Software Engineering and Management

# Automated Testing of Java ABNF Grammar Parsers with QuickCheck

Pablo Álvarez Yáñez

Göteborg, Sweden 2007

REPORT NO. 2007/40

# Automated Testing of Java ABNF Grammar Parsers with QuickCheck

PABLO ÁLVAREZ YÁÑEZ


Department of Applied Information Technology
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY AND CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Automated Testing of Java ABNF Grammar Parsers with QuickCheck
PABLO ÁLVAREZ YÁÑEZ



Report no 2007:40
ISSN: 1651-4769
Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology
P O Box 8718
SE – 402 75 Göteborg
Sweden
Telephone + 46 (0)31-772 4895

Göteborg, Sweden 2007

# Automated Testing of Java ABNF Grammar Parsers with QuickCheck

**Pablo Álvarez Yáñez**
Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology

**Abstract**

I present a new technique to test Java ABNF grammar parsers using Erlang/QuickCheck, a property based test tool. Its development was lead through the case study of the Java reference implementation of the SIP protocol. The result was successful in the case study, and the technique was also used with some other parser to prove its portability.
The study revealed the importance of a complete knowledge about the randomly generated complex strings as the first step before fixing the found errors. The developed infrastructure provides that information, and combined with the use of a debugger with the parser, gives a powerful tool to find and fix obscure errors inside the parser code.

**Keywords** : Property Based Testing, Parser Testing, Protocol Testing

# 1 Introduction

This paper describes the development of a new technique for parser testing based on the use of QuickCheck, an Erlang tool to automatically test code against formal specifications. The technique is focused on Java implementations of ABNF grammar parsers. ABNF[1] is a metalanguage promoted by the IETF[2] to describe the syntax of protocol messages[3]. So we can contextualise the problem as a part of protocol testing. However it is general enough to fit any grammar based software in further works.
The investigation is structured following a case study using the SIP protocol, and its reference implementation, wrote in Java by the NIST[4]: JainSIP version 1.2[5]. This technique was later used with some other parsers, demonstrating that it is applicable outside the case study.
This study revealed the importance of a good understanding of the input in order to delimit and identify each part of the failing parsed strings as the first step before trying to fix the code. A simple string message is not enough to understand the error in complex grammars. Detailed information about the generation process following the grammar tree is required. The infrastructure developed during this study provides that input information, and combined with the ability of QuickCheck [AHJW06] to shrink inputs until the minimum failing test case, is extremely helpful to understand why a parser is failing parsing a concrete string. The use of a debugger in the parser side with the shrunk strings once the fault was delimited gave a quick and easy way to identify and even fix some *a priori* strange faults.
The developed technique showed successful results working with the JainSIP parser, identifying and fixing several errors in a mature version of a reference implementation. It showed the power of this technique in a real world scenario.

# 2 Erlang/QuickCheck

Quviq QuickCheck is a property based random testing tool redesigned for Erlang from the original Haskell QuickCheck. In addition to the original, it also provides automatic reduction of failing test cases. Quviq QuickCheck is a product of Quviq AB [AHJW06, SA05].
QuickCheck is a property based tool, which means the programmer has to write the properties he wants to check, comparing an expected result with the actual result. QuickCheck provides a set of Erlang macros to make the translation easier of mathematical properties to code.
For example, the logical statement

$\forall I \in int(), \forall L \in list(int()).$
$not(lists : member(I, lists : delete(I, L)))$

can easily be translated into the property:

[1] RFC4234 - Augmented BNF for Syntax Specifications: ABNF
[2] Internet Engineering Task Force. http://www.ietf.org
[3] http://en.wikipedia.org/wiki/ABNF
[4] National Institute of Standards and Technology from the USA
[5] https://jain-sip.dev.java.net/

```
prop_lists_delete() ->
 ?FORALL(I, int(),
  ?FORALL(L, list(int()),
   not lists:member(I,lists:delete(I,L)))).
```

QuickCheck also provides some library functions for complex data generation, and the interesting ability of *shrinking*: after a failing test, QuickCheck prints the information of the original fail, as well as the reduction up to the minimal failing input. The possibility to view all values generated in the shrinking process, also exists. This shrinking ability provides much more information in order to find the causes of an error in the original code. For complex input generation, the programmer can also define how the *shrinking* process should work, using the macros provided.
A simplified problem faced during this work will be used as an example to better understand how this QuickCheck ability helps in the testing process. Consider the following grammar, defined in ABNF mode (alpha defines a letter, sp a blank space):

```
message = method sp *(option)
method = A | B
option = ; *(alpha | special)
special = % | $ | #
```

If the parser fails parsing the character #, and the failing string is

```
A ;as#ds%;g#%s$f%sd;ju$df;a$%
```

QuickCheck would reduce the example, after the shrinking process, to a much more understandable test case:

```
A ;#
```

With the generation libraries, well formulated properties and the shrinking ability, QuickCheck can be a very powerful tool finding obscure errors in complex programs [AHJW06].

# 3 Session Initiation Protocol

SIP is an application-layer control protocol that can establish, modify, and terminate multimedia sessions. It can be used for Internet telephone calls, multimedia distribution, and multimedia conferences. The latest version is defined in the RFC3261, released in June 2002. It is widely used as signaling protocol in voice over IP programs like Windows Messenger[6], Apple iChat[7] or Ekiga[8]. Its main advantages above other similar protocols like H.323 are[9]:

- Lightweight. SIP only have six methods (and some others in extension RFCs), it does not know about the details of a session, it just initiates, terminates and modifies sessions.
- Transport independent. SIP can use TCP, UDP, ATM and so on.

[6] http://www.microsoft.com/windowsxp/evaluation/features/communication.mspx
[7] http://theappleblog.com/2005/06/03/reintroducing-sip-free-calling-for-all/
[8] http://ekiga.org/. Also known as GnomeMeeting
[9] http://www.sipcenter.com/ and http://en.wikipedia.org/wiki/Session_Initiation_Protocol

- Text based (simple human readable messages).

SIP has two types of messages: requests and responses. They start with a request and status line, respectively, followed by a set of option headers (see Figure 1). The explanation of states and state change will be omitted here. It is not interesting for the parsing process. Further information about SIP can be found in its RFC. The reference implementation of this protocol,

```
INVITE sip:UAB@example.com SIP/2.0
Via: SIP/2.0/UDP 10.20.30.40:5060
From: UserA <sip:UAA@example.com>;tag=589304
To: UserB <sip:UAB@example.com>
Call-ID: 8204589102@example.com
CSeq: 1 INVITE
Contact: <sip:UserA@10.20.30.40>
Content-Type: application/sdp
Content-Length: 141
```

Figure 1: Example SIP request message

made in Java by the NIST, has been used as a case study in order to develop the technique. It was coded under the JAIN/SIP project[10], with the last release on November 2006, version 1.2 TCK. Due to the fact that this implementation has been developed by an agency of the US Federal Government, there are no copyrights on it. It can be used for any purpose without a license agreement. Its API is also the JSR32[11], in final release status. The JainSIP stack has been used in numerous commercial and research projects[12].
The parsers can be found under the package *gov.nist.javax.sip.parser*. More than seventy parser classes are provided. Each kind of line has its own parser, as well as some important and reusable parts (IP address for instance). Hence the testing was made with individual lines instead of whole messages. The request line (the first line in the Figure 1) has the most complex parser, and therefore it was the most intensively tested class. The individual line testing is not a lack of generalization (since the parse of the whole message is based on them), and makes it easier to find errors. Besides, the technique would be the same in the case that other parsers provide only one method to parse entire messages (see section 10.1 - Testing Other Parsers).

# 4 Motivations and Related Studies

Protocol stack implementations are very special programs since they are designed to communicate different devices in a network. A large amount of different implementations of the same protocol in different languages, architectures and devices, even with some parts implemented in hardware, must understand the messages they are sharing in the same way Hence, interoperability is a vital topic. And it is based on the understanding of the messages by the parser.
However, protocol testing usually only relies on state change verification [BP94, MP94].

[10] Java APIs for Integrated Networks. https://jain-sip.dev.java.net/
[11] Java Specification Release 32, http://jcp.org/en/jsr/detail?id=32
[12] http://en.wikipedia.org/wiki/Session_Initiation_Protocol

QuickCheck has already shown its power testing a commercial implementation of the H.248 protocol [AHJW06]. This high level testing must be done, but parser testing is also very important. The parser is a concrete part of the stack with its own specification. And a tested parser in the early stages of development simplify the test of higher parts of the stack, knowing that the problem is not related with a wrong parse. Moreover, a passed state change verification does not imply a completely correct parser job, since not every field in the messages is related to state change.
Anyhow, it is usually carry out manually, specifying a set of messages to be tested. Because of the fact that it is a labour intensive process, great care must be taken in order to achieve a good rule coverage. However, the extreme complexity of protocol grammars, with a large amount of possible combinations, is too expensive to manually address, and usually interesting test cases are overlooked even with carefully constructed test suites. Due to the importance of the parser here, it should be done deeper.
An automated testing approach, based on hundred test cases randomly generated, seems to fit perfectly for this purpose. The subjective point of view of a human tester is lost, and rule coverage criterion rely on a mature testing tool.
Some studies have been done with parser testing in compilers, specially [HP05]. But the chosen approach, based on a selection of an *a priori* good test suite according to rule coverage statistics, is not applicable working with QuickCheck.

# 5 Goals

The main goal of this project will be the development of an easy and reliable method to test Java ABNF grammar parser with Erlang/QuickCheck. To achieve this, two subgoals should be fulfilled:

- Provide a fast and easy-to-use connexion between the Java parser and the Erlang testing node.
- Achieve an easy way to find and analyse faults, identifying and delimiting the underlaying errors in the parser.

# 6 The first approach

## 6.1 Selection of properties

The first step in any property based test approach, is the specification of the desired properties in an abstract way. This is usually one of the most challenging phases [FB97]. However, the desired properties chosen for a parser are at first sight quite simple (*parse* means a call to the parser and *pp* the pretty print of the parsed object):

1. $\forall string \in grammar \Rightarrow parse(string)\ valid$
2. $\forall string \in grammar \Rightarrow pp(parse(string)) = pp(parse(pp(parse(string))))$

The first property is obvious: any well formed string must be correctly parsed. In the second, the pretty printed first parse of the string should be compared with the pretty printed second one. This is due to the fact that during the parse, blank spaces, version numbers or escape characters are usually changed or removed. Hence, a comparison with the original string would fail. With this property it is assured that the parse keep the meaning of the message. If the meaning is changed, the equality or the second parse would fail due to a different understanding of the string. Instead of the comparision of the two pretty printed strings, in Java the equality operator directly with the parsed objects could be used, if it is available.

### 6.2 Implementation with String Generator

The first approach was a direct generation of strings according to the grammar specification. Since Erlang syntax follows a BNF mode, the translation of the specification as an ABNF grammar into an Erlang module to be used as generator is rather simple. ABNF operators, like concatenation or star functions, as well as its core rules (digits, characters, new line, etc.) had to be implemented. The shrinking functionality was specified on these files. The module *eqc_abnf* provides the method to construct the generator automatically from the ABNF, as well as the module *eqc_rfc4234* provides the core rules and operations for ABNF grammars.
With the generator and the ABNF primitives implemented, the properties can be translated to QuickCheck

```
prop_string_one(Generator, NonTerminal, Type) ->
 ?FORALL(String, apply(Generator,NonTerminal,[]),
    begin
      FlatString = lists:flatten(String),
      ?CLIENTMOD:validateString(?SERVER,Type,FlatString)
    end).

prop_string_two(Generator, NonTerminal, Type) ->
 ?FORALL(String, apply(Generator,NonTerminal,[]),
    begin
      FlatString = lists:flatten(String),
      ?CLIENTMOD:parseAndCompareString(?SERVER,Type,FlatString)
    end).
```

In this case, the translation is almost direct from the mathematically formulated properties. Type is used to choose among different parser types. However, to start testing a sort of infrastructure to call the Java parser has to be provided. Actually the ?CLIENTMOD and ?SERVER macros make reference to that infrastructure (see below for details)

### 6.3 The Java Connexion

In order to achieve the communication between Java and Erlang, the IDL[13] compiler provided with the standard Erlang distribution[14] has been used. The IDL compiler translates an IDL file into a programming language specific client stub and server skeleton files. It manages the use of the interface with Java[15] and the marshaling of data types[16]. It is transparent, quick and easy

[13] Interface Definition Language
[14] http://www.erlang.org/doc/doc-5.5.4/lib/ic-4.2.12/doc/html/ic.html
[15] JInterface library
[16] http://www.erlang.org/doc/doc-5.0.1/lib/ic-4.0.5/doc/html/ch_idl_to_erlang_mapping.html

for the programmer. Once the IDL file is defined with the headers of the functions to call from the Erlang node (see Figure 2), the Erlang compiler has to be used to automatically generate the skeletons and stubs:

```
$> erlc '+{scoped_op_calls,true}' '+{be,erl_genserv}' parsertesting.idl
$> erlc '+{scoped_op_calls,true}' '+{be,java}' parsertesting.idl
```

The first command will generate the client stubs and headers (oe_parsertesting.erl, testing_parsertest.erl, oe_parsertesting.hrl, testing.hrl and testing_parsertest.hrl). The macro ?CLIENTMOD from the point above makes reference to the module testing_parsertest, which implements the holders for the functions defined in the .idl file.

```
-define(CLIENTMOD,'testing_parsertest').
```

The second command generates the Java skeletons. They form the package *testing* in the Java Server node (see Figure 3). The own Java node has to be programmed as well, the *server.Server* class in the diagram[17]. This file holds the main method and acts as the server node, receiving the messages from the Erlang node. The macro ?SERVER in the point above is a reference to this node (*javaServer* is the name the node is registered with. It can be changed in the configuration file).

```
-define(SERVER,{testing_parsertest_impl, list_to_atom("javaServer@localhost")}).
```

To get communication, the programmer only needs to run the Java program, which will act as an Erlang node, sharing a cookie with the Erlang part (editable in the configuration file too). Therefore, the tester only needs to write the implementation of the functions specified in the IDL file, extending the abstract class *server.ServerImpl*. These methods will call the concrete parser functions, and will return the desired values. Besides, the Java server has been designed using a factory pattern to load a concrete implementation file (see Figure 3). So, the same program can hold the implementation of several parsers, and switching among them only means a change in the configuration file.

```
module testing  {
  interface parsertest {
    string parseString(in string type, in string stringToParse);
    boolean validateString(in string type, in string stringToParse);
    boolean parseAndCompareString(in string type, in string stringToParse);
  };
};
```

Figure 2: Used IDL file, parsertesting.idl

Three methods have been designed in order to perform the tests:

- String parseString( String type, String stringToParse)
  Parses the string *stringToParse* according to the parser specified with *type*. Usually complex parsers provide different specific parsers for each kind of line instead of one for the

[17] See similar program in lib/ic-4.2.11/examples/all-against-all/server.java under the Erlang installation folder

whole message (see point 3) . This is the case of the used SIP parser. The variable *type* is used to choose among these parsers.
Returns the pretty printed parsed string, or some string showing the fault in case the string is not correctly parsed (usually the exception raised is a good choice).

- boolean parseAndCompareString( String type, String stringToParse)
  Parses the string *stringToParse* and then it parses again the pretty printed string of the first parse. Returns the equality between the returned parsed objects.

- boolean validateString( String type, String stringToParse);
  Parses the string *stringToParse* and returns true if it is correctly parsed, false other way.

It is also strongly encouraged to print out the received string. It becomes really useful to follow and understand the shrinking process. With the second property, it is also really useful to know directly if there is an exception parsing the first string, parsing the second one, or if the error is due to the non equality between them. This information will avoid the use of a debugger checking the properties. It is too slow and uncomfortable to use the debugger while the properties are being checked. To use it with the final shrink string is a better idea. The function *str_checker:check_string(Type, String)* is provided to communicate directly with the parser (Type has the same meaning than in the Java functions).

# 7 Early Results

To check the properties, QuickCheck provides the method *eqc:quickcheck(property())*, which executes the desired property one hundred times, stopping and shrinking if it finds an error. The first tests revealed faults in numerous line types and with both properties. It was quite surprising working with a mature version of the reference stack implementation.
To find the errors is quite easy, but even if a debugger is used to check what is going wrong in the Java part, to understand it is much more difficult. Some kind of errors such as buffer overflows can be identified now. But with a complex grammar, and a complex parser, we would need something more to solve it.
An example of execution with Request-Line (the first line in a request message. See point 3 and Figure 1) will illustrate what is happening. Executing the property, an error is found after only four tests. The initial failing string is:

```
INVITE sips:[7:52];user=ip?%F4=%75 SIP/05.3
```

And after shrinking QuickCheck returns:

```
INVITE sips:[0] SIP/0.0
```

The request line is formed by the method (INVITE), the URL field, (sips:[0]) and the protocol version (SIP/0.0). If we try to debug the parser using this string as input, an exception will be raised in the section which parses the URL (Parse Exception: Unexpected Token). The string *sips:* shows that it is working in safe mode (safe sip) and the origin of [0] is not evident. However the cause of the fault was not possible to discover. More information would be desirable in order to connect the fault with the underlaying code:

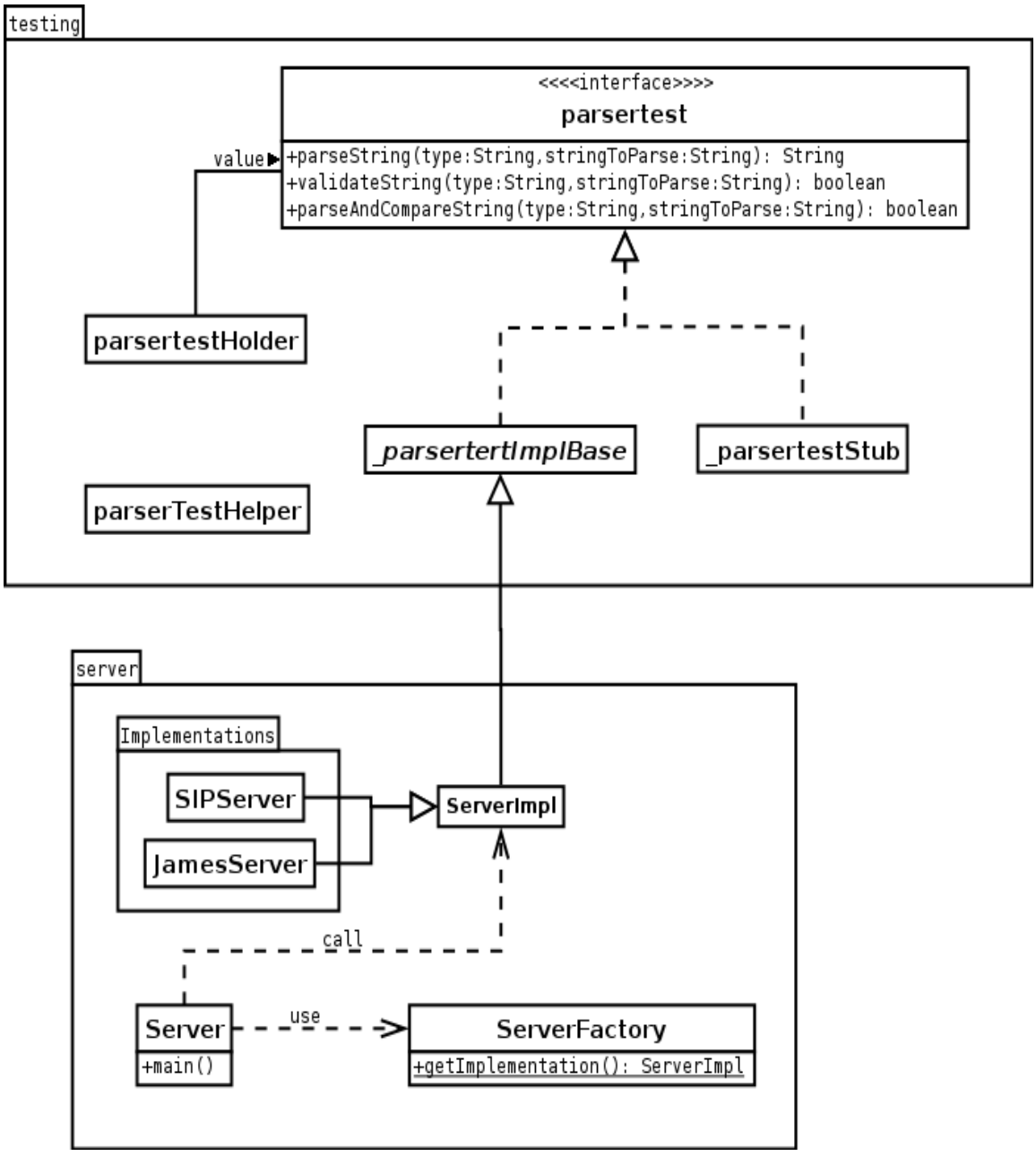


Figure 3: UML class diagram of the JavaServer program

- Which non terminal produces each character of the string. It can be done by hand in the RFC, but with complex grammar crawling among alternatives and optionals through dozens of non terminals seems to be a wrong approach. This is the first step to guess what can be going wrong.
- The whole stack of calls to non terminals that are producing the string. It can be useful, for example, to realize if the problem is related with just one bracket, or if the whole structure [0] means *something* within the grammar. It becomes more useful with strings like

  ```
  BYE o://4253.3.m-o0.n.:380/%D9@@;;;$;&=./=%E6&;&$;/,;+ SIP/356729.709
  ```

- The ABNF core rule calls. The core rules are defined in the appendix B.1 of RFC4234 (the ABNF definition), which contain the definition of certain basic rules such as alpha, digit, new line, blank space, etc. They are implemented in the eqc_rfc4234 module, with the ABNF operators. The importance of this point is to know if the problem is related to a concrete character specified on the grammar, or if it is produced with the alpha ABNF core rule, for example.

# 8 The Tree Approach

The first approach, testing the parser by parsing randomly generated strings, showed a important weak point trying to locate what is causing the error in a failing test case. Therefore, a new way to carry out the test process in three phases was constructed. It starts with a completely new representation of the generator, which now, instead of directly strings, produces trees with all the information about which terminals and non terminals are being used. Then, a concrete tree is picked, and finally it is converted into a string ready to be parsed. So, if the parsing fails, QuickCheck should show the shrunk string, as well as its own concrete tree representation.
This new tree representation store every non terminal used during generation, the symbolic representation of the ABNF operations, and the ABNF core rules. In this form it can provide all the necessary information commented in the point above. Each non terminal in the generator should be in the form:

```
'Non-Terminal'() ->
        {tree, 'Non-Terminal', [attribute]}.
```

Where attribute can be any other tree or an ABNF operation in a symbolic representation. The resultant generator should look like:

```
'SIP-message'() ->
        {tree,'SIP-message',
        [{call,?mod,alternative,[[
                'Request'(),
                'Response'()]]
        }]}.

'Request'() ->
        {tree,'Request',
        [{call,?mod,concat,[[
                'Request-Line'(),
```

```
                {call,?mod,star,['message-header'()]},
                'crlf'(),
                {call,?mod,optional,['message-body'()]}]]
        }]}.

'Request-Line'() ->
        {tree,'Request-Line',
        [{call,?mod,concat,[[
                'Method'(),
                'sp'(),
                'Request-URI'(),
                'sp'(),
                'SIP-Version'(),
                'crlf'()]]
        }]}.
```

## 8.1 Working with Trees

In the first two phases of this new approach, the tree representation has to be managed. But QuickCheck does not know how to work with it. New functions to operate with trees had to be written. The module *eqc_trees* was designed to provide the needed functions:

**eval_tree(Tree)** evaluates a tree into a string ready to be parsed. Useful to test if the generator works properly.

**format_simp(Tree)** reformats the tree into a more readable representation.

**pick_tree(Tree)** chooses one concrete tree expanding the star and repeat operators and picking one of the possible alternatives within the alternative operator. It also adds the shrinking abilities for the concrete tree.

**tree_to_symbolic(Tree)** transforms the tree into a standard symbolic representation letting QuikCheck to work with it natively.

A new implementation of the ABNF core rules and operator in symbolic mode is also needed. It was implemented in the module *eqc_rfc4234_sym*. With these functions, the new properties can now be written.

## 8.2 Implementing the Properties with Trees

This new implementation will not be too close to the formal representation as it was in the first approach. The necessity of the concrete tree representation of the final string causes the use of three nested ?FORALLs. The first one produces a generator in tree mode, the second one picks a concrete tree and the last one produces the string. In the body of the property, a ?WHENFAIL macro was also used in order to print out a more readable representation of the tree on the screen in the failing test cases.

```
prop_tree_one(Generator, NonTerminal, Mode) ->
  ?FORALL(Tree, noshrink(apply(Generator,NonTerminal,[])),
    ?FORALL(Tree_simp, eqc_trees:pick_tree(Tree),
      ?FORALL(String,eqc_gen:eval(eqc_trees:tree_to_symbolic(Tree_simp)),
      begin
```

```
        FlatString = lists:flatten(String),
        ?WHENFAIL(
          io:format("~p\n~s\n",[eqc_trees:format_simp(Tree_simp),FlatString]),
          ?CLIENTMOD:validateString(?SERVER,Mode,FlatString))
      end))).


prop_tree_two(Generator, NonTerminal, Mode) ->
  ?FORALL(Tree, noshrink(apply(Generator,NonTerminal,[])),
    ?FORALL(Tree_simp, eqc_trees:pick_tree(Tree),
      ?FORALL(String,eqc_gen:eval(eqc_trees:tree_to_symbolic(Tree_simp)),
      begin
        FlatString = lists:flatten(String),
        ?WHENFAIL(
          io:format("~p\n~s\n",[eqc_trees:format_simp(Tree_simp),FlatString]),
          ?CLIENTMOD:parseAndCompareString(?SERVER,Mode,FlatString))
      end))).
```

Should be noticed that a call to *noshrink()* is used with the first generator. It is due to the fact that QuickCheck does not know how to shrink that kind of structure. Without that call, QuickCheck would try to shrink it, producing a huge amount of repeated strings which would make the process much slower.

# 9 Results of the Tree Approach

This new representation and the way to work with it is quite different from the original approach. To illustrate how it works, an example of execution and the comments about how to use the information will be showed.
Like in the first approach, a call to *eqc:quickcheck(property())* is needed. The first property, involving only one parse will be used, and again the request line. After some tests an exception is raised, stopping the QuickCheck execution and starting the shrinking process. The execution now is a little bit slower, and a big amount of text appears on the screen. That information is not valuable. Only the shrunk tree and the final string is needed, printed out in the screen at the end of the process. The original failing string is:

```
OPTIONS sip:42.5.00.323;user=%_+ SIP/772.6004047
```

and the final shrunk string is:

```
OPTIONS sip:0.0.0.0;user=%+ SIP/0.0
```

The obtained shrunk tree is not always the exact representation of the string. Part of the shrinking process is executed in the last ?FORALL, so the shrunk tree can be reduced even more in the last part. Even so, the identification of the leafs in the final string is easy. After some *pruning*, the final tree is:

```
{'Request-Line',[[{'Method',[{'OPTIONSm',["OPTIONS"]}]},
            {call,sp},
            {'Request-URI',[{'SIP-URI',
                    [["sip:",
                         {hostport,
                              [[{host,
                                   [{'IPv4address',
                                        [[{call,digit}],
```

```
                                    ".",
                                    [{call,digit}],
                                    ".",
                                    [{call,digit}],
                                    ".",
                                    [{call,digit}]}]}]}],
                        [{'uri-parameters',
                             [[{'->OPT',
                                  [[";",
                                      {'uri-parameter',
                                          [{'user-param',
                                              [["user=",
                                                   {'other-user',
                                                       [{token,
                                                            [[{'->OPT',
                                                                 ["%"]},
                                                            "+"]]}]}]}]}]}]}]}]}]}],
                {call,sp},
                {'SIP-Version',[["SIP",
                "/",
                      [{call,digit}],
                       ".",
                      [{call,digit}]]]},
                {call,crlf}]]}
```

The last piece of valuable information comes from the last part of the shrinking process printed out by the Java server:

```
...
-------------------------------------
OPTIONS sip:0.0.0.0;user=+ SIP/0.1

valid
-------------------------------------
OPTIONS sip:0.0.0.0;user=%+ SIP/0.0

ERROR
-------------------------------------
OPTIONS sip:0.0.0.0 SIP/0.0

valid
-------------------------------------
OPTIONS sip:0.0.0.0;user=+ SIP/0.0

valid
-------------------------------------
```

Looking at all this information now, it can be said that the problem is related with the character '%'. Looking for it in the tree, it is identified as produced by the 'token' non terminal, as part of an 'uri-parameter'. The non terminals involved are defined in the original ABNF grammar as:

```
user-param        =  "user=" ( "phone" / "ip" / other-user)
other-user        =  token
```

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

Trying the same string with the rest of special characters in 'token', errors were raised with the characters '%' and '`'. With more executions of the property, this error appeared in some other 'uri-parameter' related trees, with these two special characters generated with the 'token' non terminal.
Hence, the lack of information of the first approach is now solved, and errors are perfectly delimited and put in context.
The parser code can now be seen with a different perspective, and debugger steps are much more understandable now. Actually, in the file *gov.nist.javax.sip.parser.URLParser*, the debugger shows the use of a function called *paramNameOrValue()* parsing those characters. Here is the code of the function:

```
/*
 * Name or value of a parameter.
 */
protected String paramNameOrValue() throws ParseException {
        StringBuffer retval = new StringBuffer();
        while (lexer.hasMoreChars()) {
                char next = lexer.lookAhead(0);
                if (next == '['
                        || next == ']'
                        || next == '/'
                        || next == ':'
                        || next == '&'
                        || next == '+'
                        || next == '$'
                        || isUnreserved(next)) {
                        retval.append(next);
                        lexer.consume(1);
                } else if (isEscaped()) {
                        String esc = lexer.charAsString(3);
                        lexer.consume(3);
                        retval.append(esc);
                } else
                        break;
        }
        return retval.toString();
}
```

These characters are parsed by the *isUnreserved(next)* function, because there is not any special comparison with them. But they are not unreserved characters (unreserved is another non terminal in the SIP grammar).
Hence, with the function specification, the tree, the shrunk information and the execution path seen with the debugger, this seems to be the place where these characters should be *consumed*.

```
...
char next = lexer.lookAhead(0);
        if (next == '['
                || next == ']'
                || next == '/'
                || next == ':'
                || next == '&'
                || next == '+'
                || next == '$'
                || next == '%'  //Added by Pablo Álvarez Yáñez
```

```
                    || next == '`'  //Added by Pablo Álvarez Yáñez
                    || isUnreserved(next)) {
                retval.append(next);
                lexer.consume(1);
            } else
    ...
```

Actually, if a comparison with these two characters is added here, the exception is not raised any more, and any new different error is produced. The fault seems to be solved, quickly and without a strong knowledge of the parser code!.

## 9.1 Found Errors

It is important to notice that errors in parsers are extremely difficult to discover, and they are usually detected as side effects. Some strings can be recognised by the parser but with a different parse tree that the proper one. It could give problems in higher layers of the protocol stack but sometimes is not easy to identify at parser level.
Detailed below is a extensive explanation of the errors found out with Request Line and its parser using the above described technique. Similar errors have also been found in some header line parsers.

1. Error parsing characters "%" and "`" in parameter values, described above.
   - Method used: Tree mode, one parse property.
   - Symptoms: Exception raised in strings with these characters generated by the token non terminal as part of an uri-parameters subtree.
   - Status: Fixed.
   - Solution: Addition of a comparison with the characters in the *URLParser.paramNameOrValue()* function as described above.
   - Example shrunk string:

     ```
     OPTIONS sip:0.0.0.0;user=% SIP/0.0
     ```

2. SIPS mode not considered in RequestLineParser, despite it is an option in the constructor of the *gov.nist.javax.sip.address.GenericURI* class as a field of *gov.nist.javax.sip.header.RequestLine*.
   - Method used: Tree mode, one parse property.
   - Symptoms: Detected as a side effect due to failures parsing "[" and "]" characters in sips mode.
   - Status: Open. Sips mode removed from generator to keep on testing.
   - Example shrunk string:

     ```
     REGISTER sips:f0;[ SIP/0.0
     ```

3. Error parsing character ' in query headers for URLs.
   - Method used: Tree mode, one parse property.

- Symptoms: Exception with unexpected token raised parsing this character, generated by the non terminal mark as part of a query header value.
- Status: Fixed.
- Solution: Addition of a comparison with this character parsing query option values in the *URLParser.hvalue()* function.
- Example shrunk string:

  ```
  BYE sip:0.0.0.0?%0=' SIP/0.0\n
  ```

4. Error getting absolute URI requests with characters [ and ]. Actually this is the same problem that the one shown in point 2. In that case, SIPS URIs were parsed as absolute URIs, but a deep analysis showed a difference between these two kinds of strings which motivate its treatment as different errors.

   - Method used: Tree mode, one parse property.
   - Symptoms: Error in absoluteURI requests with IPv6 references (IPv6 references appear between brackets).
   - Status: Open.
   - Example shrunk string:

     ```
     ACK e://[0] SIP/0.0
     ```

5. Error parsing queries in the form *query = value* without any value. The parser removes the character = from the string during the first parse.

   - Method used: Tree mode, two parses property.
   - Symptoms: Exception raised during second parse trying to get the value of the query at *gov.nist.javax.sip.parser.URLParser.qheader()*
   - Status: Open. This is a semantic error. The construction is allowed by the grammar, but a *query = value* query is meaningless without value.
   - Example shrunk strings:
     (a) `BYE sip:d0?+= SIP/0.0`
     (b) `BYE sip:d0?+ SIP/2.0`

     where (a) is the original string and (b) is the returned string after the first parse.

6. Error using the equality operator to compare two RequestLine objects. The error is outside the parser, trying to get the ttl parameter inside the URL Object. Despite the parameter is well identified and stored, there is some error trying to get it for the comparison.

   - Method used: Tree mode, two parses property.
   - Symptoms: Exception raised trying to access to the ttl parameter.
   - Status: Open. Due to the fact that this is not a parser error, any solution was adopted here. The deletion of the ttl parameter from the generator can be seen as a solution to keep on testing, once has been checked out that the error is not in the parser.
   - Example shrunk string:

     ```
     INVITE sip:[0];ttl=0 SIP/0.0
     ```

### 9.2 Characterization of the Errors

Having in mind the errors described above, a general taxonomy of parser errors can be made. It is very useful to identify the type of the error before trying to fix the parser, because usually a wrong approach is used. A brief explanation to solve or avoid each kind of error is given here.

**Problems related with special characters.** They are the most usually found errors. Special characters produced by some non terminal in the grammar are sometimes forgotten in the parser construction. With the tree mode properties, these errors are usually well delimited, and to fix them is not very hard . Anyway, if the bug can not be solved, the generation of the character can be avoided changing the generator.

**Wrong recognition of a non terminal** Typical parser error. Some subtree is identified as something different from what it is, and the error raises in some time trying to parse it with a wrong function. They are difficult to delimit, because the root of the subtree where the error starts is not obvious. Its symptoms are usually side effects during the parsing. The worse case is that the parser consumes the string without any error but with a wrong identification of the structures. Try to identify some *a priori* quite different strings with the same error can be useful to find the root of the subtree.

**Non implemented part of the protocol** Sometimes, the stack implementation only implements part of the protocol. To be able to test, the grammar should be restricted to the implemented part. It can be more difficult than it seems, and the described technique used with a debugger can help to identify the non terminals involved. Sometimes this problem also appears with new releases of the RFC which adds new non terminals.

**Semantic errors** Usually found with the two parses property. A string generated according to the grammar specification is meaningless to the parser and it returns some modified string. The second parser would fail trying to parse it, or the comparison could return false. The solution to adopt here is in the hands of the programmer. A change in the grammar can be an option. Anyway, these meaningless strings are usually not produced in real protocol runs.

**Bugs in parts of the stack outside the parser** These errors are not real parser errors, but they affect in some way the testing process. They should be treated as any other standard software error. No solution based on this technique is possible here.

## 10 Analysis

The combination of the tree representation with QuickCheck gave amazing results with the used SIP parser. It was not only able to identify and delimit several errors in a mature version of a reference implementation, but also strongly helped in the process to fix them. Can be said that this is the most important success of the developed technique. It could even allow, in a real world testing process, a role separation between the original programmer and the tester without a so strong code knowledge,.
The given infrastructure makes the internal use of QuickCheck and Erlang really easy and

transparent for a potential tester without those skills, only needing to call some functions from an Erlang shell. The internal use of the new tree representation with its own functions does not add a strong load to the testing process, and execution and shrinking are still rather fast.
The Java connexion was simplified to the implementation of three methods extending an abstract class, and the modification of some parameters in a configuration file. It avoids the always tedious work of integrate two different programming platforms.
The characterization of the most usual error types made in the point 9.2, could be helpful for a tester once an error is identified, helping in the not always easy decision of what to do with the error or feature found. Besides, the possibility to avoid some non terminal generation, changing the original grammar, can be really useful. It would allow to keep on testing while part of the team try to fix the found bugs, making testing process agiler.
Hence, can be said that the expected goals were accomplished, and obtained results were even over initial expectations. However, the target of the thesis was the development of a general technique to test Java ABNF grammar parsers, not only the test of a single SIP parser. So, even if the technique seems to be designed to fit any other parser with the described characteristics, the study must be completed setting up and testing some other parsers.

## 10.1 Testing Other Parsers

As was said above, to choose a parser which can be used with this software, it should fulfil three factors:

1. An ABNF defined grammar. Most of IETF protocols specify their syntax in this mode. Some old protocol releases are specified using other representations, but usually new RFCs with the grammar updated into ABNF exist (SMTP for instance).

2. The parsing functions should be accessible from the outside. Most of protocol implementations have special parsing classes or the parse function inside the data type classes. But sometimes, to improve performance, the parsing is made at the same time the corresponding action of the protocol is performed. It can be solved changing the action function calls for fake functions, leaving only the parsing part and its exceptions. But it requires an extra work, and it is not always possible.

3. Data types should provide a pretty printer. In Java the toString() method is usually implemented for every object. But if this is not the case, the tester should write his own one if the access to the fields which form the complete string are public.

Another point to think are the cross references in the grammar. Usually in a RFC the defined grammar uses non terminals specified on another RFC. For instance, URIs and emails are usually specified calling their representation on the RFC2822 - Internet Message Format or RFC3986 - Uniform Resource Identifier (URI): Generic Syntax. A generator for those grammars can be generated and compiled, and those functions imported from the original generator. But sometimes this extra effort is not worthy. On one hand, it makes generation much slower. Generated trees could be huge, and it could even cause memory overflows. On the other, those structures are probably not parsed, and they are only stored as normal strings. The knowledge of how the program performs the parsing of those structures should guide the tester to call the

outer function or just change it for a standard string (changing the call to an URI reference for just "http://reference/", for example).
So, some applications were chosen in order to test the portability of the developed software testing other parsers.

### 10.1.1 The java.net Package from the standard JDK

The *java.net* package, provided with the standard JDK, offers some solutions for implementing networking applications. A parser for URLs and URIs can be found there. These Java classes follow the format defined by RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, amended by RFC 2732: Format for Literal IPv6 Addresses in URLs. A new version exists, RFC 3986, released in January 2005 which obsoletes RFC2396 and also add the introduction of IPv6 addresses. However, the Java version used (1.5.0) still use the old release. Since the new release do not notify changes in the URI and URL syntax, as can be read in its introduction, it was used trying to avoid complex cross references.
During testing, some errors were revealed. A new non terminal introduced in the new release in anticipation for future releases, is obviously not implemented (IPvFuture). Some errors related with the new way IPv6 addresses are defined were also noticed.

### 10.1.2 Apache James

The Apache JAMES Project delivers a rich set of open source solutions, written in Java, related to internet mail and news[18]. Among some other functions, it provides a parser for email address within the Mailet API, following the specification from RFC 2821 - Simple Mail Transfer Protocol (SMTP). This was the tested part.
This specification makes some references to RFC2822 - Internet Message Format, so both grammars were generated and compiled, and the needed function imported. The parsing is made within the constructor of the *org.apache.mailet.MailAddres* class, so in the Java server only was necessary the creation of an MailAddres object with the generated string. The testing infrastructure was ready in minutes. Can be said that the extraction of the grammar from the RFCs was the most difficult part.
Some errors were found and identified. As usual, errors with special characters, semantic errors (error with empty quoted strings, see section 7.2), and even an error in the ABNF formulation from the RFC were discovered (a problem with the precedence of operators. Brackets were included to solve that).

### 10.1.3 jSDP

jSDP is a Java library that enable users to manipulate SDP messages[19]. SDP, Session Description Protocol, is specified on the RFC4566. SDP provides a standard representation to convey media details, transport addresses, and other session description metadata to the participants of voIp calls, streaming videos or video-conferences.

[18] http://james.apache.org/
[19] http://jsdp.sourceforge.net/

The library provides one class for each kind of line in the message. The parsers are accessible as public methods from those classes. The generator cross references were avoided, due to the fact that the structures are simply stored and not parsed. The Java server implementation was really easy and it was ready in minutes.
Several errors were found. The parsers were based in a matcher structure, not very flexible, and they failed with non standard characters.

## 11 Discussion and Further Work

Despite the fact that the designed technique was successful, the underlaying code, above all the part related with tree management, works outside the functions QuickCheck provides. It makes functions to work in a tricky way to be able to use QuickCheck abilities like shrinking. As a result, the properties are much complexer, being necessary three nested ?FORALL's, and the final tree sometimes is not the exact representation of the shrunk string, due to the double shrinking. It will make the tester to prune by hand some of the subtrees in some cases. The way QuickCheck works makes also the screen output quite obscure, and the unnecessary information appearing on the screen could confuse the tester in the first approach.
One of the program major restrictions of this work is its limitation to ABNF grammars. ABNF is a standard used by the IETF to use as metalanguage to describe protocols. However, sometimes old protocols were released with its own syntax representation, even for protocols released by IETF. Others, like Jabber for instance, are based on XML schemas. This is not a limitation of the technique itself, but a extension to cover other grammar representations could be interesting. It would be generalizable for all kind of parsers, not only for protocols, and potentially for every type of grammar based software. The only requirements are the input generation based on a grammar, and the necessity of the generation tree to understand the output.
Another possible extension point is the inclusion of other programming languages. Now the technique can be used with Java, and of course, natively with Erlang. The IC compiler works with C as well, so an extension for this language would be quite easy. Other languages could be used communicating through sockets. It would need bigger changes in order to manage the connexions in the Erlang node.

## 12 Conclusion

As a conclusion, can be said that the described technique gave promising results working in a real world scenario, finding obscure errors in a widely used reference implementation. Future works in this area based on this work could have both theoretical and industrial success in the world of grammar based software.

## 13 Acknowledgments

I wish to thank Göteborgs Universitet and Universidade de A Coruña for giving me the chance to carry out my thesis project here. This year was an unforgettable experience and a very valuable time for my academic career.

Special thanks to my supervisor, Thomas Arts, and his patient reviews, ideas and comments. This work would not be possible without him. Thanks Thomas! Also thanks to Agneta Nilson for her orientating comments during the Thesis Preparation Course.
I also would like to thank my friend Amanda Amner for her effort in helping me to correct this report.

## References

[AHJW06] 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 Press.

[BP94] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA, 1994. ACM Press.

[FB97] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[HP05] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 104–113, New York, NY, USA, 2005. ACM Press.

[MP94] Milena Mihail and Christos H. Papadimitriou. On the random walk method for protocol testing. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 132–141, London, UK, 1994. Springer-Verlag.

[SA05] Hans Svensson and Thomas Arts. A new leader election implementation. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 35–39, New York, NY, USA, 2005. ACM Press.