Test-Sequence Generation with HOL-TestGen With an Application to Firewall Testing

Achim D. Brucker and Burkhart Wolff

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

Abstract HOL-TestGen is a specification and test case generation environment extending the interactive theorem prover Isabelle/HOL. Its 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. Although originally designed for black-box unit-tests, HOL-TestGen's underlying logic and deduction engine is powerful enough to be used in test-sequence generation, too.
We develop the theory for test-sequence generation with HOL-TestGen and describe its use in a substantial case-study in the field of computer security, namely the black-box test of configured firewalls.
Key words: symbolic test case generations, test sequence generation, black box testing, theorem proving, Isabelle/HOL, computer security

1 Introduction

Today, essentially two software validation techniques are used: *software verification* and *software testing*. As far as symbolic verification methods and model-based testing techniques 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 [12], which can be viewed as *new abstraction techniques* reducing infinite models to finite and checkable ones,
- new approaches to generate *counter-examples*, 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 test case generation techniques, and
- ways to log the implicit *testing hypothesis* underlying a test and to make them explicit.

B. Meyer and Y. Gurevich (Eds.): TAP 2007, LNCS 4454, pp. 149–168, 2007.

The HOL-TestGen system [5, 4, 3] 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. The core of HOL-TestGen is a test case generation procedure that decomposes a *test specification* (TS), i. e., test-property over a program under test, into a semantically equivalent *test theorem* of the form:

$$[\![\mathrm{TC}_1; \ldots; \mathrm{TC}_n; \mathrm{THYP}\ \ H_1; \ldots; \mathrm{THYP}\ \ H_m]\!] \Longrightarrow \mathrm{TS}$$

where the TC_i are the *test cases* and THYP is a constant (semantically defined as identity) used to mark the explicit *test hypotheses* H_j that are 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.

In this sense, the test theorem bridges the gap between test and verification. Testing can be viewed as systematic weakening of specifications.

HOL-TestGen has been applied to unit-tests; for example, [5] discusses tests of insert and delete operations for library implementations of red-black trees. In this paper, however, we show that the procedure can also be used for sequence testing of locally non-deterministic reactive systems as well: instead of using an automaton, we build a test-specification based on its input traces. We apply this technique to a substantial case study in the field of computer security, namely the black-box test of a *configured network firewall*. As firewalls are part of today's IT security infrastructure, testing their correct behavior is a rewarding task and, and as we will see, a challenging application for specification based testing.

This paper consists of two parts: In part one, we introduce HOL-TestGen, its explicit test hypothesis generation and its potential for sequence test generation conceptually. In part two, we outline the firewall problem domain, present formal test plans based on these concepts for a concrete configuration, and evaluate them by some empirical data.

2 Foundations

2.1 Isabelle

Isabelle [10] 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 HOL, which we choose as framework for HOL-TestGen.

While Isabelle/HOL is usually coined as "proof assistant," we use it as symbolic computation environment. Implementations on Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution and rewriting, and the overall environment provides a large collection of components ranging from documentation generators and code-generators to (generic) decision procedures for datatypes and Presburger Arithmetic.

Isabelle can easily be controlled by a programming interface on its implementation level in SML in a logically safe way, as well as in the Isar level, i. e., a tactic proof language in which interactive and automated proofs can be mixed arbitrarily. Documents in the Isar format, enriched by the commands provided by HOL-TestGen, can be processed incrementally within Proof General (see Section 3) as well as in batch mode. These documents can be seen as formal and technically checked test plan of a program under test.

Isabelle processes rules and theorems of the form $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, also denoted as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$. They can be understood as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}." Further, Isabelle provides a built-in meta-quantifier: $\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$ for representing "fresh free variables not occurring elsewhere" thus avoiding the usual provisos on logical rules. In particular, the presentation of sub-goals uses this format. We will refer to assumptions A_i also as *constraints* in this paper.

2.2 Higher-order Logic

Higher-order logic (HOL) [6, 2] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e. g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

Isabelle/HOL provides also a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories. Furthermore, it provides the means for defining data types and recursive function definitions over them in a style similar to a functional programming language.

3 The HOL-TestGen System: An Overview

HOL-TestGen is an *interactive* (semi-automated) test tool for specification based tests. Its theory and implementation has been described elsewhere [5, 3]; here, we briefly review main concepts and outline the standard workflow. The latter is divided into four phases: writing the *test specification* TS, generation of *test cases* TC along with a *test theorem* for TS, generation of *test data* TD, i. e., constraint-free instances of TC, and the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test. (See Figure 1 for the overall workflow.) Once a test theory is completed, documents can be generated that represent a formal test plan. The test plan containing test theory, test specifications, configurations of the test data and test script generation commands,

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

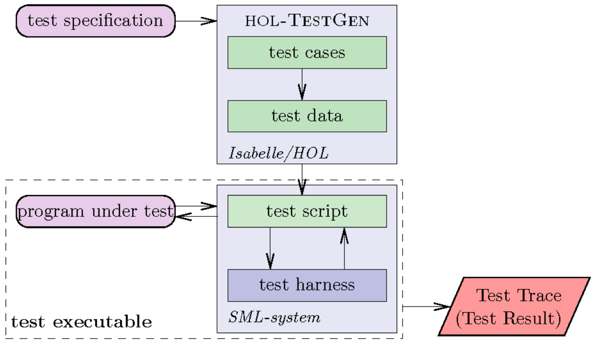

Figure 1: Overview of the Standard Workflow of HOL-TestGen

possibly extended by proofs for rules that support the overall process, is written in an extension of the Isar language [11]. It can be processed in batch mode, but also using the Proof General interface interactively, see Figure 2. This interface allows for interactively stepping through a test theory in the upper sub-window while the sub-window below 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, HOL-TestGen produces a *test*

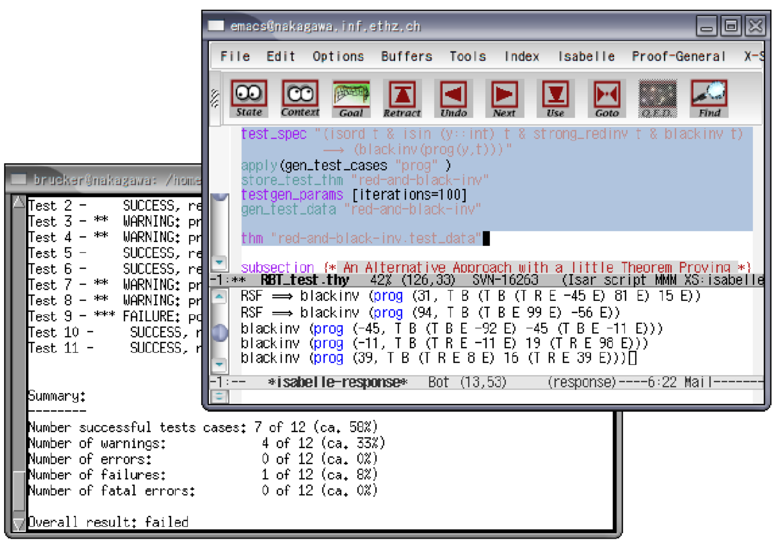

Figure 2: A HOL-TestGen Session Using Proof General

script driving the test using the provided *test harness*. The test script together with the test harness stimulate the code for the program under test built into the *test executable*. Executing the *test executable* runs the test and yields a *test trace* showing errors in the implementation (see lower window in Figure 2).

4 Test Case Generation with Explicit Test-Hypothesis

In this section, we describe the test case generation procedure of HOL-TestGen. It is driven by an exhaustive backward-application of the tableaux calculus presented in Section A combined with certain normal-form computations eliminating certain forms of redundancy. Interleaved with this partitioning process (similar to the DNF of Dick and Faivre [7]), test hypothesis rules are generated on the fly and applied to certain subgoals in a backward manner. In the following, we only present two well-known kinds of test hypothesis. Following the terminology of Gaudel [9], these are called *uniformity* and *regularity* hypothesis.

4.1 Inserting Uniformity Hypothesis

Uniformity hypothesis have the form:

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

where THYP is a constant defined as the identity; this constant is used as marker to protect this type of formulae from other decomposition steps in the generation procedure. Semantically, this kind of hypothesis expresses 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.

The derived rule in natural deduction format expressing this kind of test theorem transformation reads as follows:

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

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

Tactically, these hypothesis were introduced *at the end* of the test case generation process, i. e., when all other rules can no longer be applied. Using a uniformity hypothesis for each (non-THYP) clause allows for the replacement of free variables by meta-variables which can be instantiated by ground terms during the test data selection phase later. This transformation is logically sound. For example, for a test specification if $x \leq 0$ then *ioprg* x else *ioprg* $-x$, the test case generation produces for the program *ioprg* under test the test theorem:

$$\mathit{test}: \text{ if } 0 \leq x \text{ then } \mathit{ioprg}\ x \text{ else } \mathit{ioprg}\ -x$$

1. $0 \leq ?x \implies \mathit{ioprg}\ ?x$
2. $\mathrm{THYP}((\exists x.0 \leq x \to \mathit{ioprg}\ x) \to (\forall x.0 \leq x \to \mathit{ioprg}\ x))$
3. $?y < 0 \implies \mathit{ioprg}\ -?y$
4. $\mathrm{THYP}((\exists x.\ x < 0 \to \mathit{ioprg}\ -x) \to (\forall x.\ x < 0 \to \mathit{ioprg}\ -x))$

The test-data selection phase will easily generate the instances of the test cases *ioprg* 3 and *ioprg* $(-(-4))$ (satisfying the constraints) to be used in a black-box test. If we have the implementation of *ioprg* in our hands, we could also verify the test-hypothesis; provided that execution paths in the concrete program correspond to test classes, we gain knowledge from the test for the verification.

4.2 Inserting Regularity Hypothesis

In the following, we address the problem of test case generation for quantifiers (or, equivalently: free variables) ranging over recursive datatypes such as lists or trees. As an introductory example, we consider the membership predicate of an element in a list defined by the following recursive rules:

$$\begin{aligned} x \ \mathrm{mem}\ [\,] &= \mathrm{false} \\ x \ \mathrm{mem}\ (y\#ys) &= \mathrm{if}\ y = x\ \ \mathrm{then}\ \ \mathrm{true}\ \ \mathrm{else}\ \ x\ \ \mathrm{mem}\ \ ys \end{aligned} \tag{1}$$

which occurs as "precondition" in the example test specification:

$$\boxed{x\ \ \mathrm{mem}\ \ S \rightarrow \mathit{ioprg}\ \ x\ S}$$

For testing recursive data structures, Gaudels [9] suggested the introduction of a *regularity hypothesis* as one possible form of a test hypothesis:

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

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

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

The equalities introduced by this rule lead together with the simplification rules shown in Equation 1 of the predicate mem to the following result of the test case generation (we omit the uniformity hypothesis insertion here):

$\mathit{test}: \ x\ \ \mathrm{mem}\ \ S \rightarrow \mathit{ioprg}\ \ x\ S$

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

and, again, it is an easy game for a random-based test-data-selection method to provide constraint free instances of the test cases.

4.3 Principles of Test-Sequence-Generation in HOL-TestGen

Considering the previous subsection more closely, one easily recognizes that it also holds the key for the principles of test sequence generation in HOL-TestGen: since a finite automaton can be converted into (mutual) recursive acceptance predicate accept on input lists, this scheme of a test specification can also be used for specifying the test of a transition function $iopra :: \alpha \Rightarrow \sigma \Rightarrow \sigma$ option under test, which takes some input of type α and some state of type σ and can produce a successor state (the α option type contains the constructors Some a and None). Together with the recursively defined Mfold-combinator:

$$\text{Mfold}\ []\ \ \sigma\ \ \mathit{ioprg} = \text{Some}\,\sigma$$

$$\text{Mfold}\ (in\#H)\ \ \sigma\ \ \mathit{ioprg} = \begin{cases} \text{Mfold}\ \ H\ \sigma'\ \ \mathit{ioprg} & \text{if } \mathit{ioprg}(in, \sigma) = \text{Some}\,\sigma', \\ \text{None} & \text{otherwise.} \end{cases}$$

it is now possible to lift an individual (partial) function *ioprg* to be run in a complete sequence by using the following scheme of a test specification:

$$\boxed{\text{accept}\ \ S \rightarrow P(\text{Mfold}\ \ S\ \sigma_0\ \ \mathit{ioprg})}$$

where σ_0 is the initial state. After HOL-TestGen synthesized a trace S and suitable input for variables occurring in P, a test driver running the test sequence can be generated. Note that the function *ioprg* can in particular log the complete run of a system and make the test verdict depending on this log, i. e., the complete history of inputs and outputs in the real system trace.

4.4 An Infra-structure 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 *ioprg*, i. e., in situations where the test-driver and the external program under test represent a communicating system.

In the following, we describe a special instance of the overall scheme discussed in Section 4.3. As fundamental modeling assumption of this instance, we require that the test-driver can be built upon an “i/o stepping function” $\mathit{ioprg} :: \iota \Rightarrow \sigma' \Rightarrow (o \times \sigma')$ option. This function takes an input of type ι, an internal state of type σ' only managed by itself, and returns the observable output of type o plus the result state of one step of the system under test. We allow *ioprg* to fail, depending on the concrete realization inside the test harness. This could represent timeouts or other forms of misbehavior of the system under test. Further, we assume a function: $\mathit{post} :: \sigma \times \sigma' \Rightarrow \iota \Rightarrow o \Rightarrow$ bool that, depending on the observer state, the *ioprg* state, the (concrete) input and the (concrete) output decides that the behavior of *ioprg* conforms to the specification in this step. We assume *ioprg* to be a function in the mathematical sense, so identical runs with the same inputs will produce the same outputs; *which* outputs were chosen is unimportant as long as *post* remains satisfied. The choice of the output and the successor state is non-deterministic in this sense, and even the stimulation sequence automaton may be non-deterministic. We call these assumptions on non-determinism occurring in the system under test *local non-determinism*, in contrast to *deep non-determism* occurring in testing theories such as [8] and at least partially in their test system implementation.

The key element for the instantiation of the scheme of Section 4.3 lies in the generic definition of an adapter function that builds a stepping function from this i/o stepping function. As a suitable abstraction over a history log, we integrate into this adapter an environment of type σ that keeps track of values exchanged at runtime of a test which were bound to symbolic variables occurring in *abstract traces*. The latter were gained from standard protocols by replacing values which were only known at runtime; thus, we will be able to tackle with a quite common class of reactive systems.

As a prerequisite, we need the two functions $\mathit{rebind} :: \sigma \Rightarrow o \Rightarrow \sigma$ and $\mathit{subst} :: \sigma \Rightarrow \iota \Rightarrow \iota$. The former extracts from a concrete output a new binding for corresponding variables occurring in abstract output; the latter replaces variables occurring in abstract input to the corresponding values exchanged in the previous system run. Wiring everything together, we get the following definition:

$$
\begin{aligned}
&\mathrm{observer}\ \mathit{rebind}\ \mathit{subst}\ \mathit{post}\ \mathit{ioprg}\ \mathit{in}\ (\sigma,\sigma') \equiv \mathrm{let}\ \mathit{in}' = \mathit{subst}\,\sigma\,\mathit{in}\ \mathrm{in}\\
&\quad \mathrm{case}\ \ \mathit{ioprg}\ \ \mathit{ioprg}'\ \ \sigma'\ \mathrm{of}\ \ \mathrm{None} \Rightarrow \mathrm{None}\\
&\qquad\qquad\qquad\qquad\qquad \mid \mathrm{Some}(\mathit{out},\sigma''') \Rightarrow \mathrm{let}\,\sigma'' = \mathit{rebind}\,\sigma\,\mathit{out}\ \mathrm{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{if}\ \mathit{post}(\sigma'',\sigma''')\ \mathit{in}'\ \mathit{out}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{then}\ \mathrm{Some}(\sigma'',\sigma''')\ \mathrm{else}\ \mathrm{None}
\end{aligned}
$$

The adapter function observer essentially runs *ioprg* on its state and on the *in* resulting from *subst*; the resulting *out* leads to an update of the observer state. Occurring errors were propagated. The function observer is fully executable and is compiled to a part of the test driver.

4.5 An Example

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. Figure 3 shows the abstract protocol (containing variables and constraints over them) and its sub-protocol containing just the input stimulation sequence.

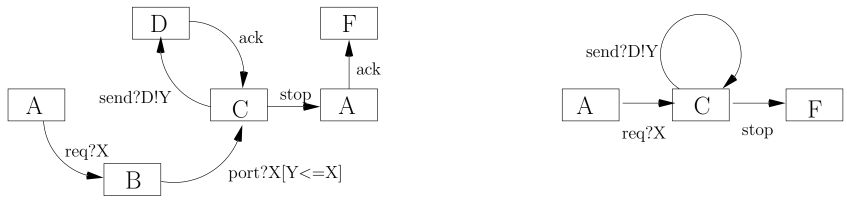

Figure 3: An abstract protocol automaton and the resulting stimulation sequence automaton

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

$$\mathrm{vars} = X \mid Y$$

and specify the combined type of abstract and concrete input and output events:

$$\begin{aligned}
\mathrm{InEvent} &= \mathrm{req}\,\mathrm{chan} \mid \mathrm{reqA}\,\mathrm{vars} \mid \mathrm{send}\,\mathrm{data}\,\mathrm{chan} \mid \mathrm{sendA}\,\mathrm{data}\,\mathrm{vars} \mid \mathrm{stop}\\
\mathrm{OutEvent} &= \mathrm{port}\,\mathrm{chan} \mid \mathrm{portA}\,\mathrm{vars} \mid \mathrm{ack}
\end{aligned}$$

The definition of subst is now straight-forward:

$$\begin{aligned}
&\mathrm{subst}\; \mathit{env}\; (\mathrm{req}\;\; n) &&= \mathrm{req}\, n\\
&\mathrm{subst}\; \mathit{env}\; (\mathrm{reqA}\;\; v) &&= \mathrm{req}(\mathrm{lookup}\; \mathit{env}\;\; v)\\
&\mathrm{subst}\; \mathit{env}\; (\mathrm{send}\;\; d\, n) &&= \mathrm{send}\, d\, n\\
&\mathrm{subst}\; \mathit{env}\; (\mathrm{sendA}\;\; d\, v) &&= \mathrm{send}\, d(\mathrm{lookup}\; \mathit{env}\;\; v)\\
&\mathrm{subst}\; \mathit{env}\; \mathrm{stop} &&= \mathrm{stop}
\end{aligned}$$

as well as defining rebind

$$\begin{aligned}
&\mathrm{rebind}\; \mathit{env}(\mathrm{port}\, n) &&= \mathit{env}(Y \mapsto n)\\
&\mathrm{rebind}\; \mathit{env}\, \mathrm{ack} &&= \mathit{env}
\end{aligned}$$

and the definition of the post-condition:

$$\begin{aligned}
&\mathrm{post}'\; (\mathit{env}, x, \mathrm{req}\;\; n, \mathrm{port}\, m) &&= (n \leq m)\\
&\mathrm{post}'\; (\mathit{env}, x, \mathrm{send}\;\; z\;\, n, \mathrm{ack}) &&= \mathrm{true}\\
&\mathrm{post}'\; (\mathit{env}, x, \mathrm{stop}, \mathrm{ack}) &&= \mathrm{true}\\
&\mathrm{post}'\; (\mathit{env}, x, y, z) &&= \mathrm{false}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{post} && :: (\mathrm{vars} \rightharpoonup \mathrm{int}) \times \mathrm{unit} \Rightarrow \mathrm{InEvent} \Rightarrow \mathrm{OutEvent} \Rightarrow \mathrm{bool}\\
&\mathrm{post}\;\; x\, y\, z && \equiv \mathrm{post}'(\mathrm{fst}\, x, \mathrm{snd}\, x, y, z)
\end{aligned}$$

Here, $\alpha \rightharpoonup \beta$ denotes partial functions and is just a synonym for $\alpha \Rightarrow \beta\,\mathrm{option}$.

The predicate post checks the constraint that the server must return a port within a previously communicated range. The abstract inputs like sendA Data X will be converted to concrete input send Data 23 if 23 has been communicated previously by the server under test; the explicit variable management is done once-and-for-all in the observer adapter.

The automaton for the set of stimulation traces results from a direct translation of the diagram above:

$$\begin{aligned}
\text{stimTrace}'\ (A, (\text{reqA}\ \ X)\#S) &= \text{stimTrace}'(C, S)\\
\text{stimTrace}'\ (C, (\text{sendA}\ \ d\,Y)\#S) &= \text{stimTrace}'(C, S)\\
\text{stimTrace}'\ (C, [\text{stop}]) &= \text{true}\\
\text{stimTrace}'\ (x, y) &= \text{false}
\end{aligned}$$

$$\begin{aligned}
&\text{stimTrace} \quad :: \text{InEvent} \Rightarrow\ \text{listbool}\\
&\text{stimTrace}\, s \equiv \text{stimTrace}'(A, s)
\end{aligned}$$

Finally, we state the test specification for a reactive sequence test. Note that its pattern is an instance of the sequence test (see Section 4.3) which is again an instance of the pattern post $x \rightarrow$ post x ($iopry\ \ x$) in Section 4.2:

$$\boxed{\begin{array}{c}\text{stimTrace}\, \mathit{trace} \longrightarrow\\ \text{success}(\text{Mfold}\, \mathit{trace}((X \mapsto \mathit{init}), ())(\text{observer}\, \text{rebind}\, \text{subst}\, \text{post}\, \mathit{ioprog}))\end{array}}$$

where success :: α option $\Rightarrow$ bool is an auxiliary function that yields true for values of the form Some E. Applying our test case generation and test data generation procedures takes only a few seconds, including the generation of the test script containing the abstract input sequences plus the test program run over them; this test program also contains the compiled versions of observer, subst, rebind, etc.

For the test depth $k = 4$ of the test case-generation procedure we already reach path coverage in the stimulation protocol automaton and therefore implicitly on the protocol automaton shown in Figure 3.

5 Case-Study: Testing Firewall Configurations

In many institutions, an unrestricted connection of the internal network to he Internet is classified as a security risk. *Firewalls* as means to restrict network traffic are therefore widely used in todays IT infrastructures. As security infrastructure crucially depend on them, testing their correct behavior is an important and rewarding task. As we will see, it is also an interesting application for specification based testing. The complete specification is part of the HOL-TestGen distribution [1].

If we have the implementation of *ioprg* in our hands, we could also verify that it represents an automaton; the minimal path length covering all vertexes in this automaton gives a bound for k.

5.1 A Bluffers Guide to Firewalls

In a computer network, e. g., based on TCP/IP, a message from A to B is encapsulated in one or more *packets* which contains the content of the message and routing information. The routing information of a packet mainly contains its source address (where does the packet come from), its destination address (where should the packet go to) and the protocol (e. g., http, smtp) used on top of transport layer (e. g., TCP/IP).

In its simplest form, a firewall is just a *stateless packet filter* which just filters (i. e., rejects or accepts it) traffic from one network to another based on the destination address, source address and the protocol, the *policy* used. The policy is the specification (or configuration) of the firewall which describes which packets should be denied and which should be rejected. In some cases, stateless filtering is not enough, some application protocols, like ftp or most of the protocols used for Internet telephony such as Voice over IP (VoIP) have an internal state of which the firewall must be aware of. For example, some connections are only allowed within a specific state of the protocol.

Figure 4 illustrates a simple and common setup of a firewall, separating three

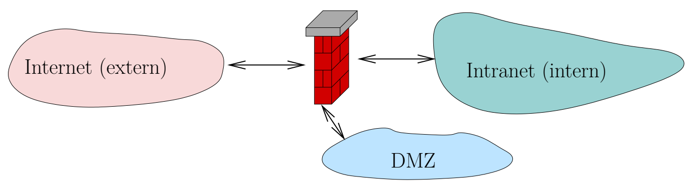

Figure 4: A simple firewalling scenario.

networks: the external (potentially dangerous) Internet, the internal network that has to be protected (intranet) and a network that is somewhat in-between, the demilitarized zone (DMZ). The DMZ is usually used for servers (e. g., the Web server and the Mail server) that should be accessible both from the outside (Internet) and the internal network (Intranet) and thus underlie, a more relaxed policy than the intranet. An example for a simple firewall policy is shown in Table 1 in an informal way. Such a policy uses a first-fit pattern matching strategy, i. e., the first match overrides later ones. For example, a packet from the Internet to the intranet is rejected (it only matches the last line of the table) whereas a

source	destination	protocol	action
DMZ	Intranet	any	deny
Intranet	Webserver	http	accept
Internet	Webserver	https	accept
Intranet	Mailserver	smtp	accept
Intranet	Mailserver	imap	accept
Intranet	Mailserver	imaps	accept
any	any	any	deny

Table 1: A simple Firewall Policy

http-packet from the Intranet to the Web server is accepted (second line of the table. The lines of such a table are also called *rules*; together, they build the *policy* of a firewall.

In the remainder of this section, we will briefly introduce a formal HOL model of networks and policies; it will turn out that these concepts can be used uniformly both for stateless packet filters and statefull application level firewalls. This model forms the basis for several test case generation scenarios that validate a firewall implementation against its specified policy.

5.2 A Formal Firewall Model

Packets and Networks. As a prerequisite, we need a formal models of protocols, packets and nets. We model protocols as an abstract data types, e. g., the most common ones are declared by:

$$\mathrm{protocol} := \mathrm{ftp} \mid \mathrm{http} \mid \mathrm{https} \mid \mathrm{voip} \mid \mathrm{smtp} \mid \mathrm{imap} \mid \mathrm{imaps} \mid \mathrm{unknown}\ .$$

As we do not want to depend on a specific representation of addresses and package content, we introduce the abstract types $\alpha\,\mathrm{src}$ and $\alpha\,\mathrm{dest}$ for the source and destination address and $\beta\,\mathrm{content}$ for the content. Moreover, we introduce an unique identifier id for each packet. Thus, the type of a package defined straight-forward as:

$$(\alpha, \beta)\,\mathrm{packet} := \mathrm{id} \times \mathrm{protocol} \times \alpha\,\mathrm{src} \times \alpha\,\mathrm{dest} \times \beta\,\mathrm{content}$$

Further, we define projectors, e. g., getId, getSrc, for accessing the different components of packet directly.

As a next step, we model networks, or just *nets*, and parts thereof (*subnets*). To be as abstract as possible at this stage, we model nets as an axiomatic type class [10]. For the purpose of this paper, it suffices to know that a net is a set of sets of addresses, i. e.,

$$\alpha\ \mathrm{subnet} = (\alpha :: \mathrm{net})\,\mathrm{set}\,\mathrm{set}$$

where $(\alpha :: \mathrm{net})$ requires that the types we use to instantiate α are members of the type class net. This definition allows us to model firewall policies that restrict the traffic between sub-networks and also between single hosts (addresses). For checking, if a given address is part of a subnet, we define the following operator:

$$a \sqsubset S \equiv \exists s \in S.\, (a \in s) \qquad \text{with type } \alpha\,\mathrm{adr} \Rightarrow \alpha\,\mathrm{subnet} \Rightarrow \mathrm{bool}.$$

The Firewall Policy. From an abstract point of view, a policy is a partial mapping of packets to decisions, e. g., deny or accept. The datatype:

$$\alpha\,\mathrm{out} := r\,\mathrm{accept}\,\alpha \mid \mathrm{deny}$$

for decisions allows for modeling the modifications of return packages; Thus, our model can capture address-translation techniques (network address translation

(NAT)) realized by some firewalls as well.[2] The type of a policy follows directly from this:

$$(\alpha, \beta)\,\text{policy} := (\alpha, \beta)\,\text{packet} \rightharpoonup ((\alpha, \beta)\,\text{packet})\,\text{out}$$

where $\alpha \rightharpoonup \beta$ denotes the partial mapping (i. e., type synonyms to $\alpha \Rightarrow \beta$ option; cf. Section 4.2). In our model, rules and policies have the same type, i. e., we can introduce a type synonym:

$$(\alpha, \beta)\,\text{policy} := (\alpha, \beta)\,\text{rule}$$

for rules. Moreover, the override operator for partial mappings ($\_ ++ \_$) allows for nicely combining several rules to a policy. For example, $r_2 ++ r_1$ combines the rules r_1 and r_2 where r_1 overrides (has higher precedence) r_2. We can define several *generic rules combinators* at this abstract level (without concrete format of addresses) that substantially simplify the formalization of a concrete policies. For example, the usual two “catch-all” rules for accepting or denying all traffic were expressed as:

$$\begin{aligned} \text{allowAll}\,p &\equiv \text{Some}(\text{accept}\,p) && \text{with type } (\alpha, \beta)\,\text{rule, and} \\ \text{denyAll}\,p &\equiv \text{Some}(\text{deny}) && \text{with type } (\alpha, \beta)\,\text{rule.} \end{aligned}$$

Many other combinators for restricting traffic based on its source, destination or protocol can already be defined on this abstraction level. A rule restriction all packets coming from subnet s can be defined as

$$\text{allowAllFrom}\,s \equiv \text{Some}\,\text{allowAll} \restriction_{\{p \mid (\text{getSrc}\,p) \sqsubset s\}}$$

with type $(\alpha :: \text{net})\,\text{subnet} \Rightarrow (\alpha, \beta)\,\text{rule}$, and where $\_ \restriction \_$ is the restriction operator on partial mappings.

IPv4. At this point, we make the packet address format more concrete. We specify the underlying transport protocol, e. g., IPv4 or IPv6. For our example, we use tcp combined with ip version 4. In this setting, an address consists out of an unique 32 bit number, represented as four-tuple and a port:

$$\begin{aligned} \text{ipv4Ip} &:= \text{int} \times \text{int} \times \text{int} \times \text{int} \\ \text{port} &:= \text{int} \\ \text{ipv4} &:= \text{ipv4Ip} \times \text{port} \end{aligned}$$

Based on these definitions, we can define further combinators (rules) that are specific to tcp/ip addresses, i. e., they can accept or reject packages based on an ip address and a port.

[2] However, in reality, a firewall policy can describe more fine-grained how packets are denied, e. g., some packages could be silently discarded (this is often called *drop*) or the packet could be rejected, causing an error message is send to the origin.

5.3 Testing Stateless Firewalls

Our abstract firewall model, presented in the last section, allows for the direct formalization of the informal policy given in Table 1. First we have to define the subnets of type ipv4 subnet, based on their ip address ranges, e. g.:

$$\text{intranet} \equiv \left\{\left\{\big((a,b,c,d),p\big)\,\middle|\, (a=192)\wedge(b=168)\right\}\right\}$$
$$\text{webserver} \equiv \left\{\left\{\big((a,b,c,d),p\big)\,\middle|\, (a=172)\wedge(b=16)\wedge(c=70)\wedge(d=4)\right\}\right\}$$

Grouping the rules of our informal policy with the same source and same destination, define:

$$\begin{aligned}
\text{DmzIntranet} &\equiv \text{denyAllFromTo dmz intranet}\\
\text{toWebserver} &\equiv \text{allowProtFromTo http intranet webserver}\\
\text{toMailserver} &\equiv \text{allowProtFromTo smtp intranet mailserver}\\
&\quad ++\,\text{allowProtTo imap mailserver}\\
&\quad ++\,\text{allowProtTo imaps mailserver}
\end{aligned}$$

The *test specification* for the stateless firewall case is now within reach: we just state that the *firewall under test (fut)* has the same filtering function behavior as our given combined policy:

$$\boxed{\mathit{fut}(x) = (\text{denyAll} ++ \text{DmzIntranet} ++ \text{toWebserver} ++ \text{toMailserver})(x)}$$

Applying our test case generation and test data generation procedures results, after 72 hours running time on a modest equipped workstation, in 828 test cases, among them:

$$\begin{gathered}
\mathit{fut}(9, \text{smtp}, ((6,2,8,5),0), ((7,3,8,1),1), \text{content}) = \text{Some deny}\\
\mathit{fut}(8, \text{http}, ((6,6,10,3),6), ((4,7,5,9),1), \text{content}) = \text{Some deny}\\
\mathit{fut}(2, \text{imaps}, ((6,2,10,7),9), ((172,16,70,5),3), \text{content})\\
= \text{Some}(\text{accept}(2, \text{imaps}, ((6,2,10,7),9), ((172,16,70,5),3), \text{content}))\\
\mathit{fut}(6, \text{imaps}, ((9,7,9,10),9), ((172,16,70,5),0), \text{content})\\
= \text{Some}(\text{accept}(6, \text{imaps}, ((9,7,9,10),9), ((172,16,70,5),0), \text{content}))
\end{gathered}$$

Overall, testing stateless packet filters is quite similar to classical unit testing of stateless software. The test-data selection is trivial in this example.

5.4 Testing Statefull Firewalls

The well-known file-transfer protocol file transfer protocol (ftp) is based on a dynamic negotiation of a port number which is then used as channel to communicate the file content between the sender and the receiver. Thus, a stateless firewall can only provide a very limited form of network protection if ftp is involved, whereas a statefull firewall that observes the inner state of the ftp session can open the negotiated port dynamically. Testing statefull firewalls, where the filter functions change over time, requires test-sequence generation.

A Statefull Firewall Model. First we model the internal state of a statefull firewall as a tuple of a store and the current policy (that can change during a transition):

$$(\alpha, \beta, \gamma)\,\mathrm{FwState} = \alpha \times (\beta, \gamma)\,\mathrm{Policy}$$

One possibility is, to model the store as the list of accepted packages:

$$(\beta, \gamma)\,\mathrm{history} = (\beta, \gamma)\,\mathrm{packet\,list}$$

A transition from state to state is a mapping from the packet that fired the transition, the current state to the new state:

$$(\alpha, \beta, \gamma)\,\mathrm{FwStateTrans} = (\beta, \gamma)\,\mathrm{packet} \times (\alpha, \beta, \gamma)\,\mathrm{FwState} \rightharpoonup (\alpha, \beta, \gamma)\,\mathrm{FwState}$$

Moreover, for combining state transitions, we define two combinators: orelse takes the first defined transitions

$$f\,\mathrm{orelse}\,g \equiv \begin{cases} \mathrm{Some}\,y & \text{if } f\ x = \mathrm{Some}\,y, \\ \mathrm{None} & \text{otherwise,} \end{cases}$$

and repeat repeats as long as the transitions is defined:

$$f\,\mathrm{repeat}\,g \equiv \begin{cases} \mathrm{Some}\,z & \text{if } f\ x = \mathrm{Some}\,y \text{ and } f\ y = \mathrm{Some}\,z \\ \mathrm{None} & \text{otherwise.} \end{cases}$$

Modeling the file transfer protocol (ftp). During an ftp session, the server (normally located in the Internet) opens a data connection to the client (e. g., located in the intranet) using a port that is negotiated: Figure 5 shows an abstract trace of an ftp session: the client initializes the session by sending a init message to the server, the client answers with a port request containing a dynamic port

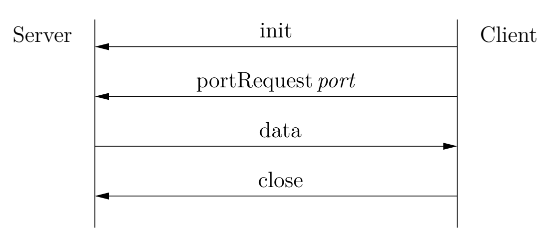

Figure 5: A sample trace of an file transfer protocol (ftp) run

for the data connection and then the server sends the data to the client using this dynamic port. Eventually, the client will close the connection and the firewall has to close the data port. We model the communication as follows:

$$\mathrm{ftpMsg} = \mathrm{init} \mid \mathrm{portRequest}\, \mathit{port} \mid \mathrm{data} \mid \mathrm{close} \mid \mathrm{other}$$

Further, we will use the id part of a package to distinguish several ftp-sessions. We model state transitions of the ftp protocol as recursive predicates. First we define a generic state transition for messages that do not change the policy and special transition for the portRequest (that opens the data port) and close (that closes the data port). As an example, we present the simple generic transition (see [1] for the remaining details) that is defined recursively based on the definitions:

$$\begin{aligned} \mathrm{ftpSt_f}((a, \mathrm{ftp}, c, d, e), (\mathit{in}, \mathit{policy})) &\equiv \mathrm{if}\, \mathrm{accept}(a, \mathrm{ftp}, c, d, e)\mathit{policy} \\ &\qquad \mathrm{then}\, \mathrm{Some}((a, \mathrm{ftp}, c, d, e)\#\mathit{in}, \mathit{policy}) \\ &\qquad \mathrm{else}\, \mathrm{Some}(\mathit{in}, \mathit{policy}) \\ \mathrm{ftpSt_f}(x, (\mathit{in}, \mathit{policy})) &\equiv \mathrm{None} \end{aligned}$$

The state machine modeling ftp can be defined using the orelse combinator for combining the singe transitions:

$$\mathrm{ftpSt} \equiv \mathrm{ftpSt_{portRequest}}\, \mathrm{orelse}\, \mathrm{ftpSt_{close}}\, \mathrm{orelse}\, \mathrm{ftpSt_f}$$

Using the repeat combinator, we can easily model arbitrary runs of the protocol.

Testing ftp. We have to clarify the test purpose first: for example, one could aim for testing one or more correct protocol runs (with or without interleavings), or for illegal protocol runs. Here, we show a test for single, legal protocol runs. We define a recursive acceptance predicate isFtp testing for legal ftp traces. We assume a simple test scenario with a initial policy only allowing ftp sessions (initiated using port 21, the control port of the ftp protocol) from the intranet to the Internet:

$$\mathrm{ftpPolicy} \equiv \mathrm{allowAll} +\!+\, \mathrm{allowProtFromToPort}\, \mathrm{ftp}\, \mathrm{intranet}\, \mathrm{internet}\, 21$$

The accept-predicate for traces in the sense of Section 4.3 is defined on the basis of the ftp protocol machine together with some additional constraints:

$$\begin{aligned} \mathrm{accept}(t) = t \in \{x \mid \mathrm{isFtp}\ c\, s\, i\, x\} \\ \wedge\, \mathrm{isInIntranet}\, c \wedge \mathrm{isInInternet}\, s \wedge \mathrm{getPort}\, s = 21 \end{aligned}$$

using predicates (isInIntranet and isInInternet) for checking if an address is within a specific subnet. The key stone of our test section is the test specification:

$$\boxed{\mathrm{accept}(t) \rightarrow \mathit{fut}\, t = \mathrm{Mfold}\, (\mathrm{rev}\, t)\, ([], \mathrm{ftpPolicy})\ \mathrm{ftpSt}}$$

which is an instance of the test specification scheme discussed in Section 4.3. Using our test method, we receive four test cases which each represent different ftp traces. The test case generation took about 5 minutes. For space reasons, we omit the quite involved code of the generated test script here; the interested reader is referred to [1]).

6 Conclusion

It comes perhaps as a surprise that conceptually—viewed from a strict datatype centric angle and using a powerful logic—sequence testing is just a special case of unit testing. Instead of one input to be send to the system under test to receive one output, a *list* of input is generated to receive a *list* of outputs; the rest is the usual monadic trickery to represent i/o in a functional setting and the use of abstract test traces instead of concrete ones.

One might question the practical relevance of this observation since the length of the considered sequences is fairly small in the firewall study ($k = 4$ in our ftp example, and $k = 8$ in experiments with VoIP protocols, where the slow-down was already considerable). However, the example in Section 4.5 can easily be blown up to protocol-lengths of 100; test case generation including test script generation still takes less than a minute (see HOL-TestGen example suite). It is therefore the combination between richness of data-structures, the branching-factor in the automaton, *and* the length of the protocol, which may represent a fundamental barrier to our approach, not the length alone. So far, we do not see that this is different from any other tool-supported test case generation approach.

The combination of theorem proving and test data generation is a fruitful one, in particular to control the state-space explosion which is in our case an explosion of test cases for testing the filter-function of firewalls. Using theorem proving techniques for simplifying firewall policies can reduce dramatically both, the overall time for generating test cases as well as the number of generated test data. For example, within HOL-TestGen, we can formally prove the following equality which formalizes the fact, that a global allow-all rule will override a direct predecessor with the more specific allow rule:

$$(\mathrm{allowAll} \mathbin{++} \mathrm{allowAllFromTo}\ x\ y) = \mathrm{allowAll}$$

Thus, proving equalities and using them for the "logical massage" of policies in test-specifications will eliminate redundant test cases by computing a semantically equivalent, but "simpler" policy with respect to time and space consumption.

Our integrated approach to unit and sequence testing also paves the way for combined scenarios: it is straight-forward to formulate test specifications that "guide" a statefull firewall in a specific state and to compute test cases that test the specific filter-function in this state.

Finally, there is the possibility to verify test-hypothesis generated throughout the test theorem generation phase. In our view, a specification-based test is clearly an approximation to verification. A test has the advantage to be potentially based on more abstract data than the concrete program. Once generated, test data can be used for fast checks that a (complex, black-box) program conforms to the test specification. Such fast checks can be of crucial importance in a software development process, e. g., when checking in a new version of a program into the version management system of a development project. In later stages, a full review and even a verification of the test hypothesis might be in order; depending on the degree of abstraction of the test specification with respect to the concrete program, the test cases can help to structure and simplify this code-verification task.

Acknowledgment

We thank Lukas Brügger for valuable discussions on the subject of firewall testing and the work he did during his semester thesis.

References

[1] The HOL-TestGen Website. `http://www.brucker.ch/projects/hol-testgen/`.
[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, Orlando, May 1986.
[3] A. D. Brucker and B. Wolff. HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zürich, Apr. 2005.
[4] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Formal Approaches to Testing of Software (FATES 05)*, LNCS 3997, pages 87–102. Springer-Verlag, Edinburgh, 2005.
[5] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395, pages 16–32. Linz, 2005.
[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670, pages 268–284, Apr. 1993.
[8] L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. In *FATES/RV 2006*, Sept. 2006.
[9] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915, pages 82–96. Aarhus, Denmark, 1995.
[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002.
[11] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD thesis, TU München, München, Feb. 2002.
[12] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

A Tableaux Calculus for HOL

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

(a) Quantifier Introduction Rules

$$\frac{}{t = t} \qquad \frac{}{\text{true}} \qquad \frac{P \quad Q}{P \wedge Q} \qquad \frac{\begin{array}{c}[\neg Q]\\ \vdots\\ P\end{array}}{P \vee Q} \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ Q\end{array}}{P \to Q} \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ \text{false}\end{array}}{\neg P} \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ Q\end{array} \quad \begin{array}{c}[Q]\\ \vdots\\ P\end{array}}{P = Q}$$

(b) Safe Introduction Rules

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

(c) Unsafe Elimination Rules

$$\frac{\text{false}}{P} \qquad \frac{P \wedge Q \qquad \begin{array}{c}[P, Q]\\ \vdots\\ R\end{array}}{R} \qquad \frac{P \vee Q \quad \begin{array}{c}[P]\\ \vdots\\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots\\ R\end{array}}{R} \qquad \frac{P \to Q \quad \begin{array}{c}[\neg P]\\ \vdots\\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots\\ R\end{array}}{R}$$

$$\frac{\exists x.\ P\ x \qquad \bigwedge x.\ \begin{array}{c}[P\ x]\\ \vdots\\ Q\end{array}}{Q} \qquad \frac{P = Q \qquad \begin{array}{c}[P, Q]\\ \vdots\\ R\end{array} \qquad \begin{array}{c}[\neg P, \neg Q]\\ \vdots\\ R\end{array}}{R}$$

(d) Safe Elimination Rules

$$\text{if}\ \ P\ \ \text{then}\ \ A\ \ \text{else}\ \ B = (P \to A) \wedge (\neg P \to B)$$

(e) Rewrites

Table 2: The Standard Tableaux Calculus for HOL

B A Sample Derivation

In the following, we show, how the test case generation procedure inside HOL-TestGen synthesizes input data by a fully automatic symbolic constraint solution process. We pick the example of Section 4.2:

$$x \text{ mem } S \to \textit{ioprg } x\ S$$

Since S is the only free variable of list type, the procedure picks it, derives a datatype exhaustion theorem (as shown in Section 4.2) on the fly and applies it. The following proof-state is the result:

1. $S = [] \Longrightarrow x \text{ mem } S \to \textit{ioprg } x\ S$
2. $\bigwedge a.\ S = [a] \Longrightarrow x \text{ mem } S \to \textit{ioprg } x\ S$
3. $\bigwedge a\ b.\ S = [a, b] \Longrightarrow x \text{ mem } S \to \textit{ioprg } x\ S$
4. $\text{THYP}(3 \leq |S| \to x \text{ mem } S \to \textit{ioprg } x\ S)$

Variable propagation, simplification with the rules of (1) in Section 4.2 and the implication introduction rule from Table 2b yield the following state:

1. $\text{false} \Longrightarrow \textit{ioprg } x\ []$
2. $\bigwedge a.\ \text{if } a = x \text{ then true else false} \Longrightarrow \textit{ioprg } x\ [a]$
3. $\bigwedge a\ b.\ \text{if } a = x \text{ then true else if } b = x \text{ then true else false}$ $\Longrightarrow \textit{ioprg } x\ [a, b]$
4. $\text{THYP}(3 \leq |S| \to x \text{ mem } S \to \textit{ioprg } x\ S)$

Thus, the constraints for the first test case are not satisfiable anymore and it can be erased. In the sequel, we apply the simplification of the conditional of Table 2e and the safe elimination rule for conjunction Table 2c.

1. $\bigwedge a.\ \textit{ioprg } x\ [x]$
2. $\bigwedge a\ b.\ [\![a = x \to \text{true}; a \neq x \to (x = b \to \text{true} \wedge x \neq b \to \text{false})]\!]$ $\Longrightarrow \textit{ioprg } x\ [a, b]$
3. $\text{THYP}(3 \leq |S| \to x \text{ mem } S \to \textit{ioprg } x\ S)$

Now, the safe elimination rule for implication in Table 2c effectively produces a series of case splits; variable propagation and elimination of contradictory clauses simplify the proof state again. Thus, cascades of conditionals were eliminated.

Finally, the elimination of superfluous quantifiers result in the proof state shown in Section 4.2.