Testing Telecoms Software with Quviq QuickCheck

Thomas Arts
IT University of Göteborg, Gothenburg, Sweden
and Quviq AB
thomas.arts@ituniv.se

John Hughes
Chalmers University, Gothenburg, Sweden
and Quviq AB
rjmh@cs.chalmers.se

Joakim Johansson
Ulf Wiger
Ericsson AB, Älvsjö, Sweden
joakim.l.johansson@ericsson.com
ulf.wiger@ericsson.com

Abstract

We present a case study in which a novel testing tool, Quviq QuickCheck, is used to test an industrial implementation of the Megaco protocol. We considered positive and negative testing and we used our developed specification to test an old version in order to estimate how useful QuickCheck could potentially be when used early in development.

The results of the case study indicate that, by using Quviq QuickCheck, we would have been able to detect faults early in the development. We detected faults that had not been detected by other testing techniques. We found unclarities in the specifications and potential faults when the software is used in a different setting. The results are considered promising enough to Ericsson that they are investing in an even larger case study, this time from the beginning of the development of a new product.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods

General Terms Verification

Keywords Test Automation, Property Based Testing

1. Introduction

This paper describes a case study intended to evaluate a novel software testing tool, Quviq QuickCheck, in the development of telecommunications software. Quviq QuickCheck is a tool that tests running code against formal specifications, using controllable random test case generation combined with automated test case simplification to assist error diagnosis. Our case study considered the Media proxy under development at Ericsson, the world's leading provider of telecommunication equipment, among which mobile telecommunications systems and internet multimedia subsystems. We wanted to know whether QuickCheck was applicable at all to real telecoms software, which typically implements large and complex protocols, and must live up to very high quality standards. Would QuickCheck enable us to find bugs faster than conventional testing? Would it find subtile bugs, or just "obvious" ones? Would it potentially reduce testing time? Would it find obscure bugs, and help to improve final product quality? Our study is small and qualitative, but it suggests that the answer to all of these questions is a resounding "Yes".

The rest of the paper is structured as follows. In section 2 we give an introduction to Quviq QuickCheck, and in section 3 we explain our case study, giving some background on the software testing methods already used at Ericsson. In section 4 we explain our approach in detail, with samples of the testing code and a description of the extensions we made in parallel to QuickCheck, to better support this kind of testing. In 5 we present the results we obtained, and in 8 we draw conclusions.

Erlang'06 September 16, 2006, Portland, Oregon, USA.

2. Quviq QuickCheck

Quviq QuickCheck is a property-based testing tool, developed from Claessen and Hughes' earlier QuickCheck tool for Haskell [3] and a re-design for Erlang [2]. Apart from adaption to an Erlang setting, Quviq QuickCheck includes a number of extensions, of which the most significant is an ability to simplify failing test cases automatically. Quviq QuickCheck is a product of Quviq AB.

A user of QuickCheck writes properties that are expected to hold, as Erlang source code making use of the QuickCheck API. For example, one property of the standard list reversal function is

```
prop_reverse () ->
  ?FORALL({Xs,Ys},
          {list(int()),list(int())},
          lists:reverse(Xs++Ys)
          ==
          lists:reverse(Ys) ++ lists:reverse(Xs)).
```

This can be read as the logical statement

$$\forall(\texttt{Xs},\texttt{Ys}) \in \texttt{list(int())} \times \texttt{list(int())}.$$
```
  lists:reverse(Xs++Ys) =
    lists:reverse(Ys) ++ lists:reverse(Xs)
```

Here `?FORALL` is an Erlang macro, which takes a *pattern* as its first parameter: `?FORALL(X,S,P)` *binds* pattern `X` to a value in the set `S`, within the property `P`. By writing properties in this style, the QuickCheck user can build up a (usually partial) *formal specification* of the code under test, which is checked against the implementation by QuickCheck.

Checking the property runs 100 random test cases drawn from the given sets, and reports success if all tests pass. If any test fails, the failing test case is printed. For example, if the user mistakenly formulated the property above as

```
lists:reverse(Xs++Ys)
==
lists:reverse(Xs) ++ lists:reverse(Ys)
```

(in which `Xs` and `Ys` are swapped in the last line), then QuickCheck might report

```
Failed! After 8 tests.
{[-2],[2]}
```

where the value printed is the failing test case that was bound to the pattern `{Xs,Ys}` by `?FORALL`.

Quviq QuickCheck enables the user to run many tests with little effort, and provides a *direct payoff* for formulating formal specifications, thus encouraging developers to do so. In comparison to automated regression testing, which runs the *same* tests repeatedly, random testing of the sort that QuickCheck performs has the potential to find many more defects. A consultant experience shows that 85% of software defects are found the *first* time a test is run [5]—regression testing will mainly check that the same mistake is not made twice, while QuickCheck also searches for new innovative ways of which a bug may have been introduced.

In fact, QuickCheck prints more information than we saw above—once found, the counterexample is simplified as far as possible. In this case, this leads to the output

```
Failed! After 8 tests.
[-2],[2]
Shrinking....(4 times)
[0],[1]
```

The effect of "shrinking" in this simple example is just to reduce the absolute values of the numbers in the test case—as a result, we can at least see that the failure of the test does not depend on the particular values `2` and `-2`. In general, simplifying test cases can lead to a dramatic reduction in their size and complexity—see Hildebrant and Zeller for some very convincing examples [9]. Simplifying failing test cases is usually the first stage of diagnosing a fault, and is often very time consuming. Automating this process can lead to substantial savings.

Knowing that the final failing test case is *minimal* provides still more information. For example, we can see that neither `Xs` nor `Ys` can be the empty list in a failing test—otherwise, shrinking would have discarded one of the list elements altogether. Likewise, the elements of the two lists cannot be equal—otherwise, shrinking would have reduced the element of `Ys` to zero also. QuickCheck can also display the failed shrinking attempts—that is, successful tests which are similar to, but smaller than, the minimal failing test, thus helping the user to diagnose the fault more quickly.

In real applications, the test data are rarely just lists of integers. In the property above, `list(int())` is a *test data generator*, which captures both a set of values (lists of integers) and a probability distribution over them at the same time. When writing properties of real programs, test data of complex types is needed, and suitable generators have to be defined by the user. QuickCheck provides a rich set of macros and library functions for doing so. We will see some examples of their use in section 4 below.

3. The Media Proxy

The component we chose for our case study was the H.248 protocol interface to Ericsson's Media Proxy. The Media Proxy is the data-plane part of a Session Border Gateway, which serves as a media firewall in an IP-telephony network. At this time, the Media Proxy had been subjected to Function Test, and was undergoing System Test and approaching product release. As such, it was stable and well documented. The H.248 interface is central to the family of products to which the Media Proxy belongs, and is, due to its inherent complexity, very difficult to test using traditional methods. Specifically for the Media Proxy, a reasonably small subset of H.248 was supported, so a case study could be carried out with limited resources.

Pre-release testing usually takes 3-4 months at Ericsson, during which all the effort of the development team is devoted to finding and fixing errors. The quality standards that telecommunication products must meet are very high, and the Älvsjö team follow a disciplined approach to testing, among other things using the Erlang/OTP test server to run thousands of automated tests regularly. The team has a strong track record for quality, meeting and exceeding the five nines reliability criteria [8].

The media proxy was thus fairly well tested when we began, but it allowed us to evaluate QuickCheck's potential for finding subtile bugs late in the development process. This is interesting, even if larger benefits would be expected by formulating properties, and using QuickCheck, at a much earlier stage. That the media proxy is itself partially implemented in Erlang is not really important—we used a black-box approach, testing the proxy by sending it protocol messages and inspecting the replies, so it could equally well have been implemented in any other programming language.

The media proxy has been developed by an iterative process with an internal release every two weeks. These releases are stored and can be reconstructed, thus allowing us to move back in time. In an experiment, we used our QuickCheck properties to test an old release in order to obtain some insight in how many faults we potentially would have been able to detect if we were to use QuickCheck much earlier in the development.

The media proxy is part of a standardised approach to supporting multimedia calls, defined by the International Telecommunication Union (ITU). In this approach, incoming and outgoing media streams (sound, video, etc.) are connected to each other by a *media gateway*. The connections are established or broken by a *media gateway controller*, a separate physical unit which communicates with the media gateway using a protocol known as H.248, or Megaco (*Me*dia *Ga*teway *Co*ntroller). The media proxy is a simple case which is just intended to carry multimedia calls across a firewall. It behaves as a simplified media gateway, and thus accepts and carried out commands from a media gateway controller.

The current version of the Megaco protocol is specified by an ITU recommendation, a document of 212 pages [7]. It defines a number of commands which a media gateway controller can send to a media gateway, how the commands should be interpreted by the gateway, and the responses that may be returned. The protocol is largely stateless—the controller can send any command to the gateway at any time. In a few cases, the gateway may send a message to the controller which is *not* just a response to a previous command, but we ignored these in our case study. The code contains approximately 150,000 lines of Erlang code and ten times as much C code. We tested merely the control software, which is implemented in Erlang.

The most important elements of the gateway state are *contexts* and *terminations*—or, in plain English, calls and subscribers. Each termination (subscriber) is associated with a number of media streams, which may be in a variety of states (inactive, sending, receiving, etc). Calls are initiated by creating a context and placing a termination in it. *Add* commands are used to insert terminations into contexts; all the terminations in the same context are participants in the same call, and (by default) see and hear each other—streams with the same stream identifier in each termination in a context are (by default) connected to each other. *Modify* commands are used to change the properties of streams in terminations—for example, activating them once all the participants in a call have been added. *Subtract* commands are used to remove terminations from contexts; when the last subscriber is removed from a call, the call ends. Nine other commands are defined in the standard, but these are the most important, and are the only ones we included in our testing.

The Megaco standard is very general, and permits any number of terminations in the same context, with any number of streams in

each one, connected to each other in any topology. The media proxy is much simpler than this: it is restricted to at most *two* terminations per context, with at most *five* streams, with the default all-hear-all connections between them. There are many other simplifications and restrictions on the commands that can be sent, and the replies that may be received. This is acceptable, because (the first release of) Ericsson's media proxy is only intended to be used together with an Ericsson media gateway controller, and these two products can be designed to work together. It is important, though, to specify clearly what subset of the standard protocol the Ericsson products will use. This is done in an internal Ericsson document, the *Interwork Description* (IWD), a further 183 pages. This document, and the ITU standard itself, were our main sources of information when developing QuickCheck properties and generators.

4. Our Approach

We used a black-box testing approach for the Media proxy in which Quviq QuickCheck sends messages to the proxy and receives the replies. The replies are analyzed and cheched against expectations. In addition, crashes of the Media proxy can be observed, so that QuickCheck is aware of test cases resulting in a system failure.

4.1 Generating H.248 Messages

Our first task was to develop QuickCheck generators for H.248 messages. H.248 permits messages to be sent either in a text form, or in a binary form, and the standard contains a grammar for each. The text form of messages is specified by an ABNF grammar [4], while the binary form is specified in ASN.1 [6]. However, the Media Proxy represents messages as Erlang records, and we had a library available to convert these to and from the text and binary forms. We could thus generate Erlang data-structures, and use this library to encode them as text or binary data.

We initially tried generating random messages according to the ABNF syntax, but these were invariably rejected by the Media Proxy because they did not fulfill the conditions stated in the IWD. Thus we wrote generators which essentially were an implementation of these conditions. For example, messages may contain a *media descriptor*, which the standard specifies in ASN.1 as follows:

```
MediaDescriptor ::= SEQUENCE
{ termStateDescr TerminationStateDescriptor OPTIONAL,
  streams CHOICE
  { oneStream   StreamParms,
    multiStream SEQUENCE OF StreamDescriptor
  } OPTIONAL,
  ...
}
```

The IWD specifies that the termination state descriptor is not used, but that the streams must be present, and that there may be more than one of them. This is represented by the following QuickCheck generator:

```
mediadescriptor(Streams) when Streams=/=[] ->
  {mediaDescriptor,
   #'MediaDescriptor'{
      streams =
        case Streams of
          [{Id,Mode}] ->
            oneof([{oneStream,streamParms(Mode)},
                   {multiStream,[stream(Id,Mode)]}]);
          _ -> {multiStream,
                [stream(I,M) || {I,M} <- Streams]}
        end}}.
```

In this code, the teletype font is just constructing Erlang data structures to represent messages—just what would appear in a traditional test case. The italic font encodes the logic from the IWD—that there must be at least one stream, and that the possible representations differ depending on whether there is one stream, or several. The bold font is the only QuickCheck operation that appears here, expressing the fact that when there is only one stream, it may be represented either using the `oneStream` tag, or using the `multiStream` tag with a single stream. When a media descriptor containing one stream is actually generated, QuickCheck makes a random choice between these two alternatives. The function **oneof** returns a generator, not a stream descriptor, but QuickCheck allows any data-structure containing a generator to be used as a generator itself, which permits generators for complex types such as this to be written with a very lightweight syntax—QuickCheck functions need be introduced only where a random choice must be made.

To take another example, each stream in a media descriptor is described by a collection of *stream parameters*, specified in ASN.1 as follows:

```
StreamParms ::= SEQUENCE
{ localControlDescriptor
       LocalControlDescriptor OPTIONAL,
   localDescriptor
       LocalRemoteDescriptor OPTIONAL,
   remoteDescriptor
       LocalRemoteDescriptor OPTIONAL,
   ...,
   statisticsDescriptor
       StatisticsDescriptor OPTIONAL
}
```

The IWD specifies in addition, that "LocalControl will be included in all cases except when no media (m-line) is defined in the remote SDP", the remote SDP being the remote descriptor appearing among the stream parameters above. Thus there is a dependency between the appearance of the local control descriptor, and the form of the remote descriptor. There are essentially two cases for stream parameters, with and without an m-line in the remote descriptor, and we have to ensure that valid stream parameters are generated in each case. The QuickCheck generator which does so appears as follows:

```
streamParms(Mode) ->
  ?LET(RemoteMediaDefined, probably(),
       case RemoteMediaDefined of
         true ->
          #'StreamParms'{
           localControlDescriptor =
             localControl(Mode),
           localDescriptor =
             localDescriptor(RemoteMediaDefined),
           remoteDescriptor =
             remoteDescriptor(RemoteMediaDefined)};
         false -> ...
       end).
```

Here `?LET` is a QuickCheck construction which binds a variable, `RemoteMediaDefined`, to the value generated by `probably()`—a random boolean which is more often true than false. (It was considered that cases where an m-line *is* defined are more interesting to test). The final result is then generated by the third argument of `?LET`, in which `RemoteMediaDefined` is used to ensure that a local control descriptor is indeed generated if a remote media line is to be included, and (in the call to `remoteDescriptor`) that a remote media line is indeed included if we decided that it should be.

As these examples illustrate, it is quite straightforward to write generators that ensure that generated data meet the requirements stated in the IWD.

However, it is easy to make a mistake when writing these generators, and generate messages with the wrong structure. These mistakes could in principle be discovered by a static type-checker, but since Erlang lacks such a thing, we had to discover them by testing. We made use of the existing libraries for encoding Megaco messages as text, and wrote a QuickCheck property requiring that all messages generated by our generators could be encoded as text, and then decoded again to produce the same result. We found numerous mistakes in our generators by this method.

4.2 Testing Message Sequences

Most of the testing we carried out consisted of sending message sequences to the media proxy, and inspecting the replies they generated. Attention was focused in the first place on *positive testing*, that is, testing which aims to establish that the system under test responds as expected to *valid* inputs. Thus we wanted to generate message sequences that would conform to the proxy's expectations, rather than unexpected sequences that should trigger an error reply. In addition, we also looked at some *negative testing*, where in a sequence with valid inputs, an *invalid* input is generated. The proxy is in those cases expected to respond with an error reply.

4.2.1 Testing valid sequences

We generated sequences of Add, Modify, and Subtract commands, and checked that all the commands in each sequence could be executed by the proxy, with a correct "success" result being returned. The main constraint that we had to satisfy to ensure valid command sequences was that no context should ever contain more than two terminations—so sending three Adds in a row to the same context was not allowed. Moreover, Modify and Subtract commands must refer to existing terminations—which requires keeping track of the terminations which have been generated, of their identifiers (which are returned by the proxy), and of the subtractions which have already been performed.

We decided therefore to generate test cases consisting of a sequence of commands and assertions, each being a call to a suitable Erlang function. Since we expect this kind of testing to be common, we built a generic command-sequence testing module on top of the QuickCheck core, and constructed the media proxy tests in terms of the new module. In order to keep to valid command sequences, we constructed an abstract model of the proxy state, and used it to generate and recognise sequences fulfilling the conditions above. These two steps are discussed in the following two subsections.

4.2.2 A Module for Testing Command Sequences

Although it is easy to define an Erlang function which makes a sequence of calls, we wanted to represent such test cases *symbolically*—so that they could be displayed for the user, automatically simplified, and analysed to filter out invalid sequences. We therefore generated test cases as Erlang data-structures, and defined an interpreter responsible for actual test execution. While all of this would be quite possible using the core QuickCheck functionality, it is not exactly convenient, which is why we built an additional module `eqc_commands` to make this kind of testing easy. Although we think of `eqc_commands` as extending the functionality of QuickCheck, in fact it uses the same interface to the QuickCheck core as any other testing code. So although it was convenient for Quviq to supply it in this case, this was not necessary—`eqc_commands` could be written by any sufficiently skilled QuickCheck user.

To illustrate how we use `eqc_commands`, consider a simple example using the command `use(N)` to use a resource with number `N`, and the assertion `available(N)` which checks that `N` is *not* in use. This models a situation in which `use` commands have side-effects that can invalidate future assertions; this is what we wish to discover by testing.

We define a property that states that assertions always succeed, no matter which commands have previously been executed (of course, this property will fail).

```
prop_commands() ->
  ?FORALL(Cmds,make_commands(commands()),
     begin put(resources,[]),
           case run_commands(Cmds) of
             {ok,_} -> true;
             {error,_,_} -> false
           end end).
```

Here the `put` initialises the list of resources in use, while `run_commands` runs a list of commands, returning `ok` if no exceptions are raised and all assertions succeed. The commands themselves are generated by `commands()`:

```
commands() -> ?LAZY(
  frequency(
    [{1,[]},
     {5,?SET(OK,?MODULE,use,[choose(1,5)],
             commands())},
     {5,?ASSERT(?MODULE,available,[choose(1,5)],
                commands())}])).
```

The `frequency` function chooses between weighted alternatives, in this cases generating command lists with an average length of 11. `?SET` generates a command to call `use`, with an argument between one and five, binding the result to `OK`. `?ASSERT` generates an assertion calling `available`, with an argument between one and five. In both cases, the last argument of the macro generates the rest of the list of commands. (The enclosing `?LAZY` introduces *lazy evaluation* of the generator—without this, `commands()` would recurse infinitely).

When we tested this property, we obtained the following failing test case:

```
Failed! After 1 tests.
[{assert,test,available,[2]},
 {assert,test,available,[1]},
 {assert,test,available,[4]},
 {set,1,test,use,[2]},
 {assert,test,available,[5]},
 {assert,test,available,[2]},
 {set,2,test,use,[2]},
 {set,3,test,use,[1]},
 {assert,test,available,[2]},
 {assert,test,available,[5]}]
```

This is the value of `Cmds` in the property—as we can see, `?SET` and `?ASSERT` generate Erlang terms representing the function calls to be made during a test. The numbers occurring in `{set,...}` terms are *indices* which can be used to refer to the value returned by a command in later commands or assertions.

This failing test case does provoke an error, but is not particularly perspicuous. Fortunately, QuickCheck goes on to simplify the failing test as follows, using simplification strategies built into `eqc_commands`:

```
Shrinking........(8 times)
[{set,1,test,use,[2]},{assert,test,available,[2]}]
```

Simplification discards commands and assertions which do not contribute to the error, leaving precisely one assertion—the one that failed—and the one command which caused the assertion

failure. Such simplified test cases are an excellent starting point for debugging.

In this simple example, no use is made of the results returned by `use`, but in reality the result of one call is often needed to construct the arguments of later ones. This is why `?SET` binds the result to a variable (`OK` in the example above). Yet this result is not available while the test case is being *generated*—only when it is *run*—so what is `OK` bound to at generation time, and how can it later be used?

Our solution is to bind such variables to *an expression* that can be evaluated during test execution to yield the result of the call. These expressions take the form of Erlang data structures, and in the example above would be `{var,1}`, `{var,2}` and `{var,3}`. When such expressions appear in generated test cases, our interpreter replaces them by the value returned by the corresponding command, before each command or assertion is executed. Our interpreter also recognises structures of the form `{call,Module,Function,Args}` and interprets them as function calls. This gives us a simple symbolic representation for tests that process and use the results from earlier commands in later ones—something that is essential for testing the media proxy.

4.2.3 Tracking the Proxy State

In order to restrict testing to *valid* command sequences, we needed to predict the proxy state as the sequence is executed. We introduced an *abstract model* for this purpose. We needed to keep track of all the *terminations* created by the command sequence, of the *context* that each belonged to, and of the *streams* associated with each termination, together with their modes. We modelled the state by an Erlang record

```
-record(state, termination=[]).
```

where `termination` is a list of pairs of *termination identifiers* and abstract *termination states*. Termination states were also represented by a record:

```
-record(termstate, context, streams=[]).
```

where `context` is the *context identifier* of the context in which the termination resides, and `streams` is a list of *stream identifiers* (small integers) and *modes* (`inactive`, `sendRecv`, `sendOnly`, `recvOnly`). Since termination and context identifiers are allocated during test execution by the media proxy itself, then the *abstract* state used during test generation contains only *expressions that will evaluate* to the correct identifiers during test execution, as described above.

Given such an abstract proxy state, we can decide whether or not each command in a test case is valid. For example, an *Add* command is valid if the context it is adding to is either newly created (and so originally empty), or contains only a single termination. *Add* commands appear in our test cases as a call to the function `send_add`, with parameters the context being added to, the streams of the termination being added, and the request message which should actually be sent. The validity of *Add* commands is checked by a clause in the function `valid_cmd`, as follows:

```
valid_cmd(S,{set,V,_,send_add,[Cxt,Streams,Req]}) ->
    lists:member(Cxt,[?megaco_choose_context_id |
                      singletoncontexts(S)]);
```

Here `?megaco_choose_context_id` indicates to the proxy that it should allocate a new context, while `singletoncontexts(S)` extracts the list of contexts containing only a single termination from the abstract proxy state—so an *Add* command is valid under precisely the conditions stated above. The preconditions for *Modify* and *Subtract* commands are checked by other clauses.

To decide whether an entire *sequence* of commands is valid, it is also necessary to track changes to the abstract state caused by each command. We defined a function `state_after(S,Cmd)` to compute the new abstract state after execution of command `Cmd` in state `S`. In the case of an *Add* command, a new termination is added to the abstract state, with a termination identifier extracted from the result of the *Add*, and a context identifier extracted from the result if the context was newly created, and taken from the command itself if an existing context was used:

```
state_after(
  S,
  {set,Reply,_,send_add,[Cxt,Streams,Req]}) ->
  Context =
    if Cxt==?megaco_choose_context_id ->
         {call,
          ?MODULE,get_amms_reply_context,[Reply]};
       true -> Cxt
    end,
  #state{
    termination =
      [{{call,
         ?MODULE,get_amms_reply_termid,[Reply]},
        #termstate{context=Context,
                   streams=Streams}}
       | S#state.termination]};
```

In this code, `Context` is defined to be an *expression* which will evaluate to the context of the new termination at test execution time, and similarly the termination identifier is an expression which will apply `get_amms_reply_termid` to extract the real termination identifier at test execution time. The two `get_amms_...` functions are defined in the module containing this specification, and just select the right field from the reply. We are careful to reuse the existing expression for a context which already exists, so that we can easily tell at test generation time whether or not two context identifiers will denote the same context when the test is executed. Similar clauses update the abstract state after a *Modify* or *Subtract*.

Using `valid_cmd` and `state_after`, it is simple to define a function `valid_commands` which tests whether an entire sequence of commands will be valid, when executed in the initial abstract state.

Of course, it would be very costly to generate arbitrary sequences of commands, and then keep only those which happen to be valid. We therefore wrote a generator tailor-made to generate valid command sequences. We tracked the abstract state during generation also, defining `possible_cmd(S)` to generate a command which is valid in state `S`, together with an assertion checking that the command succeeded. It is always possible to generate an *Add* command, while *Modify* and *Subtract* commands can be generated only if there is a termination available to operate on. We generated commands which, by construction, satisfy `valid_cmd`, for example choosing the context that we add a termination to to be either `?megaco_choose_context_id` or an element of `singletoncontexts(S)`, so that the condition in `valid_cmd` is trivially satisfied. By using `state_after` to compute the abstract state after the generated command, we can ensure that the commands that follow are also valid according to `valid_commands`. Although there is a certain amount of repetition here—we express the same conditions both via a predicate and a generator—in fact, most of the code is common.

The reason we need *both* a generator and a predicate, is that even if the command sequence we initially generate is valid, once we start shrinking the sequence by dropping commands, there is no guarantee that it will remain valid. For example, it is valid to add two terminations to a context, subtract one, and then add a third,

because at no time are there more than two terminations in the context. Yet if this sequence were shrunk by discarding the subtract, then it would become invalid. We need not only a generator that produces valid sequences initially, but a predicate that we can use to recognise valid sequences during shrinking.

The property we finally used for testing was as follows:

```
prop_commands_succeed() ->
  ?FORALL(
    Cmds,
    make_commands(possible_cmds(initialstate())),
    ?IMPLIES(valid_commands(Cmds),
       case run_commands(Cmds) of
         {ok,_} -> true;
         {error,Results,Reason} -> false
       end)).
```

The precondition `valid_commands(Cmds)` ensures that any invalid sequences are discarded during shrinking.

4.2.4 Testing invalid sequences

The Interwork Description describes which sequences of messages are valid, and as long as all systems communicating with each other are based on the same IWD, then invalid messages need not be considered. For this reason, testing is mainly focused on valid message sequences.

Negative testing, that is sending a sequence containing an invalid message, is of interest for estimating compatibility with the full H.248 protocol. Since the IWD restricts the H.248 protocol, there are valid H.248 message sequences that are invalid in the context of the proxy. In the future, the media proxy may perhaps be connected to controllers from other suppliers, which may well generate such invalid message sequences. Such future releases of the proxy should at least reply with an error message when receiving a message that is valid according to the H.248 protocol, but invalid according to the IWD. Thus we included some negative testing in our case study.

There are numerous examples of invalid messages, and we only concentrated on two obvious cases: adding three terminations to a context, and using an arbitrary termination identifier in an Add request. In both cases the expected behaviour of the proxy is to reply with an error message.

Generating such invalid message sequences works in exactly the same way as generating valid messages. An Add request with an arbitrary termination identifier can only be sent in the case when a context already exists, and so at least one Add request has to precede this invalid request. Similarly, adding a third termination to a context must be preceded by two other Add requests. We use the abstract state to determine whether we can generate an invalid message. The result of the message is compared with the expected error reply and a fault is detected when that comparison fails. Note that invalid messages may be generated at any point in a message sequence, and indeed, a single sequence may contain multiple invalid messages.

Although rather straightforward, there is one issue one has to pay special attention to here. In the test case one should be able to distinguish between sending a valid and an invalid message. This is necessary in order to formulate the `valid_commands(Cmds)` precondition and to update the state correctly. For example, recall from Sect. 4.2.3 that we defined sending an Add request to be valid if the context either was a new context or one with one termination added.

```
valid_cmd(S,{set,V,_,send_add,[Cxt,Streams,Req]}) ->
    lists:member(Cxt,[?megaco_choose_context_id |
                      singletoncontexts(S)]);
```

If we just use the function `send_add` to send the third termination to an existing context, the precondition will rule out this test case. The solution is, of course, to create a new function, e.g., `send_third_add` and allow that only when the context already contains two terminations (remember that we may drop an Add request when shrinking!). The abstract state that we maintain while creating a test case is updated according to the expected result of adding a third termination—namely that since we expect the proxy to reply with an error message, the abstract state remains unchanged.

5. Results

As we developed generators and properties, we ran tests as often as possible, and of course encountered many errors in our own code. Often it was quite difficult to establish whether a failure was due to a misunderstanding on our part, or a genuine error in the system under test. However, after around six days of developing our QuickCheck specification, we had found five errors in the proxy itself. These are explained in the first subsection below.

We observed that the proxy was already working rather well when we began our experiment. In order to evaluate QuickCheck as a development tool, we also tested a version of the proxy from several months earlier, using the same QuickCheck properties. The results of this experiment are presented in the second subsection.

The faults we detected have all been reported to the development team. The seriousness of each fault was discussed and counter measures were taken.

5.1 Faults Found

The first fault we found was not really in the proxy, but in the Megaco encoding and decoding itself. Looking back at the ASN.1 specification of `StreamParms` in section 4.1, we see that *all* of its components are optional—and this is also true according to the IWD. This might lead one to conclude that it is valid to *omit* all the fields from a `StreamParms`—but when QuickCheck generated such a message, the encode-decode property that we used to test our message generators failed. On closer inspection, we found that while the ASN.1 grammar in the ITU standard allows for a `StreamParms` with zero fields, the ABNF grammar in the same standard insists that at least one must be present—it doesn't matter which one, but at least one must be there. It seems, then, that the standard itself is inconsistent on this point, which illustrates the dangers of constructing two descriptions of the same thing, with no automated check that they agree. (It is possible, but unlikely, that the *intention* is to permit streams with no parameters when the binary interface, specified in ASN.1, is used, but not when the text interface, specified in ABNF, is used). In any case, the encoder and decoder we were using encoded messages as text, and thus should have followed the ABNF grammar in the standard and rejected streams with no fields—but the encoder *accepted* them, while the decoder rejected them. However one chooses to interpret the standard, the choice should be made consistently, and so this reveals a fault in the software under test.

The other faults were provoked by minimal command sequences, generated as described above.

- If a single termination is added to a context, and then that termination is modified, then there is a crash inside the proxy. Modification is intended to be used to activate the streams in a context once both participating terminations have been added, and the designers had simply assumed that both terminations would be available when a modify was attempted.
- Adding one termination to a new context, and then subtracting it again, caused a crash in the proxy—on the first day we tried it. When we tried to duplicate the failure on the following day,

it no longer occurred. It turned out that the fault had just been fixed by the developers, via a patch which was not installed on the machine which we first used.

- Adding two terminations to a context, and then trying to modify one of them, results in a crash in the proxy *if the terminations do not have the same number of streams*. For example, if one termination carries audio and video streams, and the other carries only audio, then attempting to activate the streams leads to a crash. This was really the same underlying problem as the first fault above: the designers had assumed that each stream in a termination being modified has a "partner" in the other termination, but if the number of streams differs then this is not the case.
- The last fault we found arises when a termination is added to a new context; a second termination is added, and then subtracted; a third termination is added and then subtracted; then a fourth termination is added and subtracted. On this final subtract, there is a crash in the proxy. This failing case was particularly interesting because it illustrated the power of shrinking: the original test consisted of over 160 commands, but shrinking reduced it to just these seven. It is also a test that a human tester would hardly be likely to try. The problem was caused by a failure to clean up data-structures correctly on a subtract, leading to an incorrect state after the first subtract, but survivable until the third one.

Once our initial approach was decided, and the basic message generators written, we found that each fault required around half a day of work to find. After the second and fourth faults were found, we also needed to revise our specification to allow *Modify* commands only when two terminations with equal numbers of streams were present—thus documenting the bug in the specification via a stronger precondition. Without these changes, every run of QuickCheck found the same bug: especially considering that failing test cases are simplified, QuickCheck is very likely to report the "most common bug" on every run.

5.2 Testing an Old Version

As a second experiment, we rebuilt an old version of the proxy and tested it against the same specification, to see how QuickCheck would perform on a presumably buggier version of the same code. We rebuilt the second release after serious testing started. (The first release was merely an attempt to get the infrastructure in place.) Testing this old release proved to be a little more difficult than expected, since the interface had changed somewhat between the old and the new versions, and our QuickCheck specification had to be revised to conform to the old interface. We also had to strengthen preconditions after each bug was discovered, to prevent QuickCheck from reporting the *same* bug all the time, as discussed above. In total, we spent around 6 hours on this experiment.

The version we tested was responsible for two "trouble reports" in Ericsson's fault database. In six hours of testing, QuickCheck revealed a total of nine faults. One of these corresponded to a trouble report—the other trouble report was at the level of the underlying Session Data Protocol (SDP), which our QuickCheck specification did not test at all. (We just supplied suitable constants for SDP parameters in our message generators.) Given only a few days for developing the QuickCheck properties and generators, the result indicates that one can boost testing efficiency with QuickCheck.

We have tried to identify whether any of the other eight faults was caused by an error that was discovered via another, later, trouble report. This turned out to consume too much time for the engineers, since the trouble reports may well describe another fault than the one we observed, nevertheless caused by the same error. A rough guess was that one or two of the faults detected may be caused by a later discovered error.

We went through the list of trouble reports and estimated whether our QuickCheck approach would be able to detect a similar fault provided that we had had time to extend the specification. Some faults are typically hardware related or of other kinds that cannot be detected with our testing approach. Our educated guess is that we could at least detect half of the faults underlying the trouble reports. In addition, it is very likely that we would have identified faults that have not shown as trouble report.

5.3 Invalid sequences

Since negative testing has had low priority in the product development, it is not surprising to see that both kinds of invalid sequences we generate are causing a fault.

Normal termination identifiers look like "ip/12/162/1", where the numbers are depending on the hardware. It turns out that the developers have assumed the first of those four values always to be "ip" or at least some atom. Therefore, sending the termination identifier "0/0/0/0" (after shrinking from four arbitrary numbers) results in a crash of the proxy. Given the resulting failing test, we can rather quickly conclude that the numbers do not matter for this failure, since they have shrunk to zero. Therefore, it must be something with the first value being a number. This demonstrates how shrinking helps in analysis of the fault. After fixing the error in the code, we get proper error replies, even on a context like "12/162".

An Add request for a third termination to an existing context is expected to be replied upon with an error message. It turns out, however, that a reply never is returned. The hardware is not asked to add the third termination, thus somewhere the software cancels the request, but a reply is never generated.

6. Discussion

Overall, we are very pleased with the results of this case study. Although the proxy was already well tested when we started, we found a number of significant faults, with a modest amount of work per fault. We did discover, though, that *domain expertise* is essential for constructing this kind of QuickCheck specification—even the apparently complete documentation we were provided with leaves room for interpretation. This is of course true for all informal protocol descriptions.

The specification we developed provides *precise documentation* of how we interpreted the IWD. This turned out to be very useful in fault analysis.

We did encounter debate over whether or not all the faults we found were "real faults". The proxy is initially designed to work together with a specific Media Gateway Controller (MGC), under development at a different site within Ericsson, and developers responded that Ericsson's MGC would never send some of the message sequences we found. However, later versions of the same software will be expected to interoperate with MGCs from other suppliers, and at that stage the faults we found will be significant.

We were actually quite surprised by this objection, because the purpose of the IWD is just to document the assumptions that the media proxy may make about the MGC, and the restrictions we found were needed (specifically on *Modify* commands) are not stated there. Thus it seems that knowledge about which sequences will or will not be used has been communicated more informally. The IWD *does* contain a number of test cases in an appendix, which sure enough do not do the kinds of things that our generated tests do. It is common to include such sequences in the requirements for a system, and of course, they will then be used as a basis for testing. The risk is that developers of the proxy interpret them as *specifying* what sequences must be handled, while the developers of

the controller interpret them as *examples* of sequences that may be sent. Developing a QuickCheck specification forced us to formalise and document these extra restrictions, and we believe this is a valuable contribution, even if informal communication between the development teams has in this case proved sufficient.

There is a clear request from the users to be able to distinghuish positive and negative testing. However, the generators for valid message sequences are a subset of those for invalid sequences, since one needs to be able to bring the system in a certain state before certain invalid messages can be sent. A generator for arbitrary valid and invalid message may have a distribution of test cases that either hardly ever generates a negative test or far too often, depending on the likelyhood to bring the system in a state in which the negative message can be sent. For that reason, one should equip the generators with an argument used to choose between the kind of sequences that are generated. With different properties one could emphasize the different aspects of the system.

Quviq QuickCheck uses Erlang as a testing metalanguage (and also as the implementation language for QuickCheck), but has its origins in Claessen and Hughes' QuickCheck tool for Haskell [3]. Of course, in this case it was convenient to use Erlang, since the system under test was an Erlang system—but it is still interesting to compare Haskell and Erlang in this rôle as test metalanguages. There are substantial differences in the tools themselves, but from a language point of view, the major differences are Haskell's static type system, and lazy evaluation.

QuickCheck depends heavily on lazy evaluation, which is available "for free" in Haskell, but must be explicitly simulated in Erlang. However, we found that in most cases this can be hidden conveniently by macros—although not always. It is sometimes necessary to indicate laziness explicitly in Quviq QuickCheck specifications using the macro `?LAZY`.

Static typing would certainly have been useful to catch errors in generators earlier, although we found QuickCheck testing to be quite a good substitute. The original QuickCheck also allows a default generator to be associated with each type, so that properties can be formulated a little more simply when the default generator is appropriate. However, in this case study, *type* information is not enough to determine which generator is appropriate—we need to know the abstract proxy state too. So in this case, the advantage of using type information to guide generation would be small. On the other hand, representing test cases symbolically in Haskell requires type definitions, potentially many such, whereas in Erlang we could simply write down the values we wanted. Moreover, the kind of processing that `eqc_commands` does of test cases would require sophisticated generic programming in Haskell, but was much simpler in Erlang. Interpreting symbolic test cases was eased by Erlang's ability to invoke a function given its *name* as an Erlang atom—Erlang provides some basic reflection via this mechanism. We conclude that Erlang is rather a suitable host language for this kind of testing. It's dubious whether our specifications could have been expressed as concisely in Haskell, even if the versions of QuickCheck were otherwise equivalent.

7. Related Work

A predecessor of Quviq QuickCheck has been used in several case studies, among which one to test a leader election algorithm [1]. In that case study, random generation of process scheduling was crucially used. Hardly any generators had to be written for that case study and shrinking was restricted to lists of scheduling events. In the case study presented in this paper we have created generators for complex messages and we have contributed with a method and library to combine those messages into sequences. Other significant differences are that we now applied QuickCheck not only on industrial software, but also with the engineers who developed the software. Moreover, the application we looked at is much larger than the leader election algorithm.

There are many other tools and techniques for random test generation and for model based testing. It is hard to make a quantitative comparison between all the tools and techniques. We have demonstrated Quviq QuickCheck to managers at Ericsson's Methods and Tools department concerned with selecting the right testing tools for the company. They turned out to be very surprised by the possibilities QuickCheck offered and claimed: "Never seen this before". The strength of the tool hides in the combination of

- having a rich programming language, Erlang, as test specification language, compared to either a restricted or proprietary language,
- being able to automatically find a minimal counter example, and
- having full control due to a flexible API.

8. Conclusion

Our case study showed that QuickCheck is applicable to black-box testing of the synchronous part of real telecommunication protocols. Even though we only looked at one particular protocol, we have developed an approach that can be used for many similar protocols. We detected faults in the Media proxy and could trace those back to either ambiguities in the Interwork Description or errors in the code. Some of the errors we found were also found by the test team, other errors had not been discovered by that team. The total time it took to find the errors was a fraction of the time the test team had spent in total. Of course, the test team found more errors than we have found, also in parts we have not even tested.

QuickCheck also found both subtle and obvious bugs. The subtle errors will never show when the proxy is used as intended. However, in a different setting, where other vendors connect to the proxy, the errors may reveal themselves. Therefore, QuickCheck helped to identify those potential problems.

The results of the case study indicate that by using QuickCheck early in the development, one would be able to detect faults quickly with little time investment. Testers would therewith be enabled to concentrate on different kinds of tests, like performance tests, load tests, hardware tests, etc. In that way, one can ensure an even higher product quality.

The results of the case study have been encouraging enough to Ericsson to invest in Quviq QuickCheck. In recently started new product development QuickCheck will be used by several engineers from the start.

References

[1] Thomas Arts, Koen Claessen, John Hughes, and Hans Svensson. Testing implementations of formally verified algorithms. In *Proceedings of the fifth conference on Software Engineering Research and Practice in Sweden*, October 20-21 2005.

[2] Thomas Arts and John Hughes. Erlang/quickcheck. In *Ninth International Erlang/OTP User Conference*, November 2003.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[4] David H. Crocker and Paul Overell. Augmented BNF for syntax specifications: ABNF. Technical report, Internet proposed standard, October 2005.

[5] Marc Fewster and Dorothy Graham. *Software Test Automation – Effective use of test execution tools*. ACM press/Addison-Wesley, 1999.

[6] International Organization for Standardization. Information processing systems – Open Systems Interconnection (OSI) – specification of Abstract Syntax Notation One (ASN.1). Technical report, 1995.

[7] Telecommunication Standardization sector of ITU. ITU-T Rec. H248.1, gateway control protocol. Technical report, International Telecommunication Union, September 2005.

[8] U. Wiger, G. Ask, and K. Boortz. World-class product certification using erlang. In John Hughes, editor, *Erlang Workshop*, Pittsburgh, 2002. ACM SIGPLAN.

[9] R. Zeller, A. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:183–200, February 2002.