Testing a SIP decoder with QuickCheck - extended abstract

Hans Nilsson, Ericsson
Hans.R.Nilsson@ericsson.com

October 30, 2008

1 Introduction

SIP[4] is a protocol that recent years has gained high attention in the telecoms industry for services in connection with telephony over IP.

The messages in the protocol are text based and the syntax is defined by Augmented BNF[2] in RFCs. Unfortunately, the SIP grammar is not suitable as input to a traditional parser generator without radical re-writing. Left for the decoder/encoder implementor is a monotone programming session lasting for weeks.

The result is like all such code usually full of errors which makes systematic testing necessary. This paper describes a successful automatic test of a SIP decoder/encoder using QuickCheck[1]. The test code was generated from the BNF grammar and was therefore free from the test case implementators miss-understandings and errors.

In traditional testing, a number of test cases are programmed where the test object is given known input and the result is compared to the known output in the test case. This has some disadvantages. Only the cases that the tester can imagine and write as a program are tested. Usually only a few parameters in the test cases are varied in such code and only within the limits the tester believes are relevant.

There are alternatives gaining a growing interest. One is *property based testing* where the tester specifies the properties the test object shall have. The test system generates test cases to show that the properties are fulfilled. In the case of QuickCheck, the test input is randomly generated according to the specification. When an error is found, so called *shrinking* is applied and the tester is presented a *minimal* example that triggers that error.

Property based testing therefore generates a large number of test cases, often with value combinations that are a surprise for a human reviewing the generated test data. In that way a very good coverage is obtained.

2 The test

With QuickCheck the tester has to supply two things:

1. a *generator* that specifies the type of test input data that QuickCheck shall randomly generate

2. a *property* that decides if a result is valid or not with the actual input

In a simple example - a square function - the generator could be the QuickCheck built-in function generating integers. Those integers are given as arguments to the square function to be tested. The property could be very simple, for example just testing that the function return value is a positive integer or be more exact and really checking that the value is the square of the input.

For the SIP decoder/encoder there were two alternatives: either generate the internal Erlang form or the external text form. The text form was selected because that generator could be automatically obtained from the BNF in the SIP specifications.

The property shall test, that for all SIP messages M:

1. the semantics of M in text format is the same as the semantics of M decoded into the internal representation

2. the semantics of M in the internal representation is the same as M encoded into the text format

Why not just compare the syntax? Obviously the internal form differs from the text form. Two different messages in text form could however be semantically equal due to differences in case, number of blanks, line breaks and even line ordering. By some kind of normalization it could be possible to compare the syntax. Such a normalization is not trivial, and is probably error prone.

To avoid trying to extract the semantics out of both text form and internal form or to write a normalizer, a simpler but anyway useful middle way was chosen:

```
decode(M) == decode(encode(decode(M)))
```
where M is a message generated in text format by QuickCheck.

This is actually a syntactical normalization of the messages, since the decoder is such that differences in case etc are lost. Surely some test cases are missed but the testing is hopefully "good enough" - more about this later.

2.1 Example of generators and properties

The BNF is large - about 1000 lines in 20 RFCs. As an example of the direct generation[1] of code from BNF we could look at a sligthly edited example from [4]:

```
SIP-message    =  Request / Response
Request        =  Request-Line
                  *( message-header )
                  CRLF
                 [ message-body ]
Request-Line   =  Method SP Request-URI SP SIP-Version CRLF
Method         = REGISTERm / INVITEm / ACKm / OPTIONSm
                / BYEm / CANCELm / REGISTERm / extension-method
INVITEm           =  %x49.4E.56.49.54.45 ; INVITE in caps
```

[1]Thanks to Joe Armstrong who gave me a BNF parser

```
extension-method  =  token
token             =  1*tok-char
tok-char          = (alphanum / "-" / "." / "!" / "%" / "*" / "_"
                    / "+" / "`" / "'" / "~" )
```

Naively converted to generators this will be :

```
eSIP_message() -> oneof([eRequest(), eResponse()]).
eRequest() -> [eRequest_Line(),
               list_of(emessage_header()),
               eCRLF(),
               oneof([[], [emessage_body()]])
              ].
eRequest_Line() -> [eMethod(), eSP(), eRequest_URI(),
                    eSP(), eSIP_Version(), eCRLF()].
eMethod() -> oneof([eREGISTERm(),eINVITEm(),eACKm(),eOPTIONS(),
                    eBYE(),eCANCEL(),eREGISTER(),eextension_method()]).
eINVITEm() -> "INVITE".
etoken() -> [etok_char() | list_of(etok_char())].
etok_char() -> oneof([ealphanum(),45,46,33,37,42,95,43,96,39,126]).
eextension_method() -> etoken().
```

where `oneof/1` is a QuickCheck function that selects on of the elements in the list in the argument and `list_of/1` generates a list (maybe empty) where the elements is generated by the generator in the argument of `list_of/1`.

The property to test is basically:

```
prop_sip_decode_encode() ->
     ?FORALL(M, eSIP_message(),
             decode(M) == decode(encode(decode(M))) ).
```

The `?FORALL(Var, Generator, Property)` macro is a QuickCheck provided macro that generates a value by calling `Generator`, assigns it to `Var` and finally calls `Property` which returns `true` if the property is fulfilled.

3 Problems

There are of course some problems with the naive generation:

1. The BNF is not always correct. `"1:FF"` is for example a correct IPv6 address according to both the SIP grammar[4] and the IPv6 grammar[3]!
2. Some constructs results in infinite loops
3. Different branches in the BNF tree have different sizes of the sets of possible values, but the branches have same probability to be chosen when a value is to be generated. This gives the unwanted situation that some test cases will have higher probability than others. In the worst case, some constructs could be left untested while others are tested more than once.

4. When calling `eSIP_message/0`, *all* generator functions will be called. The resulting data structure and fun's will have the possibility to generate *any* SIP construct. Out from this, QuickCheck will only use one path and the rest will be thrown away.

The obvious solution for the BNF problems (1-2) is to re-write the BNF. For the unbalanced probability problem 3), the solution[2] is to add weights depending on the sub tree sizes. The size was simply calculated as 1 for a leaf and as the sum of all sub trees for an internal node. The QuickCheck function `frequency/1` takes as argument a list of {`Probability, Generator`} as argument, and selects one `Generator` depending on the `Probability`.

Problem 4) was solved with inserting the `LAZY` macro around all generator bodies. The result is that for a generator `g()` as argument, a `fun() -> g() end` will be returned instead. QuickCheck will then only eval `g()` if needed.

The resulting generator code for the example is:

```
eSIP_message() -> ?LAZY(frequency([{463,eRequest()},
                                   {377,eResponse()}])).
eRequest() -> ?LAZY([eRequest_Line(),
                     list_of_smaller(emessage_header()),
                     eCRLF(),
                     oneof([[], [emessage_body()]])
                    ]).
eRequest_Line() -> ?LAZY([eMethod(), eSP(), eRequest_URI(),
                          eSP(), eSIP_Version(), eCRLF()]).
eMethod() -> ?LAZY(oneof([eREGISTERm(),eINVITEm(),eACKm(),eOPTIONS(),
                          eBYE(),eCANCEL(),eREGISTER(),eextension_method()])).
eINVITEm() -> ?LAZY("INVITE").
etoken() -> ?LAZY([etok_char()|list_of_smaller(etok_char())]).
etok_char() -> ?LAZY(frequency([{5,ealphanum()},{1,45},{1,46},
                                {1,33},{1,37},{1,42},{1,95},
                                {1,43},{1,96},{1,39},{1,126}])).
eextension_method() -> ?LAZY(etoken()).
```

4 Results and discussion

Most important: a lot of errors was found. Many of them was in legal messages of strange sorts that normally would not have been tested. Even if such messages do not occur in practice, some components are sometime present. If such an error would have been left in the decoder, there would have been some sporadic and hard-to-catch errors left.

It is interesting that no errors was found in the encoder/decoder neither when it was tested conventionally nor during heavy usage in labs and at customer premises. Another part of the SIP stack was not tested by QuickCheck and was written by the same author. Conventional testing found some errors, but when tested by QuickCheck later, additional errors was found

The QuickCheck testing was extremely valuable and saved time and therefore money also for other parts of the system, since the decoder/encoder is very

[2]Thanks to John Hughes who actually pointed out this problem and the one in 4) for me and also solved them

central in the whole system. An error here will definitely stop most other test cases.

The approach of generating the generators from the “formal” BNF specification gave test code that is as correct as possible. The limited test in the property seem to have had no impact of the final test quality.

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG ’06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[2] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF, 1997.

[3] R. Hinden and S. Deering. IP version 6 addressing architecture (RFC2373), 1998.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol (RFC3261), 2002.