state is reached where a derived formula has the appropriate form.

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

datatype Triangles := equilateral | scalene | isosceles | error

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

$$\begin{aligned}\textbf{constdefs}\ & \mathit{triangle} :: [\mathrm{nat}, \mathrm{nat}, \mathrm{nat}] \to \mathrm{bool} \\ & \mathit{triangle}\ x\ y\ z \equiv (0 < x) \wedge (0 < y) \wedge (0 < z) \wedge (z < x + y) \\ & \qquad \wedge (x < y + z) \wedge (y < x + z)\end{aligned}$$

Now we define the behavior of the triangle program by initializing the internal Isabelle proof state with the test specification TS:

$$\begin{aligned}\texttt{prog}(x, y, z) = \textbf{if}\ & \mathit{triangle}\ x\ y\ z\ \textbf{then} \\ & \textbf{if}\ x = y\ \textbf{then} \\ & \quad \textbf{if}\ y = z\ \textbf{then}\ \text{equilateral}\ \textbf{else}\ \text{isosceles} \\ & \textbf{else if}\ y = z\ \textbf{then}\ \text{isosceles} \\ & \quad \textbf{else if}\ x = z\ \textbf{then}\ \text{isosceles}\ \textbf{else}\ \text{scalene} \\ & \textbf{else}\ \text{error}\end{aligned}$$

Note that the variable `prog` is used to label an arbitrary implementation as the current *program under test* that should fulfill the test specification.

In the following we show how our test package `TestGen` can be applied to the automatic test data generation problem for the triangle problem. Our method proceeds in the following steps:

1. By applying `gen_test_case_tac` we bring the proof state into *testing normal form* (TNF). In this example, we decided to generate symbolic test cases up to depth 0 (discussed later) and to unfold the *triangle* predicate by its definition before the process. This leads to a formula with 26 clauses, among them:

$$\begin{aligned} [\![0 < z; z < z + z]\!] &\Longrightarrow \texttt{prog}(z, z, z) = \text{equilateral} \\ \left[\!\!\left[\begin{array}{l} x \neq z; 0 < x; 0 < z; \\ z < x + z; x < z + z \end{array} \right]\!\!\right] &\Longrightarrow \texttt{prog}(x, z, z) = \text{isosceles} \\ [\![y \neq z; z \neq y; \neg z < z + y]\!] &\Longrightarrow \texttt{prog}(z, y, z) = \text{error} \end{aligned}$$

   We call each Horn-clause of the proof state a *symbolic test case*. As a result of `gen_test_case_tac`, we can extract the current proof state and get the *test theorem* which has the form $[\![A_1; \ldots; A_{26}]\!] \Longrightarrow TS$ where the A_i abbreviate the above test cases.
2. We compute the concrete *test statements* by instantiating variables by constant terms in the symbolic test cases for "`prog`" via a random test procedure (`genadd_test_data`). The latter operation selects the test cases from the test theorem and produces the test statements (excerpt):

$$\texttt{prog}(3, 3, 3) = \text{equilateral} \qquad \texttt{prog}(4, 6, 0) = \text{error}$$

A test statement can be compiled into a test program by simply mapping all operators to external code (where `prog` is the code for calling the program under test). This can be automated with Isabelle's code-generator. If such a compilation is possible for a formula A, i.e., if A only consists of constant symbols for which this map is defined, we call A *executable*. This definition essentially rules out unbounded logical quantifiers and more arcane HOL constructs like the Hilbert-operator.

In our triangle example, standard simplification was able to eliminate the assumptions of the (instantiated) test cases automatically. In general, assumptions in test statements (also called *constraints*) may remain. Provided that all test statements are executable, clauses with constraints can nevertheless be interpreted as an abstract test program. For its result, three cases may be distinguished: (i) if one of the clauses evaluates to false, the test is *invalid*, otherwise *valid*. A valid test may be (ii) a *successful test* if and only if the evaluation of all conclusions (including the call of `prog`) also evaluates to true; (iii) otherwise the test contains at least one *test failure*. Rephrased in this terminology, the ultimate goal of the test data selection is to construct successful tests, which means that

ground substitutions (i.e. instantiations of variables with constant terms) must be found that make the remaining *constraints* valid.

Coming back to our example, there is a viable alternative for the process above: instead of unfolding *triangle* and trying to generate ground substitutions satisfying the constraints, one may keep *triangle* in the test theorem, treating it as a building block for new constraints. It turns out that a special test theorem and test data (like "$triangle(3,4,5) = \text{True}$") can be generated "once and for all" and inserted before the *test data selection* phase producing a "partial" grounding. It will turn out that the main state explosion is shifted from the test case generation to the test data selection phase, possibly at the cost of test adequacy. This technique to modularize test data generation will be discussed in Sec. 4 in more detail.

3 Concepts of Test Case Generation

As input of the test case generation phase, the *test specification*, one might expect a special format like $\text{pre}(x) \rightarrow \text{post}\ x\ (\texttt{prog}(x))$. However, this would rule out trivial instances such as $3 < \texttt{prog}(x)$ or just $\texttt{prog}(x)$ (meaning that `prog` must evaluate to True for x). Therefore, 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 this test specification, our method `gen_test_case_tac` can be separated into the following conceptual phases (in reality, these phases were performed in an interleaved way):

- *Tableaux Normal Form Computation:* via a tableaux calculus (see Tab. 1), the specification is transformed into Horn-clause normal form (HCNF).
- *Rewriting Normal Form Computation:* via the standard rewrite rules the current specification is simplified.
- *TNF Computation:* by re-ordering of the clauses, the calls of the program under test are rearranged such that they only occur in the conclusion, where they must occur at least once.
- *TNF Minimization:* redundancies, e.g., clauses subsumed by others, are eliminated.
- *Exploiting Uniformity Hypothesis:* for free variables occurring in recurring argument positions of primitive recursive predicates, a suitable *data separation lemma* is generated and applied (leading to a test hypothesis *THYP*).
- *Exploiting Regularity Hypothesis:* for all Horn-clauses not representing a test hypothesis, a uniformity hypothesis is generated and exploited.

After a brief introduction of concepts and use of Isabelle in our setting, we will follow the sequence of these phases and describe them in more detail in the subsequent sections. We will conclude with a discussion of coverage criteria.

3.1 Concepts and Use of Isabelle/HOL

Isabelle [17] is a generic theorem prover of the LCF prover family; as such, we use the possibility to build programs performing symbolic computations over formulae in a logically safe (conservative) way on top of the logical core engine: this is what `TestGen` technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church's higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Isabelle's proof engine is geared towards Horn-clauses (called "subgoals"): $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, written $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". A *proof state* in Isabelle contains an implicitly conjoint sequence of Horn-clauses ϕ_1,...,ϕ_n and a *goal* ϕ. Since a Horn-clause

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

is logically equivalent to

$$\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1},$$

a Horn-clause normal form (HCNF) can be viewed as a conjunctive normal form (CNF). Note, that in order to cope with quantifiers naturally occurring in specifications, we generalize the idea of a Horn-clause to Isabelle's format of a *subgoal*, where variables may be bound by a built-in meta-quantifier:

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

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$; this mechanism is used to generate test theorems. The meta-quantifier $\bigwedge$ is used to capture the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as free variables. 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.

3.2 Normal Form Computations

In this section, 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 [10] presented as (derived) natural deduction rules. The core tableaux calculus is shown in Tab. 1 in the Appendix. Note, that with the notable exception of the elimination rule for the universal quantifier (see Tab. 1(c)), 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.

Horn-clauses can be normalized by a number of elementary logical rules (e.g., False $\Longrightarrow P$ = True), the usual injectivity and distinctness rules for constructors implied by data types and computation rules resulting from recursive definitions. Both processes together bring an original specification into *Rewriting HCNF.*

However, these forms do not exclude clauses of the form:

$$[\![\neg(\texttt{prog}\ x = c); \neg(\texttt{prog}\ x = d)]\!] \Longrightarrow A_{n+1}$$

where `prog` is the program under test. Equivalently, this clause can be transformed into

$$[\![\neg(A_{n+1})]\!] \Longrightarrow \texttt{prog}\ x = c \vee \texttt{prog}\ x = d$$

We call this form of Horn-clauses *testing normal form* (TNF). More formally, a Horn-clause is in TNF for program under test F if and only if

- F does not occur in the constraints, and
- F does occur in the conclusion.

Note that not all specifications can be converted to TNF. For example, if the specification does not make a suitably strong constraint over program F, in particular if F does not occur in the specification. In such cases, `gen_test_case_tac` stops with an exception.

3.3 Minimizing TNF

A TNF computation as described so far may result in a proof state with redundancies. Redundancies in a proof state may result in superfluous test data and should therefore be eliminated. A proof state may have:

1. several occurrences of identical clauses
2. several occurrences of clauses with subsuming assumption lists; this can be eliminated by the transformation

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![P; Q; R]\!] \Longrightarrow A;}{[\![P; R]\!] \Longrightarrow A;}$$

3. and in particular, clauses that subsume each other after distribution of $\vee$; this can be eliminated by the transformation

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B; \quad [\![R; Q]\!] \Longrightarrow A \vee B;}{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B;}$$

The notation above refers to logical transformations on a subset of clauses within a proof state and not, as usual, on formulae within a clause. Since in backward proofs the proof state below is a refinement of the proof state above, the logical implication goes from bottom to top.

3.4 Exploiting Uniformity Hypothesis for Recursive Predicates

In the following, we address the key problem of test case generation in our setting, i.e.; recursive predicates occurring in preconditions of a program. As an introductory example, we consider the membership predicate of an element in a list:

$$\begin{array}{lll} \textbf{primrec} & x\ mem\ [\,] & = \text{False} \\ & x\ mem\ (y\#ys) & = \textbf{if}\ y = x\ \textbf{then}\ \text{True}\ \textbf{else}\ x\ mem\ ys \end{array} \tag{1}$$

which occurs as precondition in an (abstract) program specification:

$$x\ mem\ S \rightarrow \texttt{prog}\ x\ S$$

For the testing of recursive data structure, Gaudel suggested in [13] the introduction of a *uniformity hypothesis* as one possible form of a test hypothesis, a kind of weak induction rule:

$$\frac{\begin{array}{c}[|x| < k] \\ \vdots \\ P\ x\end{array}}{P\ x}$$

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

For all variables in clauses that occur as (recurring) arguments of primitive recursive functions, we will use a testing hypothesis of this kind — called *data separation lemma* — in an exercise in poly-typic theorem proving [19] described in the following.

The Isabelle/HOL data type package generates definitions of poly-typic functions (like case-match and recursors) from data type definitions and derives a number of theorems over them (like induction, distinctness of constructors, etc.). In particular, for any data type, we can assume the size function and reduction rules allowing to compute $\big|[a, b, c]\big| = 3$, for example. Moreover, there is a standard *exhaustion-theorem*, which for lists has the form

$$[\![y = [\,] \Longrightarrow P; \bigwedge x\ xs.\ y = x\#xs \Longrightarrow P]\!] \Longrightarrow P$$

Now, since we can separate any data x belonging to a data type τ into:

$$x \in \big\{z :: \tau.\ |z| < d\big\} \vee x \in \big\{z :: \tau.\ d \leq |z|\big\} \tag{2}$$

i.e., x is either in the set of data smaller d or in the remaining set. Note that both sets are infinite in general; the bound for the size produces "data test cases" and not just finite sets of data. Consequently, we can derive for each given type τ

and each d a destruction rule that enumerates the data of size $0, 1, \ldots, k-1$. For lists x and $d = 2, 3$, it has the form:

$$x \in \{z :: \alpha \text{ list}.\ |z| < 2\} \rightarrow \big(x = [\,]\big) \vee \big(\exists a.\ x = [a]\big)$$
$$x \in \{z :: \alpha \text{ list}.\ |z| < 3\} \rightarrow \big(x = [\,]\big) \vee \big(\exists a.\ x = [a]\big) \vee \big(\exists ab.\ x = [a, b]\big) \qquad (3)$$

Putting equation (2) together with the destruction rule (3), instead of the unsafe uniformity hypothesis in the sense of Gaudel we automatically construct the safe data separation lemma, i.e. an exhaustion theorem of the form:

$$\frac{\begin{array}{ccccccc} [x = [\,]] & & [x = [a]] & & [x = [a, b]] & & \\ \vdots & & \vdots & & \vdots & & \\ P(x) & \bigwedge a. & P(x) & \bigwedge a\ b. & P(x) & THYP\big(3 \leq |x| \rightarrow P(x)\big) & \end{array}}{P(x)}$$

The purpose of this rule in backward proof is to split a statement over a program into several cases, each with an additional assumption that allows to "rewrite-away" the x appropriately. Here, the constant $THYP :: \text{bool} \rightarrow \text{bool}$ (defined as the identity function) is used to label the test hypothesis in the proof state. Since we do not unfold it, formulae labeled by $THYP$ are protected from decomposition by the tableaux rules shown in Tab. 1.

The equalities introduced by this rule of depth $d = 3$ allow for the simplification of the primitive recursive predicate *mem* which leads to further decompositions during the TNF computation. Thus, for our test specification:

$$x\ mem\ S \rightarrow \texttt{prog}\ x\ S$$

executing `gen_test_case_tac` results in the following TNF:

1. $\texttt{prog}\ x\ [x]$
2. $\bigwedge b.\ \texttt{prog}\ x\ [x, b]$
3. $\bigwedge a.\ a \neq x \rightarrow \texttt{prog}\ x\ [a, x]$
4. $THYP(3 \leq |S| \rightarrow x\ mem\ S \rightarrow \texttt{prog}\ x\ S)$

The simplification of the *mem* predicate along its defining rules (1) leads to nested "**if then else**" constructs. Their decomposition during HCNF computation results in the constraint that the lists fulfilling the precondition must have a particular structure. Even the simplest "generate-and-test"-method for test data selection will now produce adequate test statements, while it would have produced mostly test failures when applied directly to the original specification.

The handling of quantifiers ranging over data types can be done analogously: since $\forall x.\ P(x)$ is equivalent to $\forall x : UNIV.\ P(x)$ and since the universal set $UNIV = \{z :: \tau.\ |z| < d\} \cup \{z :: \tau.\ d \leq |z|\}$, the universal quantifier can be decomposed into a finite conjunction for the test cases smaller than d and a test hypothesis $THYP$ for the rest.

From the above example it follows that the general form of a test theorem is $[\![A_1; \ldots; A_n; THYP(H_1); \ldots; THYP(H_m)]\!] \Longrightarrow TS$. Here the A_i represent the test cases, the H_i the test hypothesis, and TS the testing specification.

3.5 Exploiting Regularity Hypothesis

After introducing the regularity hypothesis and computing a TNF (except for clauses containing *THYP*s), we use the clauses to construct another form of testing hypothesis, namely the *regularity hypothesis* [13] (sometimes also called *partitioning hypothesis*) for each test case. This kind of hypothesis has the form:

$$THYP(\exists x_1, \ldots, x_n.\ P\ x_1, \ldots, x_n \rightarrow \forall x_1, \ldots, x_n.\ P\ x_1, \ldots, x_n)$$

This means that whenever there is a successful test for a test case, it is assumed that the program will behave correctly for *all* data of this test case.

Using a uniformity hypothesis for each (non-*THYP*) clause allows for the replacement of free variables by meta-variables; e.g., for the case of two free variables, we have the following transformation on proof states:

$$\frac{[\![A_1\ x\ y; \ldots; A_n\ x\ y]\!] \Longrightarrow A_{n+1}\ x\ y}{[\![A_1\ ?x\ ?y; \ldots; A_n\ ?x\ ?y]\!] \Longrightarrow A_{n+1} ?x\ ?y;\ \ THYP((\exists xy.\ P\ x\ y) \rightarrow (\forall xy.\ P\ x\ y))};$$

where $P\ x\ y \equiv A_1\ x\ y \wedge \ldots \wedge A_n\ x\ y \rightarrow A_{n+1}\ x\ y$. This transformation is logically sound. Moreover, the construction introduces individual meta-variables into each clause for the ground instances to be substituted in the test data selection; this representation allows for partial instantiation of variable with constant terms and is also a prerequisite for structured test data selection as discussed in Sec.4.

3.6 Coverage Criteria: A Discussion

In their seminal work, Dick and Faivre [12] propose to transform the original specification into disjunctive normal form (DNF), followed by a case splitting phase converting the disjunctions $A \vee B$ into $A \wedge B$, $\neg A \wedge B$ and $A \wedge \neg B$ and further (logical and arithmetic) simplifications and minimizations on the disjunctions. The resulting cases are also called the *partitions of the specification* or the *(DNF) test cases*. The method suggests the following test adequacy criterion: a set of test data is *partition complete* if and only if for any test case there is a test data. Consequently, a program P is tested adequately to partition completeness with respect to a specification S if it passes a partition complete test data set.

Our notion of a *successful test*, see Sec. 2, is a HCNF based adequacy criterion. DNF and HCNF based adequacy result in the same partitioning in many practical cases, as in the triangle example, while having no clear-cut advantage in others. Since the DNF technique has the disadvantage of producing a double exponential blow-up (the case splitting phase alone can produce an exponential blow-up) while HCNF computation is simply exponential, and since HCNF-computation can be more directly and efficiently implemented in the Isabelle proof engine, we chose the latter.

HCNF adequacy subsumes another interesting adequacy criterion under certain conditions, namely *branch coverage* with respect to the specification. Branch coverage means that in any (mutual) recursive system of functions, all reachable branches, e.g., of the **if** P **then** A **else** B statements, were activated at least

once. For a mutual recursive system consisting only of *primitive* recursive functions, (i.e., with each call the size of data will decrease exactly by one), it can be concluded that if the testing depth d is chosen larger than the size of the maximal strong component of the call graph of the recursive system, each function is unfolded at least once. Since the unfold results in conditionals that were translated to $(P \to A) \wedge (\neg P \to B)$, any branch will lead to a test case.

Thus, while `gen_test_case_tac` often produces reasonable results for arbitrarily recursive functions, we can assure only for primitive recursions that the underlying HCNF adequacy of our method subsumes branch coverage.

4 Structured Test Data Selection

The motivations to separate test data selection from test case generation are both conceptual and technical. Conceptually, test data selection is a process where we would also like to admit more heuristic techniques like random data generation or generate-and-test with the constraints. Test data selection yields sequences of ground theorems (no meta-variables, no type variables); this paves the way for evaluation by compiled code, a new approach is needed to cope with the unavoidable state explosion in the late stages. A purely technical motivation for this separation is Isabelle-related: within a test theorem, it is not possible to instantiate polymorphic type variables α in different ways when generating test statements, however, this flexibility may be desirable.

The generation of a multitude of ground test statements from one test theorem containing the test cases and the test hypothesis is essentially based on a random-procedure followed by a test of the satisfaction of the constraints (similar to QuickCheck). For each type, this default procedure may be overwritten in `TestGen`-specific generators that may be user defined; thus, the usual heuristics like trying $[0, 1, 2, \mathit{maxint}, \mathit{maxint}+1]$ can be easily implemented, or the counterexample generation integrated in Isabelle's arithmetic procedure can be plugged in (which, in our experience, is difficult to control in larger examples).

Now we will discuss the issue of structured test data generation. Similar to theorem proving, the question of "how many definitions should be unfolded" is crucial; exploiting suitable abstractions is the major weapon against complexity. In our first attempt to generate a test theorem for the triangle example (see Sec. 2), the auxiliary predicate *triangle* is unfolded in the test specification. This resulted in the aforementioned 26 cases. If we do not unfold it, the resulting test theorem has only 10 test cases, but contains "abstract constraints" such as:

$$\begin{aligned}
[\![\mathit{triangle}\ z\ z\ z]\!] &\Longrightarrow \mathtt{prog}(z,z,z) = \text{equilateral}\\
[\![\neg \mathit{triangle}\ z\ z\ z]\!] &\Longrightarrow \mathtt{prog}(z,z,z) = \text{error}\\
[\![y \neq z; z \neq y; \mathit{triangle}\ z\ y\ z]\!] &\Longrightarrow \mathtt{prog}(z,y,z) = \text{isosceles}
\end{aligned}$$

Thus, a substantial part of the proof state explosion can be postponed by treating *triangle* as a building block in the constraints or, in other words, by generating more *abstract* test cases.

Now, if we could generate a *local test theorem* for *triangle* as such, generate the *local* test data separately and resolve the resulting test statements for it into the test theorem for the global computation, the state explosion could be shifted to the test data selection. The trick can be done as follows: we define a trivially true proof goal for:

$$\texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z \Longrightarrow \texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z$$

unfold *triangle* and compute TNF(`prog`). When folding back *triangle* via the assumption we get the following local test cases:

$$\begin{array}{ll}
\neg \mathit{triangle}\ 0\ y\ z & \neg z < x + y \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\neg \mathit{triangle}\ x\ 0\ z & \neg x < y + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\neg \mathit{triangle}\ x\ y\ 0 & \neg y < x + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\left[\!\!\left[\begin{array}{c} 0 < x; 0 < y; 0 < z; \\ z < x + y; x < y + z; y < x + z \end{array} \right]\!\!\right] \Longrightarrow & \mathit{triangle}\ x\ y\ z
\end{array}$$

which can easily be converted into *abstract test statements* such as *triangle* 1 1 1. When resolving the latter in all combinations into the abstract global test theorem, instances for variables with randomly generated constants were made superfluous. Thus, the test statements of previously developed theories can be reused when building up larger units. Of course, when building up test data in a modular way, this comes at a price: since the local test statements do not have the same logical information available as their application context in a more global test theorem, the instantiation may result in unsatisfiable constraints. Nevertheless, since the criterion for success of a decomposition is clear — at the very end we want constraint-free test statements achieving a full coverage of the TNF— the implementor of a test has more flexibility here helping to deal with larger problems. In our example, there is no loss at all: test data for the local predicate is valid for the global goal, and by construction, the set of test statements is still complete for HCNF coverage.

5 Applications

We applied our method to specifications of two widely used variants of balanced binary search trees: AVL trees and red-black trees. These case studies were performed using Isabelle 2003 compiled with SML of New Jersey running on Linux with 512 MBytes of RAM, and an Intel 1.6 GHz P4 processor.

5.1 AVL Trees

In 1962 Adel'son-Vel'skiĭ and Landis [3] introduced a class of balanced binary search trees (called AVL trees) that guarantee that a tree with n internal nodes has height $O(\log n)$. Based on an AVL-theory from the Isabelle library we generated test cases for the following invariant: if an element y is in the tree after insertion of x in the tree t then either $x = y$ holds or y was already stored in t. Based on the depth 3, this *test specification* leads to an amazing 236 test cases which were computed in less than 30 seconds.

5.2 Red-Black Trees

A widely used variant of balanced search trees was presented by Bayer [4]. In this data structure, the balancing information is stored in one additional bit per node. This is called "color of a node" (which can either be red or black), hence the name *red-black trees*. A valid (balanced) red-black tree must fulfill the following two invariants:

- *Red Invariant:* each red node has a black parent.
- *Black Invariant:* each path from the root to an empty node has the same number of black nodes.

We aimed for testing a "real-world" implementation of red-black trees and decided to test the red-black trees provided in the standard library of SML of New Jersey (SML/NJ) [2]. There, red-black trees are used for implementing finite sets and maps which are intensively used throughout the SML/NJ compiler itself.

Our specification is based on the formalization [16] of the SML/NJ red-black trees (based on version 110.44 of SML/NJ). The specification starts with the basic data type declaration for binary trees:

datatype color $= R \mid B$
α tree $= E \mid T$ color (α tree) (α item) (α tree)

In this example we have chosen not only to check if keys are stored or deleted correctly in the trees but also to check if the trees fulfill the balancing invariants. Therefore our specification has to formalize the red and black invariants. This is done by the following recursive predicates:

consts
redinv :: (α item) tree $\Rightarrow$ bool
blackinv :: (α item) tree $\Rightarrow$ bool

recdef *redinv* "measure ($\lambda t.$ (size t))"
"*redinv E* = True"
"*redinv* (T B a y b) = (*redinv* $a \wedge$ *redinv* b)"
"*redinv* (T R (T R a x b) y c) = False"
"*redinv* (T R a x (T R b y c)) = False"
"*redinv* (T R a x b) = (*redinv* $a \wedge$ *redinv* b)"

recdef *blackinv* "measure ($\lambda t.$ (size t))"
"*blackinv E* = True"
"*blackinv*(T *color* a y b) = ((*blackinv* a) $\wedge$ (*blackinv* b)
$\wedge$((*max_B_height* a) = (*max_B_height* b)))"

We use the following test specification for checking if the delete operation fulfills these invariants:

$$(\textit{redinv}\ t \wedge \textit{blackinv}\ t) \rightarrow (\textit{redinv}\ (\textit{delete}\ x\ t) \wedge \textit{blackinv}\ (\textit{delete}\ x\ t))$$

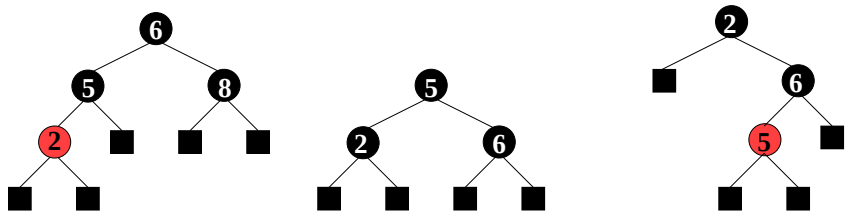
In other words, for all trees the deletion operation maintains the red and black invariant. For testing purposes, we instantiated *item* with Integers. The test case generation takes less than two minutes and results in 348 test cases. Among them

$$\begin{aligned} &\mathit{delete}\ 8\ (T\ B\ (T\ B\ (T\ R\ E\ 2\ E)\ 5\ E)\ 6\ (T\ B\ E\ 8\ E)) \\ &\qquad = (T\ B\ (T\ B\ E\ 2\ E)\ 5\ (T\ B\ E\ 6\ E)) \end{aligned}$$

which describes that the deletion of the node 8 in the tree shown in Fig. 1(a) must result in the tree shown in Fig. 1(b). This test case revealed a major error in the standard library of SML/NJ. Using a simple SML test script one observes:

```
val input = T (B,T (B,T (R,E,2,E),5,E),6,T (B,E,8,E))
- val output = delete(input,8);
val output = T (B,E,2,T (B,T (R,E,5,E),6,E))
```

Obviously, the black invariant does not hold for `output` (see Fig. 1(c)).



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

Figure 1. Test Data for Deleting a Node in a Red-Black Tree

This example shows that specification based testing can find efficiency bugs: combinations of insert and delete operations of the SML/NJ implementation easily lead to trees that degenerate to sorted lists. In our case, the revealed flaw has not been detected in the last 12 years, although red-black trees are widely used within the SML/NJ compiler itself. Fixing this bug will presumably lead to a perceptible performance gain of the SML/NJ compiler.

Based on our definitions, the bug could be reproduced by QCheck/SML [1], a QuickCheck-like random testing tool. Although this particular bug can even be found without using a hand-programmed test data generator, the QuickCheck method imposes to write one in general. Moreover, our method allows to conclude that certain coverage criteria are fulfilled and makes all underlying test hypotheses explicit. Further, our approach can profit from the underlying theories for data-types offering the potential for problem-specific case splits.[2]

[2] ... such as $[\![P(\texttt{minBound} :: \texttt{Int}); a \neq \texttt{minBound} \Longrightarrow P(-a)]\!] \Longrightarrow P(-a)$ which also produces the mentioned problem for `abs(x-2)>=0` (namely $x = \texttt{minBound} + 2$) after unfolding `abs` to `if x >= 0 then x else -x`.

6 Conclusion

We have presented the theory and implementation of a test case generator for unit tests. In contrast to [20] (which also provides a recent survey), which attempts to analyze imperative programs with non-trivial data-structures, our approach is focused on functional programs. Since imperative programs can be provided with a functional interface (by compiling a functional call to a statement sequence consisting of (i) initialization, (ii) executing constructors representing data types, (iii) calling the program under test, and (iv) checking the result), this is not a real limitation of our approach except if complex reference structures have to be analyzed. We demonstrated the practical feasibility of our approach by testing functions from the SML/NJ library, which revealed a major bug leading to inefficiency in basic data structures of the SML/NJ compiler.

In our opinion, test data generation is an activity that clearly needs *some* user interaction: as in model-checking, one has to experiment with the form of the specifications and basic parameters (depth of *data separation*, the level of abstraction, the decision which definitions should be unfold, etc.) in order to get a feasible test data set for the test of a "real program". Therefore, we believe such an activity is best supported by an integration into an *interactive* theorem proving environment such as Isabelle. Since `TestGen` is ca. 400 lines of SML code that is loaded into Isabelle, we still consider our approach fairly "lightweight". Nevertheless, `TestGen` is at present the only implementation of a test case generator that combines state-of-the-art deduction technology based on derived rules (formally proven inside Isabelle) with a powerful logic.

We believe that there is another line of criticism against Dijkstra's verdict. A successful test together with explicitly stated test hypotheses is not fundamentally different from program verification: all sorts of modeling assumptions were made, adding test hypothesis is just one more of them. The nature and trustworthiness of these assumptions may be different, but a clear-cut line between testing and verification does not exist.

6.1 Future Work

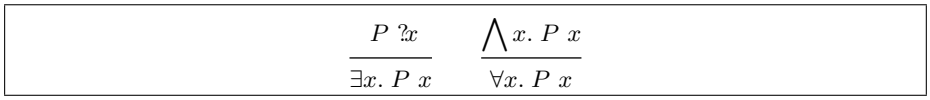
We see the following lines of extension of our work:

1. *Investigating the test hypothesis*: a new test hypothesis (like congruence hypothesis on data, for example) may dramatically improve the viability of the approach. Furthermore, it should be explored if the verification of the test hypothesis for a given abstract program offers new lines of automation.
2. *Better control of the process*: at the moment, our implementation can only be controlled by very globally applied parameters such as depth. The approach could be improved by generating the test hypothesis and the test data depending on the local context within the test theorems.
3. *Integration tests*: integrating/combining our framework into behavioral modeling leads to the generation of *test sequences* as in [15, 18].
4. *Generating test data for many-valued logics* such as *HOL-OCL* [6] should make our approach applicable to formal methods more accepted in industry.

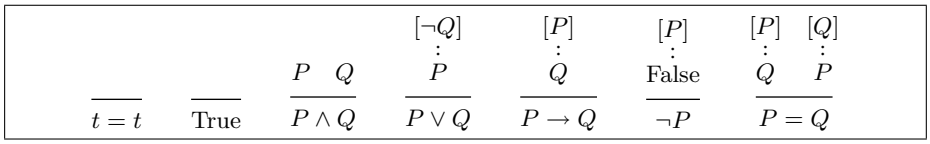
References

[1] QCheck/SML. `http://contrapunctus.net/league/haques/qcheck/`.
[2] SML of New Jersey. `http://www.smlnj.org/`.
[3] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
[4] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.
[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*. Number 58 in Advances In Computers. 2003.
[6] A. D. Brucker and B. Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In C. Muñoz, S. Tahar, and V. Carreño, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.
[7] A. D. Brucker and B. Wolff. A case study of a formalized security architecture. In T. Arts and W. Fokkink, editors, *FMICS'03*, volume 80 of *Electronic Notes in Theoretical Computer Science*, Roros, 2003. Elsevier Science Publishers.
[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.
[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.
[10] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of Tableau Methods*. Kluwer, Dordrecht, 1996.
[11] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 1972.
[12] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *FME 93*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.
[13] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *LNCS*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.
[14] S. Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.
[15] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - a tool for distributed systems specification. In *FTRTFT 96*, volume 1135 of *LNCS*, pages 467–470. Springer-Verlag, 1996.
[16] A. Kimmig. Red-black trees of smlnj. Studienarbeit, Universität Freiburg, 2003.
[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
[18] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In E. Brinksma and J. Tretmans, editors, *Proc. Formal approaches to testing of software*, pages 47–60. BRICS, 2001.
[19] K. Slind and J. Hurd. Applications of polytypism in theorem proving. In D. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 103–119. Springer-Verlag, Rome, Italy, 2003.
[20] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.
[21] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

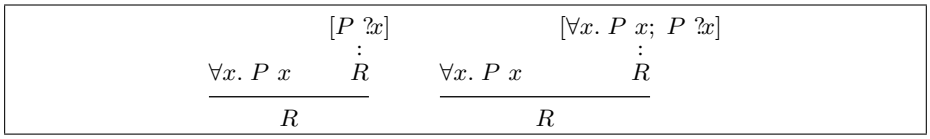
A Appendix



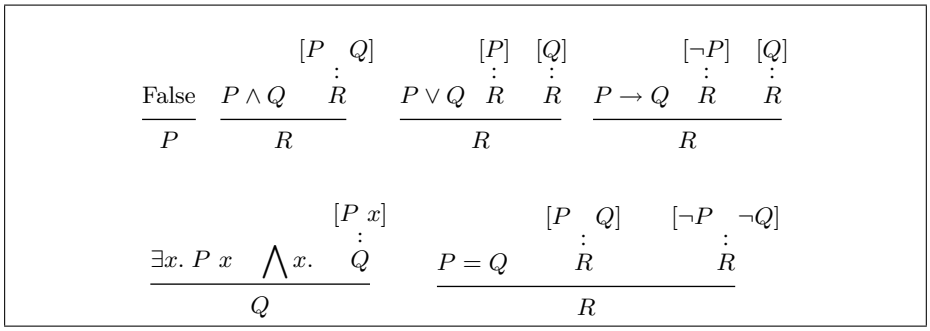
(a) Quantifier Introduction Rules



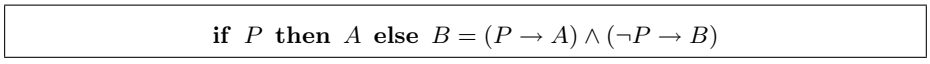
(b) Safe Introduction Rules



(c) Unsafe Elimination Rules



(d) Safe Elimination Rules



(e) Rewrites

Table 1. The Standard Tableaux Calculus for HOL