CHALMERS

Using QuickCheck to verify Erlang implementation of GTPv2

Master of Science Thesis

JOHAN EMILSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science & Engineering
Gothenburg, Sweden, July 2011

Using QuickCheck to verify Erlang implementation of GTPv2

JOHAN EMILSSON

Examiner: JOHN HUGHES

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

Abstract

In this master thesis project, Quviq QuickCheck is used in order to perform unit testing of a stateless subsystem of Ericsson's MME, which is a central node in the 4G network. During the project, errors that have not been found in previous testing are discovered, QuickCheck test code is found to be cheap to maintain, and the specific environment is found to be ideal for QuickCheck testing.

Acknowledgments

I would like to thank my examiner and supervisor of this master thesis, John Hughes, for his expertise and the advice during the project. I would also like to thank my Ericsson supervisors, Jonas Hadin and Martin Nylin, for their help in guiding me through the Ericsson infrastructure as well as the system that has been tested. I would like to show my appreciation to Ericsson managers, Anne-Lie Börjesson and Christer Nilsson, for allowing me to perform this project and consume the time of their team members. I would also like to give a general thank you to everyone who has, in one way or another, participated in the project. Finally, a big thank you to my friends and family for all the love and support.

Nomenclature

GPRS	**General Packet Radio Service** A platform for mobile networking services.
GTPv2	**Second version of the GPRS Tunneling Protocol** The communication protocol that is used between SGWs and MMEs in the LTE network.
GTS	**GTPv2 Translation Subsystem** A subsystem of the MME that is responsible for GTPv2 communication.
IE	**Information Element** The building blocks of which GTPv2 messages consist.
LTE	**3GPP Long Term Evolution** The fourth generation of mobile networks, commonly referred to as 4G.
MME	**Mobility Management Entity** A central node in the LTE network.
PBT	**Property Based Testing** A software testing method in which the expected behavior of a software system is defined by properties.
SBT	**Scenario Based Testing** A software testing method in which the expected behavior of a software system is ensured by execution of one or more scenario(s).
SGW	**Serving Gateway** A node in the LTE network.
UE	**User Equipment** Any mobile equipment that is used in a mobile network. Usually a mobile phone.

Contents

1 Introduction

This master thesis project has been performed on behalf of Ericsson Lindholmen in Gothenburg, Sweden. One of the major products that is being developed at these facilities is the *Mobility Management Entity* (MME), which is a central node in the *3GPP Long Term Evolution* (LTE, commonly referred to as 4G) network. During the course of this project, *property based testing* (PBT) is performed on one of the subsystems of the MME.

PBT is a concept that is related to the concepts of *specification and model based testing.* Test methods based on these concepts (such as QuickCheck, see section 3) are widely used in the academic community, and several studies have suggested that QuickCheck offers a number of benefits compared to conventional testing when used in the software development industry [2, 3, 4, 5, 6].

This section describes the purpose, scope and limitations of the work performed in the thesis.

1.1 Purpose

The purpose of this thesis is to study the potential benefits and drawbacks of using *QuickCheck* for unit testing on a software module that implements a data communication protocol (see section 4). The commissioner is also interested in gaining an understanding of how QuickCheck would work in their organization, in what situations the use of this tool would be beneficial and where/how QuickCheck could be efficiently implemented in the current development organization.

1.2 Scope

The scope of this thesis contains a number of deliverable items that together will constitute a foundation on which an analysis regarding the matter at hand can rest.

- Implementation
  - QuickCheck unit testing of the protocol module described in section 4.
  - QuickCheck unit testing of a legacy version of the same module.
- Analysis

  - Time consumption needed in order to accomplish QuickCheck unit testing that gives the same (or higher) degree of confidence regarding the correctness of the code, compared to the currently used SBT method.
  - Complexity of the QuickCheck implementation in terms of the amount of learning that is needed in order perform QuickCheck unit testing as described above.
  - Importance of detected defects (if any), compared to defects discovered by the currently used SBT method.
  - The thesis will also contain an analysis regarding how well suited QuickCheck is for use in the environment at hand. It is possible that QuickCheck is better suited for testing in other environments and/or for different purposes than the unit testing that will be implemented during the course of this thesis. This part of the analysis will involve topics such as what level of support the use of QuickCheck offers the developers, besides from finding bugs, compared to the currently used SBT method.

1.3 Limitations

The practical work performed in this thesis is limited to a stand alone implementation of QuickCheck unit testing of the protocol module described in section 4. The implementation is not to be integrated with existing test environment.

The theoretical work will be focused on the organization surrounding the development of the relevant subsystem. The results and conclusions of the thesis will be applicable to development organizations similar in size and environment. The thesis is not aiming at reaching conclusions about the use of QuickCheck in a wider perspective.

2 Software quality and testing

Correctness[1] and stability[2] are important measures of quality for any software developer. For some software, such as for safety- and performance critical systems, correctness and stability might even be considered as the most important measures of quality.

In order to achieve software that meets high quality demands with respect to these measures of quality, developers test the software they produce in order to discover failures, errors and faults. Software testing is generally regarded as an expensive and demanding activity, and usually consumes a large portion of the total budget of any software project. It is therefore relevant and interesting to perform research in this topic and to evaluate alternative approaches of software testing.

Conventionally, testing in software projects is performed at a number of different levels of abstraction. This thesis is focused on a unit testing level, where a unit is the smallest testable object in a software system (usually a function). This means that programmers write tests that ensures the correctness of the code they produce.

2.1 Scenario based testing

The most common way of performing unit testing is by an approach that in this report will be referred to as *scenario based testing* (SBT). This approach includes an initialization of test data, an execution of the software that is to be tested, and a statement of what the result of the execution is expected to be. In other words, the programmer creates scenarios (also known as test cases) that are used in order to ensure the behavior of the software. Consider a function `date_to_day` that is supposed to calculate the day number from a date, such that if the function is applied on the date the first of January, the result should be the day number 1. Similarly, applying the function to the last of December would result in the day number 365. For simplicity, assume that the function does not take in consideration whether the current year is a leap year or not.

```
test_date_to_day() ->
  Date = {1,1},
  DayNumber = date_to_day(Date),
  DayNumber == 1.
```

[1]Correct software is software that behaves according to a predefined specification.
[2]Stable software is software that does not crash unexpectedly.

In the first line, some test data is set up. In the second line, the call to the function that is being tested is made and the results are stored. The last line contains an assertion that controls that the results are as expected. In SBT, tests like these are commonly grouped together in so called test suites that are used to ensure the behavior of some software using several scenarios. In this case, it would probably be a good idea to also test the function with the last date of the year and with one or more dates in between. Please note that the example above is written as an Erlang function that results in a boolean value. The example is not meant to constitute an actual example of some SBT tool, but rather to describe the basic principles behind SBT.

SBT is generally accepted as a good practice and it comes with a number of benefits besides finding errors in the code. For instance, a test case can serve as design specification for the function, and if test cases are set up to be executed automatically, SBT ensures that change and refactoring does not compromise the integrity of the software. There are however drawbacks with SBT. In order to cover all execution paths in a function, the programmer will have to write a large number of scenarios, something that is both tiresome and time consuming.

2.2 Property based testing

Another way of achieving unit testing is by a method called *property based testing* (PBT). This concept is related to *specification and model based testing* and is based on formal specification of software. In PBT, the desired behavior of the software is described with properties. A property is, in other words, a rule that describe how the software that is being tested is allowed to behave. Different PBT tools are likely to utilize and test properties differently, the fundamental idea behind PBT is however likely to be similar regardless of the tool that is being used. The primary idea of PBT is to use the properties that are defined in order to generate test cases, and to verify that the property holds for every test case. The advocates of PBT claims that this approach comes with many advantages compared to conventional SBT, for instance that it enables thorough testing with less effort[2]. In this project the PBT tool QuickCheck has been used (see section 3).

2.3 Negative testing

The SBT example in section 2.1 is a typical example of positive unit testing, a function is being applied to correct input. In general, positive testing means testing that is performed with non-faulty test data. The opposite is called negative testing, faulty data or faulty states are used in order to assure that the software that is being tested creates and handles errors correctly. In order to fully test the behavior of a system, both positive and negative testing is necessary. The concepts of positive and negative testing are separated from those of property and scenario based testing.

3 QuickCheck

QuickCheck is a PBT tool that was originally developed by Claessen and Hughes as a lightweight testing tool for programs written in Haskell[1]. Quviq AB has since developed a version of the tool for Erlang programs, called Quviq QuickCheck[2]. This section describes the idea behind QuickCheck testing, and shows how QuickCheck can be used in practice. Besides the concepts that are described in this section, QuickCheck also has support for testing stateful systems and C code. However, these aspects of QuickCheck have not been used during the course of this thesis and are thus left out of this section.

3.1 Properties

A property in QuickCheck is basically a logical statement that defines how the software that is being tested is expected to behave. Consider the example from section 2.1 that tests the function `date_to_day`. To ensure the behavior of this function using QuickCheck one needs to consider how to specify the behavior of a function that calculates the day number from a date. An oracle[3] could be used, but one could also specify the behavior manually. In this simple example it is not difficult to catch the behavior of the function using a simple property. Consider the following example, that also makes use of the inverse function of `date_to_day`, `day_to_date`, that calculates a date given a day number:

```
prop_day_number() ->
    ?FORALL(D, date()),
            D == day_to_date(date_to_day(D))).
```

This property states that every date `D`, that is generated by `date()` (see section 3.2), is equal to the result of applying both `date_to_day` and its inverse to `D`. QuickCheck will use the generator `date()` to generate random dates and check that the property holds for every generated test case. This kind of a property is, in this report, referred to as a round trip property. This kind of property might give a false security regarding the correctness of the functions, if they have a common misunderstanding of how the conversion between date and day numbers are supposed to work.

[3]An oracle, in this context, is an equivalent function that is known to be correct.

3.2 Generators

QuickCheck offers a number of built in generators, such as `int()` that generates a small random integer. QuickCheck also enables the user to combine the built in generators in order to generate more complex data. The generator `date()` might look something like this:

```
date() ->
    {choose(1,12), choose(1,31)}.
```

The built in generator `choose(N,M)` generates a random value that is inclusively bounded by `N` and `M`. This means that `date()` would generate a tuple of two elements where the first element is an integer between 1 and 12 (a month), and the second element is an integer between 1 and 31 (a day). This generator would generate correct dates in most cases, but there is also a possibility that it generates an incorrect date. The month of June for instance, can have at most 30 days. The tuple {6,31} would, in other words, be an incorrect date.

As described in section 2.3, both positive and negative testing is needed in order to fully determine the correctness of a software system. The recommended approach to achieve this in QuickCheck is to design generators that can produce both correct and incorrect test data, and to write a validating function for the data in order to be able to determine the correctness of the system, regardless of the correctness of the input data. One of the reason behind this recommendation is the simple fact that generators like these are less complex. If the generator `date()` was to generate only correct dates it would have to be more complex. However it is also desirable to be able to control the distribution of faulty data, therefore two simple date generators could be used; one that mostly generates correct dates, and one that mostly generates incorrect dates. These generators can be combined into a complete date generator that also gives the user control over fault distribution in the following way:

```
date() ->
    fault(bad_date(), good_date()).

bad_date() ->
    {int(), int()}.

good_date() ->
    {choose(1,12), choose(1,31)}.
```

The generator `fault()` is used to give the user control over fault distribution, `bad_date()` will mostly generate incorrect dates and `good_date()` will mostly generate good dates. The generator can now be used in the following way: `fault_rate(1,5,date())`, which tells QuickCheck that one in five dates should be generated using the faulty generator, which is the first of the two arguments of the generator `fault()`. This approach renders simple generator code, and gives the user a possibility to control fault distribution.

3.3 Validation

In order to determine that a function works as intended, there is a need to determine whether an error was caused by incorrect input and that there are no errors in input that does not result in errors. In order to achieve this, there is a need for validation. To determine whether a date is incorrect or not, the following validation function could be used:

```
valid_date({M,D}) when D >= 1 ->
    if
      lists:member(M, [1,3,5,7,8,10,12]) -> D =< 31;
      lists:member(M, [4,6,9,11])        -> D =< 30;
      M == 2                             -> D =< 28;
      true                               -> false
    end;
valid_date(_) ->
    false.
```

This function returns a boolean value, and states that a valid date has a day that is greater to or equal to one, and that if the month has 31 days - the day should be smaller than or equal to 31, and so on. Recall the property from section 3.1, this property now needs to be slightly updated in order fully capture the behavior of the function `date_to_day`:

```
prop_date() ->
   ?FORALL(Date, fault_rate(1,5,date()),
           case date_to_day(Date) of
              bad_date -> not valid_date(Date);
              Day      -> Date == day_to_date(Day) and
                              valid_date(Date)
           end).
```

In this property, it is assumed that the atom `bad_date` is returned by the function in case of a bad date. If that happens, the property states that the generated date should not be a valid date. That is, the validation function should return false. If no error is discovered, the property states that the same round trip property as in the previous version of the property should hold and that the generated date shall be valid. This property does not fully capture the behavior of the function `day_to_date`, as its error handling is not being utilized.

An execution of the property would, assuming that the functions it tests work as intended, looks like this:

```
1> eqc:quickcheck(prop_date()).
................................................................
..................................
OK, passed 100 tests
true
```

QuickCheck generates 100 test cases by default, the number of tests can easily be adjusted manually. If QuickCheck manages to find input that falsifies the property, this input will be displayed for the user. The input will also be simplified by a process that is called *shrinking*. In this process, numbers are reduced, lists are shortened and so on, in order to find the smallest possible case that causes the same error. The shrunken input is also displayed. The concept of shrinking is not widely used in this project, and will thus not be described in any more detail in this report.

3.4 Collect

In order to visually display the fault distribution of the generated data, QuickCheck has a built in property `collect(Data, prop())` that collects statistics of the generated data `Data`, and then runs the property `prop()`. Collect can be used in order to gather statistics of the data produced by the generator `date()`:

```
prop_collect() ->
   ?FORALL(Date, fault_rate(1,5,date()),
           collect(valid_date(Date), true)).
```

This property does not test anything, it just gathers statistics. The result would look something like this:

```
1> eqc:quickcheck(prop_collect()).
................................................................
..................................
```

```
OK, passed 100 tests
82% true
18% false
true
```

This output shows that about 20% of the dates that are generated are incorrect. The use of the property `collect(Data, prop())` during this project is not described in this report, however the full implementation can be seen in the appendix.

4 Environment

In order to provide the reader with an understanding of the context of the thesis work, this section describes the environment of the software that is tested.

The *3GPP Long Term Evolution* (LTE, also commonly referred to as 4G) network is a network that allows mobile devices to communicate and to use the internet in accordance with the 4G standards. This thesis project includes PBT of one of the subsystems of the *Mobility Management Entity* (MME) node.

4.1 Mobility Management Entity (MME)

The MME is a central node in the LTE network; it is connected to a large number of base stations and a number of other MMEs. It keeps track of the location of all *User Equipment* (UE, e.g. mobile phones) that is connected to any of the base stations. The MME is also responsible for setting up data communication bearers between the base stations and the *Serving Gateway* (SGW). A simple LTE network is depicted in figure 1, the MME uses the second version of the *GPRS Tunneling Protocol* (GTPv2) for communication with the SGW and other MMEs. The other protocols in the figure will not be relevant for this thesis.

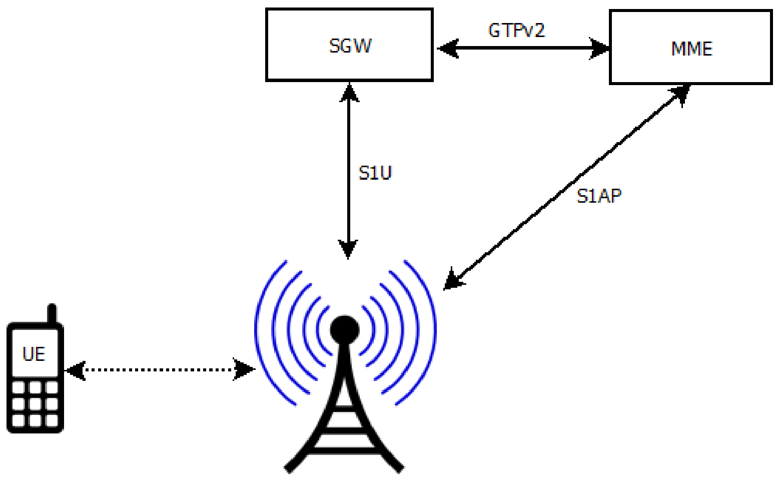

Figure 1: A simple sketch of the relevant parts of an LTE network.

4.2 GTPv2 Translation Subsystem (GTS)

The subsystem in the MME that has been tested in this thesis will in this report be referred to as the *GTPv2 Translation Subsystem* (GTS). This subsystem is more or less a protocol module that implements the protocol GTPv2 according to 3GPP's specifications[7]. The GTS is responsible for processing, packing and unpacking data traffic that is being sent and/or received to/from the SGW and/or other MMEs. The subsystem receives binary GTPv2 messages (see section 4.3), validates the data and unpacks it into Erlang records[4] that are passed on to the receiving internal subsystem, see figure 2. The GTS is also capable of doing the reversed task; receiving Erlang records from internal subsystems of the MME, translating them into GTPv2 compatible data packets and sending the data to the receiver according to the GTPv2 standard, see figure 3.

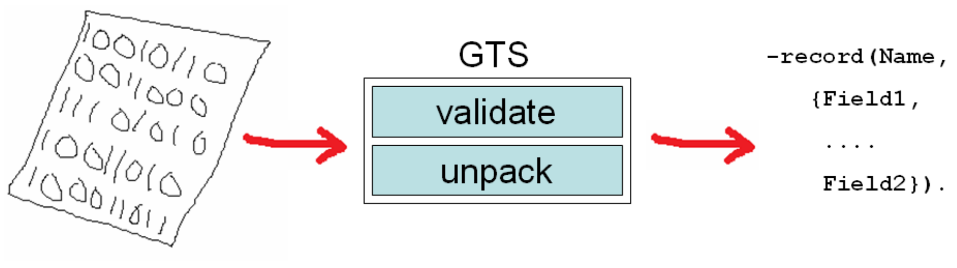

Figure 2: Binary GTPv2 messages are validated and unpacked into Erlang records by the GTS.

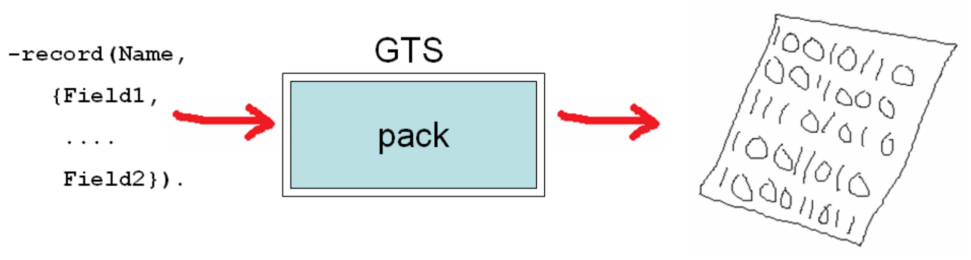

Figure 3: Erlang records are received by the GTS and gets packed into binary GTPv2 messages.

[4]A record is an Erlang construct that enables named access to data objects, similar to associative arrays in Python or structs in C.

4.3 GTPv2 - Messages and Information Elements

In order to give the reader of this report a chance to understand the details of the QuickCheck testing (section 6.3) that is performed in this thesis, this section contains a description of GTPv2 messages according to 3GPP's specifications[7].

A message consists of one or more *information elements* (IEs). The 3GPP specification[7] states which IEs can be contained in a specific message, which IEs are mandatory for a message and so on. An IE that is considered mandatory for a certain message needs to be included in the message in order for the message to be considered valid. A GTPv2 message is in other words a string of binary IEs. A typical IE is coded in the following way:

```
<<ID, Length:16, Spare:4, Instance:4, Data:Length/bytes>>
```

The above syntax is Erlang bit syntax[5], something that is frequently used in the QuickCheck testing that is performed in this project. The meaning of each field of the bit string is described in table 1.

Name	**Size**	**Comment**
ID	8 bits	A unique identifier, the type of the IE
Length	16 bits	The length of the Data field contained in the IE, this value is interpreted as bytes
Spare	4 bits	These bits are not used, nor evaluated. Commonly set to 0000
Instance	4 bits	If more than one IE of the same type is used in a message, the Instance value is used to tell them apart
Data	Length bytes	The actual data content of the IE

Table 1: The contents of an IE.

The specification also contains information regarding how the data contained in a specific IE should be coded. If the data is not coded according to the specification, the IE is considered invalid. IEs that are mandatory for a specific message needs to be valid in order for the message to be considered valid.

The full GTS implementation consists of validation, unpacking and packing of 33 GTPv2 messages. In these messages, there are a total of 41 IEs.

[5]Erlang bit syntax was built into Erlang in order to make it easier to handle binary data.

5 Method

In order to achieve the goals of this thesis a number of activities are performed.

5.1 Preparation and pre-studies

The current testing environment and the relevant development chain at large are studied in order to assess how the use of QuickCheck could be beneficial. The results of these studies are also useful for interpreting the results of the actual QuickCheck unit testing.

The thesis also contains studies of previous use of QuickCheck at Ericsson, in other companies and in academia, in order to gain an understanding of how QuickCheck has worked in similar environments, how QuickCheck is beneficially utilized, and how to avoid mishaps that might have been performed by others in the past.

5.2 Implementation and measurements

In order to study the benefits and drawbacks of QuickCheck unit testing, QuickCheck is used to perform unit testing of the GTS (see section 4.2). In order to attain valuable results from the testing, measurements of some sort need to be made.

The most straightforward measurement that is made is the time consumption that is needed in order to achieve a thorough unit testing with QuickCheck. This measurement is compared to an estimation of the time consumption needed in order to achieve similar results with the current unit testing techniques that are used in the relevant organization. The estimation is based on informal interviews with developers from the concerned organization.

The complexity of the test code is also an interesting measurement, however it is not a trivial thing to measure. In order to assess the amount of effort that is needed to learn how to write the properties and test data generators necessary in order to conduct QuickCheck unit testing, informal interviews with experienced Erlang developers from the concerned organization are conducted.

The thesis also contains an adaptation of the QuickCheck code for testing of a legacy version of the GTS. Measurements and estimations of the overall effort and in particular time consumption needed in order to achieve this adaptation are used in order to study the maintainability of QuickCheck test code, compared with the test code produced by the currently used SBT environment.

Whether there is a difference in criticality between the defects that are detected by the currently used SBT technique and those that are discovered by QuickCheck is yet another matter that is difficult to measure. The criticality of errors might be even harder to value than the previous measurements, especially at a unit testing level since there is no saying about the failure that a unit level error might cause in the system that the unit belongs to. However, an analysis of this issue is conducted by comparing the methods of the currently used SBT method and PBT with QuickCheck with respect to code coverage and other factors that are related to certainty of correctness.

As mentioned in section 2.1, SBT can be used for other purposes than finding errors. By conducting informal interviews with members of the concerned development organization, these purposes are determined. By simply using QuickCheck for unit testing in the affected environment, it is possible to determine whether QuickCheck testing offers the same support.

6 Results

This section describes the findings of the project, as well as the QuickCheck test code that has been produced.

6.1 Current development environment

The software developers and designers at the concerned department of Ericsson use a number of testing strategies and tools in order to ensure the quality of their products. Besides unit testing and code reviews, simulated and actual environments are used for system and integration testing of the systems.

This project is focused on a unit testing level, and uses the unit testing environment at the concerned department as a benchmark in order to put the results of the QuickCheck testing in context. The current unit testing consists of an SBT tool that is developed internally at Ericsson, the testing is divided into test suites that are set up to be run automatically when code is committed to the code base. The SBT tool also has support for displaying the code coverage of tests suites, and helps with cross system dependencies. Some of the developers that have, in one way or another, been involved in this project, say that the main purpose of the unit testing environment is to evaluate the design of different components rather than to find actual coding errors. The SBT tool however, seem to be used by developers mainly for the specific purpose of ensuring the correctness of the code.

The current unit testing environment seem to be appreciated for the support it gives to the developers, but there also seem to be some drawbacks. The majority of the developers that have participated in the pre-studies of this thesis say that the test code produced by the tool often becomes complex and hard to understand. They also have problems with having to spend large amounts of time in order to maintain the test code, due to the inherent tight coupling with the code that is being tested.

6.2 Previous use of QuickCheck

The benefits and drawbacks of QuickCheck testing have been studied on several occasions.

Arts et al.[2] used QuickCheck for testing of an Ericsson media proxy and were able to find several faults that had not been detected during previous testing of the proxy. They were also able to discover that the specification of the protocol that was used in the media proxy included several unclarities.

During a master thesis project at Linköping University, Granberg and Jernberg[4] preformed QuickCheck testing on Ericssons radio base stations that are used for mobile communications. They found that QuickCheck was able to find faults that conventional testing would not find, that QuickCheck code is easier to maintain and that QuickCheck testing requires slightly higher effort and has a higher learning threshold than conventional testing.

In a case study performed at Erlang Training and Consulting Ltd., Boberg[3] used QuickCheck for system testing of a message gateway product. He concluded that QuickCheck testing increased the number of faults that were discovered during system testing.

In more recent studies, Koral and Hoffmann[5] were able to conclude that QuickCheck testing requires more thinking and less work when they used the tool to test Motorola mobile network products. Crowe[6] used QuickCheck to test radio base station C code and stated that QuickCheck is highly appropriate for testing of concurrent systems.

These examples all give reasons to think that QuickCheck is a powerful testing tool, and that property based testing is a concept that might become a well used testing approach in the software industry.

6.3 Implementation

This section describes the test code that has been written in order to perform QuickCheck unit testing of the GTS. The test code has a modular design that consists of one module that generates and validates individual IEs, one module that validates and sanitizes entire messages, and one module per message that is being tested. As described in section 4.3, different messages consist of different combinations of IEs. Each message also has a different set of mandatory IEs. The modular design of the test code allows for reuse of generation and validation of IEs, as well as validation and sanitation of entire messages while still leaving room for the individual differences of the different messages. A diagram of the test code design is depicted in figure 4.

Validation is performed in order to evaluate error messages from the GTS. If the GTS returns an error message saying that a mandatory IE is missing or is incorrect in the generated message, the test code needs to verify that such an error is indeed present in the message. There is in other words a need to be able to validate whether a message and/or an IE is correct or incorrect.

Sanitation is performed in order to be able to check that round trip properties hold. The round trip properties that are being studied are those of a generated messages, and the result of applying both the unpack and pack functions of the GTS to that generated messages. When the GTS unpacks a message, unnecessary data is removed as well as incorrect IEs that are not

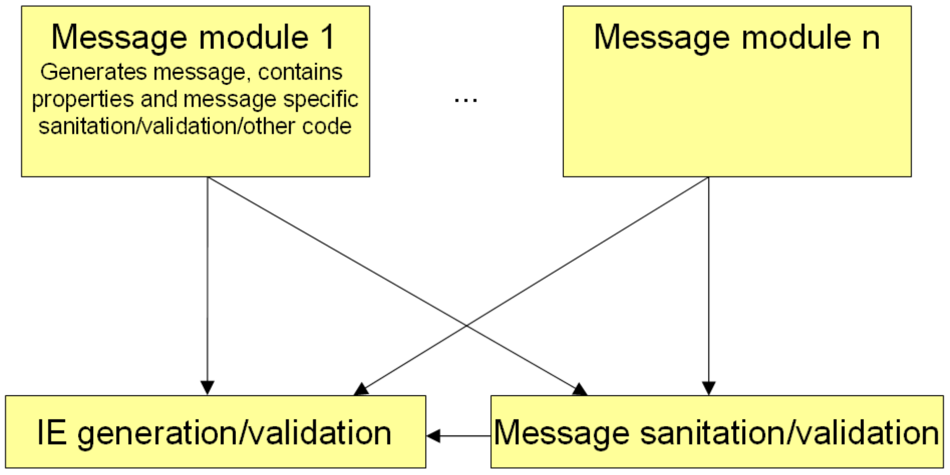
Message module 1
Generates message, contains properties and message specific sanitation/validation/other code
...
Message module n
IE generation/validation
Message sanitation/validation

Figure 4: The modular design of the test code.

mandatory (mandatory IEs that are incorrect would provoke an error). This sanitation needs to be performed by the test code in order to check the round time properties.

6.3.1 IE generation and validation

The module that generates and validates IEs is capable of generating correct and incorrect IEs of each type that can be present in any of the messages that are used for testing of the GTS in this project. The module is also capable of validating each of the IEs. The complexity of the IEs, both in terms of generation and validation, varies heavily. The following is an example of generation code for one of the more simple IEs:

```
ie_cid(Instance) ->
  fault(bad_ie(?IE_CHARGING_ID, Instance), good_cid(Instance)).

good_cid(Instance) ->
  ?LET(CID, binary(2),
       <<?IE_CHARGING_ID, 2:16, 0:4, Instance:4, CID/binary>>).
```

The ?LET macro is generally used when the data that is generated depends on previously generated data. In this case it is needed because Erlang does not allow function calls in bit syntaxes. The macro ?IE_CHARGING_ID represents the ID of the IE. The generation of IEs is performed in the same manner as in the example of section 3.2 when it comes to faulty data. The generator `bad_ie(Id, Instance)` is a generator that is used for faulty creation of all IEs in the module.

Validation of IEs also includes some sanitation, this is needed since there is some information in an IE that is not being evaluated or read by the GTS. Some IEs are allowed to be longer than they need to be. In this situation, data that comes after the needed data is discarded. The following is an example of simple validation of an IE that illustrates this matter:

```
valid_mei(<<?IE_MEI, _:16, Spare:4, Instance:4, Data:8/bytes,
            _Rest/binary>>) ->
    case valid_bcd(Data) of
        true  -> {ok, <<?IE_MEI, 8:16, Spare:4, Instance:4,
                        Data:8/bytes>>};
        false -> nok
    end;
valid_mei(_) ->
    nok.
```

What this validator states is basically that every IE that has the correct ID, that is long enough, and from which it is possible to get 8 bytes of valid data (according to the function `valid_bcd(Data)`) is a correct IE of this type. All other IEs are considered invalid. The function `valid_bcd(Data)` validates that binary data is encoded according to a specific notation that is frequently used in the GTPv2 protocol for the purpose of encoding decimal digits.

The validator also performs some sanitation, unnecessary data is discarded. In the case of a valid IE, a sanitized version of the IE is returned.

The way an IE is validated and generated has been determined by referencing the 3GPP specification[7] as well as some internal Ericsson documents that specify which parts of the relevant protocol that the products in questions comply to.

6.3.2 Message validation and sanitation

The module that validates and sanitizes messages uses the IE generation and validation module in order to validate and/or sanitate the individual IEs of the messages it validates and/or sanitizes.

Validation of messages is performed in order to evaluate whether or not an error that is returned from the GTS is correct. An error message from the GTS contains information about the error that has occurred, in particular it contains the ID of the causing IE and the type of error that has occurred.

This is all the information that is needed in order to determine the validity of the error. For example, one type of error occurs when a mandatory IE is not included in a message. This error is validated by checking that the IE is indeed mandatory for the message in question and that it is not included in the generated message.

Sanitation of messages is performed by using the validators in the IE generation and validation module to exclude incorrect IEs and unnecessary data from the message. The exact nature of how validation and sanitation is performed in this module has, as in section 6.3.1, been determined by referencing the 3GPP specification[7] and internal Ericsson documents.

6.3.3 Message modules

In order to test the way the GTS handles different GTPv2 messages a message module has been implemented for each message that is tested in this project. A message module contains information about which IEs that are mandatory in the message, for validation purposes. A message module also contains a generator that utilizes the IE generation and validation module in order to generate the IEs that constitute the specific message.

Naturally, the message modules also contain the round trip property that is used for the actual testing. These properties basically states that unless there is an error, a generated message that has been sanitized should be equal to the result of applying the pack and unpack functions of the GTS to the generated message. If there is an error, the error should be valid for the property to hold. The following code is a slightly simplified version of such a property:

```
prop_sym() ->
   ?FORALL({Message, MandatoryIEs}, gen(),
           try
             sanitate(Message) == gts:pack(gts:unpack(Message))
           catch
             throw:Response ->
                {CausingIE, ErrorID} = get_error(Response),
                validate_error(CausingIE, Error, Message,
                               MandatoryIEs)
           end).
```

The variables `Message` and `MandatoryIEs` are bound to the values generated by the generator `gen()`. `Message` is the binary message that has been generated, and `MandatoryIEs` is a list of the IDs of the IEs that are mandatory in the message. If an exception is thrown, the error message is parsed, and then the error is evaluated.

6.4 Time consumption

The time used for creating the test code used in this project has been measured throughout the course of the project. All of the work has been performed by one person.

The total time consumption of the programming is 128h. The test code tests the GTS' behavior in four different messages, which are listed in table 2. These messages requires 23 different IEs. The measurement of time consumption has been divided into the time needed in order to achieve generation and validation of IEs (shown as Generation in table 2), and the time needed in order to create properties, validation of errors, and sanitation of messages (shown as Properties in table 2). Time consumption has been measured with respect to the message the test code appertain to. The messages in table 2 are listed in order of implementation.

Message	**Generation**	**Properties**
Create Session Request	69h	32h
Create Bearer Request	3h	11h
Delete Bearer Request	0h	2h
Downlink Data Notification Failure Indication	0h	1h
Total	72h	56h

Table 2: Time consumption of the implementation of the QuickCheck testing that has been performed in this project.

As seen in table 2, the time consumption needed in order to implement testing for a message dramatically decreased after the implementation of the message Create Session Request. This is partly due to the fact that many messages contains the same type of IEs, and partly because Create Session Request contains a large number of IEs. The decrease in time consumption is also due to that all message modules follow the same pattern, and much of the code created for the first message could be reused. Naturally, one also gets an increased understanding of both the system that is being tested as well as of QuickCheck after some time.

The messages that have been selected for testing was chosen in consultation with supervisors at Ericsson.

In order to put this measurement in context, a number of developers of the concerned organization were asked to estimate the amount of time that would be needed in order to achieve unit testing with the currently used SBT tool that would fully cover the behavior of the GTS in terms of the messages that have been tested using QuickCheck. Three answers were recieved; one developer estimated the time consumption of this task to somewhere between 20h and 40h. Two other developers both estimated the time consumption to 160h.

6.5 Code complexity

Measuring code complexity is not trivial. In this report, a highly non-scientific attempt of measuring code complexity is made by showing the QuickCheck test code to experienced Erlang developers of the concerned organization. All developers that participated in this aspect of the thesis project found the test code to be fairly simple and straightforward. The concepts of QuickCheck testing are somewhat challenging due to the differences between PBT and the SBT technique that is currently being used by the organization. However, all developers that looked at the test code were able to understand how the concepts of generation, validation, sanitation and properties are used in order to achieve a thorough unit testing of the GTS. Some developers were even able to suggest improvements to the test code.

Another interesting angle regarding the complexity of QuickCheck test code is the fact that the author of this report, with very limited experience with Erlang and QuickCheck, and with no previous knowledge of the GTS or indeed any telecommunication software system, was able to implement QuickCheck unit testing for a significantly large part of the code of the GTS. This was done in a matter of months (including preparations, pre-studies etc.) and resulted in the discovery of errors that had not previously been found with the currently used SBT technique (see section 6.6).

6.6 Discovered errors

For obvious reasons, the errors that were discovered during the course of this thesis will not be covered in detail in this report.

As one might expect when it comes to testing of a relatively mature product that is currently being used in the LTE network, not a great deal of errors were discovered. However, at least two errors were found. The first of these errors could be considered a major flaw, it has to do with the way the GTS handles multiple instances of mandatory IEs in a message. The error can cause the GTS to treat messages that, according to the protocol specification[7] are correct, as incorrect. This error is present in each of the messages that have been tested during this project. The other errors consist of a few minor deviations from the specification of specific IEs.

6.7 Maintainability

In order to make an attempt of determining the maintainability of QuickCheck unit test code, an adaptation of the QuickCheck test code of this project was made for testing of a legacy version of the GTS. The legacy version of the system is from 2009, and is significantly different from the current version. The legacy version of GTS uses lists instead of Erlang bit syntax in order to handle and manipulate the binary data of GTPv2 messages. There are also some differences in the protocol specification of the different versions of the GTS.

The adaptation of the test code went surprisingly smoothly, mainly due to the fact that the QuickCheck test code is separated from the code of the GTS except from the two function calls that are performed in order to unpack and pack a binary GTPv2 message (calls that are made in each of the message modules, as described in section 6.3). The vast majority of the test code was not changed at all; the only changes that needed to be made was a to account for some minor differences in the protocols of the different versions of the system and the structure of the data that is passed on to the GTS was changed from bit syntaxes to lists. The concepts of generating binary messages, sanitizing messages and validating errors were possible to preserve in the testing of the legacy version of the GTS.

The total time consumption of the adaptation was measured to 12h.

7 Analysis

This section contains a discussion of the results found during this master thesis project, as well as reflections regarding the overall user experience of QuickCheck in the specific context of this project.

7.1 Effort

The high initial time consumption points towards a high initial cost of implementing QuickCheck unit testing. In order to get started with QuickCheck testing in this specific context, a large number of generators and validators of IEs have to be written. However, the total time consumption of the testing should be considered as reasonable. Once full QuickCheck testing of the first message has been implemented, the cost of creating test code for other messages drops dramatically. This is caused by the fact that many messages share the same IES, and the tests can reuse the generators and validators that already exist. It should also be taken into account that the use of any new tool will cause a high initial cost as it demands for the developers to get to know and understand the tool.

The QuickCheck unit testing performed in this project included four messages and 23 IEs. The total number of IEs that is handled by the GTS is 41. A reasonable estimation for a QuickCheck unit testing of the GTS is the double effort of the testing performed in this thesis, 256h. This estimation might be slightly excessive since the rate at which testing for new messages can be produced increases as more and more IEs are included in the code base.

As for the complexity of QuickCheck testing, no scientific conclusions can be reached. However, the results that were found at least give some indication that QuickCheck testing might be a relatively simple tool for unit testing. However, this does not mean that unit testing per se becomes easy with the use of QuickCheck. Thorough unit testing demands fully covering test data, which demands developers to write generators that supply them with all variations of data that is needed in order to cover all possible execution paths. In other words, you still need to use your brain in order to achieve good unit testing, even with QuickCheck.

In terms of maintainability, QuickCheck seems to have a very low cost. A likely cause of this is the fact that QuickCheck test code is largely independent of the code that is being tested. The developers that work within the concerned organization have reported that the test code of the currently used SBT tool requires large efforts in maintenance. This cost seems to be significantly reduced with the use of QuickCheck. However, it should be noted that the QuickCheck testing of this project only uses one property per message in order to ensure the behavior of the GTS. If this number is bigger, it is possible that the maintenance cost will increase. No further analysis of this matter is performed in the report. Theoretically, it should be possible to use the discovered errors of the legacy version of the system to perform some sort of analysis of the efficiency of the QuickCheck test code. Unfortunately this is not performed in this report, due to time constraints.

7.2 Efficiency

As mentioned in section 6.4, an attempt of estimating the time consumption needed to perform equally thorough unit testing with the currently used SBT tool was performed in this project. Unfortunately, not enough answers were received in order to be able to reach any scientific conclusions. However, the estimations that were received at least gives some reason to think that the time consumption of the two different tools are of the same magnitude.

Code coverage might be used as a measure of the efficiency of test code, in the terms of how much code coverage is produced by some unit of effort. However, in terms of QuickCheck testing, the measurement of code coverage is only useful for determining whether or not the generators produce the data that is needed in order to fully cover all execution paths in the software that is being tested (under the assumption that the code that is being tested does not contain unreachable code). In other words, even though the QuickCheck unit testing would have a similar code coverage as that of the currently used SBT tool, it results in a considerably larger number of test cases being executed, which might imply a more thorough unit testing.

One might, however, argue that it is difficult to fully assure the expected behavior of the GTS by formulating properties such as the round trip properties that have been used in this project. The concept of round trip properties is useless in the case when there is a mutual misunderstanding of the specification in both the packing and unpacking functionality of IEs in the GTS, this case has to be considered as rather unlikely.

Out of the errors found during this project, one is of the type that is not likely to have been discovered by common testing techniques. The error causes a failure that only occurs when certain combinations of IEs are received by the GTS, combinations that were generated by the QuickCheck test code. Naturally, it is impossible to say if that a person never would come up with a test case that would constitute such a combination, but it is at least safe to say that no such combination is currently a part of the existing test suites. The other discovered error is likely to be caused by sloppy reading of the specifications, this error should have been discovered by any testing technique.

During this project, no indication that suggests that QuickCheck testing would be less capable than the currently used SBT tool in terms of finding coding errors has been found.

7.3 QuickCheck in the specific context

Not surprisingly, QuickCheck performs very well when used for testing functional, stateless software. The specific context has to be regarded as ideal for QuickCheck testing as it is stateless and purely functional. This does not imply that it is in any way impossible to perform QuickCheck testing in different environments, the available literature shows that this is not the case [2, 3, 4, 5, 6]. But the environment in which this thesis project is performed enables the QuickCheck test code to be almost as simple as the basic examples that are contained in this report in order to explain the concepts of QuickCheck testing, see section 3.

As described in secion 6.1, the currently used SBT tool offers some support for the developers besides actual execution of scenario based test cases. This support includes such things as visualization of code coverage. This kind of support is however not an inherent capacity of the testing technique per say, but a result of the shell that has been built around the technique in order to form the actual tool. The code coverage part of the SBT tool is actually a utilization of the code coverage functionality that is part of Erlang. QuickCheck is a lightweight testing tool, and has as such not any support for visualizing code coverage, for instance. QuickCheck does however have the capability to display statistics of the test data that is being generated, by the use of the property `collect` (that has not been discussed in detail this report). In other words, this kind of comparison between the different tools is rather deceptive, since the tools are built for different purposes. With that said, there is no denying that the currently used SBT tool contains more functionality. One could argue that it is possible to overcome this matter by integrating QuickCheck and the currently used SBT tool, or by building a similar shell around QuickCheck.

During this project, QuickCheck tests have been executed from command line. No kind of integration with the existing test environment of the concerned organization has been made. If QuickCheck is to be successfully used for testing in the organization, one needs to make the developers embrace the tool. This requires both that developers gets a chance to realize the advantages of property based testing, for instance by education and time to play around a bit with the tool. It also requires that QuickCheck is easy to use, both in terms of integration with the existing environment, and in terms of general usage. During the project, Quviq QuickCheck's procedures for verifying licenses have caused some minor difficulties. These problems are easy to overcome by some manual hacking, but if QuickCheck is to be used in some kind of integration with the existing development environment, problems like these needs to be permanently solved. Reoccurring issues like these are likely to make developers dislike the tool.

QuickCheck does not need to be fully integrated into the testing environment to work successfully. However, for a successful implementation of QuickCheck testing, the organization need to make sure that the use of QuickCheck does not imply additional work for the developers, that developers somehow can create QuickCheck test suites and that the QuickCheck tests are run automatically, just as the test suites of the currently used SBT tools are. There is no indication that these matters should pose any technical difficulties, however they do require effort and determination.

8 Conclusions

The major benefits of PBT in general, and QuickCheck in particular include the ability to quickly generate large amounts of test cases. The major drawbacks of these concepts include the difficulties of writing test data generators, and formulating properties that fully assure the wanted behavior of some software. In the case of this project, these difficulties have been somewhat experienced. In order to assure the correctness of software, one needs to carefully think about the test data that is used as input. This is true for all testing techniques, including QuickCheck.

In terms of benefits, the results of this project can be used to conclude that the random test data input of QuickCheck makes it possible to find errors that is not being found by the currently used testing tools of the concerned organization. The results of the project can also be used to argue that the environment of this study is ideal for QuickCheck testing, that the effort needed in order to achieve unit testing with QuickCheck seem to be of the same magnitude as that of the currently used SBT technique, and that the QuickCheck test code imply lower maintenance cost than that of the code produced by the currently used unit testing tool.

The project also shows no indications that implementation of full QuickCheck testing of the GTS should pose any technical or organizational difficulties, assuming that the implementation is carefully planned and executed.

One can also conclude that the currently used SBT tool and Quviq QuickCheck are tools that have been built for different purposes. QuickCheck is a rather lightweight testing tool, however highly capable of finding errors, and the currently used SBT tool is a larger and heavier tool with a graphical interface and support for code coverage visualization. Comparing the the two tools as equals is thus rather misleading.

To summarize, QuickCheck seems to be a highly appropriate and capable tool for unit testing of the relevant software. With some effort, it would be possible to use QuickCheck in such a way that its benefits clearly outweigh its drawbacks.

9 Future work

Recommended future work for the concerned organization at Ericsson include such things as extending the existing QuickCheck test code in order to fully cover the GTS, this would imply more reliable data that can be used in order to assess the efficiency of QuickCheck testing on a larger scale. The product of this project can, besides being used for thorough unit testing of the GTS, also be used to perform system testing of the MME. The generators and validators of the existing test code can be utilized in order to generate large amounts of GTPv2 traffic that can be used to trigger execution of all software in the MME that makes use of the GTPv2 protocol. Even though it is hardly mentioned in this report, QuickCheck contains a state machine for testing of stateful systems, and an interface for testing of C code. One could argue that these concepts need to be used in order to evaluate the full potential of QuickCheck testing.

In terms of academia, it would certainly be interesting to see more studies of the use of QuickCheck and PBT in general. It would be particularly interesting to see a successful comparison of PBT using QuickCheck and traditional SBT. It is the firm opinion of the author of this report that PBT, or possibly another concept related to model or specification based testing, is the wave of the future in software testing. However, a reliable comparison of the two different approaches is difficult to achieve. It would require large efforts in terms of finding suitable projects and developers. To carry out such a study in a commercial organization is probably difficult, assuming that the two approaches are not proven to be equally efficient, due to the nature of commercial businesses.

References

[1] Koen Claessen and John Hughes, *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. ICFP '00, Montreal, Canada. ACM, 2000.

[2] Thomas Arts, John Hughes, Joakim Johansson and Ulf Wiger, *Testing Telecoms Software with Quviq QuickCheck*. Erlang '06, Portland, USA. ACM, 2006.

[3] Jonas Boberg, *Early Fault Detection with Model-Based Testing*. Proceedings of the 2008 SIGPLAN workshop on ERLANG. ACM, 2008.

[4] Andreas Granberg and Daniel Jerberg, *NBAP message construction using QuickCheck*. Master thesis report. Linköpings University, 2007.

[5] Raghav Karol and Torben Hoffmann, *Mission Critical with Erlang and QuickCheck - Quality Never Sleeps*. Talk given at Erlang User Conference 2010, Stockholm. Erlang Solutions Ltd, 2010.

[6] Graham Crowe, *Using Erlang for Testing non-Erlang Products*. Talk given at Erlang User Conference 2011, San Fransisco. Erlang Solutions Ltd, 2011.

[7] 3rd Generation Partnership Program, *3GPP TS 29.274 V9.4.0*. 3GPP, 2010.

A Source code

In this section, the source code is displayed. The code has been somewhat altered in order to protect the implementation details of the actual GTS. In some cases, removed code is denoted as `[---]`. The concept of Bearer Contexts have not been discussed in the report but can be seen in the code frequently, a Bearer Context is an IE that consists of one or more IEs. A Bearer Context, as a message, can have mandatory IEs.

A.1 IE generation and validation

This module has been stripped of most code. A selection of generation and validation of some IEs can be seen below, the selection describes the logic of the different kinds of IEs that are part of the test code of this project.

```
-module(eqc_ie_gens).
-export([ie_imsi/1, ie_ebi/1, ie_mei/1, ie_paa/1,
         ie_serving_network/1, valid_ie/1, valid_b_context/2]).
-include_lib("eqc/include/eqc.hrl").

%%%%%%%%%%  BAD_IE  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This generator is used for generation of faulty IEs of all sorts.
% The frequency of the different types of fault that can occur can
%  be discussed.

bad_ie(Id, Instance) ->
    frequency([{5, bad_ie_data(Id, Instance)},
               {5, <<>>},
               {2, bad_ie_length(Id, Instance)}]).
bad_ie_data(Id, Instance) ->
    ?LET(Length, choose(1,30),
        ?LET(Data, binary(Length),
            <<Id, Length:16, 0:4, Instance:4, Data/binary>>)).
bad_ie_length(Id, Instance) ->
    ?LET({Length, Data}, {choose(1,30), binary()},
         <<Id, Length:16, 0:4, Instance:4, Data/binary>>).

%%%%%%%%%%  IE_IMSI  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ie_imsi(Instance) ->
    fault(bad_ie(?IE_v2_IMSI, Instance), good_imsi(Instance)).
```

```
good_imsi(Instance) ->
    ?LET(NoOfDigits, choose(6,15),
        ?LET({Length, Data}, get_bcd(NoOfDigits),
            <<1, Length:16, 0:4, Instance:4, Data/binary>>)).

valid_imsi(<<Length:16, Spare:4, Instance:4, Data:Length/bytes,
 _Rest/binary>>) when Length >= 3, Length =< 8 ->
    case valid_bcd(Data) of
        true -> {ok, <<?IE_v2_IMSI, Length:16, Spare:4,
                Instance:4, Data:Length/bytes>>};
        false -> nok
    end;
valid_imsi(_) ->
    nok.

%%%%%%%%%  IE_EBI  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ie_ebi(Instance) ->
    fault(bad_ie(?IE_v2_EBI, Instance), good_ebi(Instance)).

good_ebi(Instance) ->
    ?LET(Data, bin_digit(4, 5, 15),
         <<?IE_v2_EBI, 1:16, 0:4, Instance:4, 0:4,
   Data/bitstring>>).

% Value not checked at unpack.
valid_ebi(<<_:16, Spare:4, Instance:4, 0:4, Data:4,
            _Rest/binary>>) ->
    {ok, <<?IE_v2_EBI, 1:16, Spare:4, Instance:4, 0:4, Data:4>>};
valid_ebi(_) ->
    nok.

%%%%%%%%%  IE_MEI  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ie_mei(Instance) ->
    fault(bad_ie(?IE_v2_MEI, Instance), good_mei(Instance)).

good_mei(Instance) ->
    ?LET(Mode, oneof([imei,imeisv]),
         ?LET(Data, get_mei(Mode),
              <<?IE_v2_MEI, 8:16, 0:4, Instance:4, Data/binary>>)).

get_mei(imei) ->
    ?LET({_,Data}, get_bcd(14), <<Data/binary, 16#F0>>);
get_mei(imeisv) ->
    ?LET({_,Data}, get_bcd(16), <<Data/binary>>).

valid_mei(<<_:16, Spare:4, Instance:4, Data:8/bytes,
            _Rest/binary>>) ->
    case valid_bcd(Data) of
        true  -> {ok, <<?IE_v2_MEI, 8:16, Spare:4, Instance:4,
                Data:8/bytes>>};
        false -> nok
    end;
valid_mei(_) ->
    nok.


%%%%%%%%%  IE_PAA  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ie_paa(Instance) ->
    fault(bad_ie(?IE_v2_PAA, Instance), good_paa(Instance)).

good_paa(Instance) ->
    ?LET(IpType, choose(1,3),
         ?LET({Length, Data}, get_paa(IpType),
              <<?IE_v2_PAA, Length:16, 0:4, Instance:4, 0:5,
    Data/bitstring>>)).

% IPv4
get_paa(1) ->
    ?LET(IPv4, binary(4),
         {5, <<1:3, IPv4/binary>>});
% IPv6 (including prefix length)
get_paa(2) ->
    ?LET(IPv6, binary(16),
         {18, <<2:3, 64:8, IPv6/binary>>});
% Both IPv6 (including prefix length) and IPv4
get_paa(3) ->
    ?LET({IPv6, IPv4}, {binary(16), binary(4)},
         {22, <<3:3, 64:8, IPv6/binary, IPv4/binary>>}).
```

```
valid_paa(<<5:16, Spare:4, Instance:4, _:5, 1:3, IPv4:4/bytes>>) ->
    {ok, <<?IE_v2_PAA, 5:16, Spare:4, Instance:4, 0:5, 1:3, IPv4:4/bytes>>};
valid_paa(<<18:16, Spare:4, Instance:4, _:5, 2:3, _:8, IPv6:16/bytes>>) ->
    {ok, <<?IE_v2_PAA, 18:16, Spare:4, Instance:4, 0:5, 2:3, 64:8,
           IPv6:16/bytes>>};
valid_paa(<<22:16, Spare:4, Instance:4, _:5, 3:3, _:8, IPv6:16/bytes,
            IPv4:4/bytes>>) ->
    {ok, <<?IE_v2_PAA, 22:16, Spare:4, Instance:4, 0:5, 3:3, 64:8,
           IPv6:16/bytes, IPv4:4/bytes>>};
valid_paa(_) ->
    nok.

%%%%%%%%%%  IE_SERVING_NETWORK  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ie_serving_network(Instance) ->
    fault(bad_ie(?IE_v2_SERVING_NETWORK, Instance),
      good_serving_network(Instance)).

good_serving_network(Instance) ->
    ?LET(Data, get_mcc_mcn_digits(),
         <<?IE_v2_SERVING_NETWORK, 3:16, 0:4, Instance:4,
   Data/binary>>).

valid_serving_network(<<_:16, Spare:4, Instance:4, Data:3/bytes,
                        _Rest/binary>>) ->
    {ok, <<?IE_v2_SERVING_NETWORK, 3:16, Spare:4, Instance:4,
       Data:3/bytes>>};
valid_serving_network(_) ->
    nok.

%%%%%%%%%%  IE_B_CONTEXT  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This IE is composed of a number of other IEs, the second argument
%  of this generator is supposed to be the list of generators for
%  the IEs that you want in your bearer context.

ie_b_context(Instance, IEs) ->
    fault(bad_ie(?IE_v2_BEARER_CONTEXT, Instance),
          good_b_context(Instance, IEs)).
good_b_context(Instance, IEs) ->
    ?LET({Length, Data}, get_b_context(IEs),
         <<?IE_v2_BEARER_CONTEXT, Length:16, 0:4, Instance:4, Data/binary>>).

get_b_context(IEs) ->
    ?LET(Data, get_bctx_elems(IEs, []),
         {size(Data), Data}).

get_bctx_elems([], Acc) ->
    list_to_bitstring(lists:reverse(Acc));
get_bctx_elems([IeGen | Tail], Acc) ->
    ?LET(Data, IeGen, get_bctx_elems(Tail, [Data | Acc])).

valid_b_context(<<?IE_v2_BEARER_CONTEXT, Length:16, _:4, _Instance:4,
                  IEs:Length/bytes>>, Mandatory) ->
    valid_b_context_content(<<IEs:Length/bytes>>, Mandatory);
valid_b_context(_,_) ->
    nok.

% Validates that mandatory IEs are valid and that all mandatory
%  IEs are present in the bearer context. Unlike the other
%  valid_<name> functions, this does not sanitate data.
valid_b_context_content(_,[]) ->
    ok;
valid_b_context_content(<<>>,_) ->
    nok;
valid_b_context_content(<<ID, Length:16, _:4, Instance:4,
                          Data:Length/bytes, Next/binary>>, Mandatory) ->
    case (lists:member(ID, Mandatory)) and (Instance == 0) of
        true ->
            case valid_ie(<<ID, Length:16, 0:4, Instance:4,
                            Data:Length/bytes>>) of
                {ok, _} ->
                    valid_b_context_content(<<Next/binary>>,
                                            lists:delete(ID, Mandatory));
                nok     ->
                    nok
            end;
        false ->
            valid_b_context_content(<<Next/binary>>, Mandatory)
    end;
```

```
valid_b_context_content(_,_) ->
    nok.

%%%%%%%%%  HELPERS_gen  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generates a bit string of length Length representing a decimal
%  value between Low and High (inclusive).
bin_digit(Length, Low, High) ->
    ?LET(D, choose(Low, High), <<D:Length>>).

% Generates a bit string consisting of NoOfDigits*4 bits where each
%  sequence of 4 bits represents a decimal digit (0-9). If NoOfDigits
%  is an odd number, a 16#F is inserted before the last digit.
get_bcd(NoOfDigits) when NoOfDigits rem 2 == 0 ->
    ?LET(V, vector(NoOfDigits, bin_digit(4,0,9)),
         {length(V) div 2, list_to_bitstring(V)});
get_bcd(NoOfDigits) ->
    ?LET(V, vector(NoOfDigits, bin_digit(4,0,9)),
         {length(V) div 2 + 1, set_odd_end(NoOfDigits,
                                          list_to_bitstring(V))}).
% Inserts a 4 bit 16#F before the last 4 bits in the bit string B
%  that consists of Size*4 bits.
set_odd_end(Size, B) ->
    Size2 = Size*4-4,
    <<Start:Size2, End:4>> = B,
    <<Start:Size2, 16#F:4, End:4>>.

% Generates a binary consisting of 3 bytes representing six decimal
%  digits (0-9) to be used as mcc and mcn numbers.
get_mcc_mcn_digits() ->
    ?LET(V, vector(6, bin_digit(4, 0, 9)), list_to_bitstring(V)).

%%%%%%%%%  HELPERS_valid  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Uses the correct validator to validate an IE.
% If you want to validate a bearer context,
%  use valid_b_context(IE, BContexts).
valid_ie(<<?IE_v2_IMSI, Rest/binary>>) ->
    valid_imsi(<<Rest/binary>>);
valid_ie(<<?IE_v2_EBI, Rest/binary>>) ->
     valid_ebi(<<Rest/binary>>);
```

```
valid_ie(<<?IE_v2_MEI, Rest/binary>>) ->
     valid_mei(<<Rest/binary>>);
valid_ie(<<?IE_v2_PAA, Rest/binary>>) ->
     valid_paa(<<Rest/binary>>);
valid_ie(<<?IE_v2_SERVING_NETWORK, Rest/binary>>) ->
    valid_serving_network(<<Rest/binary>>).

% Validates a bcd string.
valid_bcd(<<16#F:4, D2:4>>) when D2 =< 9 ->
    true;
valid_bcd(<<D1:4, D2:4>>) when D1 =< 9, D2 =< 9 ->
    true;
valid_bcd(<<D1:4, D2:4, Rest/binary>>) when D1 =< 9, D2 =< 9 ->
    valid_bcd(Rest);
valid_bcd(_) ->
    false.

% Validates a sequence of mcc and mcn digits.
valid_mcc_mcn_digits(<<>>) ->
    true;
valid_mcc_mcn_digits(<<Data:4, Rest/bitstring>>)
  when Data >= 0, Data =< 9 ->
    valid_mcc_mcn_digits(Rest);
valid_mcc_mcn_digits(_) ->
    false.
```

A.2 Message validation and sanitation

Some of the code of this module has been removed. However most functions are still present in order to give the reader a chance to understand how the code is used.

```
-module(eqc_common).
-export([packed_to_list/1,
          sanitate_message/2,
          get_error/1,
          validate_error/5,
          get_ie/3]).

% Module comments:
% This module contains some of the functions that are used
%  by the properties of each message. It is mainly functions
%  for validation of errors and sanitation of messages.

% Converts a packed binary GTPv2 message into a sorted list of
%  binary IEs.
packed_to_list(Packed) ->
    [---].

% Removes faulty IEs from a binary GTP message, returns a
%  sorted list of binary IEs.
sanitate_message(Message, BContexts) ->
    [---].

% Extracts relevant information from an exception that is
%  thrown when a message contains an invalid mandatory IE, or
%  when a message lacks a mandatory IE. Returns a tuple with
%  the integer values {Error, CauseIE}.
get_error([---]) ->
    case PayLoad of
        [---] ->
            {Error, CauseIE};
        [---] ->
            {Error, CauseIE}
    end.

% Validates an error. Returns true or false, true if the error
%  is actually found in the message and false otherwise.
validate_error(?MANDATORY_IE_MISSING, CauseIE, Message,
               MandatoryIEs, _) ->
    % CauseIE is mandatory and not in the message.
    lists:member(CauseIE, MandatoryIEs) and
        case get_ie(CauseIE, 0, list_to_binary(Message)) of
            not_found -> true;
            bad_ie    -> true;
            _         -> false
        end;
% Special case for bearer context errors.
validate_error(?MANDATORY_IE_INCORRECT, ?IE_v2_BEARER_CONTEXT,
               Message, MandatoryIEs, BContext) ->
    % BContext is mandatory, but incorrect.
    lists:member(?IE_v2_BEARER_CONTEXT, MandatoryIEs) and
```

```
        case get_ie(?IE_v2_BEARER_CONTEXT, 0,
                    list_to_binary(Message)) of
            bad_ie    -> true;
            not_found -> false;
            IE        ->
                 case eqc_ie_gens:valid_b_context(IE, BContext) of
                             nok -> true;
                             _   -> false
                         end
        end;
validate_error(?MANDATORY_IE_INCORRECT, CauseIE, Message,
               MandatoryIEs, _) ->
    % CauseIE is mandatory, but incorrect.
    lists:member(CauseIE, MandatoryIEs) and
        case get_ie(CauseIE, 0, list_to_binary(Message)) of
            not_found -> false;
            IE        -> case eqc_ie_gens:valid_ie(IE) of
                             nok -> true;
                             _   -> false
                         end
        end;
validate_error(_, _, _, _, _) ->
    false.

% Returns a binary IE, not_found or bad_ie, based on whether the
%  IE is found in the message or not.
get_ie(ID, Instance, <<ID, Length:16, Spare:4, Instance:4,
       Data:Length/bytes, _Rest/binary>>) ->
    <<ID, Length:16, Spare:4, 0:4, Data:Length/bytes>>;
get_ie(ID, Instance, <<_DifferentID, Length:16, _:4, _:4,
       _:Length/bytes, Rest/binary>>) ->
    get_ie(ID, Instance, <<Rest/binary>>);
% This case will occur when the ie has a stated length that is
%  longer than the rest of the message.
get_ie(ID, _, <<ID, _Rest/binary>>) ->
    bad_ie;
get_ie(_,_,_) ->
    not_found.
```

A.3 Message modules

The module of the message Downlink Data Notification Failure Indication has been removed as it serves no educational purpose.

A.3.1 Create Session Request

```
-module([---]).
-export([prop_sym/0, prop_collect/0]).
-include_lib("eqc/include/eqc.hrl").

%%%%%%%%%%%%%%%% PROPERTIES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prop_sym() ->
    ?FORALL({Message, MandatoryIEs, BContexts}, gen(),
            try
                % Pack and unpack Message.
                Unpacked =
                    [---](list_to_bitstring(Message)),
                Packed = [---](Unpacked),
                SortedPacked = eqc_common:packed_to_list(Packed),

                % Clean message from faulty IEs (that will be
                %  discarded during unpacking).
                CleanMessage =
                    eqc_common:sanitate_message(list_to_binary(Message),
                                                BContexts),

                % No crash (all mandatory IEs present and correct),
                %  compare sorted lists of IEs.
                SortedPacked == CleanMessage
            catch
                throw:Response ->
                    {Error, CauseIE} = eqc_common:get_error(Response),
                    % Mandatory IEs are assumed to have instance 0.
                    {_, BContext} = lists:keyfind(0, 1, BContexts),
                    eqc_common:validate_error(Error, CauseIE, Message,
                                              MandatoryIEs, BContext)
            end).
```

```
prop_collect() ->
    ?FORALL({Message, _, _}, gen(),
            collect(
              try
                  [---](list_to_bitstring(Message)),
                  ok
              catch
                  throw:Response ->
                      eqc_common:get_error(Response)
              end, true)).

%%%%%%%%%%%%%%%% GENERATOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen() ->
    {shuffle([
          fault_rate(1,10,eqc_ie_gens:ie_imsi(0)),
          fault_rate(1,10,eqc_ie_gens:ie_msisdn(0)),
          fault_rate(1,10,eqc_ie_gens:ie_mei(0)),
          fault_rate(1,10,eqc_ie_gens:ie_uli(0)),
          fault_rate(1,10,eqc_ie_gens:ie_serving_network(0)),
          fault_rate(1,10,eqc_ie_gens:ie_rat_type(0)),
          fault_rate(1,10,eqc_ie_gens:ie_indication(0)),
          fault_rate(1,10,eqc_ie_gens:ie_f_teid(0)),
          fault_rate(1,10,eqc_ie_gens:ie_f_teid(1)),
          fault_rate(1,10,eqc_ie_gens:ie_apn(0)),
          fault_rate(1,10,eqc_ie_gens:ie_selection_mode(0)),
          fault_rate(1,10,eqc_ie_gens:ie_pdn_type(0)),
          fault_rate(1,10,eqc_ie_gens:ie_paa(0)),
          fault_rate(1,10,eqc_ie_gens:ie_apn_restriction(0)),
          fault_rate(1,10,eqc_ie_gens:ie_ambr(0)),
          fault_rate(1,10,eqc_ie_gens:ie_ebi(0)),
          fault_rate(1,10,eqc_ie_gens:ie_pco(0)),
          fault_rate(1,10,eqc_ie_gens:ie_b_context(0,
                              [fault_rate(1,10,eqc_ie_gens:ie_ebi(0)),
                               fault_rate(1,10,eqc_ie_gens:ie_tft(0)),
                               fault_rate(1,10,eqc_ie_gens:ie_f_teid(0)),
                               fault_rate(1,10,eqc_ie_gens:ie_f_teid(1)),
                               fault_rate(1,10,eqc_ie_gens:ie_f_teid(3)),
                               fault_rate(1,10,eqc_ie_gens:ie_f_teid(4)),
                               fault_rate(1,10,eqc_ie_gens:ie_b_qos(0))])),
          fault_rate(1,10,eqc_ie_gens:ie_b_context(1,
                             [fault_rate(1,10,eqc_ie_gens:ie_ebi(0))])),
            fault_rate(1,10,eqc_ie_gens:ie_cc(0)),
            fault_rate(1,10,eqc_ie_gens:ie_ue_timezone(0))]),
     mandatory(),
     bcontexts()}.

% List of mandatory IE IDs of this message
mandatory() ->
    [
     ?IE_v2_IMSI,
     ?IE_v2_RAT_TYPE,
     ?IE_v2_F_TEID,
     ?IE_v2_APN,
     ?IE_v2_APN_RESTRICTION,
     ?IE_v2_BEARER_CONTEXT
     ].

% List of mandatory IE IDs of the Bearer Contexts of this message
bcontexts() ->
    [
     {0,[?IE_v2_EBI,
         ?IE_v2_BEARER_QOS]},
     {1,[?IE_v2_EBI]}
     ].
```

A.3.2 Create Bearer Request

```
-module([---]).
-export([prop_sym/0, prop_collect/0]).
-include_lib("eqc/include/eqc.hrl").

%%%%%%%%%%%%%%%% PROPERTIES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prop_sym() ->
    ?FORALL({Message, MandatoryIEs, BContexts}, gen(),
            try
                % Pack and unpack Message.
                Unpacked =
                    [---](list_to_bitstring(Message)),
                Packed = [---](Unpacked),
                SortedPacked = eqc_common:packed_to_list(Packed),
                % Clean message from faulty IEs (that will be
                %  discarded during unpacking).
                CleanMessage = pre_sanitate(list_to_binary(Message)),
                CleanerMessage =
                    eqc_common:sanitate_message(CleanMessage, BContexts),

                % No crash (all mandatory IEs present and correct),
                %  compare sorted lists of IEs.
                SortedPacked == CleanerMessage
            catch
                throw:Response ->
                    {Error, CauseIE} = eqc_common:get_error(Response),
                    % Mandatory IEs are assumed to have instance 0.
                    {_, BContext} = lists:keyfind(0, 1, BContexts),
                    pre_validate_error(Error, CauseIE, Message) or
                        eqc_common:validate_error(Error, CauseIE, Message,
                                                  MandatoryIEs, BContext)
            end).

prop_collect() ->
    ?FORALL({Message, _, _}, gen(),
            collect(
              try
                  [---](list_to_bitstring(Message)),
                  ok
              catch
                  throw:Response ->
                      eqc_common:get_error(Response)
              end, true)).

%%%%%%%%%%%%%%%% GENERATOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen() ->
    {shuffle([
              fault_rate(1,5,eqc_ie_gens:ie_ebi(0)),
              fault_rate(1,5,eqc_ie_gens:ie_pco(0)),
              fault_rate(1,5,eqc_ie_gens:ie_b_context(0,
                         [fault_rate(1,5,eqc_ie_gens:ie_ebi(0)),
                          fault_rate(1,5,eqc_ie_gens:ie_tft(0)),
                          fault_rate(1,5,eqc_ie_gens:ie_f_teid(0)),
                          fault_rate(1,5,eqc_ie_gens:ie_f_teid(1)),
                          fault_rate(1,5,eqc_ie_gens:ie_f_teid(3)),
                          fault_rate(1,5,eqc_ie_gens:ie_b_qos(0)),
                          fault_rate(1,5,eqc_ie_gens:ie_pco(0))]))
              ]),
     mandatory(),
     bcontexts()}.

mandatory() ->
    [
     ?IE_v2_EBI,
     ?IE_v2_BEARER_CONTEXT
     ].

bcontexts() ->
    [
     {0, [?IE_v2_EBI,
          ?IE_v2_EPS_BEARER_TFT,
          ?IE_v2_BEARER_QOS]}
     ].

%%%%%%%%%%%%%%%% HELPERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Special pre-validation for this message, it has some
%  odd rules that cannot be displayed here.
% This validator only returns true if the message specific
%  error is found.
pre_validate_error([---]) ->
    [---].

% Special pre-sanitation for the same reason as above
pre_sanitate(Msg) ->
    [---].
```

A.3.3 Delete Bearer Request

```
-module([---]).
-export([prop_sym/0, prop_collect/0]).
-include_lib("eqc/include/eqc.hrl").

% Module comments:
% Since this message contains no mandatory IEs, no valid
%  crashes can occur, and thus - no crashes are checked,
%  any crash is considered a failure.

%%%%%%%%%%%%%%%% PROPERTIES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prop_sym() ->
    ?FORALL({Message, BContexts}, gen(),
            begin
                % Pack and unpack Message.
                Unpacked =
                    [---](list_to_bitstring(Message)),
                Packed = [---](Unpacked),
                SortedPacked = eqc_common:packed_to_list(Packed),

                % Clean message from faulty IEs (that will be
                %  discarded during unpacking).
                CleanMessage =
                    eqc_common:sanitate_message(list_to_binary(Message),
                                                BContexts),

                % Compare sorted lists of IEs.
                SortedPacked == CleanMessage
            end).

% Will only collect 100% ok.
prop_collect() ->
    ?FORALL({Message, _}, gen(),
            collect(
              try
                  {request, _} =
                      [---](list_to_bitstring(Message)),
                  ok
              catch
```

```
                  throw:Response ->
                      eqc_common:get_error(Response)
              end, true)).

%%%%%%%%%%%%%%%% GENERATOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen() ->
    {shuffle([
              fault_rate(1,10,eqc_ie_gens:ie_ebi(0)),
              fault_rate(1,10,eqc_ie_gens:ie_ebi(1)),
              fault_rate(1,10,eqc_ie_gens:ie_b_context(0,
                             [fault_rate(1,10,eqc_ie_gens:ie_ebi(0)),
                              fault_rate(1,10,eqc_ie_gens:ie_cause(0)),
                              fault_rate(1,10,eqc_ie_gens:ie_pco(0))])),
              fault_rate(1,10,eqc_ie_gens:ie_pco(0)),
              fault_rate(1,10,eqc_ie_gens:ie_cause(0))
              ]),
     bcontexts()}.

bcontexts() ->
    [
     {0, [?IE_v2_EBI,
          ?IE_v2_CAUSE]}
     ].
```