Manual, deterministic testing versus stochastic testing using Quickcheck

Fredrik Hoback (fredrik.hoback@gmail.com), Adam Jouda (it01_tjo@kth.se)

Master's Thesis 2*20 credits
Supervisor at Ericsson: Martin Filimonovic & Johan Karlsson
Examiner: Johan Montelius

Royal Institute of Technology
School of Information and Communication Technology
SE-164 40 KISTA
SWEDEN

Abstract

The goal of our thesis was to make a comparison between the testing systems used at Telefonplan today and a tool based on Quickcheck, a tool developed by Quviq. We conclude that Quickcheck has a big potential and can be used as an automated test tool for conformance testing Ericsson's Media Gateway. It has some features that is invaluable when doing conformance testing such as the ability to randomize sequences of messages as well as random generating data in these messages. Another feature is that it has the ability to shrink failing sequences to the minimum failing sequence which can be used to find unstable states in the Media Gateway. We managed to build a prototype which can be used for further development of a real instrument to do conformance testing. But Quickcheck cannot replace todays tools used for testing at Ericsson. It lacks the ability to guarantee that test cases are performed which in turn does not guaranties product quality. It also requires the developers to have vast Erlang and Quickcheck knowledge to make this tool work. There is also the issue of cost effectiveness. The tools used today were developed at Ericsson and are therefore free, whilst Quickcheck is a licensed product with personal licenses for each user.

Contents

Acknowledgments

We would foremost like to thank Thomas Arts and John Hughes at Quviq. Without their help we would still be sitting here scratching our heads. They spent several hours answering our emails and gave invaluable help.

Then we would like to thank our supervisors at Ericsson, Martin Filimonovic and Johan Karlsson. They answered all our questions and they still answered even though we asked the same questions over and over again.

We would also like to thank Nicolas Zervos for giving us this great opportunity of working with this thesis here at Telefonplan.

Finally we would like to thank Johan Montelius, our examiner at KTH. He gave us all the necessary guidelines in order to succeed.

Chapter 1

Introduction

1.1 Motivation

Today automated testing is vital for system development. It ensures quality and gives a good test coverage. It also makes function test and conformance test easier. This is especially true in the telecom business, where different manufactures choose how much to implement and conform to different standards. Companies need to guarantee that their product will work together with other manufacturers. Another issue is the uptime. Usually the system downtime is 5 minutes per year and so involuntary crashes must be kept to a minimum. This is why automated testing is so crucial. It provides a way to do conformance testing and also do basic testing to avoid system crashes much easier. Automated system test also gives time and labour efficiency. A manual/static testing environment requires a lot of time to construct and can be difficult to maintain and adjust. This makes system testers avoid large test cases. An automated test tool allows system testers to conduct large test cases but also gives them a larger pool to construct test cases from. In this sense the test coverage is larger.

IMS (IP Multimedia Subsystem) is framework for the next generation mobile networks. It will provide IP services to the end user and make Internet integration easier. Consisting of a collection of different functions linked with standardized interfaces, telecom companies like Ericsson must make their equipment conform to the ruling standards and protocols in order to make them interconnect with other telecom companies' equipment. Therefore protocol conformance testing is an important part of guaranteeing interoperability between different nodes in the IMS.

1.2 Problem specification

Protocol conformance and function testing is a tedious process. In order to make it more efficient, a tool developed by Quviq[1] can automate test cases using a predefined set of rules and validate the outcome. Using random generators and a defined premise and a set of rules the test case can be automated to fit your scenario. This thesis propose a prototype solution built upon Quickcheck. Using Quickcheck we will build an application that can be used to run test cases provided by Ericsson that are used when todays systems are tested.

1.3 Goals

The goal is to implement a test tool prototype based on Quickcheck for the Media gateway. The second goal is to compare our prototype to the tools used today at the department in issues, like maintenance, test case coverage, easiness to use and if it is possible to build a generic tool to be used with other protocols. A requirement was to implement at least 5 test cases according to our supervisor.

1.4 Related work

The developers of Quickcheck John Hughes and Thomas Arts have done some earlier work at Ericsson together with Joakim Johansson and Ulf Wiger [8]. Their white paper concentrate on Ericsson's Media Proxy. They were able to find both subtle and obvious bugs when running their implementation against the Media Gateway. Both the Media Gateway and the Media Proxy communicates using H.248. They also had the opportunity to test their implementation against two different versions of the Media Proxy and showed that they discovered more errors in the old version, as one could expect (but new versions also introduce new bugs so this may not always be the case). Some of the obvious bugs found made the Media Proxy crash because of a faulty state. They concluded that using Quickcheck in early production and development would make it easy with little effort to detect faults and in this way ensure product quality.

1.5 Thesis outline

The first part of this paper concentrates on the underlaying protocols and system in which is interesting for this thesis. We will then continue to describe the different tools used by us and the test department. Later we introduce our implementation and how we designed it and why we did in certain ways. Finally we will show the result and conclusion of our work. We will also have a discussion

[1] www.quviq.com

at the end and some reflection on future work that can be done. In order to make this thesis easy to read and follow we will keep code examples to a minimum.

Abbreviations

2.5G	2nd generation mobile networks that are packet switched
2G	2nd generation mobile networks
3G	Third generation mobile networks
ATM	Asynchronous Transfer Mode
BIF	A Built-In Function in the Erlang
GEGGA	GEneric meGaco simulAtor with an extra G
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part
MEGACO	Media Gateway Control Protocol
MGC	Media Gateway Controller
MGW	Media Gateway
MP	Media Proxy
NGN	Next Generation Networking
PBX	Private Branch eXchange
PCM	Pulse Code Modulation
PDA	Personal Data Appliance
PID	Process IDentifier
POTS	Plain Old Telephoning Service
PSTN	Public Switched Telephone Network
PTA	Protocol Tester Application
RTP	Real-time Transport Protocol
SCTP	Simple Control Transport Protocol
SDP	Session Description Protocol
SGC	Signaling Gateway Controller
SIP	Session Initiation Protocol
STASI	STochAStic Interrogator
TDM	Time Division Multiplexing
TTL	Time To Live
VoIP	Voice Over IP
VTS	Visual Test Server

Syllabus

In this thesis we use the following terms interchangeably:

- Command/message refers to a H.248 message such as Add for example
- H.248 and MEGACO refers to the same protocol
- Application/Implementation/Program refers to our software developed by us

Chapter 2

Background

This section will begin by describing the background to our thesis. First we will look at different types of testing, their advantages and disadvantages then move onto describing the overall system IMS and some nodes within the IMS which are important to this thesis. This section will also describe necessary protocols, underlaying structure and connection models.

2.1 Testing

The idea of testing is that to help identify the correctness, completeness, security, and more important the quality of the system to be tested. The system can be both hardware and/or software, but we mainly focus on testing the software. Testing is a process of technical investigation. It is intended to reveal potential errors and gather quality-related information about the system. The terms of quality can differ between one tester and another, so it is recommended to set a common specification that determines some set of guidelines a system tester can follow. Some of the common quality attributes are capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

Testing follows some criticism or guidelines that compares the behavior and state of the product against a predefined specification.

When testing a software, engineers should distinguish between software fault and software failures. What is meant with failures is that the software does not work properly according to what a user expects. Meanwhile software faults are programming errors that may or may not reveal itself as a failure. A fault can turn into a failure if it meets some computational conditions. A fault can also turn into a failure when the software that has been tested on some specific hardware or compiler is ported to a different hardware platform or a different compiler.

A software tester has to have confidence in the system to be tested so that the organization can be confident and ensures that the software has an acceptable defect rate.

An independent group of software testers can perform software testing after the development of the system but before shipment to customers. Testing can occur simultaneously with the project development and it is a continuous process until the whole project finishes. Another common practice is to develop test suites during support escalation procedures. This is referred to as regression testing and ensures that the future updates of the software don't repeat the already known mistakes.

2.1.1 Manual versus automatic testing

There are both some negative and positive issues regarding whether to use manual testing (tests run by human) or automated testing (run by some automated tool) [3]. Manual testing should for example be used if the tester is going to run a very small number of times or the test is expensive to automate. Many system developers believe that test automation is expensive relative to its value. Others recommend automating one hundred percent of all tests. What you automate depends on the tools you use. If the tools have any limitations, the tests are preferably done manually.

Here is a list of advantages and disadvantages of using manual and automating testing [5] respectively:

Advantages of automation

- **Reliable:** Automated testing eliminates human errors because tests perform precisely the same operations each time they are run
- **Repeatable:** Software can be tested in terms of how they react under repeated execution of the same operations
- **Comprehensive:** A tester can build a test suite that covers every feature of the application
- **Reusable:** Reusing tests on different versions of an application
- **Quality:** Better quality software, run more tests in less time with fewer resources
- **Fast:** Tests run by automated test tools run much faster than tests that run by humans
- **Cost:** Reduces the cost as the number of resources for regression test are reduced. Long term costs is reduced

- It is good to run automation against code that changes frequently
- Covers and tests a large test matrix. Automated tests can be run at the same time on different machines, whereas the manual tests would have to be run sequentially.

Disadvantages of automation

- **Cost:** Writing the test cases and writing/configuring the automated tool costs more initially than doing it manually
- Some parameters can not be done automated, for example, if you can not tell the font color via code or the automation tool, it is a manual test
- High skills are required to write the automation test scripts
- Small changes in for example the GUI requires that the test script need to be changed or replaced by a new one. New updates for existing products that are going to be tested may require to replace the test script entirely.

Advantages of manual testing

- If the test case only runs a very few times it should be a manual test. It requires less cost than automating it
- Allows the tester to perform more ad-hoc (random testing). More bugs can be found this way
- Short term costs are reduced
- The more time testers spend testing a module the greater the odds to find bugs.

Disadvantages of manual testing

- Tests running manually can be very time consuming
- For every release you must rerun the same set of tests which can be tiresome
- Large projects usually have hundreds of classes and other class libraries. It would be very difficult to write separate test suites for each class in the project
- Repeating the exact test in the future can be difficult because the tests are run by humans and humans can forget
- When doing tests manually the result is verified manually. Even if the result is wrong, you may not realize it and the result may then be regarded as successful.

2.1.2 Stochastic versus deterministic testing

Stochastic testing refers to the technique of using data or parameters inside a test case that differ between test runs. You can not predict what value that will be chosen, the only thing that you have knowledge of is the limits that you predefine.

Deterministic testing refers to the technique of using the same set of parameters and data inside a test case that does not differ between test runs. All data are given in advance and if you want to test a new set of parameters you have to rewrite your test suite.

Both methods have its advantages and disadvantages. Whether to choose one of them is up to the system designer and tester. Of course one can implement part of the code deterministic and the other part stochastic and in that way you can achieve some kind of hybrid solution where you can combine the advantages of both methods.

Here follows a list of advantages and disadvantages of using stochastic and deterministic testing respectively:

Advantages of stochastic testing

- Achieving greater test area coverage
- Different set of data and parameters can be run each time
- Reduces the amount of testing time
- The potential of finding more errors and system bugs.

Disadvantages of stochastic testing

- Much more expensive to implement a stochastic testing suite
- Requires high programming skills to design and implement the required generators
- New product updates may require testing different set of parameters and/or data which in turn requires configuring or rewriting the test code
- Difficult to reduce the scope in a way that makes stochastic testing the same as deterministic testing.

Advantages of deterministic testing

- Easy and straight forward to implement
- Independent of software upgrades.

Disadvantages of deterministic testing

- Smaller coverage test are
- Need to change the parameters and data for each test case
- Time consuming, going through a whole test suite that requires testing different set of parameters and data can take a significant long time.

2.1.3 Positive and negative testing

In this thesis we concentrate on two types of testing, namely: positive and negative. There may be other definitions on what constitutes a positive or a negative test. But what is meant with positive and negative testing throughout this thesis is going to be defined here.

A positive test or test case is when a the test to be performed is legal. For example, a message that is conforming to ruling standards of some kind. As this thesis concentrate on conformance testing of protocol a correct message in this sense is a message that conforms to the H.248 framework.

A negative test case on the other hand is when a test case contains an illegal message. The message could contain faulty parameters or it may be a part of an illegal sequence. For example, a subtract is done before anything has been added. A faulty parameter is for example a port number set to -1.

2.2 IP multimedia subsystem

The IP (Internet Protocol) Multimedia Subsystem (IMS) is the key part of the Third Generation (3G) networks. 3G aim to merge the two popular paradigms that exists in todays communication technology systems, the IP and the cellular networks into one coherent IP network. The IMS aims to provide access to all services that the Internet provides no matter where you are or what system your are using for the moment as long as you have access to a 3G device.

The benefits of the Internet is its ability of providing new services because it uses open protocols. This means that any person who wants to develop a new service has a widespread knowledge of Internet protocols because in the end the people who are developing new services are the people who are going to use them.

The benefits of the cellular networks is that it provides coverage virtually everywhere. Users are not limited of using their terminals inside cities only but almost everywhere. There also exists roaming agreements between operators of different countries that allow users to access cellular services when they are abroad.

3G networks consists of different domains, such as the circuit-switched domain and the packet-switched domain. Clarifying the benefits of IMS in 3G require understanding of these two domains.

The circuit-switched domain is used in 2G networks, but for example GPRS which is a 2G standard runs over IP. The purpose of these circuits are to carry voice, video and instant messages. 2G terminals can act as a modem to transport IP packets over a circuit which is much slower than the traditional packet-switched networks. This is the main reason of substituting circuit-switched into the more adequate packet-switched networks in cellular technology.

Packet-switched domain provides IP access to the Internet. Data transmission is much faster than that in circuit-switched networks and has more bandwidth. Users can surf the web over any broadband connection. For instance a user can install a VoIP client in his/her 3G terminal and start making VoIP calls over the packet-switched domain.

So the question of why using IMS, if full Internet functionality is already available for 3G users through the packet-switched domain, arise here. The answer to this question is: Using IMS is good for Quality of Service (QoS), charging, transparency, cost effectiveness and integration of different services.

QoS is required of getting all the benefits of real time multimedia sessions. The IMS provides session establishment synchronized with QoS. Charging of multimedia sessions will be controlled by IMS in a more sophisticated way. For instance instead of charging money per byte downloaded one can charge money per session or per instant message sent regardless of the size in bytes. The last reason of using IMS is integration of services. For example an operator is developing a service and may want to use services developed by third parties, combine and integrate them with its service before selling the whole new product [10].

2.2.1 Media Gateway Controller

As the name proposes it is essentially a controller. It controls one or more MGWs depending on implementation. The controller also converts different signaling protocols such as ISUP and SIP, ISUP being from the old POTS network. Both SIP and ISUP is used to control call setup, call disconnect and other things like start the "ringing" on the other end when a call is placed. The controller controls the MGW over H.248 which in turn is run over SCTP, UDP or TCP.

2.2.2 Media Gateway

A Media Gateway is a translator between different telecommunication networks such as PSTN, NGN, 2G, 2.5G and 3G or PBX. Media Gateways handle multi-

media communication over ATM or IP. Because of the functionality of the MGW that connects different types of networks, it becomes trivial that one of its main functionalities is to convert between different types of encoding and transmission techniques.

As an example of conversion between one codec to another is a call originating from a PSTN network ending in a VoIP terminal. Transferring occurs from a circuit-switched node to a packet-switched (IP) node. From the PSTN side the MGW uses one or more PCM (Pulse Code Modulation) time slots transferring the IMS media through the packet-switched network using the Real-Time protocol (RTP). MGWs are controlled by one or more MGCs. Signalling between the MGC and the MGW is achieved mainly by H.248 (Megaco) or also by SIP. Figure 2.1 shows the MGC and MGW interaction with the SGC performing control signal conversion.

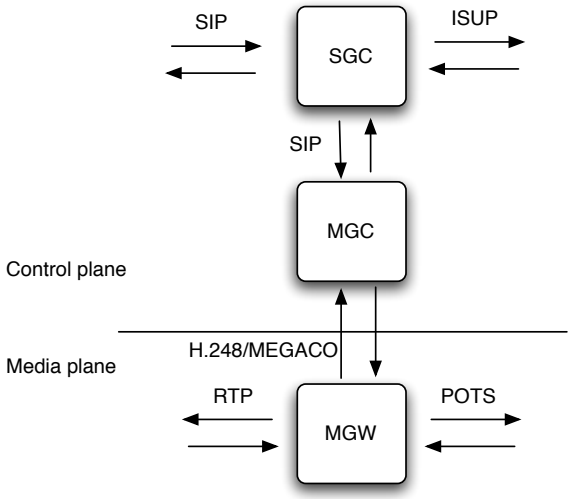

Figure 2.1: MGC and MGW interaction

2.3 TDM - Time Division Multiplexing

TDM is a way to transfer digital content over a medium (it is possible to multiplex analog content as well but it is rarely used) . The channel is divided into time slots, or so called TDM frames. In this way it is possible to transfer more logical channels over one physic channel (multiplexing them). This technology is used in the GSM networks to divide each radio frequency into time slots for simultaneous access. TDM is also used in the old traditional telephone networks, like PSTN.

2.4 H.248/Megaco

Megaco (Media Gateway Control Protocol) is a signaling protocol used between MGC/SGC and the MGW[9]. It is a general framework suitable for gateways and multipoint control units. The general connection model for this framework describes logical entities within the MGW which can be controlled by a MGC. There are three main abstractions: contexts, terminations and streams. Within a context there may be several terminations and within a termination there may be several streams. The protocol is synchronous so for each message sent it expects a reply.

In figure 2.2 we can see the position of H.248 in the OSI model.

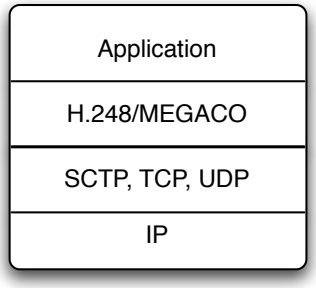

Figure 2.2: H.248 in the OSI model

2.4.1 Contexts

A context[9] is an association between terminations where the context describes who is connected to who. There is also a `NULL` context containing all terminations not associated with any other termination. Each termination is unique and therefor a termination can only belong to one context and the maximum numbers of termination within a context is a MGW property. For example a MGW that only offer point-to-point connectivity allows at most two terminations per context.

Adding terminations to a context is done with the `Add` command and it must also specify which context the termination is to be added to or it will create a new context containing the termination added. Removing terminations from a context is done with the `Subtract` command if there are no terminations left the context is deleted. It is also possible to move terminations between different contexts with the `Move` command. Contexts also include attributes which describe them it and can be viewed with an `Audit` command for statistic purpose for example. Each context has a unique `ContextID` . It also has a priority which

is used by the MGW to give precedence to contexts. Priority 0 is lowest and priority 15 highest. An emergency indicator is also present for emergency calls.

2.4.2 Terminations

Terminations in the H.248 framework[9] is a logic entity just like a context. They are described by their properties which also contain a `TerminationID` that is given at the creation of a termination, these are unique to each termination. A termination is added to a context with the `Add` command or removed with the `Subtract` command. There exists two kinds of termination, semi-permanent or ephemeral, where semi-permanent might represent a TDM channel opposed to ephemeral which represents RTP streams which in turn only exists when information is sent. The semi-permanent termination exists only in the `NULL` context.

A termination can be programmed to react to signals and/or events in the MGW. It is possible to modify terminations, for example add an event or signal that the termination should respond to or to change codecs (for sound or video). Terminations can also accumulate statistics which can be reported to the MGC via the `AuditValue` command or when a termination is subtracted from a context. As in the case of a context a termination has some properties and descriptors which can be set and if not they take the default value.

There are many different types of terminations and to achieve interoperability between MGC and MGW terminations are grouped into so called packages. This means that terminations realize a set of packages which can be explored with the `AuditValue` command. In order to support a package a termination must support all properties of that package such as signals, events, statistics defined within the package.

2.4.3 Streams

Streams are entities within each termination. A termination can hold several streams which is identified by its `StreamID` which in turn is set by the MGC. The stream maybe audio, video or text depending on what is supported in the gateway. As with the Termination and Context streams have descriptors. The Stream descriptor specifies all the parameters for the media stream included in a termination. Within the stream descriptor there are 3 subsidiary descriptors: `Local`, `LocalControl` and `RemoteControl`.

Stream descriptors:

- `Local`: Refers to the media received by the MGW. It contains the SDP part.

- `LocalControl`: This descriptor contains several properties: `Mode`, `ReserveValue`, `ReserveGroup`. `Mode` can be either one of send-only, receive-only, send/receive, inactive or loopback. This is the mode of the stream so if it is set to receive-only no media will pass out of the context. Inactive is the default value. `ReserveValue` or `ReserveGroup` is set to true or false. If true then the MGW shall reserve resources, if available, for the resources requested. If it cannot comply, it shall respond with error 510 (insufficient resources).
- `LocalRemote`: As with the `Local` descriptor it describes the media sent by the MGW. Which will be described in section 3.5.

2.4.4 Messages

The H.248 framework describes commands used to modify the logical entities within the connection model. Most commands are used by the MGC to manage the MGW but `Notify` and `ServiceChange` commands are used by the MGW to inform the MGC of changes. Listed below are all the commands[9] described in the connection model.

- **Add:** Adds a termination to a context. If no context exists then a context is created.
- **Subtract:** Removes a termination from a context, if the termination removed was the last in a context the context is deleted.
- **Modify:** Modifies the properties of a termination.
- **Move:** Move a termination to another context.
- **AuditValue:** Returns the statistics of a termination, such as current state, events and signals.
- **Notify:** Used by the MGW to notify the MGC about events in the MGW.
- **AuditCapabilities:** Returns all the values for termination properties, events and signals that is supported by the MGW.
- **ServiceChange:** Is used by the MGW to register at the MGC. The MGC should answer with a `ServiceChange` reply. It is also used to notify the MGC of termination changes at the MGW. `ServiceChange` messages are also used to announce handover by the MGC.

H.248 messages can be coded as either plain text or compact text. The plain text version of a message is shown below as is the compact version [2] but we will skip the compact message for the rest of the messages because it was not used during our implementation and it is harder to read and understand. But as one can clearly see the compact message holds the same parameters, the only

difference is the abbreviations for the full length word. We also chose not to show all the messages in plain text versions above but only those of importance to this study.

```
MEGACO/1 [124.124.124.222]
  Transaction = 9998 {
    Context = - {
      ServiceChange = ROOT {
        Services {
          Method = Restart,
          ServiceChangeAddress = 55555,
          Profile = ResGW/1,
          Reason = "901 MG Cold Boot"
          }
       }
    }
}
```

As for the compact version of the same message.

```
!/1 [124.124.124.222] T=9998{
C=-{SC=ROOT{SV{MT=RS,AD=55555,PF=ResGW/1,RE="901 MG Cold Boot"}}}}
```

Add

All messages begin with `MEGACO/x` where the x indicates a version number. Following the version number is either an IP address[1], IP address with port, domainName or deviceName. `Transaction` which is an identification number chosen by the sender and is used by the answering side in the reply message to confirm which message the reply answers. Transaction numbers are ranged from 1 to a large number and there are timers set so that a transaction number cannot be repeated for at least 30 seconds or more, this is to allow the MGW to wait for retransmissions. Then we have `Context` which indicates which context this message applies to. It can either be a number (1-438597384759 where the last number indicates the `NULL` context) or a wildcard which tells the MGW to either choose a random context or choose a new context. Then follows a field that tells the receiver (MGW or MGC) what kind of a message it is, as listed above. This field is changed to `AddReply,SubtractReply` and so on if the message is a reply to an `Add` or `Subtract` message.

```
MEGACO/2 mgc
  Transaction = 51654332 {
    Context = $ {
      Add = $ {
```

[1] IPv4 or IPv6

```
        Media {
          Stream = 62708 {
            LocalControl {
              Mode = Inactive,
              ReservedGroup = ON,
              ReservedValue = ON
            },
            Local {
              v=0
              c=IN IP4 $
              m=audio 36 RTP/AVP 125
              a=rtpmap:125 AMR/8000
              a=fmtp:125 octet-align=1
              a=ptime:40
            },
            Remote {
              v=0
              c=IN IP4 10.11.129.155
              m=audio 7460 RTP/AVP 125
              a=rtpmap:125 AMR/8000
              a=fmtp:125 octet-align=1
              a=ptime:40
           }
        }
      }
   }
}
```

Subtract

A subtract message tells which Context and within this context which Termination to remove. When a termination is deleted its streams are removed as well. The statistic descriptor also tells the MGW to send statistics along with the SubtractReply. There is also a subtract message to subtract all terminations within the MGW. In order to do this, one sets the `contextID` to 438597384759 and sets the termination to a wildcard.

```
MEGACO/2 mgc
  Transaction = 42442128 {
    Context = 33563011 {
      Subtract = ephemeral/0xd41035d5 {
        Audit {  }
        }
   }}
```

Modify

The `Modify` message is very much like the `Add` message but with additional descriptors for changing the streams, adding an event or signal. With the modify command you modify a termination within a context both specified in the message.

```
MEGACO/2 mgc
  Transaction = 42445163 {
    Context = 33563011 {
      Modify = ephemeral/0xd61035d7 {
        Media {
          Stream = 3169 {
            LocalControl {
              Mode = SendReceive,
              ReservedGroup = ON,
              ReservedValue = OFF
            },
            Remote {
              v=0
              c=IN IP4 10.11.136.145
              m=audio 11764 RTP/AVP 120
              a=rtpmap:120 G726-32/8000
              a=ptime:30
              }
            }
          }
       }
    }
}
```

Move

Move is used when one want to move a termination from a context to another. Here the field context refers to the context which you want to move the specified termination given in the termination field.

```
MEGACO/2 mgc
  Transaction = 12345564 {
    Context = 3423455 {
      Move = ephemeral/0xd71035d8 {
      }
   }
}
```

2.4.5 Call setup

A simple call setup between a MGW and a MGC is done via H.248 messages with their specific properties. If the MGW is started or has recently restarted the first message sent is a `ServiceChange` to register itself with a MGC. The MGC answers with a `ServiceChangeReply` either accepting or rejecting the registration. Figure 2.3 shows all the messages from a registration to a call setup phase and when the call is tiered down with the `Subtract` command.

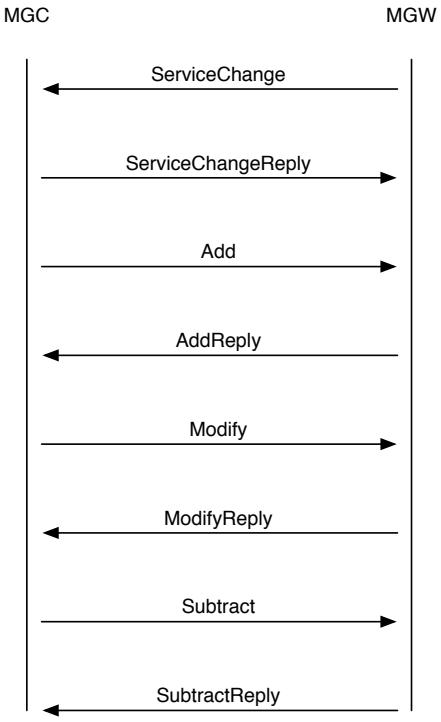

Figure 2.3: Call setup message passing between MGC and MGW

In figure 2.4 the logical entities created inside the MGW is shown. A context is created with the Add command if non existed earlier as well as a termination. These entities can all be modified during the call with the `Modify` command.

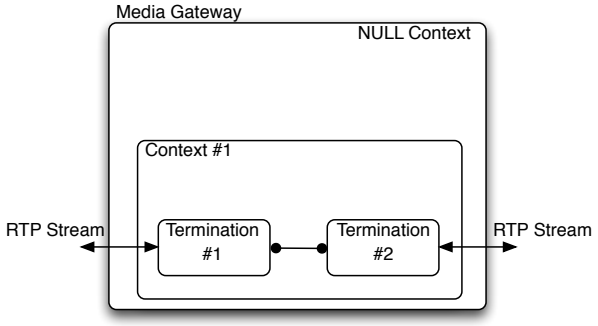

Figure 2.4: Context with terminations for two way audio conversation

2.4.6 Message passing

There are several ways to send messages. A message can be a single command (as shown in previous section) or can contain several commands in different ways described below. All these ways of sending and receiving messages are legal but may not be supported and there are differences that are important to show for the continuance of this report. The reason sending multiple commands in a message is to reduce the overhead.

Multiple actions

This is still just one message but it contains several actions for different for the same termination or for different terminations within one context. The actions are executed by the MGW in the order they are given in the message. As seen the two actions are: `Subtract` and `Modify`. Both actions refer to the same context (33563041).

```
MEGACO/2 mgc
Transaction = 48643304 {
  Context = 33563041 {
    Subtract = ephemeral/0x06103607 {
      Audit {  }
      },
      Modify = ephemeral/0x08103609 {
        Media {
          Stream = 64225 {
            LocalControl {
              Mode = SendReceive,
              ReservedGroup = OFF,
              ReservedValue = ON
            },
          Remote {
            v=0
            c=IN IP4 10.11.185.146
            m=audio 31886 RTP/AVP 120
            a=rtpmap:120 G723/8000
            a=fmtp:120 annexa=yes
            a=ptime:60
            }
          }
        }
      }
   }
}
```

The reply for this message will look like this:

```
MEGACO/2 mgw
Reply = 48643304 {
  Context = 33563041 {
    Subtract = ephemeral/0x06103607,
    Modify = ephemeral/0x08103609
  }
}
```

Multiple contexts

Multiple contexts refer to when sending a message that contains several contexts in one message. Each command is processed in the receiving order.

```
MEGACO/2 mgc
Transaction = 51802831 {
  Context = $ {
    Add = $ {
      Media {
        Stream = 15373 {
          LocalControl {
            Mode = Inactive,
            ReservedGroup = OFF,
            ReservedValue = OFF
          },
        Local {
          v=0
          c=IN IP4 $
          m=audio 36 RTP/AVP 105
          a=rtpmap:105 G726-32/8000
          a=ptime:40
        },
      Remote {
        v=0
        c=IN IP4 10.11.194.159
        m=audio 17123 RTP/AVP 105
        a=rtpmap:105 G726-32/8000
        a=ptime:40
      }
     }
   }
 }
},
  Context = 33563065 {
```

```
    Subtract = ephemeral/0x2b10362c {
      Audit {  }
    }
  }
}
```

The reply:

```
MEGACO/2 mgw
Reply = 51802831 {
  Context = 33563066 {
    Add = ephemeral/0x2c10362d {
      Media {
        Stream = 15373 {
          Local {
            v=0
            c=IN IP4 10.11.12.30
            m=audio 20496 RTP/AVP 105
            a=rtpmap:105 G726-32/8000
            a=ptime:40
          }
        }
      }
    }
  },
  Context = 33563065 {
    Subtract = ephemeral/0x2b10362c
  }
}
```

Multiple transactions

Multiple transactions in one message looks like the two types above but it gives each command a transaction number. Though there are more differences. **First:** each transaction in this message will get a separate reply. **Second:** the MGW may not process each transaction in the order they are given. **Third:** the replies may not come in the processed order. Also the MGW will send a reply for each transaction number given.

```
MEGACO/2 mgc
Transaction = 1289839401 {
  Context = 33555830 {
    Add = $ {
      Media {
         LocalControl {
```

```
            Mode = Inactive
               },
                 Local {
                   v=0
                   c=IN IP4 $
                   m=audio $ RTP/AVP 8
                   a=ptime:10
               },
                 Remote {
                   v=0
                   c=IN IP4 10.11.244.42
                   m=audio 63018 RTP/AVP 8
                   a=ptime:10
                 }
             }
          }
     }
}Transaction = 1289847899 {
   Context = 33555830 {
     Modify = ephemeral/0x2b10072c {
       Media {
         LocalControl {
           Mode = Inactive
         }
       }
     }
  }
}
```

There would be, as previous stated, a reply for each transaction in the message but we will satisfy by just showing one.

```
MEGACO/2 mgw
Reply = 1289847899 {
  Context = 33555830 {
    Modify = ephemeral/0x2b10072c
      }
}
```

All these methods of sending and receiving messages are part of the H.248 standard described in [9]. It is up to the manufacturer to support these different methods. As we will show later, not all are supported or even used.

2.4.7 TDM termination

In the previous section we described the different logical entities inside the MGW. In all the messages above the terminations shown are IP terminations, these are all ephemeral while a physical channel, like a TDM channel is semi-permanent. A MGW, as said in the beginning, can connect to different types of networks and there are different profiles for each kind of connection. A physical channel is seldom released when a connection (someone hangs up) is terminated. The physical channel remains open while the logical channel that is multiplexed onto the physical channel is terminated. To add a logical channel to a TDM channel one must send an `Add` message to specify which TDM channel and which logical channel on that TDM channel you want to use. A typical `Add` message:

```
MEGACO/2 mgc
Transaction = 1041176792 {
  Context = $ {
    Add = tdm3/26 {
      Media {
        Stream = 3982 {
          LocalControl {
            Mode = Inactive,
            ReservedGroup = OFF,
            ReservedValue = ON
          }
         }
       }
     }
   }
 }
```

The MGW handles 64 physical TDM channels and each physical channel handles 32 logical channels (although one of the 32 channels for each physical are for signaling). The field `Add=tdm3/26` tells the MGW which physical channel, in this case 3 and which logical channel, 26 it wants to use.

2.5 SDP

The SDP part of a H.248 message is to describe the streaming media [7]. Here is the SDP part of an `Add` message in the Remote descriptor.

```
v=0
c=IN IP4 10.11.129.155
m=audio 7460 RTP/AVP 125
a=rtpmap:125 AMR/8000
a=fmtp:125 octet-align=1
```

```
a=ptime:40
```

The SDP field consists of a number of lines in the form:

```
<type>=<value>
```

The `v` field tells which protocol version it is referring to. The `c` field contains a data connection where `IN` stands for Internet and `IP4` tells which address format is used and then the IP address that can be either multicast or unicast. One can also include a `TTL` this would be denoted as `10.11.162.80/127` in the case above where `127` is the `TTL` field. Media is described in the `m` line which contains several sub-fields.

```
m=<media> <port> <protocol> <format>
m=audio 43003 RTP/AVP 8
```

`Media` can be either text, video, audio, application or message. `Port` is the transport port to send media to. `Protocol` states which protocol used and it is dependent on the type of address used in the c field above. The last sub-field `format` is the media format description which is dependent on which protocol is used. Last in the SDP part is the attribute field `a` . There are several attributes described in the RFC.

```
a=rtpmap:125 AMR/8000
a=fmtp:125 octet-align=1
a=ptime:40
```

`rtpmap` maps the type number in the "m" line to a payload format to be used. It also provides some information about clock rate and encoding parameters. The `fmtp` field provides additional attributes about the codec used, it extends the `rtpmap` field.

```
a=fmtp:<format> <format specific parameters>
a=fmtp:125 octet-align=1
```

The last field in the above SDP is `ptime` . `ptime` is the length in milliseconds of the media in the packet sent. It is closely related to which audio codec used.

There are several more fields described in the RFC[2], and they may be interesting to the reader but we will satisfy by describing those used in this thesis.

2.6 SCTP

SCTP (Stream Control Transmission Protocol) and TCP (Transport Control Protocol) are two reliable transport protocols in contrast to UDP which is unreliable in a sense that data is not guaranteed to reach the end node. Typically

[2]the RFC can be found at: http://tools.ietf.org/html/rfc4566

transport protocols are chosen in way that fits the media being transported. As for IMS, SCTP has been chosen to transport H.248 messages between MGC and MGW this is because SCTP unlike TCP requires that all data is delivered before processing. This is not the case in TCP where, lets say, a web browser can start processing data as soon as it arrives displaying a web page. In this contrast SCTP makes sure that all data arrives reliable before processing it. This makes SCTP more suitable for sending signaling messages. Reliable transport protocols like TCP and SCTP also provide congestion control which is not present in unreliable protocols like UDP (although it can be implemented in the application layer).

2.6.1 SCTP association

To establish a connection (association in SCTP terms) between two nodes one of the nodes begins with sending an INIT (Initiation) chunk [6]. It also sets a timer waiting for INIT-ACK, if the timer expires before an INIT-ACK is received the sender retransmits an INIT. After a number of retries it is reported to the upper layer telling that client was unreachable. The receiving side processes the INIT chunk and generates values needed to enter an established association and derives a MAC (Message Authentication Code) which is used to verify the sender. These values are then hashed (MD5 or SHA-1) with a secret key and put in a cookie and sent back to the sender in an INIT-ACK. It then forgets about the transmission and enters a closed state. The receiver of the INIT-ACK assembles the cookie from INIT-ACK and puts it in a COOKIE-ECHO which is then returned. Upon reception of a COOKIE-ECHO message the receiver unpacks the data contained in the cookie and uses the MAC therein to verify that it was the same cookie it sent in with the INIT-ACK message. In this way it knows that it is talking to the same node who originated the INIT message. Now after the COOKIE-ECHO message, the node can answer with a COOKIE-ACK message and both sides will enter an established state. The methods using cookies protects against TCP SYN attacks[1]. See figure 2.5 for message passing with normal association establishment.

2.6.2 Data transmission

SCTP data transmission also separates the transfer mode and delivery mechanism such that applications may have either strict order-of-transmission or just reliable transmission. This is done by having different streams within one SCTP association. In this way it reduces the head-of-line blocking between independent streams of messages [4]. Another issue handled by SCTP is high priority transactions where a message can be expedited before a queue of unordered messages.

SCTP also provides flow and congestion control mechanisms to the same extent as TCP. This makes SCTP easy to introduce in existing IP networks. Other

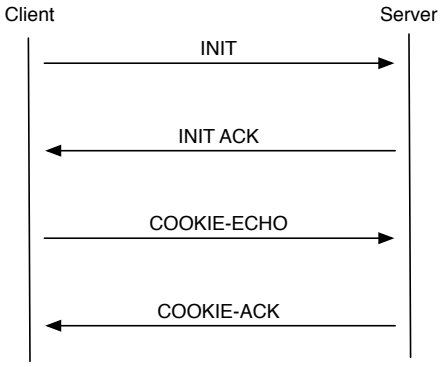

Figure 2.5: SCTP association setup

issues important for reliable connectivity is redundant path management which is handled within SCTP with multi-homed IP nodes for redundant path deployment.

2.7 Erlang OTP

The proposed thesis demanded that this implementation was developed in Erlang. Erlang is a general purpose programming language with emphasis on concurrency, distribution and fault tolerance. It is platform independent and widely used inside Ericsson's telecommunication systems. It is a runtime system which supports several systems that was developed by Ericsson in 1987 but was released as open source in 1998. In figure 2.6 we see that Erlang is platform independent using a virtual machine.

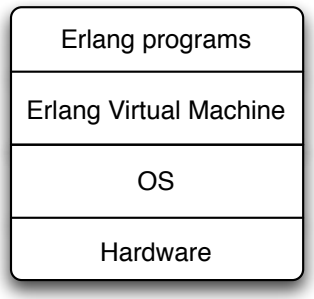

Figure 2.6: Erlang platform independency

2.7.1 Megaco-stack

OTP also includes the MEGACO/H.248 stack. It has encoders and decoders for MEGACO messages. It makes it easy to build messages in Erlang records and then run them through the encoder, the same for decoding messages. The encoder takes an Erlang record and returns a binary which one has to convert to a string in order to send it to the MGW. Decoding messages are done in the same way but the other way around.

Chapter 3

Tools

In this chapter we will describe the tools used by us and by the department for testing at Ericsson. These are all applications written in Erlang and used for different types of testing. We will also describe the tool used by us to develop our application, namely Quickcheck and gen_server. But first we will describe the environment in which we implemented our software and ran our application.

3.1 Test environment

The test environment was limited to what they used at Ericsson. In this case we used a Sun workstation running Solaris 8 with 1 GB of memory. We ran Erlang version 5.5.2.1 because of some limitations in the SCTP stack used which was developed at the department and we only got it to run on this version of Erlang.

We used a simulated MGW running on the same computer where we were developing our application. The reason for using a simulated MGW was that it was easier for us to setup and configure and if anything would go wrong we would limit the damage. We could then also use different versions of Ericsson's MGW software to test our application against.

3.2 Quviq's Quickcheck

Quickcheck[1] is a tool written in Erlang. Its main focus is on constructing test cases automatically and simplifying failing test cases. It consists of several modules[2] and features to simplify building your own application ontop of Quickcheck.

To use Quickcheck one writes properties that are expected to hold and runs random test cases drawn from a given set. If successfull Quickcheck then reports

[1] www.quviq.com

[2] refers to beam files, compiled erlang files which are executable

pass or when a test cases fails it will print out the failed test case. For easier understanding we will show a small example taken from the course material[3].

```
quickcheck_example() ->
         ?FORALL(Int, int(),
                ?FORALL(List, list(int()),
                        not lists:member(Int,lists:delete(Int,List)))).
```

This may look difficult to understand for a novice Erlang/Quickcheck user but bare with us. What this function does is to construct randomly long lists with random elements[4] in them. Then it checks each list to see if an element is accuring twice in any list. It will do this 100 times[5]. If none of the 100 lists contains the same element twice Quickcheck will show:

```
3> eqc:quickcheck(example:quickcheck_example()).
..................................................
........................................
OK, passed 100 tests
true
```

It is very easy to run 100 new test cases. A test case in this sense is a list of some length with some random integers which is checked to see if any element is repeated in the list, The element is also randomized from test case to test case. Of course the test cases can be more complex but we are just trying to illustrate an example. On the other hand if a test case fails, Quickcheck will show the following:

```
3> eqc:quickcheck(example:quickcheck_example()).
..........................................
Failed! After 42 tests.
-8
[5,-13,-8,-8,9]
Shrinking..............(16 times)
-8
[-8,-8]
False
```

First -8 indicates the element it is looking for and the list [5,-13,-8,-8,9] that failed. Then it starts the shrinking process that will shrink, in this case, the failing list to a minimum failing list. So the minimum failing list will be [-8,-8].

This is the essence of Quickcheck. One creates a property that will hold, in this case a list cannot contain the same element twice. Then when each test case

[3] A two day course given by Thomas Arts and John Hughes

[4] the elements being integers in this case

[5] if one does not restrict it by using numtest, see Quickcheck manual

is run it, is evaluated to fit one's property. If all cases holds then it will pass otherwise fail. The failing test case will be shrunk into the smallest sequence that would fail the property.

Quickcheck also provides other features like a set of random generators that can be used to construct test cases and its own state machine which we will look closer at now.

3.2.1 Generators

Quickcheck provides generators that can be used when implementing your application. There are many different kinds of generators and we will go through some of them later. When using Quickcheck, one uses these generators to write properties to generate random data. The generators have built in shrinking behavior and this is used by Quickcheck when shrinking. In the example above the generators int() and list() are used. Int() generates a random number whereas list() generates a list of elements generated by its argument. Here are some other Quickcheck generators:

choose/2	generates a number between M and N
frequency/1	does a weighted choice between elements in a tuple list
elements/1	chooses a element from a list of elements
bool/0	generates a random boolean
char/0	generates a random character

Table 3.1: Quickcheck generators

3.2.2 eqc_statem

eqc_statem is a state machine that is provided by Quviq. It works similar to the gen_server[6] in terms of interface and callback functions. The interface functions are listed in table 3 (there are several more functions not listed. They can be view in the documents accompanied with Quickcheck).

These functions are used to build sequences and run these sequences while checking each command in the sequence is legal and discarding those who are not. This is done with preconditions set by the user (so whether something is legal or not is up to the implementer of precondition).

If a precondition does not return true then the command will be discarded and thereby not included in the final sequence. When a legal sequence of legal commands (a sequence is to be viewed as a test case) has been built it is executed and

[6]http://www.erlang.org/doc/doc-5.5.4/doc/

initial_state/1	initiates the state machine and takes one argument
precondition/2	returns `true` if call C can be performed in this state S
postcondition/3	checks the postcondition of a call that was executed
next_state/3	this function is used to update the state

Table 3.2: Quickcheck's eqc_statem interface functions

checked against its postcondition. As in the case of precondition, postcondition fails the test case if a postcondition is not met. Also done, if the postcondition fails, is to shrink and discard commands which do not contribute to the failure. Using this feature one can then find the minimal sequence that fails this specific test case.

Building a sequence of commands are done in a so called symbolic state that which at execution time is replaced by the actual values returned by the command reply. A symbolic state can be as follows:

```
[{var,1}, {var,2}, {var,3}, {var,4}]
```

This is before any command has been executed. Each `{var,x}` can be seen as a symbolic command in the symbolic sequence. The above example is very simplified and cannot be used as a real symbolic state since one needs to know what to execute and replace `{var,x}` with at execution time. A better example given in the material for Quickcheck is shown below:

```
{set,{var,1},{call,erlang,whereis,[a]}}
```

Here `call` is the symbolic call to be performed when executed. This call performs `whereis` which is a BIF included in the Erlang module and the returned PID replaces `{var,1}` . So in the call one must specify module (erlang in this case), function (BIF's `whereis`) and parameters (is a list containing parameter a in this case).

When this call is executed it will be replaced by the actual PID of the return value of the BIF `whereis` . This will give you a state in which every symbolic call has been replaced by the actual return value of the executed symbolic call. In the case above you will have a list of PIDs when all the calls have been executed.

3.3 gen_server

It is a generic server which your module has as a behavior, it is used to implement a server-client relation. Your client part has some interface functions which call upon a set of callback functions within the server part. Their relation is depicted below:

```
Interface functions          Server functions
-------------------          ----------------------
gen_server:start     ----->  Module:init/1
gen_server:call      ----->  Module:handle_call/3
gen_server:cast      ----->  Module:handle_cast/2
```

There are many more interface functions and callback functions, these are just some. If the gen_server was successfully created it will return its PID. As seen above the start function is used to initiate the gen_server. The interface function `call` is a synchronous call to the gen_server which then waits until the reply comes or a timeout occurs. The opposite is the asynchronous `cast` which does not wait for a reply. The gen_server also holds a state which you only can modify within the server functions.

3.4 Protocol_tester

The protocol_tester is used to do conformance testing and can run any transport protocol supported by Erlang. You configure the protocol_tester in a text file where you specify its parameters. There are two main areas to configure: global parameters and socket parameters. Global parameters set timeouts and delays, for example how long it is going to wait for a reply before it fails a test case. Socket parameters can be set for each socket such as destination and source IP address, destination and source ports and which transport protocol to run[7].

To write a test case for protocol_tester you use the same text file. Each test case starts with something to distinguish it from others, a unique name. A test case can be a sequence of messages or just one. A simple test case could be to send an `Add` message with some specific parameters. The test case would then specify which parameters to send and what kind of response it expects from the MGW. If the reply message was according to the test case the test case is passed, if not it is failed.

Since each test case is specified in this way the protocol_tester uses this when writing a test case. You specify what you want to send and what to expect as a reply. This is a simplification of what you need to write for a test case for the protocol_tester because there are values that one cannot know before execution like `terminationID`, which is set by the MGW when receiving an `Add` message. These values are stored by the protocol_tester in variables and can be used as symbolic links when writing a test case but they will be replaced by actual values at execution time. A test case might look similar to this:

[7] TCP, UDP or SCTP are supported

```
###############
TCID:my_test_case
SEND:socket1:
MEGACO/2 <MGC>  Transaction = 3 {Context = C1 {Priority = 14, Add = T1 {Media
{LocalControl {Mode = SendReceive, tdmc/ec = off }}}, Modify = T2 { Media
{LocalControl {Mode = SendReceive}}}}}
END:
RECV:MEGACO/2 <TGW> Reply = 3 { Context = C1 { Add = T1, Modify = T2}}}
END:
###############
```

The protocol_tester uses strings to write messages and is not dependent on the Megaco stack. This makes it more flexible and one could send whatever message you would like in whichever protocol that is text based. There are also functions within protocol_tester that can be used to send messages from other modules via protocol_tester and also a function which allows you to send the reply to another module. These were used in our implementation as we will show in later sections.

3.5 STASI

STASI stands for STochAStic Interrogator. It extends the protocol_tester by making test cases more randomized. It provides a string generator which generates a random set of strings according to a function call made with parameters. This enables you to repeat a test case and for each time the parameters are changed. This makes the test coverage larger and also extends the usability of the protocol_tester. Here are some functions provided by STASI, some are very similar to what is provided by Quickcheck:

- **oneof/1:** oneof(List) will choose a element in the list
- **weight/1:** weight(tuple_list) will choose one of the tuples in a list of tuples. Each tuple has a element and a weight. The weight must be between 1-100 and the total sum of all weights in the tuple list must be equal to 100
- **choose/2:** choose(min, max) chooses a integer between Min and Max where Max must be larger or equal to Min
- **optional/2:** optional(value, frequency) is used to determine if a value should be included or not with the frequency given

3.6 GEGGA

Gegga, GEneric meGaco simulAtor with an extra G, is a highly configurable tool. As with the protocol_tester it is used to simulate the MGC. It uses the protocol_tester to establish the H.248 connection to the MGW. Gegga can be used to do more elaborate testing. Here you can build sequences of messages by calling functions with a JobID identifying your sequence. Each message is identified by a message number in the queue of messages in your sequence. Because of the flexibility it somewhat cumbersome to build your messages or message sequence. To set parameters in your message/messages one has to call the according function. For example, to set the H.248 version you call function set_version(Version). This is how you build your message by calling functions for each parameter that can be set in a message. Some functions takes more parameters in a tuple list.

Listed below are some of the functions (since there is a lot of them we are not going to show them all).

- **set_controller_id/1:** set_controller_id(ControllerID) will set the controllerID for all messages
- **set_termination_parameters/4:** set_termination_parameters(JobID, MsgNo, TermID, Parameters) takes JobID to specify which job, MsgNo to specify which message in the queue, the terminationID and a set of parameters.
- **set_localcontrol_parameters/5:** is used to set the localcontrol parameters in the localcontrol descriptor
- **set_context_parameters/3:** is used to set context parameters such as priority or emergency.

These functions are according to the ITU H.248 specifications where the parameters are the different descriptors that can be set in a message. So calling `set_local_control/5` you can specify all parameters that can be set under the localdescriptor in a message such as version, IP address, port, codec, ptime and other things related to this descriptor.

Chapter 4

In-depth Study: automated testing using Quickcheck

In this chapter we will go through design and implementation. We will begin with design issues giving the structure for our system. Later we will dig deep into implementation issues where we describe our application built upon Quickcheck and some of the other tools described earlier.

4.1 Design

As we have discussed earlier we are going to use Erlang/OTP programming language in our implementation. Our design is built upon programming three different modules. Each module will be explained in detail under the topic implementation.

For setting up the SCTP connection to the MGW and to send and receive messages to and from the MGW we used the `protocol_tester` (PTA), a tool developed at Ericsson in Telefonplan. The main reason for using the PTA was to reduce the amount of time in programming the necessary code for setting up the connection, to concentrate on our main task and to further implement some additional features.

The three modules are shown in the picture 4.1 along with the PTA and the MGW, where `proxy` resides between PTA and our simulated MGC. `proxy` module acts as its name suggests as a proxy sending and receiving messages to the PTA which in turn sends and receives messages to the MGW. The `mgw_eqc` constructs h.248 messages by calling different functions in `h248_eqc` module which in turn uses H.248 records. The `mgw_eqc` then calls functions in `proxy` module with the message to be sent as a parameter, and calls another function in `proxy` module to get the replies with the number of expected replies to be sent as a parameter.

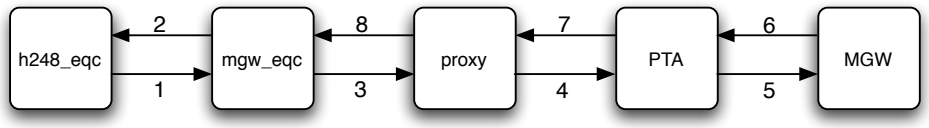

Figure 4.1: Message passing from simulated MGC to MGW

Our main module is `mgw_eqc` which actually in this case is a simulated MGC. Quickcheck's state machine (`eqc_statem`) and generators (Quickcheck generators and Erlang function generators) are used here. We have chosen to separate `eqc_statem` and megaco records in different modules to keep the whole system effective and as easy to understand as possible. Dividing a big job into smaller jobs makes the whole system easy to manage and achieve better and faster error handling.

Our three modules together with PTA and Erlang/OTP acts as a simulated Media Gateway Controller, as shown in picture 4.2.

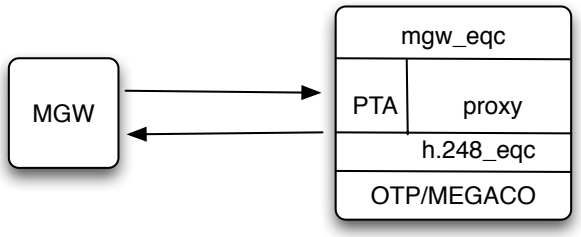

Figure 4.2: Design structure overview

4.2 Implementation

4.2.1 proxy

The module `proxy` is used to handle message receiving from PTA in an asynchronous way (non-blocking) and also receiving messages from `mgw_eqc` and further sending them to PTA. We used the Erlang `gen_server` module for message handling. We wanted to have a separate module for handling sending and receiving. This was done mainly because implementing a `gen_server` together with Quickcheck's own state machine was almost impossible, and this would have rendered the whole system non understandable. Another idea that we thought of was using the send and receive Erlang mechanisms in `mgw_eqc` but that required spawning off a new registered process from `mgw_eqc`. This also didn't work at

all together with Quickcheck. With that in hand this left us with the choice of using `gen_server` in a separate module.

Sending and receiving messages to and from PTA is asynchronous using `gen_server:cast()` function while returning the replies to `mgw_eqc` is synchronous, because when we send a H.248 message containing several transactions there would be several separate answers to each transaction from the MGW. We didn't know when all the replies would come from the MGW, so instead of using a delay function waiting for all replies, `mgw_eqc` will simply block until all replies have come. Picture 4.3 shows the message flow between `mgw_eqc` and `proxy` and between `proxy` and PTA. As seen from the picture, `mgw_eqc` calls a function

Figure 4.3: Message flow between different modules

`send_packet` (Packet) in `proxy` module where Packet is the H.248 message to be sent. `proxy` then calls a function `send_packet` (Packet) in PTA. One could ask the question of why we don't send the message directly from `mgw_eqc` to PTA. The answer to that question is we wanted a uniform system with `proxy` handling message transportation to and from the PTA. The replies are handled in a different way, `mgw_eqc` calls `get_replies` (N) in `proxy` and blocks until it gets a reply. Here we were forced to use `proxy` to get the replies back, so that `mgw_eqc` can block until all the replies have come. `proxy` gets one reply at once from PTA non-blocking and saves all the replies in a state inside `proxy` . Only when all the replies have been received the `proxy` can then send these replies to `mgw_eqc` and unblocking it.

The other job of the `proxy` module is to set up a connection with the MGW. This is done by sending a `ServiceChangeReply` with the proper parameters to the MGW. Sending a `ServiceChangeReply` is done explicitly and not in Quickcheck. After getting a green light (when a connection is established between the MGC and the simulated MGW, the MGW shows a green light telling that the connection is up) the MGW is ready to receive messages.

4.2.2 mgw_eqc

This is our main module that acts as a simulated MGC composing Megaco messages using Quickcheck macros and generators to randomize the message sequence. The module uses Quickcheck's own state machine, `eqc_statem` , as discussed before. The Megaco messages that we implemented were `Add` , `Subtract`

, `Subtract_all` , `Modify` and `Move` . Though `Move` isn't fully working due to some logical errors in our code that requires digging deeper into `eqc_statem's` functionality (but works properly if it is included in a predefined sequence).

Besides of these basic Megaco messages we implemented commands for sending multiple transactions, contexts or actions in the same message sent to the MGW. The types of messages sent in these commands are too totally randomized. For example sending two `Add` commands or one `Add` and one `Subtract` command.

When sending a message containing several transactions the MGW sees this message as multiple messages, instead of one message, each one has its own `transactionID` . The MGW processes each one of these transactions totally independent of the other ones. The problem here lies in the fact that we expect the replies for the transactions to come back as the same order as we send them, something that the MGW doesn't guaranty. For example if we send an `Add` transaction followed by a `Subtract` transaction for the same `terminationID` , the MGW may process this in the reverse order which of course will result in an error sent by the MGW. The solution to this problem was to check all the possible permutations of a message containing several transactions to be sent and doing this in the precondition. In our example the message containing an `Add` and a `Subtract` transaction for the same `terminationID` will never get sent because permuting this sequence (`Subtract` and then `Add`) will fail in precondition.

Each command is built using generators that uses Megaco records residing in `h248_eqc` module. The generators are needed to randomize different parameters such as a `contextID` , a `terminationID` and a `streamID`.

Each test case includes some random number of commands (messages). This randomization is weighted using a Quickcheck generator called `frequency`. Whenever `frequency` is not choosing the atom stop the sequence will continue growing. Changing the value of `frequency` results in changing the length of the command sequence and the generator `oneof` is used to choose between one of the available commands. `eqc_statem` will check the legality of each command chosen before including it in the test case. After sending each command the response is checked to see if it is legal and its state is updated. When the whole test case is passed the state is erased and a new state is initialized ready to save data belonging to the next test case.

The module can generate totally random sequence of Megaco messages as discussed above but one can also define a legal sequence of messages that isn't randomized. The only randomization that occurs, when using a predefined sequence, is within each message to be sent for example SDP parameters. The state

is initialized with the parameters that a user can type in using a shell. These parameters are then passed to `h248_eqc` when constructing the messages. Many problems and complexities have been experienced during programming this part of the system mainly because of the complexity that `eqc_statem` is built upon consisting of symbolic state and dynamic state compound at run time.

4.2.3 h248_eqc

This module defines functions that use Erlang records. Using records can be programmed simpler and faster than using strings but less flexible. The module defines `Add`, `Subtract`, `Move` and `Modify` message records with different in parameters. Generators are used to randomize the SDP part and the local/remote descriptors' part. This module can handle both positive and negative testing. But only SDP parameters can be set to negative (wrongly defined) parameters because they are built as strings. Everything else that isn't expected by the Megaco stack will be unacceptable by it and will return errors. That is the main negative issue against using MEGACO.

The `Add` messages can have `LocalControl,Local` and `Remote` descriptors or only the `Local` descriptor's part. We send `Add` messages with `contextID` equal to "$" (choose wildcard, that is the MGW decides the ID), and `terminationID` equal to "$" which too are decided by the MGW. The `Modify`, `Subtract` and `Move` commands chooses a random `contextID` and `terminationID` to work with. `Modify` commands can also randomize the SDP part and the `Remote` descriptor part.

Using Quickcheck's generators especially with SDP can be a tricky part. The generators do not really generate real values until at run time. When programming with generators one can not use something like:

A = choose (1,5); % Chooses a number between 1 and 5 and bind it to A

compute(A); % Calls a function called compute with the value of A

And then use A's value in the following code sections (as shown in the example with compute(A)). This will result in a compiling error complaining about A is unbound, simply because A does not get its value until run time. The solution to this kind of programming paradigm is to choose Quickcheck's own macros to get around these kinds of problems, for example `?FORALL` or `?LET` that saves the symbolically generated value in a variable and this variable can then be used only inside the body of these macros for example:

?LET(A, choose (1,5), compute(A)); % Choose a value between 1 and 5 and bind it to the variable defined under the macro ?LET, computes called with the

symbolic value of A

There are two kinds of messages, messages sent over IP and messages sent over TDM channels as discussed earlier. For an `Add` message to be sent over a TDM channel one needs to define which channel, that is a physical channel, to send the message over. These channels can be randomized using an appropriate generator.

4.2.4 H.248 message generation

This is what we chose to randomize in our implementation. Common for all messages is the Megaco version number which is the version that the MGW currently uses. Our simulated MGW uses version number 2 and a device name which we chose to call mgc.

Add

Add a termination to an existing context or to a new context `ContextID:`Identifier of context. We used the wildcard "$" which means the MGW chooses an ID. `TerminationID:` Identifier of termination. We used this expression `{megaco_term_id,true,"$"}` , where `"$"` means the MGW chooses an ID, for positive testing and the expression `{megaco_term_id,true,"*"}` , where `"*"` means all terminations, for negative testing. Randomized data: These values of the randomized data are for positive testing only. Proper values can be given for negative testing.

Type	Generators	Randomized Type/Value
TransactionID	own function	Time dependent
ContextID	elements()	Picks one of the terminations from the context previously chosen or a new one
StreamID	choose()	0-65535
Number of Streams	choose()	1-2
Stream Mode	frequency() and vector()	Weighted using frequency and paired together with the streamID using vector, the values are: Inactive, sendOnly, recvOnly, sendRecv
ReservedGroup	oneof()	ON/OFF
ReservedValue	oneof()	ON/OFF
LocalControlDescriptor, LocalDescriptor, RemoteDescriptor	frequency()	All three of them, local and remote only or local only
IP number	noshrink() and choose()	1-254
Port number	noshrink() and choose()	1024-65535
SDP number	noshrink() and choose()	96-127
Ptime	noshrink() and elements()	Depending on codec
Codec	oneof()	Depending on codec

Table 4.1: Randomized data in an Add message

Modify

Modify a termination in an existing context. Randomized data: These values of the randomized data are for positive testing only. Proper values can be given for negative testing.

Type	Generators	Randomized Type/Value
TransactionID	own function	Time dependent
ContextID	elements()	Picks one of the contexts saved in the state
TerminationID	elements()	Picks one of the terminations from the context previously chosen
StreamID	elements()	Picks one of the stream ids saved in the state
Stream Mode	elements()	Inactive, sendOnly, recvOnly, sendRecv
ReservedGroup	oneof()	ON/OFF
ReservedValue	oneof()	ON/OFF
LocalControlDescriptor, LocalDescriptor, RemoteDescriptor	frequency()	All three of them, local and remote only or local only
IP number	noshrink() and choose()	1-254
Port number	noshrink() and choose()	1024-65535
SDP number	noshrink() and choose()	96-127
Ptime	noshrink() and elements()	Depending on codec
Codec	oneof()	Depending on codec

Table 4.2: Randomized data in an Modify message

Move

Move a termination from one context to another one. If the context which has this only termination moved from, this context is deleted. Randomized data:

Type	Generators	Randomized Type/Value
`TransactionID`	own function	Time dependent
`ContextID`	`elements()`	Picks one of the terminations from the context previously chosen
`TerminationID`	`elements()`	Picks one of the terminations from one of the contexts saved in the state except the context previously chosen

Table 4.3: Randomized data in an Move message

Subtract

Subtract a termination from an existing context. If the context has this only termination then delete this context.

Type	Generators	Randomized Type/Value
`TransactionID`	own function	Time dependent
`ContextID`	no randomization	Extracts the `contextID` from the chosen `terminationID` saved in the state
`TerminationID`	`elements()`	Picks one of the terminations from the context previously chosen

Table 4.4: Randomized data in an Subtract message

Chapter 5

Results

This chapter will show the result of our work. Most of our work has been about implementing a prototype tool based on Quickcheck. The result is based on theoretical analysis more than on practical ones when comparing Quickcheck with different systems. The goal was to implement at least 5 test cases. We managed to implement 17. To construct our modules[1] we had to look at the existing test cases to see which ones can be implemented in our tool because many of the test cases can not be adjusted with the way our tool is built.

A typical test case from the test suites can be to send an `Add` message with some specific parameters that are specified in the test case. Parameters such as codec, IP address, port number or to use a specified descriptor in the MEGACO framework. It can be more elaborate by specifying a sequence of messages. In the test case there is also specified what to expect from the MGW in the reply.

A test case is defined in a test suite that gathers similar test cases. Each test suite can consist of several test cases ranging from 20 to maybe 200 and each test suite is defined to test something specific, like for example traffic test suite where the main goal is to test traffic flow in the MGW. There are many more different types of test suites. We took our test cases mainly from two test suites:

1. H.248 MP Protocol Verification (TS_ID: 1/152 41-CRA 119 0194 Uen)
2. H.248 for MGW 1.1 (TS_ID: 48/152 41-CRA 119 0194 Uen)

As seen above one of the test suites (1) were for the Media Proxy, but some of the messages are applicable for both MGW and MP. These test suites were chosen by our supervisor. There where some other issues also, as discussed earlier, with the Megaco stack that made the implementation of some test cases impossible. The test cases we managed to implement is listed in the Appendix A, attached are also the real test cases.

[1] h248_eqc, mgw_eqc, proxy

We were not able to find any bugs when testing these test cases in AD13[2]. Our main focus was to show that this tool could replace or maybe act as a supplement to existing tools.

Running a test case is done by giving the parameters according to the test case specifications. An example is shown below:

```
mgw_eqc:start(1, [add], [{version, 2}, {port,4000}, {ptime, 20}]).
```

Lets just briefly go through what this command says. First the module is specified (`mgw_eqc`) then the function within the module is called (`start`) along with some parameters that is specific to the test case that is run. The first parameter (`1`) tells our tool how many runs to perform. The second parameter (`[add]`) tells our application that it is a predefined sequence that should be run with the first message being an `Add`. Then last is the list of parameters telling what kind of parameters to put in the `Add` message. Many more parameters exists but they are explained in more detail in the `show_help` function included in our module `mgw_eqc`.

When running a negative test case one also has to specify what error code is expected so that the answer would not fail in postcondition as discussed earlier. Interestingly is the fact that we only used our tool to test small sequences of messages, even though it is mainly used to test large sequences.

The reason for not doing a practical comparison between the two systems is that the test cases do not change and the expected bugs will not differ between the systems. That in hand, makes the practical comparison a lot harder to procure.

During our coding phase we tested the MGW after each section implemented. For example testing the MGW after implementing the `Add` message. We used, for our testing, old versions of the MGW and most of the bugs that were already found we discovered them with our tool. But unfortunately we were not been able to find any new ones.

Our testing was based on constructing short message sequences by weighting the `frequency` generator to a small value. The reason for testing only small sequences was that we were concentrating on discovering practical errors, that is scenarios that can very likely happen in a real life situation. Long sequences are not likely to happen in real life and can be seen as theoretical testing scenarios. This is very easy to do with Quickcheck. Test cases implemented: see Appendix A.

[2] AD13 was at the time the latest version of the MGW

Chapter 6

Conclusions

The goal of our thesis was to make a comparison between the testing systems used at Telefonplan today and a tool based on Quickcheck. Here follows a summary of our conclusions.

Quickcheck is a powerful tool that can be used generally for testing text based protocols, like SIP, H.248, SDP etc. It can create randomly long sequences with random message data and can shrink failing sequences to a minimum failing sequence. This makes it powerful. It also provides other useful instruments, like generators that can be used without Quickcheck. Creating test cases is easy with Quickcheck and retesting them is easy. There is also the possibility to implement predefined sequences as we have shown. But there are also some disadvantages.

The main disadvantage is that it can be hard to guarantee that a specific test case has been run. But, as we see it, Quickcheck is not made to test specific test cases. It should be used to run randomly and create sequences on its own and one could maybe adjust some parameters to "point" Quickcheck in a certain direction. So if one wants to test specific test cases it is preferred to use another tool or as we did to implement the possibility to run predefined message sequences and the ability to set some parameters instead of using the provided generators. Also it could be nice if one could build a feature where one could feed it some parameters that belong to a test case and check these against the symbolic state in Quickcheck and in this way guarantee that a message or messages with some specific parameters has been run. So if Quickcheck generates a message with these parameters and gets a positive reply, it passes the test case otherwise it runs more test cases until it has generated a message containing the provided parameters.

For now Quickcheck can act as a compliment to the existing tools at the test department. We managed to implement a prototype that can be used to run conformance testing fairly easy but more work needs to be done. What Quickcheck

provides that the department did not have before is the ability to run long sequences of messages. This can be controlled by weights inside Quickcheck. To construct long test sequences before was extremely cumbersome with either `protocol_tester` or GEGGA, as described in the tools section. But more needs to be done to use Quickcheck at the test department as a tool used daily when product testing new releases.

Cost effectiveness

The test department at Telefonplan has a couple of tools that is for free. Replacing these tools with Quickcheck requires the department to buy 10-11 license which is very expensive. But as we said in earlier, our Quickcheck tool is mainly considered as a complement for today's GEGGA, STASI and `protocol_tester`. The more realistic and cost effective solution is to have 2-3 license and to have the users sharing them in a distributed system. This means that only 2-3 people at most can use Quickcheck at once. Others that want to use Quickcheck can for instance wait for the license in a shared queue.

Support and the two days course for Quickcheck add some additional costs, meanwhile support for STASI, PTA and GEGGA are for free.

Chapter 7

Future work

This section will be used to describe what more can be done to improve our prototype. All things mentioned here are to improve Quickcheck's use at the test department at Ericsson.

There is room for a lot of improvement to our prototype. First one can implement all messages that are defined in the H.248 framework. Another thing that will improve and extend Quickcheck's usability is to implement your own encoder/decoder so that the Megaco stack does not limit the test cases you want to run. This achieves a more generic tool but can be quite messy programming when constructing messages with strings for example.

One can extend the state to save considerable data such as IP and port numbers which are needed for call setup scenarios.

Integrating the system in VTS[1] and maybe emerging the three modules into one for the sake of not needing to deal with three modules all the time (like GEGGA and PTA).

[1] Visual Test Server

Chapter 8

Discussion

We believe that Quickcheck is worth developing and maintaining. It is a great tool that can be used for conformance testing here at Telefonplan. But maintaining and developing the tool that we have implemented require a good knowledge of Erlang programming and of "Quickcheck programming". What we mean about Quickcheck programming is the ability of understanding all the generators, macros, `eqc_statem` and functions in order to use these for programming a proper tool.

A system tester can easily test all the test cases implemented in the tool. Any other test case that is not supported by our tool requires digging deep into the code to make it work. This can be a very difficult and an unwanted task to for a system tester.

Quickcheck cannot and preferably won't be seen as a tool for testing predefined test suites. This is not what Quickcheck is made for. It must be seen as a stochastic automated tool for testing totally random sequences of totally random length. A system tester can, before going home from work, type a number of tests (for instance 10 000) that he/she wants to run and coming to work the day later to check the results and the eventual bugs.

We interviewed some people who works or worked with Quickcheck before here at Telefonplan. The overall opinion was that all of them criticized the documentation to be poor and in need of development. Understanding the generators, macros and other Quickcheck semantics is practically impossible to do without attending to the course. Programming with Quickcheck also requires continuous support by email, phone or even supporting at place. All of the people interviewed said that Quickcheck is worth developing and working with if they had an opportunity to do so in the future.

References

[1] CISCO. Accessed may 2007, http://cio.cisco.com/warp/public/707/4.html.

[2] Erlang.org. Accessed may 2007, http://www.erlang.org/project/megaco/encoding_comparison-v4/index.html.

[3] Exforsys Inc. Accessed may 2007, http://www.exforsys.com/tutorials/testing/automated-testing-advantages-disadvantages-and-guidelines.html.

[4] Internet Engineering Task Force (IETF) mailing list. Accessed may 2007, http://www1.ietf.org/mail-archive/web/megaco/current/msg04427.html.

[5] OSB OutSourceBazaar. Accessed may 2007, http://www.outsourcebazaar.com/index_Article_AutomatedVsManualTesing.html.

[6] Prof. Dr.-Ing Erwin P. Rathgeb. Accessed may 2007, http://tdrwww.exp-math.uni-essen.de/inhalt/forschung/sctp_fb/.

[7] IETF RFC4566. Accessed may 2007, http://tools.ietf.org/html/rfc4566.

[8] Joakim Johansson Thomas Arts, John Hughes and Ulf Wiger. Testing Telecoms Software with Quviq Quickcheck. Technical report, October 2006.

[9] International Telecommunication Union. Gateway control protocol: Version 2. Technical report, 05 2002. www.itu.int.

[10] Wikipedia.org. Accessed may 2007, http://en.wikipedia.org/wiki/IP_Multimedia_Subsystem.

[11] Wikipedia.org. Accessed may 2007, http://en.wikipedia.org/wiki/Time-division_multiplexing.

Appendices

.1 A

Here follows the test case tested from the two test suites. These are internal Ericsson documents and will therefor not accompany this thesis.

H.248 for MGW 1.1 (TS_ID: 48/152 41-CRA 119 0194 Uen)	H.248 MP Protocol Verification (TS_ID: 1/152 41-CRA 119 0194 Uen)
3.1.1	3.2.1
3.1.2	3.3.1
3.2.1	3.3.2
3.2.2	3.3.3
3.2.3	3.4.1
3.3.2	3.4.6
3.3.3	
3.4.4	
3.4.7	
3.4.8	
3.4.9	