OULUN SEUDUN
AMMATTIKORKEAKOULU

Kimmo Karppinen

AUTOMATED GENERATION FOR TEST INTERFACE

AUTOMATED GENERATION FOR TEST INTERFACE

Kimmo Karppinen
Bachelor's Thesis
Autumn 2013
Degree Programme in Information Technology and Telecommunications
Oulun University of Applied Sciences

TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, Langattomat laitteet

Tekijä(t): Kimmo Karppinen
Opinnäytetyön nimi: Automated Generation for Test Interface
Työn ohjaaja(t): Ensio Sieppi
Työn valmistumislukukausi ja -vuosi: Syksy 2013 Sivumäärä: 40 + 1 liitettä

Opinnäytetyön aiheena oli toteuttaa Nokia Siemens Networks:lle sovellus, joka päivittää rajapintaviestit, kun niihin tulee muutoksia. Lisäksi työn tavoitteena oli nopeuttaa testiympäristön päivitystä, koska päivitys pysäyttää testaamisen siksi aikaa, kunnes päivitys on tehty.

Työssä käytettiin mallina sovellusta, joka luo automaattisesti rajapintaviestejä, vaikkakin viestien muoto on vääränlainen tähän työhön liittyen. Kaikki tarvittava tieto ja tavoitteet sovellukselle tulivat ryhmältä, joka ottaa tämän uuden sovelluksen käyttöön. Viestien rakenne sovittiin tarkoituksenmukaiseksi, sovellus antaa viesteille oletusarvot ja tallentaa viestit tietokantaan.

Työn tuloksena on sovellus, joka luo sovitulla rakenteella rajapintaviestejä. Lisäksi sovellus päivittää automaattisesti viestejä, jos niihin tulee muutoksia tietokannassa. Sovellus korvaa osan manuaalisesta päivitystyöstä, joten sovellusta tullaan kehittämään lisää, jotta loputkin manuaalisen osan työt saataisiin automaattisen päivityksen piiriin.

Asiasanat:
Automaattinen, rajapinta viesti, luominen, testaus

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology and Telecommunications, Wireless devices

---

Author(s): Kimmo Karppinen
Title of thesis: Automated Generation for Test Interface
Supervisor(s): Ensio Sieppi
Term and year when the thesis was submitted: Fall 2013 Pages: 40 + 1 appendices

---

The objective of this thesis was implement application for Nokia Siemens Networks which would update the interface messages when they are changed inside the repository. One of the objectives was also to speed up the update procedure because testing cannot take place while changes in the repository are being made.

As a basis for this new application an existing application was used even though its message structure is different compared to what is needed by this new application.
All the needed information and objectives came from the team who will use the application. The objectives for this new application are that it should generate the messages in a specific format and give the messages default values and store them to the repository.

As a result of this work a new application is implemented, which generates messages in a specific format, updates the messages when change happen them inside of the repository. The application replaces the part of manual update procedures which take place when the repository is changed. In the future this new application will be improved so that the rest of the manual updates will happen automatically.

Keywords:
Automated, interface message, generation, testing

ACKNOWLEDGEMENTS

I want to thank my school supervisor Mr. Ensio Sieppi for the advice he offered during this work.

In addition, I want to thank the company of the thesis, Nokia Siemens Networks, for this opportunity. Special thanks go to my supervisor Mrs. Kirsti Simula from Nokia Siemens Networks who provided me great support during this work. I also want to thank the two teams who helped me with this work.

2013, Oulu.

Kimmo Karppinen

TABLE OF CONTENTS

1 INTRODUCTION

The goal for this thesis was to create an automated interface generation application for the Nokia Siemens Network testing environment. Interface generation is a procedure where the application creates interface messages between different systems. The messages are generated from the source repository, where all messages are defined including their properties. The messages are required be the in CU (Control Unit) format, which controls the test environment.

The existing message generation solution includes manual procedures which are time consuming and error prone. The messages are manually updated and if there is need for new message, then the testing team creates new message and gives the default values for messages and finally stores the messages in the repository. With the new application all those manual procedures will be automated.

One of the biggest challenges was to generate the interface messages structure into the CU format, because the application uses another already existing automated interface generation as a starting point and therefore the interface messages are different format. The new application should work together with other tools in the system and the generated messages be used by those tools. The possible incompatibilities of the message structures require adaptation.

2 DIFFERENT TESTING TECHNIQUES

Testing techniques are used to verify the system functionality, protocol, etc. The test environment where this work is done, uses these testing techniques to verify the system functionality.

2.1 Black-box testing

Black-box can also be called functional testing. Tester only knows what are the system's input and output. The tester does not know how the system works. The tester views the system as black-box and is unconcerned of the internal structure of the system. The tester tests the system functionality against the specification.

The advantage in black-box testing is that it tests if the system works like it is supposed to.

The disadvantage is that exhaustive input testing is not possible because it would require every input condition or combination to be tested. In addition, because there is no knowledge of the internal structure, there could be errors or mischief. Those cannot be detected with black-box testing. (1, p.29)

2.2 White-box testing

White-box testing is also called structure testing. Test is designed by examining the internal structure. Test data is driven by examining the logic of the system, without concern of the system requirements. The tester must have knowledge of the internal system structure and logic either by studying it or asking from maker of the system.

The advantage is that it tests the produced code. The errors or deliberate mischief are more likely detected because internal structure and logic is known.

The disadvantage is that it does not verify if the specifications are correct. Missing paths and data-sensitive errors are not detected. (1, p.30)

2.3 Gray-box testing

Gray-box testing is a combination of black- and white-box testing. The tester must study and understand both the requirements and internal structure and if necessary contact the developer of the system. An example of gray-box testing is that the tester notices that one certain functionality is reused in an application. The tester contacts developer to understand the internal structure then the tester can remove the unnecessary test because it may be possible to test the functionality in one test. (1, p.30)

2.4 Unit-test

Unit-test tests particular functions or modules. A programmer typically does the unit-test not a tester as it requires detailed knowledge of the internal program design and code. Unit-tests are not usually done unless the application is well designed with tight code. (1, p.32)

3 TESTING INFRASTRUCTURE

The environment, where test persons test and verify the functionality of the systems, is introduced in next chapters. The new implemented application is used in this the environment.

3.1 DSPi

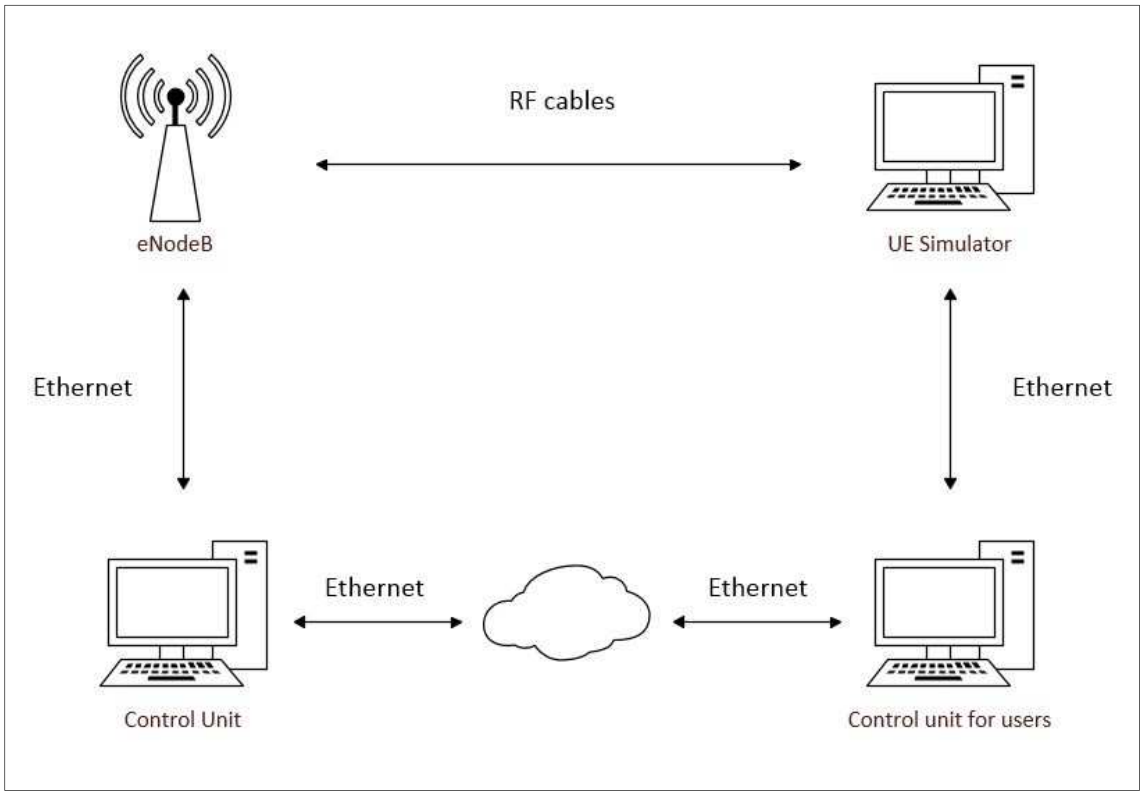
DSP (Digital Signal Processing) integration is system level testing, where this test environment is and all tests also are run.

The DSP integration system is to test and verify the functionality of the interaction between different system components in eNodeB. Example of one test is peer-to-peer communication with UE (User Equipment). All those tested components are in OSI (Open System Interconnection) layer 1 and 2. (3, s.24).

3.2 Test environment and testing work flow

In this chapter it is described how the test person performs the tests currently without the automated interface generation application.

The main control point of the testing environment is the CU (Control Unit). The CU controls the eNodeB and the control unit for users. The eNodeB is a base station and the control unit for users controls UE Simulator. The UE Simulator controls the each user that is created in the cell and the cell is created by the eNodeB (picture 1.).



PICTURE 1. DSPi general test line schema (2, s.28)

The tests are created by the test person. The test person reads the specification that defines the requirements of the feature and how that feature is made. The test for the feature is created when the test person understands how to create the test that confirms the feature is really working. Those tests are usually black-box testing.

The test person uses the CU to start the test and the result of the tests are shown in the CU. When the test starts, first the CU sends command to other parts of the testing system to start. When the systems are ready, CU start testing of all parts of the systems and confirms that all tests fulfill the requirements to pass the tests. If test fails, the CU informs the test person that the specific test fails and gives an error message.

The tests are automated using application that allows to runs test repeatedly and monitors the running test and inform the test person when the test is ready.

When the test person gets information that the test fails, the test person first checks what was the error message. There can be error messages that can be ignored because the automated verifying process is not perfect. If the error is real, then test person must do detailed investigation to find out what really caused the error. After that the test person informs the development teams about this error with the investigation results.

The commands that the CU sends and receives are created manually by the test persons.

The application will automatically create the CU command messages that are used to control the environment in different systems. (6.)

3.3 Jenkins

Jenkins is an application that is used to run those automated executions of repeated jobs as described earlier. Jenkins also monitors the jobs and will give the report of whether the execution succeeded or failed. Jenkins focuses on two jobs:

- Building/testing software projects continuously.
- Monitoring executions of externally-run jobs. Jenkins makes it easy for testing personal to notice when something is wrong.

Jenkins runs the test automatically and triggers the job when something happens in the followed target. Jenkins' features and capacity to modify the report of job were the reason why it is used to make the application automated. (4.)

4 GOALS FOR AUTOMATED TEST INTERFACE GENERATION

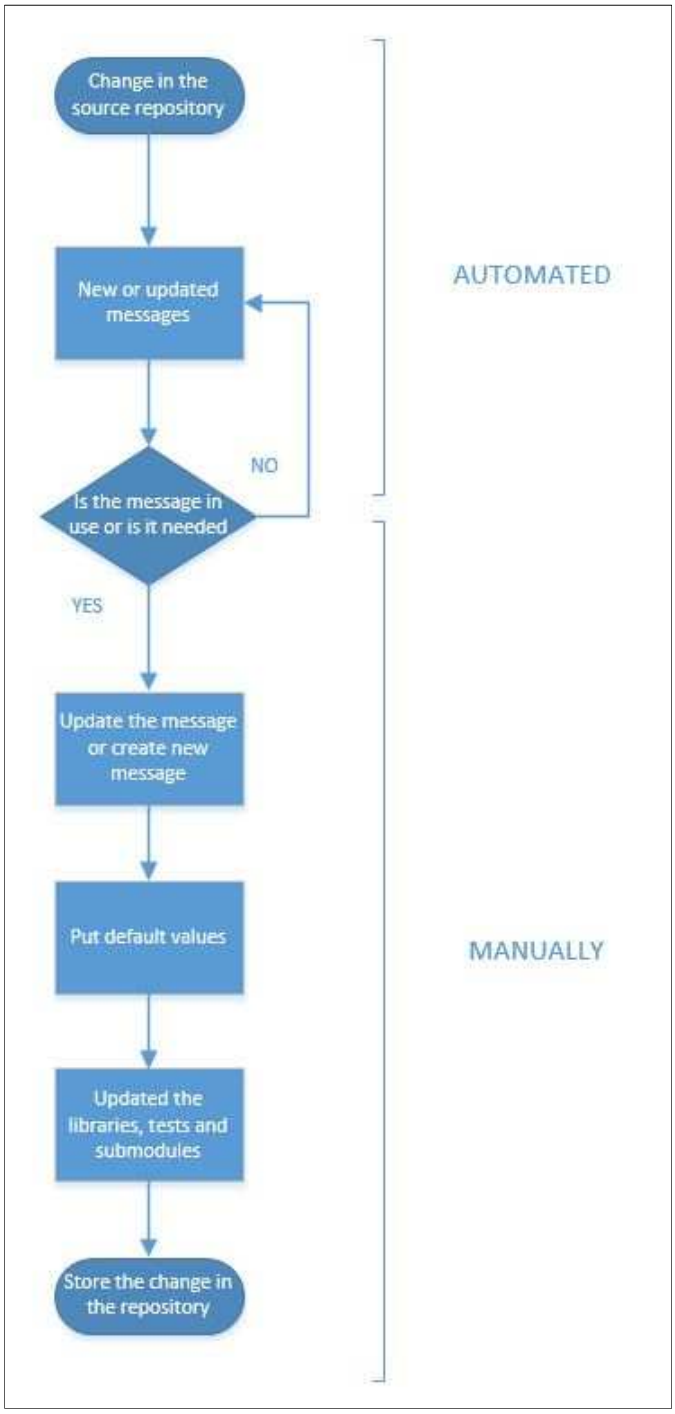
As described earlier, there are still parts of the testing system that are operated manually and those need to be automated.

The requirements for this work came from the testing team and they gave guidelines for how the application should generate messages in the CU format. In addition, when messages are generated, default values for the messages should be given. Finally, the messages with their default values are stored to the repository. All above should happen automatically.

5 CURRENT TEST INTERFACE GENERATION

In this chapter the current test interface generation is introduced.

The present test interface generation (picture 2.) is only partially automated.



PICTURE 2. Present test interface generation

Automated procedures

The current test interface generation system includes automated procedures as described in picture 2. Those are procedures such as:

- When there are changes in source repository the system informs the testing team about those changes, additions or removals
- System shows which messages have changed and what is the change

Manual procedures

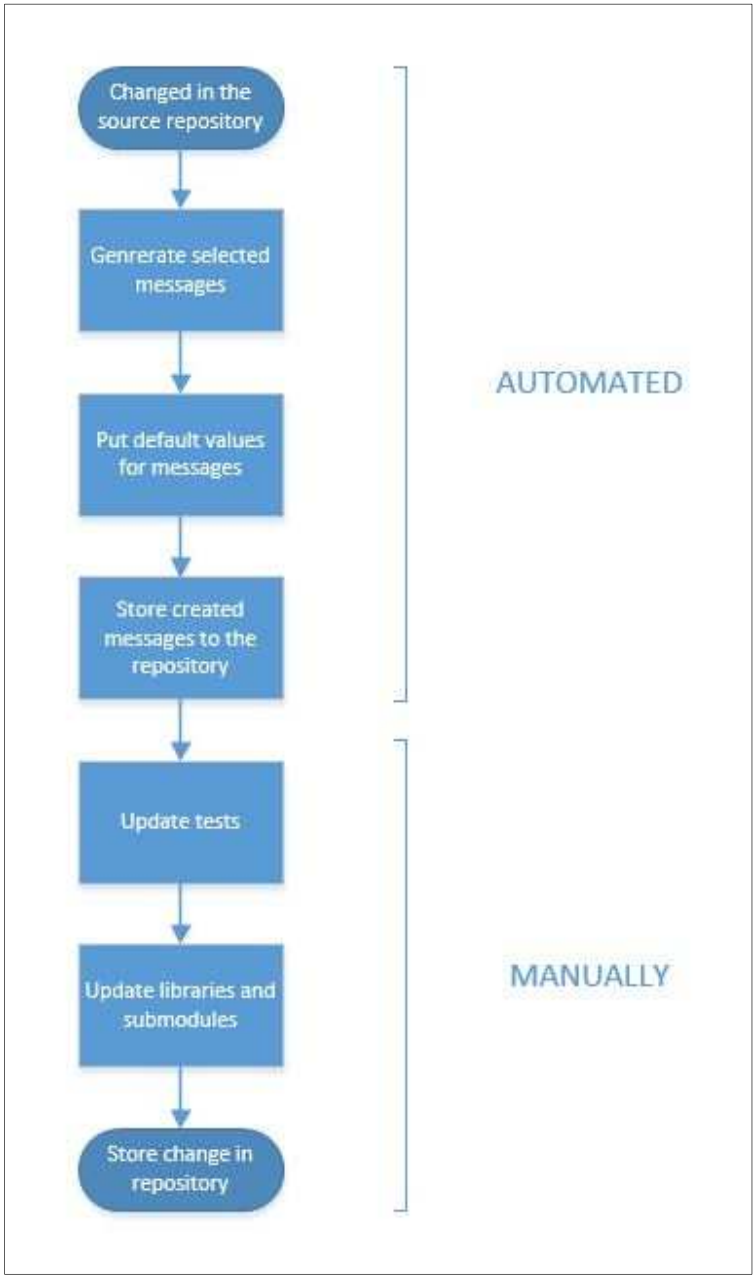
The current test interface generation system includes manual procedures as described in picture 2.Those are procedures such as:

- The testing team must check if they use the changed message
- Update the change in the they own message or create a new message
- Give the message default values if the message is new
- Update the libraries which hold all the default parameters for the tests
- Update the sub-modules which hold the parameters for the test
- Update test cases
- Store the change in the repository

6 IMPLEMENTATION OF AUTOMATED TEST INTERFACE GENERATION

In this chapter it is described how the automated test interface generation is implemented.

The new test interface generation (picture 3) automates some of the manual message generation.



PICTURE 3. New test interface generation

Automated procedures

The new interface generation system includes automated procedures as described in picture 3. Those are procedures such as:

- When there are changes in the source repository the system starts the application
- The application generates the selected messages
- The application will put default values for the messages
- The generated messages are stored in the repository

Manual procedures

The new interface generation system includes manual procedures as described in picture 3. Those are procedures such as:

- Update the test cases
- Update the libraries
- Update the sub-modules
- Store the updates into the repository

Jenkins is used as it is able to monitor the changes happened in the source repositories. When some changes happens it stores the source repository information to the temporal repository where the automated test interface generation application can read that information and starts the application.

After the application is started it fetches all the data that is needed to the message generation from the temporal repository. In addition, the application reads two files that contain messages to be generated. The application start to generate the messages, when all above is done. The application generates the messages in the CU format (picture 5.).

When the selected messages are created, the application gives default values to the messages from the messages default value files. The default value files are created manually by the test person. Parameters without special need will have zero as their default value.

When all the selected messages are created and they have their default values, the application stores all the messages files to the repository. Finally the application informs the testing team that the messages have been generated.

Before the generated messages of the application are used in the test environment, the required test cases, libraries and sub-modules are need to be updated first to be compatible with the messages. The update is needed because of the name difference of the parameters.

7 CHALLENGES DURING IMPLEMENTATION

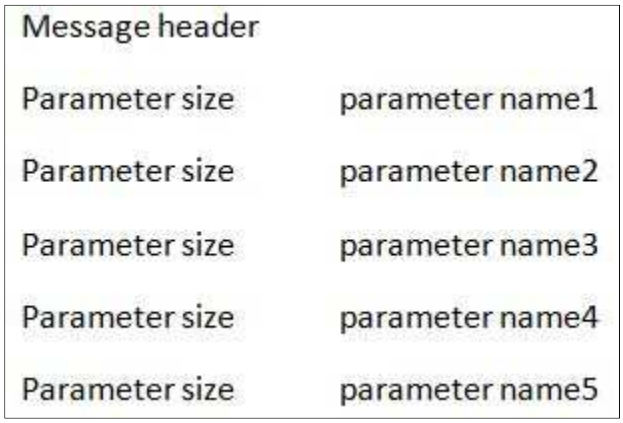
The biggest challenge was to automate the manual parts of the existing system so that the new application receives indication when something is changed in the source repository and the application will start. An other challenge was to modify the already existing automated interface messages generation, so that it generates the selected messages in the CU format, without interfering it. Below is a detailed list of the issues that required special attention during implementation.

- CU format
- Open structured field
- Open array field
- Creating file for a message
- Ability to print messages correctly
- The messages generation takes too much time
- Default values
- Unit-test
- Test the messages in the test environment
- Automation procedures

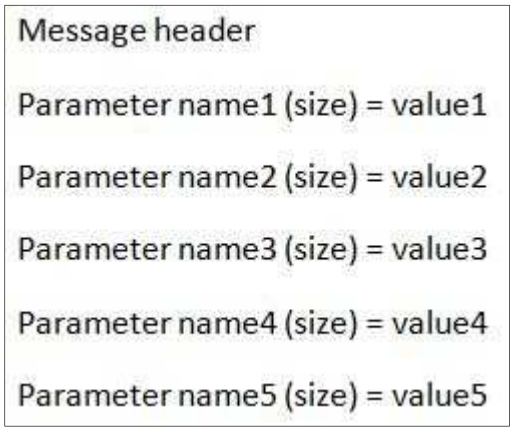
7.1 CU format

As stated earlier, it was required that this new application should be able to generate messages in a way that also other systems understand those correctly and can use those in their operation.

The OMG (Other Message Generation) creates messages in different format than what CU requires (pictures 4. and 5.). Every structure in the OMG was useless in CU. Header was totally different and could not be used and even parameters were differently defined.



PICTURE 4. Simple message in the OMG format



PICTURE 5. Simple message in the CU format

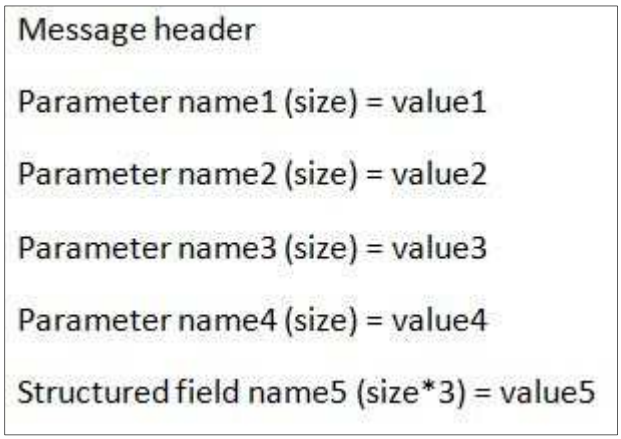
7.2 Open structured field

Structured field is like a function call. The structured field tells all the needed information so that the application can fetch all data that is inside the structured field.

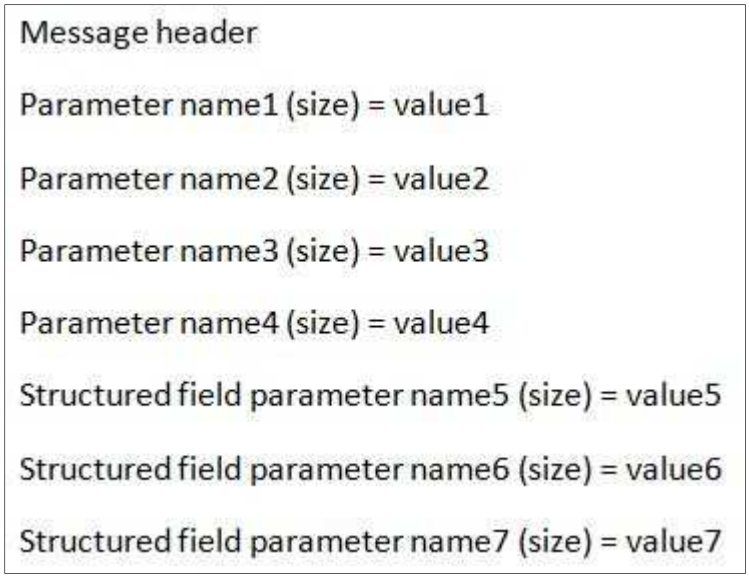
First the structured field was printed wrongly, (picture 6) because the application did not know how to open the structured field, but it showed information of the structured field. OMG opens the structured field in a totally different way, it opens the structured field in own message, after the original message is printed

(picture 8). CU requires the structured field to open immediately and print where structured field was called (picture 7).

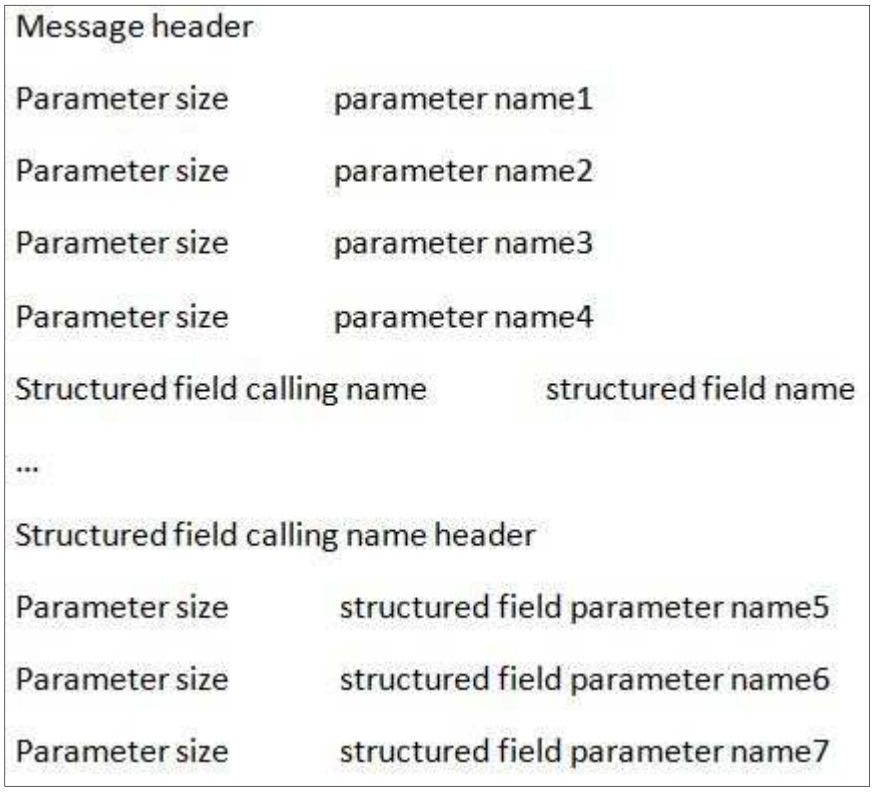
Opening structured field in a middle of the message had a side effect, which caused the execution time of the message generation raise exponentially.



PICTURE 6. Not opened structured field.



PICTURE 7. Opened structured field



PICTURE 8. Opened structure field in the OMG format

7.3 Open array fields

Array field is a group of parameters that are printed only once or multiple times.

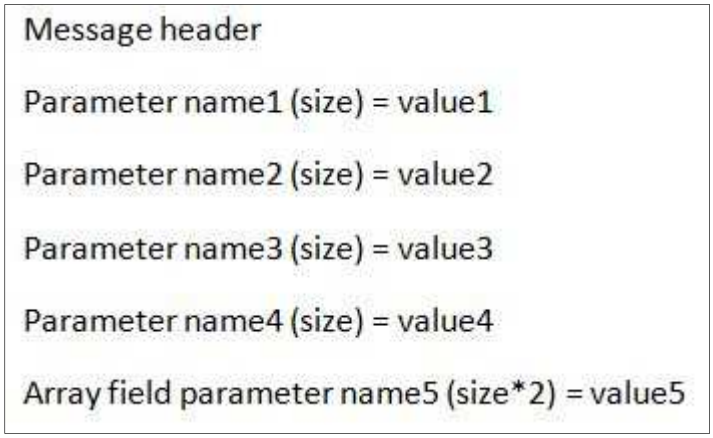
Array fields were a one of biggest challenges of the automated message generation, because there are three different kinds of array fields and array field information can be given in two ways.

- All information is in one line
- All information resides in two lines

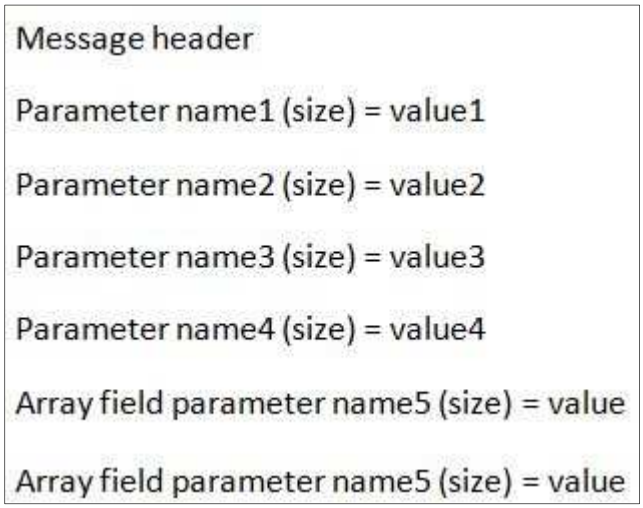
7.3.1 Simple array field

In the simple array field one line is repeated n. times.

At first the application could not print array field correctly (picture 9). The array field should have looked as show in picture 10.



PICTURE 9. Wrongly printed simple array field message.

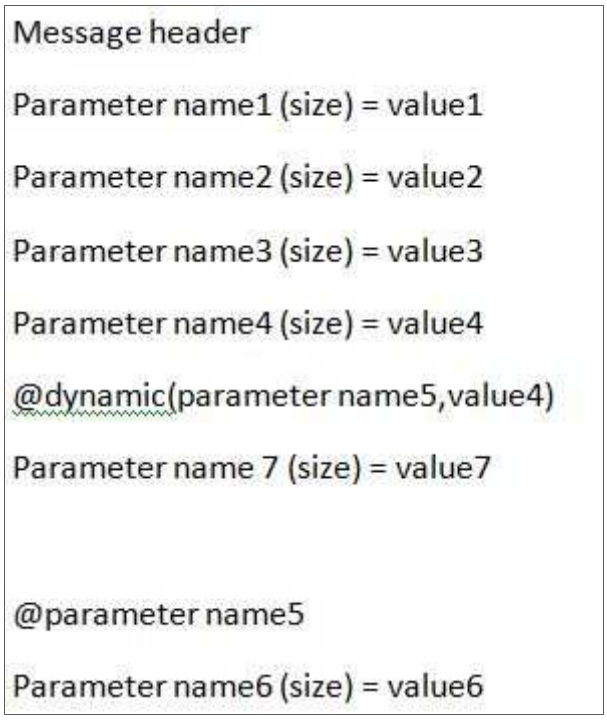


PICTURE 10. Rightly printed simple array field message.

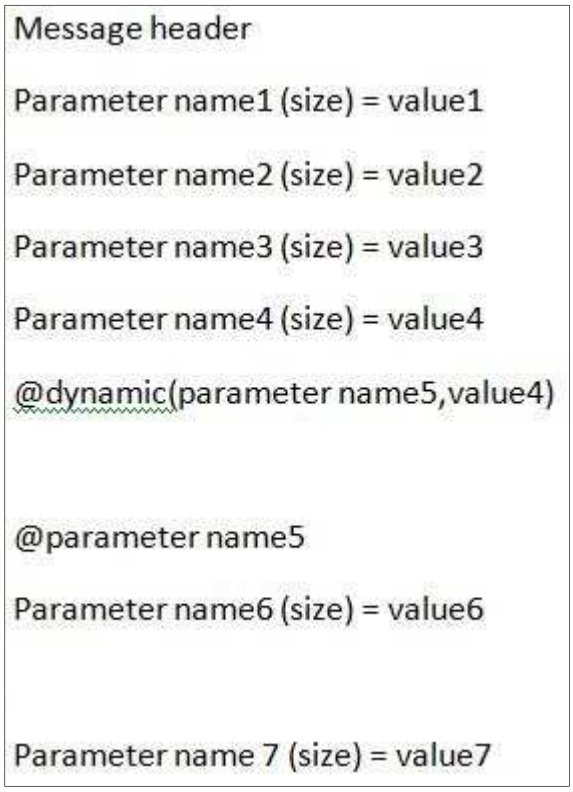
7.3.2 Dynamic size array field

Dynamic size array field is almost the same as the simple array field and structured field, with the difference that the repetition time is told before the dynamic size array field.

The dynamic size array field information is printed differently than with other array field or structured field. Firstly dynamic size array field data is printed to the end of the message and the call is left where it is needed (picture 11). At first the application printed the dynamic size array field wrongly (picture 12).



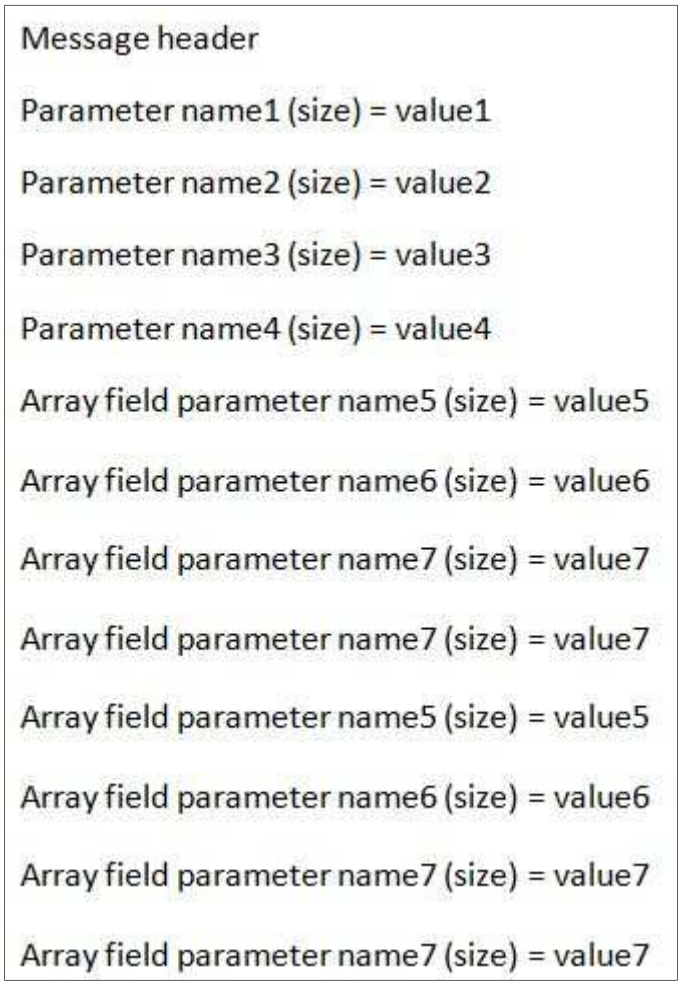
PICTURE 11. Right way printed dynamic size array field.



PICTURE 12. Wrong way printed dynamic size array field.

7.3.3 Fixed size array field

Fixed size array field is almost the same as the dynamic size array field, but the difference is that the lines tell how many times the fixed size array field is printed. Furthermore the fixed size array field is printed immediately inside the message (picture 13).



PICTURE 13. Fixed size array field in CU format.

7.4 Creating file for message

The CU required the message to be in its own file and the file name must be the message name. The file must be saved to the right folder and which folder it belongs to is told in the beginning of the message name. The message name included unnecessary information that must be left out.

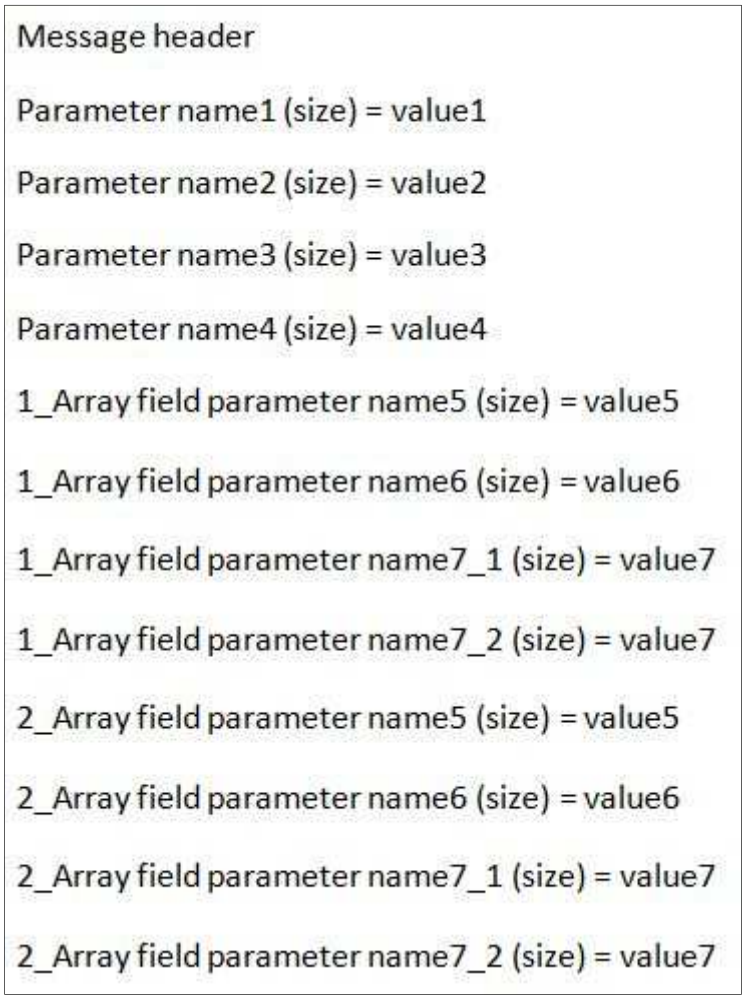
7.5 Ability to print messages correctly

When the messages had their own file, it was easier to compare them to the original files. Simple messages were almost the same, but complex messages were not even close the same.

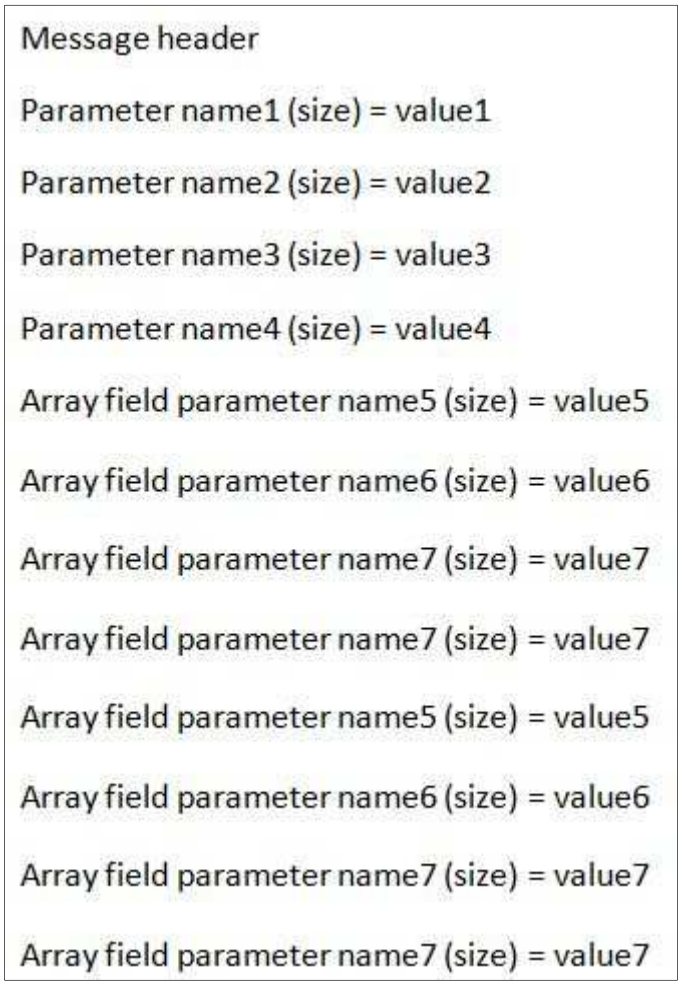
The differences in simple messages were the parameter's name, which did not need to be fixed, because the testing team wanted parameter's name to be the same as in the source repository. In the complex messages the difference was much more than the parameter's name difference. When compared the created messages to the original messages, they missed additional parameter name information, index number, and repetition number. The complex messages were printed wrongly because they contained multiple field types and the ability of the application to print was not sophisticated enough. The application also needs to open union field and be able to name each parameter individually.

Union field is almost the same as the structured field but the difference is that it contains two structured fields and only one is printed. The difference between structured fields inside the union is the parameters for different HW (Hardware) platforms and they are named differently Platform_1 and Platform_2. Some un-ion contains both parameters and structured fields, and sometimes each can be printed and other times only the other could be printed.

An individual name for parameter was needed because the CU needed to be able to select any parameter and there could not be a parameter with the same name (picture 14). Without the individual name there could be many parameters with the same name (picture 15).



PICTURE 14. Rightly placed index and repetition numbers.



PICTURE 15. Before index and repetition numbers.

7.6 Message generation takes too much time

The generation of all the selected messages took from five to ten minutes and naturally that is too long. The real cause for this was the difference between how the OMG and the application open the structured field. OMG opens structured fields after the message is printed and it does not need to search structured fields again. The application opens the structured field when it is needed and it searches the structured field again and prints it there. The search was the reason way it takes so long.

7.7 Default values

The messages need default values because other systems will not start if the messages do not have some specific default values. There was not file nor files were the generated messages could get the default values. Now the default value files were forced to be created for each message otherwise messages could not get their default values.

7.8 Unit-test and test messages in test environment

At the beginning the unit-test was made at the same time as the application, but when application functions were more complex, the unit-test stopped. All functions need they own unit-tests.

When the application could create the messages correctly and give them default values, then it was time to test the messages. The messages must work in the real test environment before they can be used by all the test persons.

7.9 Automation procedures

The application starts automatically when it notices changes in the source repository or a person adds a new or removes an old message from messages to generate the list. All people in testing team should be able to use the application and it should be easy to use. The application should report to the testing team when messages are changed and tell what the changes are.

8 IMPLEMENTATION RESULTS

In the next chapters it is described how the implementation challenges, which are explained earlier, are solved.

8.1 CU format

In the beginning it was difficult to start without knowledge on how to use OMG to generate the messages in CU format. The testing team gave guidelines how to use OMG and how to generate the messages so that they can be used in CU. Below is the function that prints simple messages in CU format.

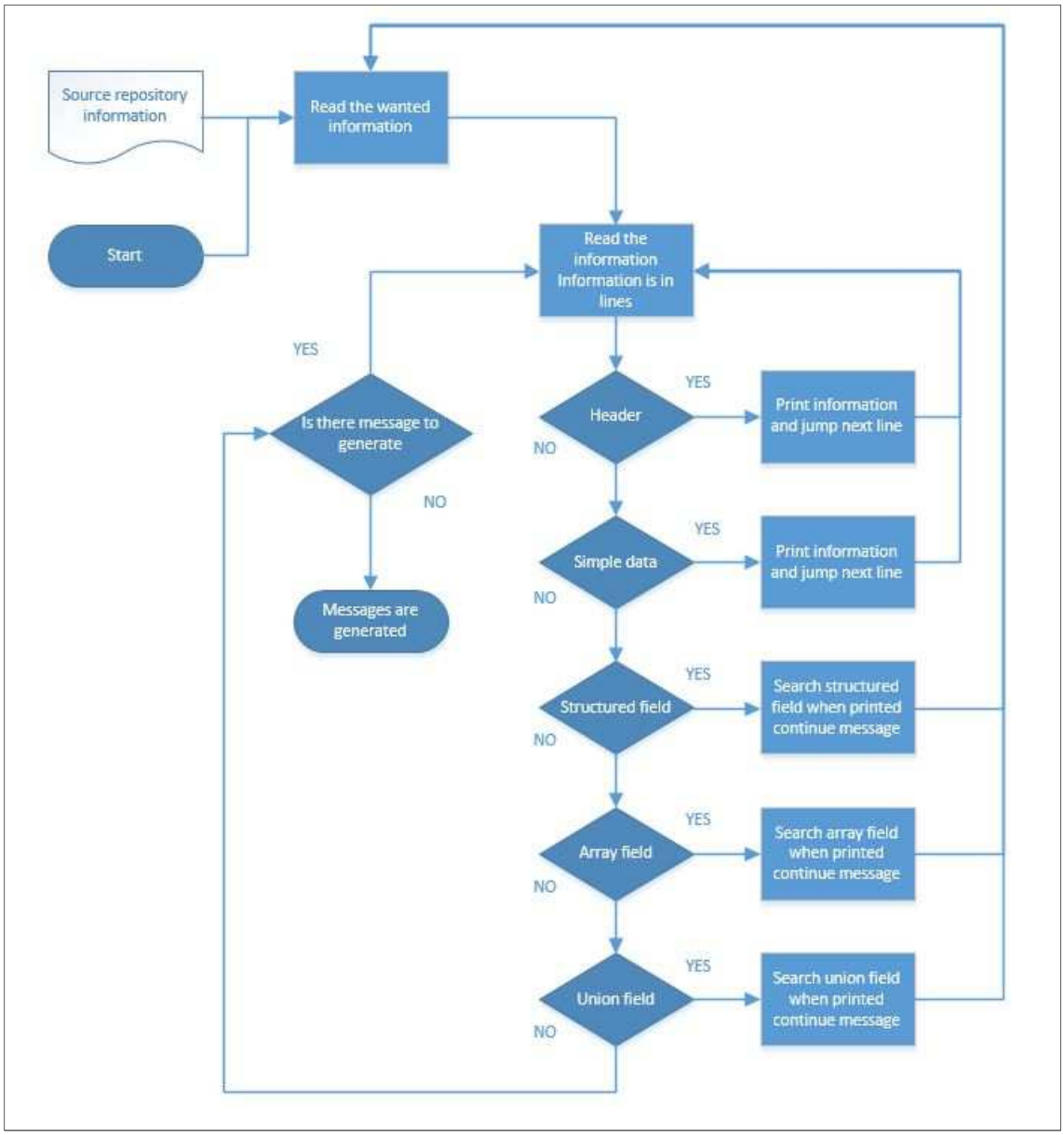
```
def _primitive_field(self, field):
    return '%s (%s) = %s\n' % (field.field_name, field.size, self._zero_content_of_bytes(field.size))
```

The example above can only print simple lines; it cannot print anything more complex but that is the base for printing all field types.

8.2 Open structured field and array field

The application could not open the structured field inside the message, because OMG cannot open the structured field inside a message.

Application has to search the structured field from the source repositories so that the structured field can be printed inside a message (picture 16). Repeated search of the structured field does have small effect to the generation time when generating the simple structured field.



PICTURE 16. How the application opens structured fields and array fields.

Array field is the same as the structured field but array field needs a function that prints array field data multiple times. Simple array fields do have small effect to the generation time of the messages.

8.3 Creating file for message

The CU needs the messages to be generated inside its own file. A function was created that recognizes the message types at the beginning of the message name and uses that type information to save the message to the correct folder. In addition, the function removes unnecessary information from the message name. Below is an example of how the application will create file in right folder.

```
For message:
   If type_1 at beginning of message name:
      Print message in file and save the file folder_1
   If type_2 at beginning of message name:
      Print message in file and save the file folder_2
   If type_3 at beginning of message name:
      Print message in file and save the file folder_3
```

8.4 Improving printing capabilities of application

The parameters inside the messages cannot be named as same, because the CU wants to be able to select each parameter.

If there are parameters with the same name, then the CU does not know which one to select. This means that all parameters must have an individual name inside a message. All functions that print need to be able to add to the parameter name either an index number or repetition number or both. Array field is the most difficult one, because some array fields are too big for the print ability, for example a one is structured array field can contain over two hundred repeated parameters and that structured field can be repeated over ten times. This makes the automated printing functions very difficult. These kind of structured messages raised the execution time exponentially.

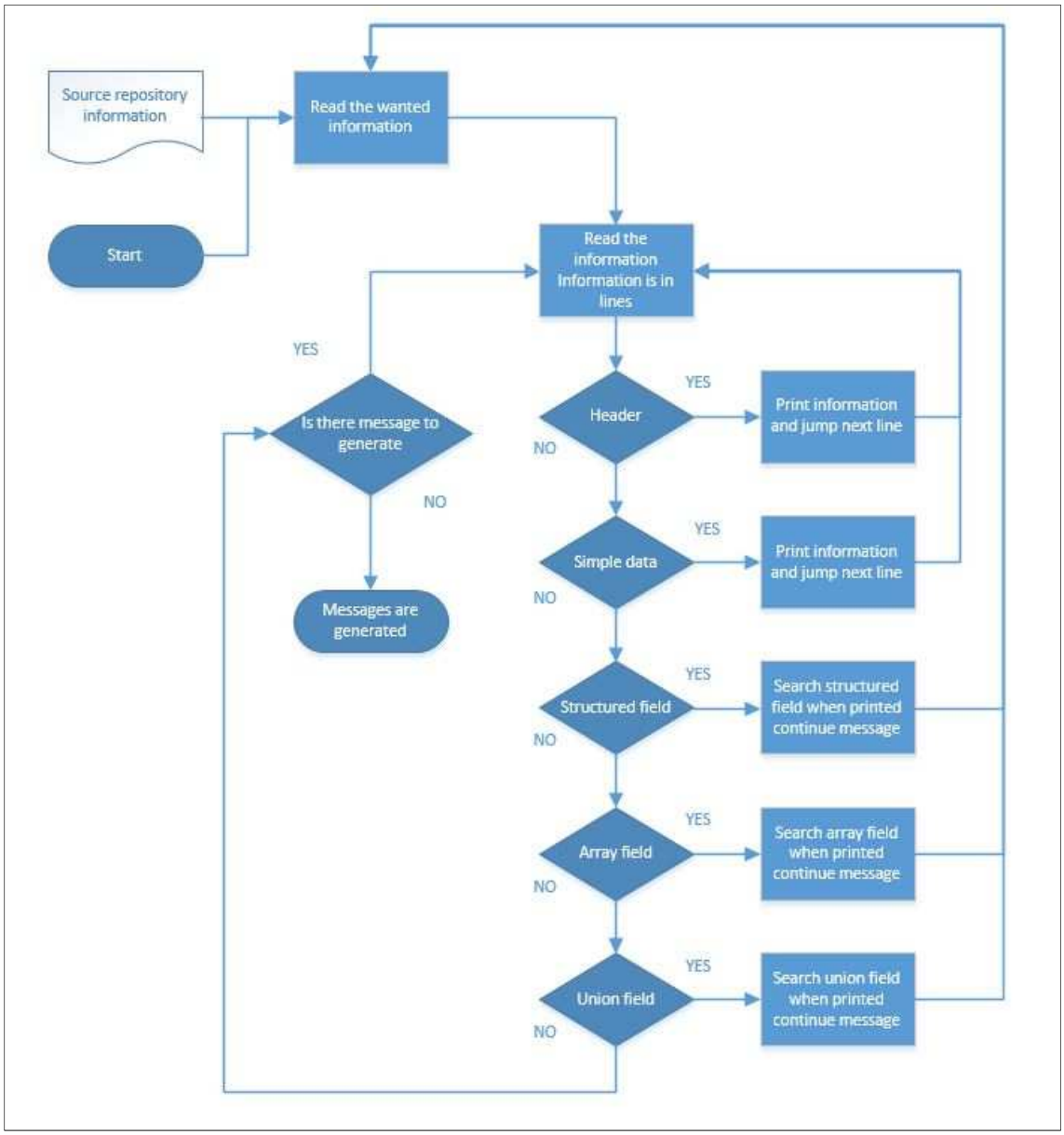
Some messages required more information to be added to the parameter name because it did not tell enough about what the parameter does. Example of this is the address parameter. The address parameters did not tell where the address pointed. So more information was needed to be added at the address name.

First it was unclear how to tell to the application which union must be printed. Then it was found out that the OMG has a way to tell in its own message generation that this one message needs the special function. That method was used as an example on how to tell to the application which HW platform parameters should be printed if union is chosen. When the platform was chosen and it was noticed that the structured field was smaller than the other platform structured field. That required a padding which is an empty parameter that will increase the structured field size so that its size is the same as the other platform structured field. The padding must be placed after the printed structured field of the platform.

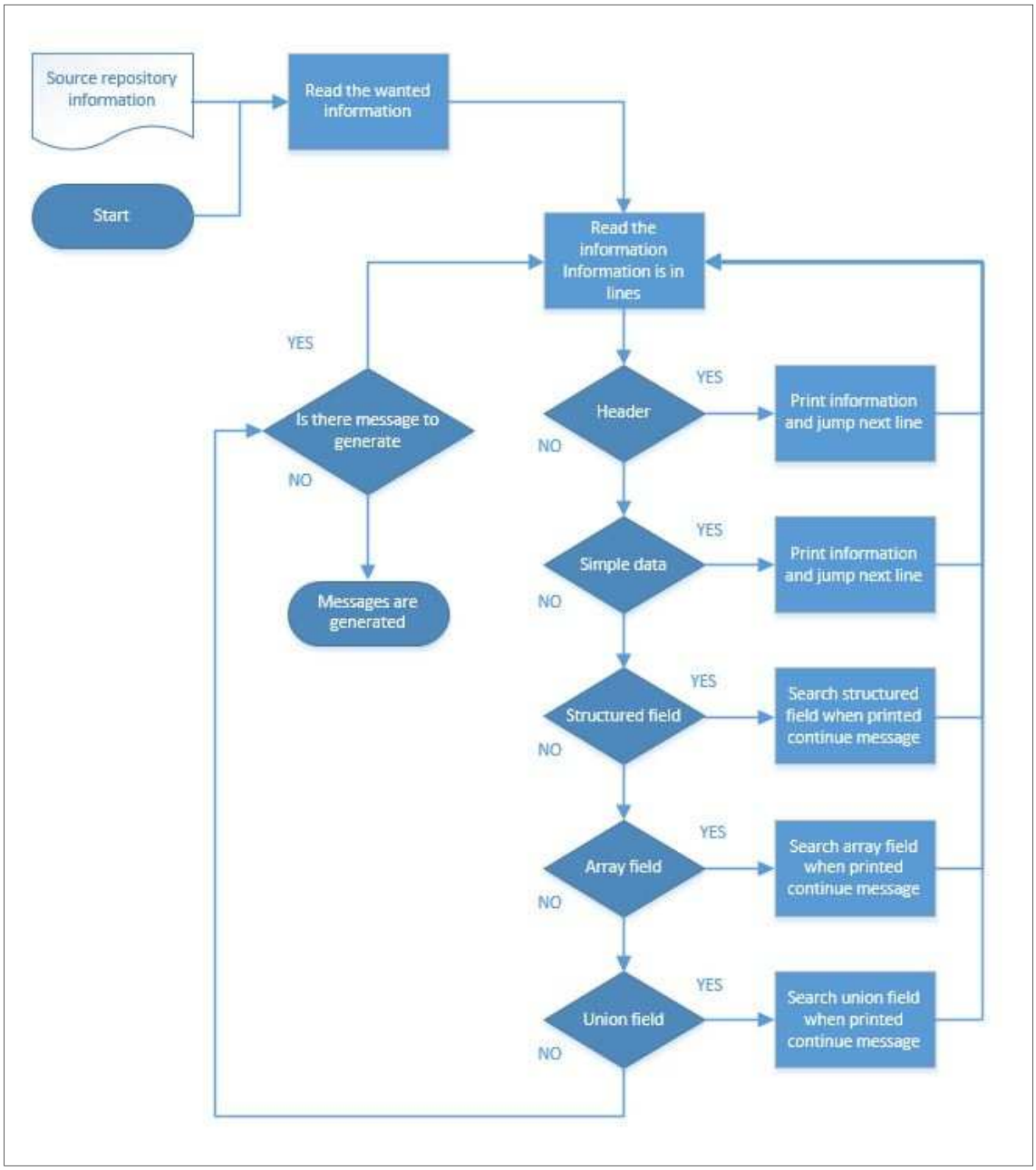
The application code became hard to read after the application could open the union, print the content of the union correctly and name the parameters individually inside the message and other functions that increased the application printing abilities.

8.5 Generating all messages is too time consuming

The main reason why the message generation with the new application took such a long time was the way how the application opens the structured field. Application reads and searches each structured field again even though that information is already read in the previous round (picture 17). This problem was solved with the solution that saves the source repository information which is searched and read. That saved information is made available directly when it is needed again (picture 18).



PICTURE 17. Slow way to open structured field.



PICTURE 18. Fast way to open structured fields

8.6 Transfer default values

There was no place or method for how to find and set the default values to the message parameters. This problem was solved by creating a default value file for each message. In addition, a function was created that compares the created message parameters with the default file parameters. If the parameter name

is the same and the parameter size is also same, then the default value can be transferred to the created message. The application searches the parameter name from the default file so the parameter order can be different than the created message or the parameter count can be different than the created message. The transfer of the default value is slower when the order of parameters is different than created message or parameters count is different.

There is a parameter that tells how many times the dynamic size array field should be printed. A function was created that prints the dynamic size array field data as many times as the parameter information tells. When the data is printed as many times as it was needed then function confirms that the message size is divisible by four. Based on that result padding parameters are needed to added or removed or nothing.

8.7 Unit-test

Making unit test for software under development is one of the essential parts of the software programming. A unit test framework was available and unit tests were made, but during the implementation the complexity of the application increased a lot making the creation of new unit test cases difficult. After getting better understanding of the whole system, it was easier to create and perform the unit test cases.

8.8 Test messages in test environment

During the testing it was noticed that the test cases used the old parameter name. Those cases were modified and tests passed. The application creates a file that contains all messages in alphabetical order in that folder. It was created because the CU needs it. The file tells the CU which messages it can use.

8.9 Automation procedures

To be able use this new application for automated message generation, it needs to be checked that testing person access rights to the repositories and have privileges for applications usage that are part of the whole system where this new application resides.

The first intention was to create automation for the application that offers local message generation without storing the messages inside the repository. This failed because of the permission problems.

The second try was so that the testing team does not see the application. The testing team sees only the generated messages, default value files and files that contain the information which messages are to be generated. The application was only visible for the job inside the Jenkins. The job starts the application.

The local message generation without storing them to the repository is possible but it requires previous knowledge and manual commands.

9 CONCLUSIONS

The main goal of this thesis' was to create automated test interface generation. Other requirements were that the generated messages should be in the CU format. Default values are given to the messages and messages with default values are stored to the repository.

All the goals for this work have been reached. The application starts automatically when it notices changes in the source repository or the messages to generate files are changed. Application generates the selected messages in the CU format, sets default values for them and finally stores them in the repository. Before the application generated messages can be used in the test environment the test cases, libraries and sub-modules need to be updated first to be compatible with the messages.

The biggest problem in this work was the OMG because using it as a basis for this new application proved to be not the best idea. Lot of new code required enabling the application to generate messages in the CU format. A better idea would have been to create this new application from scratch, but after these difficulties with the OMG noticed it was too late to be able do it from the very beginning because I could not have finished this work in time. Now, when this application has been created using OMG the code is not the easiest to understand.

One improvement is still required and that is to improve the Jenkins report that is not currently informative enough. The report should show clearly which messages are changed and what the changes are.

In addition, end users manual "HOW TO GENERATE CU INTERFACE MESSAGES IN THE AUTOMATED GENERATION" is written, which tells how to add or remove messages for generation and how to use it manually. (Appendix 1)

The work went smoothly except for couple of small problems. Help from NSN side was available when needed. The company provided me with the workplace and all the needed tools to do this work.

Mostly this work was coding, which is not my primary training but was familiar already in some level anyway. Studies required getting familiar with the test environment and how the testing team automates their tests. For the application i had to inquire a lot of things from the testing team because the testing team had the answer and they knew what they wanted for the application. An example of these inquires was the individual naming for the parameters name. In the original messages some parameters were individual named while some were not.

LIST OF REFERENCES

1. Lewis, William E. 2009. Software Testing and Continuous Quality Improvement, Third Edition. Auerbach Publications

2. Łyko, Marcin 2011. LTE Introduction & DSP Integration Wroclaw Team Presentation. NSN intra-net

3. Päätalo, Päivi. 2010. LTE base station DSP software pre-integration test environment and testing. Oulu: University of Oulu, Department of electrical and information engineering. Master's thesis

4. Kohsuke, Kawaguchi – Carr, Paul 2013. Meet Jenkins. Retrieval day: 24.6.2013. http://jenkins-ci.org/.

5. Aslmadi, Izzat. 2012. Advanced Automated Software Testing: Frameworks for Refined Practice. IGI Global

6. NSN internal discussions 15.2.13 – 14.6.13.

Appendix 1.

Only for internal use of the Nokia Siemens Network.