



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

MASTER THESIS IN COMPUTER SCIENCE AND ENGINEERING

Algorithms for Verifying Backwards Compatibility In Distributed Real-Time Systems

HUSAM ABDULWAHHAB

MAKSIMS SMIRNOVS

Department of Computer Science and Engineering
Computer Science – Algorithms, Languages and Logic Master Program
&
Computer Systems and Networks Master Program
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

The Authors grant Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose to make it accessible on the Internet.

The Authors warrant that they are the authors of the Work, and warrant that the Work does not contain text, pictures or other material that violates copyright law.

The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors have signed a copyright agreement with a third party regarding the Work, the Authors warrants hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Algorithms for Verifying Backwards Compatibility In Distributed Real-Time Systems

Husam Abdulwahhab

Maksims Smirnovs

© Husam Abdulwahhab, 2016.

© Maksims Smirnovs, 2016.

Supervisor: Mirosław Staron

Examiner: Eric Knauss

Master's Thesis 2016:NN

Department of Computer Science and Engineering

Chalmers University of Technology

University of Gothenburg

SE-412 96 Gothenburg

Telephone +46 31 772 1000

Acknowledgements

We would like to express our sincerest gratitude to our academic supervisor Dr. Mirosław Staron, who provided us with support, guidance and good feedback throughout this project. We would also like to thank our technical supervisors Håkan Oswaldsson and Thomas Sundell who helped shaping our thesis with extremely useful ideas and technical advices, and our manager at Ericsson Niklas Zetréus, who introduced us to the company's personal and made sure we have access to Ericsson's facilities. Lastly, we sincerely thank Ericsson for providing us the opportunity to carry out this study at the premises of the company.

Husam Abdulwahhab & Maksims Smirnovs, Gothenburg, June, 2016

ABSTRACT

BACKGROUND: Backwards Compatibility is a key solution for companies that are attempting to reduce the cost and effort of introducing software updates to their customers. It is also an important property in order to gain the customer's trust in accepting the updates without fearing side effects of some functionality not working properly. Therefore, it is important that the newly released software update is backwards compatible with an older version of the software of the same product.

METHOD: In this paper, the main goal is to derive algorithms for verifying backwards compatibility and implement them. The results are obtained by following a research methodology that is based on the design research. Literature review was conducted in order to identify existing methods for verifying backwards compatibility. Algorithms for verifying backwards compatibility were designed, prototyped and tested on a distributed real-time system in order to evaluate their behaviour. The implemented prototypes of the algorithms can verify backwards compatibility in distributed real-time systems that pertain to telecommunications industry.

RESULTS: Description of all the identified algorithms and strategies from academia and the industry, along with their classification into taxonomy are presented. Three algorithms that are designed by the authors which verify backwards compatibility in distributed real-time systems. The first algorithm is based on communication signals of a component during the execution of one of its tasks, the second algorithm focuses on the details of what the system is doing while executing the tasks, the third algorithm combines characteristics of the previous two algorithms. Prototypes of all the algorithm are developed and tested on various test scenarios of a software update. The combined results of the algorithms identified 8% backwards compatibility problems which are within the acceptable range when a software update is performed on the SGSN-MME product. All three algorithms provide details on what might be causing the incompatibility of the software update.

CONCLUSION: Backwards compatibility is a hard problem to achieve especially in large and complex systems such as the one this study was based on. The algorithms that were identified in this study show a lot of promise in developing automated methods for verifying backwards compatibility. This is proven with the prototypes of the three algorithms that were developed over the study period. The work that was carried over this study shows that there is a number of open gaps to be studied in the future in order to achieve full scale autonomous algorithms for verifying backwards compatibility for various components of a system.

Keywords: backwards compatibility, distributed real-time systems, automated check, execution based backwards compatibility, performance based backwards compatibility.

Contents

1	Introduction	1
2	Background	3
2.1	Backwards Compatibility	3
2.2	Sources of Backwards Incompatibility	5
2.2.1	Verifying Backwards Compatibility	6
2.2.2	Forward Compatibility	7
2.3	Scope	7
2.3.1	Real-Time Systems	7
2.3.2	Distributed Systems	8
2.3.3	Ericsson's SGSN-MME Product	8
2.3.4	Ericsson's View of Backwards Compatibility in the SGSN-MME	10
2.4	Related Work	11
2.5	Limitations	12
3	Methodology	13
3.1	Research Questions	13
3.2	Objectives	14
3.3	Selecting Research Methodology	15
3.4	Research Approach	16
3.5	Literature Review Process	18
3.5.1	Data Sources and Search Strategy	19
3.5.2	Study Selection process	20
4	Algorithms Design	21
4.1	Signal Based Algorithms	21
4.1.1	Signals, Packets & Protocols	21
4.1.2	Capturing Execution Signals	22
4.1.3	Identified Traffic Analysis Tools	22
4.1.4	Execution Signals	24
4.1.5	Identified Patterns	25
4.1.6	Verifying Backwards Compatibility	27
4.1.7	Prototyping the Algorithm	28
4.1.8	Testing of the Prototype	29
4.2	Events Based Algorithms	30

4.2.1	Counters and Events	30
4.2.2	Capturing Counters	31
4.2.3	Capturing Events	32
4.2.4	Verifying Backwards Compatibility	32
4.2.5	Prototyping the Algorithm	35
4.2.6	Testing of the Prototype	36
4.3	Unified Algorithm	37
4.3.1	Merging the Algorithms	38
4.3.2	Verifying Backwards Compatibility	39
4.4	Performance Based Algorithms	40
5	Results	42
5.1	Study Results	42
5.1.1	Literature Review Results	42
5.1.2	Empirical Results	44
5.2	Results of Implementation Phase	47
5.2.1	Results of Applying Signal Based Algorithm	47
5.2.2	Results of Applying Events Based Algorithm	49
5.2.3	Results of Applying the Unified Algorithm	52
5.2.4	Impact of the Results	53
6	Threats to Validity	54
7	Ethical Considerations	55
8	Conclusion	56

1

Introduction

SOFTWARE has become an important part of people's lives [MBN12]. Today, people use software in activities and applications such as healthcare, transportation, navigation, and many other purposes. This has created a competitive industry with new companies emerging and competing with each other in the various fields of software development [MBN12]. In order to gain an advantage over the competitors, the companies are expected to create products that have optimal performance and scalability [BG99, MBN12]. This is done through constant updating of the system which introduces new features and enhancements. These updates may result in degradation of the performance of the system, thereby increasing the need of tests to maintain the quality of the software [KP96]. The updates also introduce new functionalities into the system which may fail to work on previous version of the system, thus, making the older versions of the product obsolete very quickly [KP96]. The failure of the older products is an inconvenience to the customers who purchased these products, because as soon as they download the new firmware their systems might not function properly. Hence, it is important to check that new updates do not result in such problems. This type of quality checks is known as verification of backwards compatibility [GBM15].

The formal definition of backwards compatibility is introduced in the next section; however, in order to get an understanding of what shall be discussed, the following definition is used, "*Backwards compatibility indicates that new software can run compatibly with the software of older versions of a system without any problems*" [GBM15]. To exemplify this definition in practice, consider a group of nodes in a system that communicate with each other using a certain communication protocol. If node (A) in the system is updated to use a new non-backward-compatible communication protocol, while the rest of the nodes continue to use the old protocol, then node (A) will not be able to communicate with the other nodes as before. Backwards compatibility is very important for software developing companies that want to deploy updates to products fast and frequently [GMB15].

The implementation of backwards compatibility is not a trivial task because it becomes more complex and harder to verify in proportion to the increase of the size of software [GMB15]. Old methods of testing each feature of the new software on older hardware, to establish whether the features work or not, do not scale well with the increasing size of these products and their software. Thus, performing backwards

compatibility on large scale products can be slow and can take a long time due to the enormous size of the software that supports them. This is especially important for the telecommunication infrastructure products, as these products have the characteristics of being large scale, distributed and real-time. This type of distributed real-time systems is particularly problematic due to the complex nature of its design, as it is made up of many layers of hardware combined with an enormous amount of software that are constantly being updated with new features and functionalities. Therefore, it is important to design and develop automated algorithms that verify backwards compatibility of new updates that are introduced into this type of systems.

In this study, we aim to design and develop algorithms for verifying backwards compatibility in distributed real-time systems. Since this type of systems within the telecommunications industry tend to have major parts with several testing phases, it is important to identify backwards compatibility algorithms for the various parts of the system. This is done through a study phase and an implementation phase which are explained in the research approach. The solution of this type of problems can be quite beneficial both at an industrial and academic level. This is because the new knowledge gained from this thesis could be applied in various types of software applications within the industry, as well as solving problems within the fields in computer science and engineering.

The contributions of this study include the introduction of two new definitions of backwards compatibility which are *Execution Based Backwards Compatibility*, and *Performance Based Backwards Compatibility*. Additionally, the classification of the identified algorithms and strategies into four categories which are, *Execution Check*, *Performance Check*, *Syntax Check* and *System Check*. Lastly and most importantly, the design and development of two algorithms for verifying backwards compatibility which are *Signal Based Verification Algorithm* and *Events Based Verification Algorithm* that pertain to the execution based backwards compatibility verification, and a method by which the two algorithms are merged together into a *Unified Algorithm*.

The rest of the paper is structured as follows. Section 2 of this paper presents the background of the thesis with all the relative information pertaining to backwards compatibility. Section 3 discusses the research questions, objectives and the approach of how the thesis is carried out. Section 4 discusses in details the design of the algorithms that were developed during the period of the thesis. Section 5 presents and discusses the results of the thesis and Section 6 addresses ethical considerations of the study. Lastly, Section 7 concludes the study.

2

Background

THIS section provides information about the most important elements that constitute the background of the thesis, such as definitions of backwards compatibility, potential sources of backwards incompatibility, the scope of the thesis, limitations and other general definitions that will be used throughout the paper.

2.1 Backwards Compatibility

While the term “Backwards Compatibility” may seem straightforward in terms of meaning, the subject of backwards compatibility is broad and can have varying definitions [PR12]. This is because backwards compatibility applies to different levels of software, as well as hardware. The first and most common definition of backwards compatibility is in terms of avoiding system failures in the presence of new updates [PR12]. Based on [PR12], the general definition for backwards compatibility is “*when new updates are introduced into the system, they must not cause the system or any of its components to crash while executing their usual tasks*”. While this is a good definition, it is a general one, especially given that any new updates are thoroughly tested before they are introduced into the system in order to ensure software quality. However, while new updates may not cause a system to crash, they may end up causing one of the components of the system to behave differently. In this study, two additional definitions for backwards compatibility are presented, that is *execution based backwards compatibility* and *performance based backwards compatibility*. The main difference between the two definitions is that the former definition refers to the behavior of the system, while the latter refers to the system in terms of performance, e.g. load on the CPU or RAM.

- *Execution based backwards compatibility* - a new feature implementation is backwards compatible with an older version of the product if the new feature implementation preserves the observable behavior of the older version of the product. Whilst this definition is indeed similar to the aforementioned one, it does provide a more detailed description, namely, that the behavior of the system should remain the same with the introduction of new features. What this means is that in terms of execution, the system must exhibit the same behavior when it comes to, for instance, sending signals on a networking level.

That is, the signals sent by an older version of the system must still be present within the updated version of the system.

- *Performance based backwards compatibility* - implies that the implementation of a new feature is backwards compatible with an older version of the product if the new feature does not decrease the performance by more than 2%. This bound (2%) is defined by Ericsson and is the de facto definition used within the company, where the study is carried out. For example, this indicates that the updated system does not utilize CPU more than it is allowed to, for instance, connecting two thousand roaming users to a server should not create a load on the CPU more than 2%.

Whilst the definitions we described above are the main ones we shall refer to, there are other backwards compatibility definitions worth mentioning.

- *Binary backwards compatibility* - is that the application or component working with the old version of the binary file keep working correctly with the new version of the library with an implemented feature without recompilation of the application [PR12]. An example of this is a program that reads (1 to n) integer values. If this program is used to read float values instead, it will crash because of the mismatch in the *datatypes* within the binary file of the program, therefore, backwards compatibility at a binary level shall not hold.
- *Source level backwards compatibility* - means that applications and other dependent components have to be rebuilt and recompiled in order to function properly on different components [PR12]. However, there is no need to change the source code itself, only the binary recompilation. An example of this would be moving software to a new type of hardware such as a computer with a different CPU architecture. In this case it is not necessary to change the source code of the program; however it is necessary to recompile the program in order for it to work on the new hardware.
- *Microcode level backwards compatibility* - is backwards compatibility of software at low level programs. It is the result of changes that occur on the hardware of a system. For example, when a new functionality is added to an existing CPU design, this functionality might not be compatible with the legacy version of the software [AEF05]. Thus, verification of backwards compatibility must occur at a microcode level, which is an extremely advanced approach that requires the knowledge of computer architecture in general.

It is worth noting that while a new feature that is introduced into the system must not affect the original behavior of the system, it can expand on it. However, some updates may require a change in the original behavior of the system as part of the update, for instance, omitting sending network signals to a nearby router about the system's current state. In which case the definition of *execution backwards compatibility* may not stand firmly. Similarly, if a performed update on the system causes performance degradation by more than 1%, then the definition for *performance backwards compatibility* may not hold either. Having described several backwards compatibility approaches, each with their own unique method for verifying back-

wards compatibility at a certain level, we would like to emphasize once again that the definitions we shall mainly use throughout this paper in terms of backwards compatibility will be *execution based backwards compatibility* and *performance based backwards compatibility*.

2.2 Sources of Backwards Incompatibility

Backwards incompatibility refers to a system's inability to accept new updates and features [GBM15]. This means that updates that are introduced into the system may cause problems to the system as a whole or to one of its components. In order to get a better understanding of what is backwards compatibility, it is important to discuss the different sources of problems which can lead to a system being backwards incompatible. However, we do emphasize that this section only addresses the sources of backwards incompatibility, not the solutions to resolve the issue. Some of the identified solutions to resolving the issues of backwards incompatibility are discussed in section (2.3 Related Work) and in more details in the results of the study phase in section (5.1 Study Results). Backwards incompatibility may occur during one of the following *phases* of software development:

- *System Updates*

System updates are common and are released periodically in most of the products. They include bug fixes, enabling or disabling certain parameters, or attempts at increasing performance of certain components. However, in situations such as upon a release of an update, the system that is subjected to the change results in being backwards incompatible. For instance, it can be backwards incompatible in terms of *binary backwards compatibility* or *execution based backwards compatibility*. This can yield the result of one of the system's components crashing or performing differently from the intended behavior.

- *Feature Deliveries*

Sometimes it is the case that new features are introduced into certain components of the system, which would allow the system to achieve new tasks that it could not fulfil beforehand, for instance, enabling power saving mode via a click on the icon. These features undergo functional tests, however, as we mentioned before, it is not always the case that the newly implemented functionality is tested for backwards compatibility. For instance, in terms of execution based backwards compatibility, the feature does not affect the behavior of the system; however, it might affect the performance in a negative way. Thus, it is very likely that these features would result in some unexpected compatibility issues.

- *Software Corrections*

The majority of software products that are released will result in a number of trouble reports, which document the issues encountered during the imple-

mentation phase and possible solution of the issue. Companies try to fix these problems as soon as possible and release what is known as a “hotfix”, which contains a correction of the reported problem. However, it might be the case that the released corrections are not fully tested and in some cases, these corrections can result in more unexpected problems, which yield a compatibility issue with the legacy software.

Now that we know where backwards incompatibility may occur, it is necessary to establish some of the causes of backwards incompatibility during the three aforementioned phases [PR12]. The following key points are some of the most common reasons behind backwards incompatibility:

- Removing certain functionalities from one of the system’s components, thereby preventing the system from being able to use this function.
- Changing the structure or name of one of the functions within the system.
- Changing the number of parameters in one of the system’s functions.
- Changing the order of the parameters in one of the system’s functions.
- Changing the type of parameters that the function is expecting.
- Changing the type of value that the function is supposed to return.
- Overriding some of the existing functions in the system.
- Removing an entire module or software component from the system because it is deemed unnecessary.

Having described the potential sources of backwards incompatibility, it is fitting to mention possible solutions that can verify backwards compatibility.

2.2.1 Verifying Backwards Compatibility

The first approach that could verify backwards compatibility consists of executing various levels of testing the feature against the system, such as unit and component testing, functional testing, and full scale system and network testing. Hence, most of these tests will provide a sense of certainty that the feature will work in the system successfully and with minimum risks on the system itself. While this is a good first line of defense against incompatibility issues, it might not be enough in terms of other definitions of backwards compatibility, such as *performance based backwards compatibility* or *binary backwards compatibility*. Thus, there should be new type of tests on the feature or an update against the system, which are specifically made for the purpose of verifying most of the backwards compatibility types defined in the previous section.

One important aspect of the backwards compatibility verification algorithms is that the tests should be automated and should be able to point out where the incompatibility problems are, thus making it easier for the developers to detect these

compatibility issues and correct them. This is where the role of the authors comes into play, as the aim of this study is to establish suitable algorithms that can be automated and used for verifying backwards compatibility of new updated builds against the old ones in distributed real-time systems.

2.2.2 Forward Compatibility

Although this is a complementary subject to this thesis, it is worth explaining the concept of forward compatibility or “*upward compatibility*” as a distinction from backwards compatibility [JIH02]. Forward compatibility is “*a design pattern of a system which allows the ability of accepting input that is intended for future versions*”. This means that a system with forward compatibility design is built with expectation of future changes within that system, allowing it the capability of understanding data that is generated by future versions or new devices, which allows it to read and execute the data and reply to these devices [JIH02].

2.3 Scope

Based on the discussion in previous section about backwards compatibility, as well as the study that has been done regarding it so far, the subject of backwards compatibility is very broad. It is of paramount importance to focus the scope of our thesis on backwards compatibility in specific parts or areas. The scope of our thesis is strictly aimed at systems that contain characteristics of distributed real-time computing. From the literature review that we have performed, we can draw a conclusion that there has been little (if any) study done in terms of verifying backwards compatibility in distributed real-time systems.

2.3.1 Real-Time Systems

A real-time system is one where the correctness of the system depends on two factors. The first factor is the logical result of the computation and the second factor is the time, at which the results are generated [ST88]. What this essentially implies is that if the output is not delivered on time, something bad might happen. However, the timing constraints can be even further divided into soft timing constraints and hard timing constraints.

A soft timing constraint implies that an un-timed completion of a task is undesirable, but not critical for a system [LI00]. Whilst a hard timing constraint denotes that if a hard deadline is missed, the outcome could be fatal [LI00]. As an example, consider a mobile subscriber connected to a mobile network. A soft timing constraint in this particular case would be the user connecting to the network, e.g. connection establishment should not take more than 100 milliseconds. The time to connect the user does not need to be a hard timing constraint. However, should that user

in any case try to contact emergency services, then a hard constraint would be to prioritize the user's call because, most likely, the information in the call will be of vital importance. Hence, a task in the user's mobile phone that is responsible for converting the data from analog to digital, and afterwards encapsulating this data into a physical link frame and sending the link frame to the nearest cell tower should be performed within, e.g. 20 milliseconds. This is considered as one of hard timing constraints. This is due to that if the data is not sent on time, as we mentioned before, a fatal outcome might occur. Hence, in real-time systems, timing constraints are vital and processing of the data is done in real-time.

When new updates are introduced into a real-time system, they may result in a degradation of the performance of some of the tasks of that system. These degradations may result in a delay to some of the tasks with hard timing constraints. Therefore it is important to verify that new updates that are introduced into the system may not affect the system's performance in a negative way based on the definition above for performance based backwards compatibility.

2.3.2 Distributed Systems

A distributed system is a collection of independent, autonomous machines that are connected through a network and distribution middleware [TS14]. In a distributed system, the machines coordinate their activities and share the resources of the system through methods of synchronization, so that to a computer user, a distributed system appears as a single coherent system [TS14]. The nodes of a distributed system communicate with each other through signals of message passing using certain communication protocols. The order of communication between the nodes is very important as it defines the execution of one of the tasks of the system. When new updates are introduced into a distributed system, they may result in changes in the execution of one of the tasks of the system. This may cause the order of communication to be disturbed which results in the system crashing because the wrong message arrived to a particular node. Therefore, it is necessary to verify that the execution of the system's components is not changed unless it was done on purpose, based on the definition of the *execution based backwards compatibility*.

2.3.3 Ericsson's SGSN-MME Product

The reason behind our focus on distributed real-time systems is because it is a common type of systems that are being developed and used within the field of telecommunication. This decision was also motivated by the fact that we were able to find an ideal environment where the study can be performed. The study, investigation, implementation and testing parts of the thesis are done at Ericsson [WWW1]. Ericsson is one of the leading enterprises within the field of telecommunication and networking, with many products that are being used for communication on a global scale. Most of Ericsson's products have full scale implementation of parallel dis-

tributed fault tolerant and real-time systems, thus, their products are particularly useful for the scope of our thesis.

The particular product that the thesis is based on is the Serving GPRS Support Node - Mobility Management Entity (SGSN-MME) [WWW2]. It is a fault tolerant distributed real-time system that is responsible for the delivery of data packets from and to the mobile stations within its geographical service area. The system's tasks include packet routing and transfer, mobility management (attach/detach and location management), logical link management, and authentication and charging functions. The product is made up of several components; these components are developed by different departments within the company. The software side of the product is made up of several layers following the general standards of software architecture, which include application layer, business layer, middleware and operating system level layers, depicted in Figure (1).

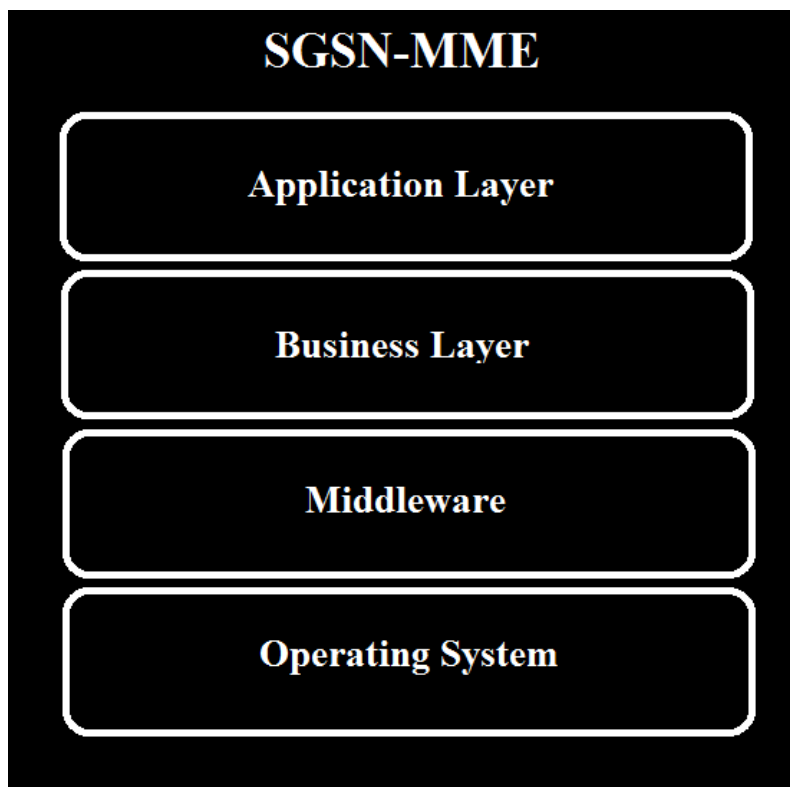


Figure 1. SGSN-MME Overview

The product also has a number of tests from its planning and development phase until its deployment. These phases are very large and complex and there are whole frameworks and teams that work on managing them. However, the test phases still follow the general standards of testing, which include unit testing, component testing, functional tests as well as system testing. The product is part of this thesis's study because it is an ideal environment for testing the prototypes of the algorithms.

2.3.4 Ericsson's View of Backwards Compatibility in the SGSN-MME

Ericsson releases a new hardware base of the SGSN-MME for its customers every few years, whilst the new software updates are released every month. The updates contain a number of firmware and software changes as well as new features, functionalities and enhancements to the system's performance. This results in a large number of software that must be constantly tested, in order to ensure that the software can run without any problems. However, in spite of the various tests, these features may end up causing unexpected problems to specific parts of the system or the system as a whole, so customers are usually reluctant to accept those updates. It is important for Ericsson to ensure their customers that the newly installed updates and features will not affect the original behavior and performance of the system in a negative way. Therefore, the intention is to provide both the company and the customers with a method that can be used to test the integrity of these updates, thereby verifying their safety on the system. This is achieved by verifying that the previous behavior of the system remains unaffected by the new features that were introduced to the product. Therefore, Ericsson is looking for the following requirements in terms of backwards compatibility:

- Narrowing the problem of backwards compatibility using proper definitions that can be related to the company's system.
- Overview of the existing strategies for verifying backwards compatibility in both academia and the industry.
- Proper classification of the strategies to identify where in the system they can be implemented.
- Recommendations on the most suitable algorithms for verifying backwards compatibility in their system.
- Prototypes of several algorithms as proof of concept, to provide an insight on their behavior in the system.
 - The prototypes must be automated.
 - The prototypes must indicate which elements of the software are incompatible.
- Indicating which phase of the software development process can an algorithm be used.

Ericsson's view on the backwards compatibility algorithms is that they are means to gain the customer's trust in accepting those updates, knowing that they will not affect the behavior and performance of the system in any way.

2.4 Related Work

There have been made several attempts at studies in the context of verifying backwards compatibility in terms of software modules, libraries and IP communication; however, to the best of author's knowledge, none of the studies that are aimed towards verifying backwards compatibility in distributed real-time systems have been published. In this section, we shall provide a couple of examples of studies that have been done in order to establish solutions for achieving backwards compatibility.

The work by [VO08] addresses the problem of ensuring IP communication in terms of communicating nodes on the network. In particular, when there is an *edge network* [VO08], and that an edge network is connected to two other edge networks via a Six/One router [VO08], which belong to different providers, the issue arises in terms of communication when there exist two nodes in different edge networks and they wish to communicate. The reason that a communication cannot be established is that the *edge addresses* have their IP addresses locally, that is, not visible to outside network. One of the solutions, which is discussed by [VO08], is to make the edge address *transit*, which would make the address globally routable. And in this particular case, the communication would take place upon such a change. However, if the edge address is globally routable, then it cannot be reached by legacy networks, where the edge address was previously established as local. Hence, the issue of backwards compatibility is that if some component of the system is upgraded, the legacy version software cannot communicate with it.

The solution that is proposed by the author is to create a protocol in Six/One router that performs a mapping between each edge address onto one unique transit address per edge, and each transit address is mapped to a single unique edge address [VO08]. Additionally to this, in the functioning protocol of Six/One router, hosts in edge networks can be reached both via their edge addresses and *transit addresses*. By fulfilling these two criteria, nodes that support legacy network can communicate via edge addresses and nodes that support transit addresses can communicate globally, thus, in this sense, backwards compatibility is achieved. However, in relation to the conducted study, such an algorithm is not feasible as the SGSN-MME system is more complex and it is not used in the context of edge networks.

Another related study by [WH11] was aimed towards developing a completely abstract trace-based semantics for class libraries in object-oriented languages, for instance, class libraries in *Java* programming language. In the context of the study performed by [WH11], a language of their choosing was indeed *Java*, and the authors verified backwards compatibility on *Java*-like sealed packages.

The approach taken by the authors to verify backwards compatibility in terms of preserving the old behavior of the legacy implementation in all possible contexts was to use standard operational semantics. This is done so that if there is change of control between the client context and the library context (the code that belongs to the library and to the client context), it must be explicit in terms of interaction labels. These interaction labels record the input/output operations between the

library and context code, for instance, parameters that were passed in to a method call [WH11]. Thus, by obtaining these labels, the authors traced them, in the sense that they abstracted how the data is represented in the *heap* (heap/stack in terms of computer memory), introduced support for hiding classes and lastly, abstracted package denotations were provided. These trace semantics allowed to monitor the behavior of the library program. Hence, once the initial behavior is obtained, all that is left to do is to perform the desired modifications to the library, and perform the proof of concept. According to [WH11], the proof for checking trace-based semantics for class libraries was based on specialized simulation relations between the program configurations of program contexts for both the old and the new library implementation. Indeed, this approach is valid, however, as with the approach by [VO08], it cannot be used for verifying backwards compatibility in distributed real-time systems nor this algorithm can be altered to reuse it in some way so that it can be applied in the context of this study.

It is worth mentioning that the work by [VO08] addresses the aspect that the study pertains to routers, which communicate in real-time, whilst the study by [WH11] highlights the software aspect only. However, as it was mentioned before, the backwards compatibility algorithms from the two papers cannot be applied to the current study, as the setting of the study is to verify backwards compatibility in distributed real-time systems.

2.5 Limitations

Having analyzed the scope of the thesis and what needs to be done in order to fulfill its objectives, the following are some of the major limitations of the thesis:

- The focus of the study pertains only to the communication products of Ericsson; this thesis does not include other products from this company or other companies.
- The study results in algorithms for verifying backwards compatibility in distributed real-time systems specifically.
- The study pertains, in particular, to the SGSN-MME product of Ericsson, however the algorithms are meant to be generic enough to be applied in distributed real-time systems in general.
- The study results in implementation of the algorithms in the form of prototypes for testing purposes.
- Certain details of the thesis will be ambiguous as to not disclose sensitive information of the company due to confidentiality.
- Due to the fact that the study is limited to the time period of the thesis, not all of the algorithms will be tested on the system through prototyping.

3

Methodology

THIS section discusses in details the process through which the thesis is carried out. It includes the objectives of the thesis, the research questions as well as explaining the research approach, and how the literature review process was carried out, along with the expected challenges.

3.1 Research Questions

Having presented information about backwards compatibility in the previous sections, it is evident that there is room for performing study regarding this matter. The research questions are formulated in the following manner so that they capture the essence of this study. The research questions are listed below:

RQ 1. What kind of algorithms and strategies can be used for verifying backwards compatibility in general?

RQ 2. Which of the identified algorithms can be used for verifying backwards compatibility in distributed real-time systems?

RQ 3. Which algorithms can be combined together to create a unified algorithm for verifying backwards compatibility?

The authors have provided several definitions for different types of backwards compatibility (described in section 2.1), therefore the first research questions is aimed at identifying all existing algorithms and strategies that can verify backwards compatibility in general, regardless to which definition the algorithm belongs. The purpose of the *RQ 1* is to identify and assess whether one of the already existing algorithms can be altered and applied to the current study. However, *RQ 2* focuses more on algorithms that address the aspects of the distributed real-time systems directly. The purpose of *RQ 3* is to identify which of the algorithms that were developed by the authors can be combined together into a single algorithm for verifying backwards compatibility. Hence, these questions constitute the formal goal of the thesis which is identifying algorithms for verifying backwards compatibility in distributed real-time systems.

3.2 Objectives

The main purpose of this thesis is to identify algorithms and strategies for verifying backwards compatibility for distributed real-time systems. This is done by fulfilling the following objectives:

- Identify the existing algorithms and strategies within academia for verifying backwards compatibility in general.
- Study the existing ideas that are proposed by the company.
- Classify the algorithms and strategies based on their types to identify where in the system they can be used.
- Formulate a hypothesis for an algorithm for verifying the component's backwards compatibility for a certain component in the system.
- Implement and verify a prototype of the algorithms that are formulated.
- Document the algorithms and their performance in the system.
- Recommend the most suitable algorithm(s) to use for verifying backwards compatibility for the system.
- If possible, formulate a unified algorithm that can work on the various parts of the distributed real time system.

The objectives, the rationale behind them as well as means of accomplishment are explained in greater detail in the subsequent *Table 1*.

Step	Method	Expected Outcome	Rationale
1.	Literature & Academic review	Algorithms & Strategies	Identifying existing algorithms and strategies in academia and industry is required to establish a reference baseline for the contribution of the thesis.
2.	Type Classification	Taxonomy	Identifying where in the system the algorithms can be used is needed to guide our further development of the method
3.	Design	Theory of the algorithm	Providing detailed design of the algorithm as a theory to be tested through prototyping
4.	Prototyping	Prototype of the Algorithm	Validating, testing and matching the results against the theorized results of the algorithm
5.	Documentation	Results & Performance	Providing detailed documentation of the results of the algorithm and its behavior in the system
6.	Comparing Results	Recommendation of Algorithms	Analysis of the algorithms results and recommending where they can be used in the system

Table 1. Research Objectives and Expected Outcomes.

The details of how these objectives will be achieved are explained in the *research approach* section.

3.3 Selecting Research Methodology

The most suitable choice for a research methodology for this thesis is *design research* [CJB04, VK04], where the goal is to create a service, feature or a full product. Another candidate for research methodology to be used in this study is *action research* [ALM99], which might be a suitable alternative. However, most theses that use this type of research methodology have a tendency of being done over a very long period of time, therefore, this research methodology had to be excluded. Even so, the general process for the *design research* described by [CJB04] cannot be completely applied to the study; therefore, the study follows a process that is similar and based on the design research. The research process that this study follows is depicted in Figure (2). The reason for choosing this research methodology is due to the fact that the thesis includes the design and creation of algorithms for verifying backwards compatibility within distributed real-time systems.

3.4 Research Approach

The study is divided into a number of phases which are illustrated in Figure (2). The first phase, which we have labelled as the *planning phase*, involves preparation for the entire study. This includes creating an acceptable thesis proposal that describes the problem and the scientific approach in dealing with it, as well as the research questions and goals of the study. This phase also includes creating a planning report that includes objectives of the study, along with their milestones and a projected time plan based on those milestones. During this phase, we have studied existing research within the field of backwards compatibility, in order to understand the current progress and establish the gap to be studied. This phase also includes studying and understanding the system that the study is based on at Ericsson, which is the SGSN-MME [WWW2].

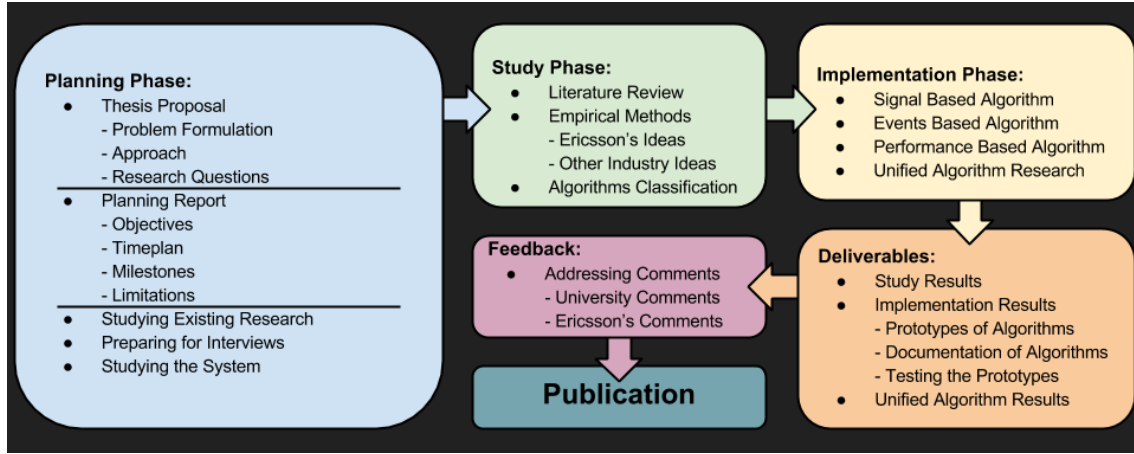


Figure 2. The Research Process

The SGSN-MME system is run through a number of phases of testing from its planning and development phase until it reaches the customer. These test phases are very large and complex and there are whole frameworks and teams that work on them. Thus, it is of great importance to have technical interviews with experts of those parts and test phases of the system, in order to gain the necessary technical understanding and narrow the scope of the study to specific manageable components. In order to ensure that we gain as much information as possible from the technical interviews with the experts, the planning phase also includes preparation for the interviews and setting up the questions early on.

The second phase, which we have labelled as the *study phase*, includes collecting and studying the existing algorithms and strategies for verifying backwards compatibility in both academia and the industry. This phase is divided into two parts; the first is collection of the algorithms and strategies. This includes performing a literature review where a number of papers are studied in order to collect the algorithms and strategies for verifying backwards compatibility. This part also involves gathering empirical algorithms and strategies for verifying backwards compatibility from the

industry, particularly ideas that are suggested by Ericsson, as well as other ideas that are recommended by other companies and people who have experience in this field. The second part of this phase involves studying the collected data and classifying them so that it would be easier to choose an algorithm based on the type that it belongs to. By the end of this phase there should be a form of a classification or taxonomy map, which contains all the algorithms and strategies that were collected, studied and classified.

The third phase, which we have labelled as *implementation phase*, includes the creation of prototypes of the selected algorithms for verifying backwards compatibility. Ideally, the algorithms should be generic enough to be applied to the various components of the system and can be used at the various test phases of the system. The algorithms will cover verifying *execution based backwards compatibility* of the component, as well as its *performance based backwards compatibility*. Since the authors intend to develop multiple algorithms, this phase was made into a cycle which will be followed while developing the algorithms; the cycle can be seen in Figure (3).

The *first step* of this cycle will involve identifying which components of the system are best for starting backwards compatibility verification. The idea is to select a component that is representative of a large portion of the system so that the algorithm can be broad enough to be applied on the other components with minimum changes. The *second step* of the cycle involves the formulation of a hypothesis for an algorithm that can be used for verifying backwards compatibility for that component.

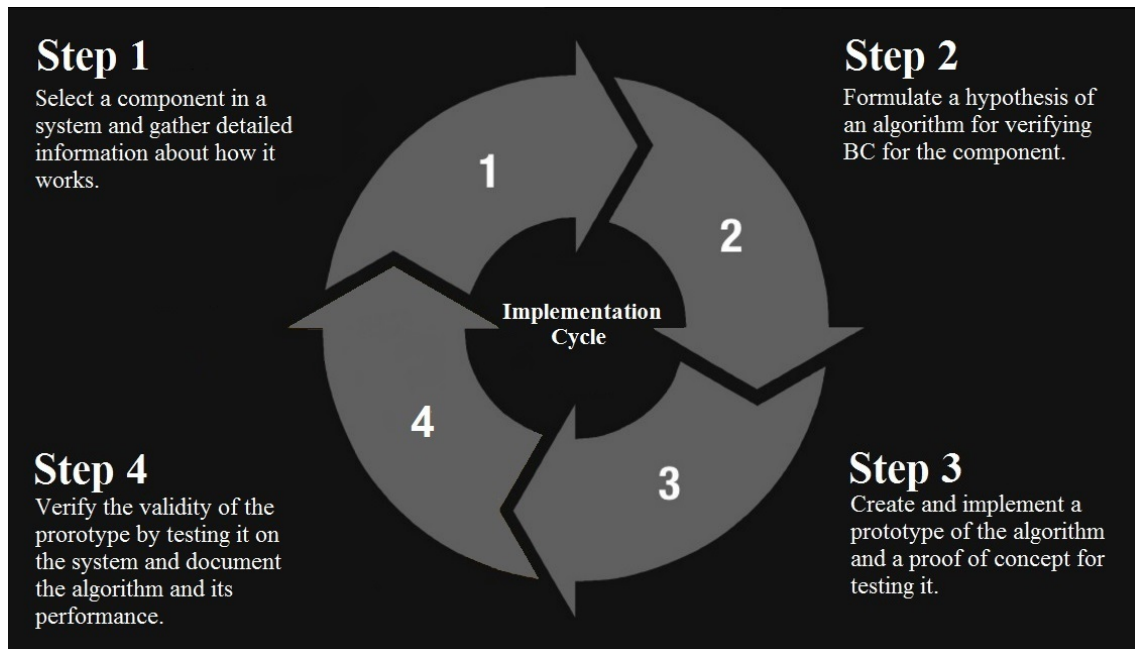


Figure 3. The Implementation Cycle

The *third step* of the cycle includes the implementation of a testable prototype for the algorithm and the creation of proofs of concept in order to test the prototype.

The *fourth step* of the cycle involves thorough testing of the prototype through the proof of concepts within the system and documenting the results of the algorithm along with the performance of the prototype.

It is worth noting that this cycle is a general one and there might be slight alterations in it, depending on the algorithm that is being developed. It is also worth noting that due to time constraints, it is not expected that this cycle will be run on all of the components and test phases of the system. However, it is assumed that some of the components of the system will share similar characteristics that will allow some of the hypothesis that are formulated to be reused.

The *implementation phase* also includes merging the formulated algorithms together into one *unified algorithm* that could be used for measuring the level of backwards compatibility for the various types of definitions that were mentioned in the background section of this thesis. This particular algorithm will be done in order to answer the last research question, whether or not it is possible to create such a unified algorithm for distributed real-time systems. In the case where we are successful in creating such an algorithm, the same cycle of development will be followed in creating and implementing a prototype for the algorithm and then test it on the system in order to verify its behavior and document its performance.

The fourth phase that is labelled as *deliverables* of this thesis, includes the results of the study phase, which are the algorithms and strategies that were collected from both the industry as well as academia, with description of each one and classification of all algorithms and strategies into a useful taxonomy map. The deliverables also include the results of the implementation phase that contain the description of the algorithm with a prototype that illustrates how it works, proofs of concept that verify the validity, as well as documentation of the results of the algorithm and the performance of the prototype on the system.

The final phase involves the presentation of the findings of the thesis to both the university and the company and addressing all *feedback and comments* that are given to the authors at which point the thesis will be ready for *publication*.

3.5 Literature Review Process

This section provides the description of the literature review process performed while trying to identify what kind of already existing algorithms and strategies are used to verify backwards compatibility in the general sense. That is, backwards compatibility is not only limited to *execution based backwards compatibility* or *performance based backwards compatibility*, but to other types of backwards compatibility definitions.

3.5.1 Data Sources and Search Strategy

In order to gather as much information as possible, the primary sources of data collection were the four digital libraries: *ACM Digital Library*, *IEEE Xplore*, *Inspec Digital library* and *Springer Link*. These digital libraries are well known and are considered among the most reliable sources of information in terms of computer science articles, journals or books.

The conducted search query had to reflect the aim of this thesis, namely, to gather knowledge about empirical algorithms and strategies that exist in the industry for verifying backwards compatibility. Therefore, the search query consisted of the following set of words, that was conducted with the aim of Boolean expressions “AND” and “OR”. The initial string query used was the following: “backwards compatibility and software”. After performing the search on the aforementioned libraries, the amount of articles obtained was limited. However, it is also the case that within the industry verifying backwards compatibility is sometimes referred to as backward compatibility, omitting the syllable ‘s’ at the end. Hence, the second query used was “backward compatibility and software”. Once again, the results in terms of articles obtained did not vary. The digital libraries produced almost identical results to the first search query (in terms of quantity and content), however, only *IEEE Xplore digital library* had slight variations in terms of quantity of the articles. The overall quantity of obtained papers is depicted in Figure (4).

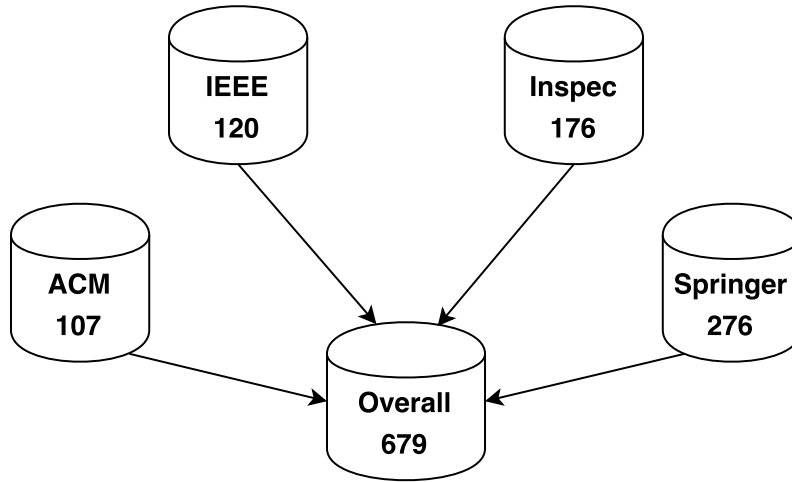


Figure 4. Articles from Digital Databases

As we can see, the amount of articles gathered was 679, however, after the study selection procedure, only a small amount of articles was included in this study. Next we proceed with explanation of the study selection process.

3.5.2 Study Selection process

Defining the criteria according to which either to include an article for further inspection or not is a vital matter. The criteria were as follows:

- The study should be about verifying backwards compatibility within the context that is software related.
- The study must be a conference publication, a journal or a magazine.
- The study must be written in English.

As exemplified in Figure (4), the initial amount of articles gathered was 679, however, after inspecting *titles* and *abstracts* of these papers, only 9 papers were selected for full paper analysis.

4

Algorithms Design

IN this section the algorithms that have been developed throughout the thesis are described in detail. For each algorithm the description of the intended behavior of the algorithm is provided, its implementation, and how it was tested.

4.1 Signal Based Algorithms

The first algorithm that is studied within the first implementation cycle of the thesis is an abstract level algorithm, which focuses on the execution sequence of the software of a component within the system. In particular, the algorithm focuses on the sequence of communication signals between the nodes of the component. The reason behind choosing this type of algorithm is the following:

- It is an ideal algorithm for following the execution of a program at a higher level, without diving into the details of what the system is doing.
- The algorithm is ideal for achieving the goal of creating a generic solution that could be applied to all kind of programs that utilize network communication.
- The system that is being studied is so complex that even a single component of a system is large enough to be a fully automated functioning system. Thus, this type of algorithm would be ideal for an abstract view of the system's behavior.
- This type of algorithm is ideal for distributed real-time systems, where the intent is to follow the external communication between computer nodes.

Before explaining how the algorithm works, it is important to disambiguate some of the details that encompass the algorithm.

4.1.1 Signals, Packets & Protocols

In computer communication, a group of nodes can communicate with each other via signals. Whenever a machine, such as a server, stationary computer, notebook, touchpad or a mobile phone, performs communication on a network level, it is

sending and receiving signals in real-time. The signals are communicated in the form of network packets, which are units of data that are sent from a source node to a destination node over a network. Each packet is numbered uniquely and includes information such as the source address, destination address, the protocol used, the size of the packet, and error detection checksums.

The sent/received signals must abide certain rules of communication. Communication protocols are rules that allow two or more nodes in a network to send and receive data. These rules define the type of data that can be transmitted, the synchronization of the communication and possible error recovery methods. There are many types of communication protocols, some are public and can be used by all entities and some are private. Some of the most popular communication protocols include TCP/IP, UDP and FTP.

4.1.2 Capturing Execution Signals

The first part of the algorithm is to register all the packets that are transmitted and received between the communicating nodes when the component is executing one of its tasks. This is important because it provides an abstract layer of the sequence of execution of the distributed nodes inside the component. Thus, the captured packets represent the essence of the execution of one of the tasks of the component. In order to capture the full trace of communication between the nodes, a traffic analysis tool must be used in order to record all the packets that are sent and received in the network, hence, it is important to identify a list of tools and decide on the most suitable one for this purpose.

4.1.3 Identified Traffic Analysis Tools

In order to monitor the communication between nodes in a network, certain traffic analysis tools are used. They are mainly used for data analysis, troubleshooting, communication protocol development. The tools take care of registering all traces of packets on a specific hardware interface that are transmitted and received between the nodes in a network. Therefore, all the necessary network packet information will be recorded.

A total number of eight network traffic analysis tools are identified. All the identified tools are open source. Most of the tools are similar, however, there is difference in the implementation of the tools themselves, as well as how the gathered network packets are captured and presented. The identified tools, along with a brief explanation of each tool, are presented below:

- *DSniff*
DSniff is a network sniffer application that allows the user to monitor the traffic on the specified interface, as well as the possibility of disrupting traffic from other clients that are not on the same computer [WWW3]. Initially, the

tool was developed to showcase how networks are insecure and how data traffic can be manipulated. Otherwise, the tool provides great possibilities for a user to either trace/tamper the ongoing network traffic [WWW3].

- *EtherApe*

EtherApe is a network traffic analysis tool. One of the main features that it provides is that the tool can display complete ingoing/outgoing network traffic on a particular interface [WWW4]. Moreover, it can summarize the protocols and the node's table, where node table contains IP addresses with which the computer is communicating in real time [WWW4]. As of today, the only available version of this product is for the UNIX operating system.

- *Netsniff-ng*

Netsniff-ng is another network utility tool that can perform analysis on the captured network packets [WWW5]. However, out of all the tools mentioned so far, in terms of its capabilities, this tool has the advantage. As an example, this tool can generate wire-rate traffic (the rate is approximately $2 * 10^8$ meters per second), replay packets, and perform an autonomous system trace route [WWW5].

- *Packet analyzer*

Packet analyzer as the name suggests, this is an application that can allow the user to monitor network traffic in real-time and in offline mode [WWW6]. That is, this program, if configured correctly, can store the passing data, for example, on a disk [WWW6]. Afterwards the data can be used for closer inspection.

- *PacketSquare*

PacketSquare is a tool for testing network protocols. It uses the “.pcap” (captured network layer packets) files for testing, analyzing, and troubleshooting the network. Moreover, the tool can analyze network traffic of more complex network systems, like router switches or firewalls [WWW7]. Most interesting capability of this tool is that with its help, it is possible to modify certain fields within protocols of the captured packets, or duplicate/delete packets [WWW7].

- *Scapy*

This program can perform packet manipulation, decode packets, generate network traffic and even do *traceroute* operations. Since *traceroute* has not been mentioned before, it is imperative that an explanation be given [WWW8]. What the *traceroute* does is that the tool is used to perform network diagnostics. In the sense that if a network expert wishes to see the path a packet took when it was sent from node A to node B, this tool will display all the intermediate routers that the packet was routed to.

- *Tcpdump*

Tcpdump is a packet analyzer that can be run on both UNIX and Windows operating systems [WWW9]. Its main purpose is to display network traffic on an interface via which the computer is communicating over the network. It

is open source, however, there are many limitations of this tool, e.g. the tool does not provide many options besides displaying the captured network layer packets [WWW9].

- *Wireshark*

Wireshark is a cross platform open source packet analyzer that runs on UNIX, Windows and Mac OS and other known operating systems [WWW10]. It is the most popular and most widely used tool for capturing and analyzing network communication [WWW10]. It includes a high level graphical user interface and offers a wide range of filtering which makes it very useful for identifying patterns. It also comes with a terminal based version called *tshark*, however it is limited compared to the full program.

4.1.4 Execution Signals

Having analyzed the capabilities of each network traffic analysis tool, it was decided that wireshark is the most suitable packet analyzing tool for the purpose of this algorithm. This is because wireshark offers great level of flexibility through the filtering option within the program; this is depicted in Figure (5). Furthermore, wireshark also includes a console based version tshark.

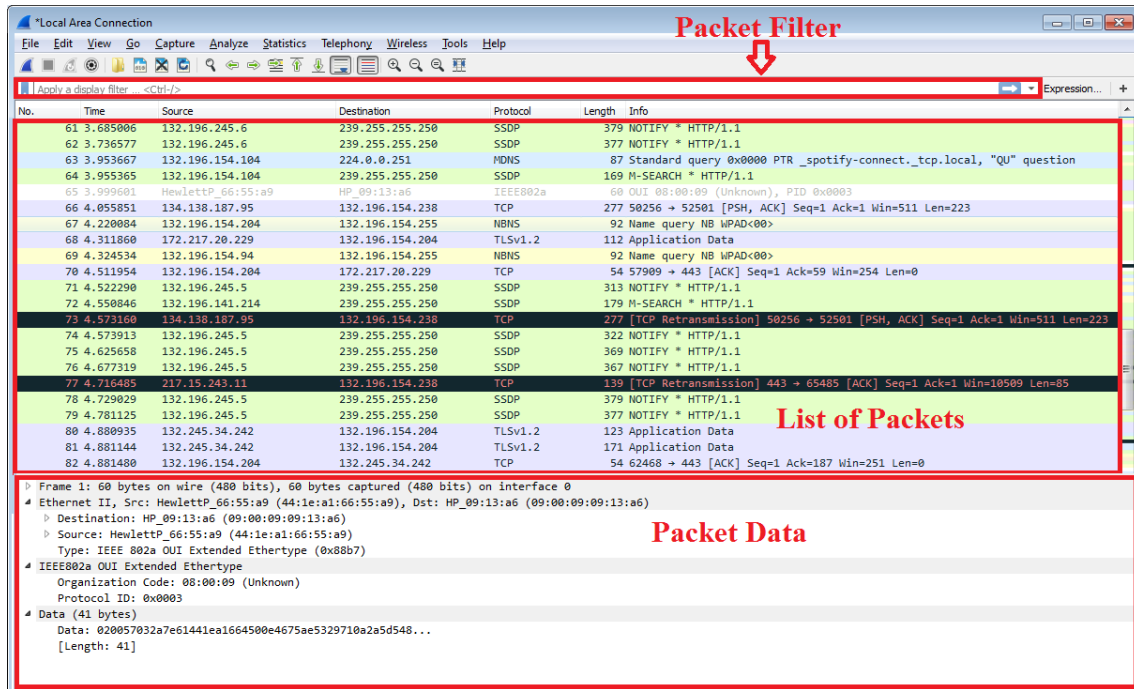


Figure 5. Example of a Wireshark Capture

Now that a suitable tool for capturing packets has been identified, we can explain the second and most important part of the algorithm, which verifies execution based backwards compatibility of a component. As mentioned before, the goal of the algorithm is to identify the execution sequence of the component through the sequence

of communication between its nodes while it is running. By using wireshark it is possible to capture all the packets that are being transmitted and received between the nodes within the component while it is executing one of its tasks. The captured packets are stored in a file of the type “.cap”.

An example of a five second capture while the machine is idling can be seen in Figure (5). We can see that even in a five seconds capture, there are already over 80 communication packets that were captured by wireshark. However, the amount of packets that are usually captured when a machine is performing a task is very large. This is due to the fact that there are many packets that are not related to the execution of the software, such as simple echoes that checks whether a node is alive or not. Thus, it becomes necessary to study the packets of the captured file and try to filter out the ones that are important for the execution of the program.

4.1.5 Identified Patterns

By closely observing the captured files with wireshark we were able to identify that certain packets share the use of certain communication protocols, which are unique to the execution of the program. These packets describe the sequence of execution of the program with details about the communication that occurs between the nodes, thus they represent the essence of the execution of that scenario. The packets and their sequence of execution and communication differ from one test scenario of the component to another. Therefore, a *pattern* is established which represents a unique sequence of packets that pertain to a set of communication protocols. In order to capture the packets that correspond to a pattern and exclude other packets that are not relevant to the execution sequence of the program, certain search queries are used. The search queries are abstract filters that are used for extracting packets that pertain to a specific pattern, thereby minimizing the size of the captured file. It is important to note that the identified patterns are not all the *patterns* that are used in every scenario of the component, as there are a large number of them. Nevertheless, they are enough to help formulate a testable prototype. The protocols that resulted from the use of those patterns are:

- **GTPV2**

Nodes that communicate using this communication protocol were filtered out using the search query “(*gtpv2* && *ipv6*) || (*gtpv2* && *ip*) && !(*icmp*)”. The query filters out the packets that use the *gtpv2* communication protocol with *ip* and *ipv6*, while excluding echo messages using which is done using !(*icmp*). The query and its resulting packets of the execution of one of the component’s tasks can be seen in Figure (6).

- **GIOP**

This sequence of packets is filtered out using the search query “giop”.

- **S1AP**

Packets communicated using this protocol are filtered out using the query “s1ap”.

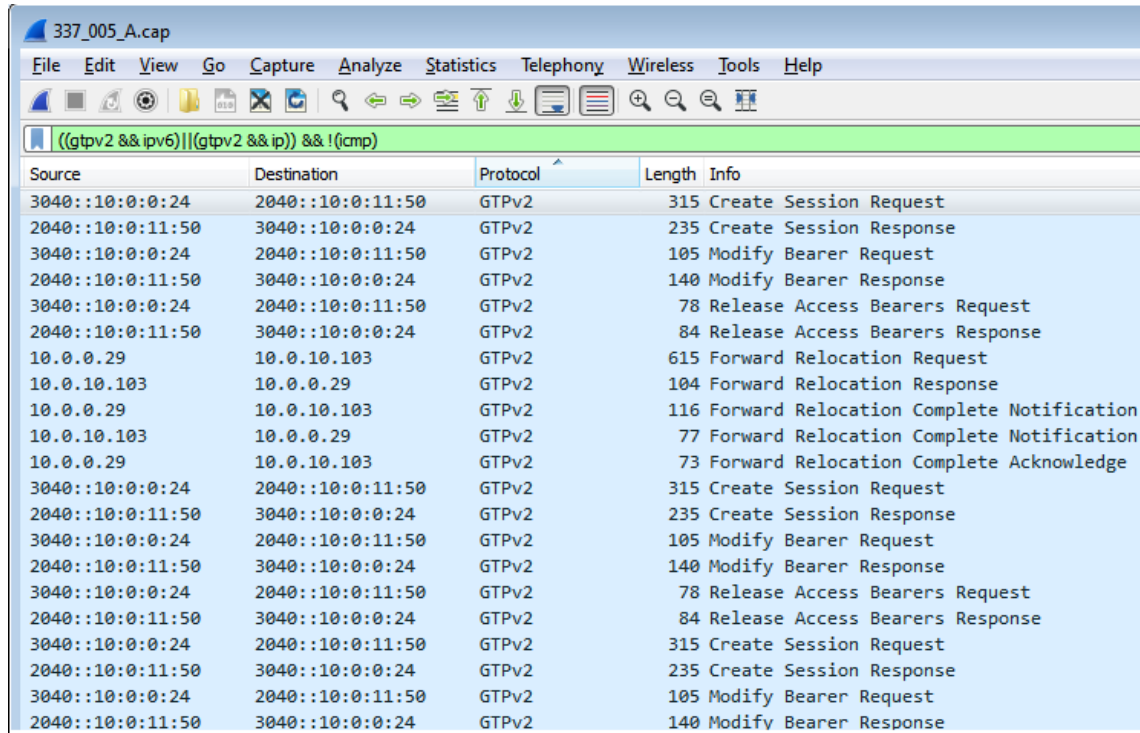
- **SCCP**

The packets that use this protocol were filtered out using the query “sccp”.

- **BSSGP**

Nodes that communicate packets using this protocol were filtered out using the query “bssgp”.

The sequence of packets that pertain to one of the above patterns are unique to a particular test scenario of the component. In order to confirm their uniqueness, wireshark was used to capture packets for one hour, without running any test scenarios of the component. The resulting packets should therefore contain no packets that pertain to any of the identified patterns, thus verifying that they are patterns that are unique to the component.



The screenshot shows the Wireshark interface with a search query entered in the filter field: `((gtpv2 && ipv6)||gtpv2 && ip) && !(icmp)`. Below the filter, a list of 20 filtered packets is displayed, all of which are GTPv2 packets. The table columns are Source, Destination, Protocol, Length, and Info.

Source	Destination	Protocol	Length	Info
3040::10:0:0:24	2040::10:0:11:50	GTPv2	315	Create Session Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	235	Create Session Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	105	Modify Bearer Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	140	Modify Bearer Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	78	Release Access Bearers Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	84	Release Access Bearers Response
10.0.0.29	10.0.10.103	GTPv2	615	Forward Relocation Request
10.0.10.103	10.0.0.29	GTPv2	104	Forward Relocation Response
10.0.0.29	10.0.10.103	GTPv2	116	Forward Relocation Complete Notification
10.0.10.103	10.0.0.29	GTPv2	77	Forward Relocation Complete Notification
10.0.0.29	10.0.10.103	GTPv2	73	Forward Relocation Complete Acknowledge
3040::10:0:0:24	2040::10:0:11:50	GTPv2	315	Create Session Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	235	Create Session Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	105	Modify Bearer Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	140	Modify Bearer Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	78	Release Access Bearers Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	84	Release Access Bearers Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	315	Create Session Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	235	Create Session Response
3040::10:0:0:24	2040::10:0:11:50	GTPv2	105	Modify Bearer Request
2040::10:0:11:50	3040::10:0:0:24	GTPv2	140	Modify Bearer Response

Figure 6. Search Query and its Resulting Packets Wireshark

The search queries are entered into the green field in wireshark, which is used for filtering the packets that correspond to a pattern. The search query consists of the protocols *gtpv2* in combination with IPv4 and IPv6 addresses. Additionally, “!(icmp)”, which prevents echo messages, is applied on the resulting query in order to obtain the desired sequence of execution of the test program, which is depicted in the blue part of Figure (6). All packets that pertain to the aforementioned patterns are filtered in a similar manner using their respective search queries.

4.1.6 Verifying Backwards Compatibility

The unique execution signals that are captured, exemplified in the preceding section, will be used for verifying whether the component's software is *execution based backwards compatible* or not. The verification process begins by capturing a trace of the execution of one of the component's tasks before an update is introduced into it. The captured file contains all the communication signals between the nodes of the component during the execution of one of its tasks. Thus, the most suitable way to capture this trace is to run one of the test cases of the component, which tests the execution of one its tasks.

Afterwards, using the patterns that have been identified within the component, all the unique packets of that test case are extracted and stored in their proper order, thereby ensuring the correct sequence of execution. This is represented in an abstract level, that is, the initial captured file with the captures signals, depicted on the left hand side of Figure (7). The exact same procedure is repeated for the execution of the same test case, however this time it is done in the build that contains the updated version of the software of that component (right hand side of Figure 7). Thereby the full execution of the test case will be recorded for both versions of the software.

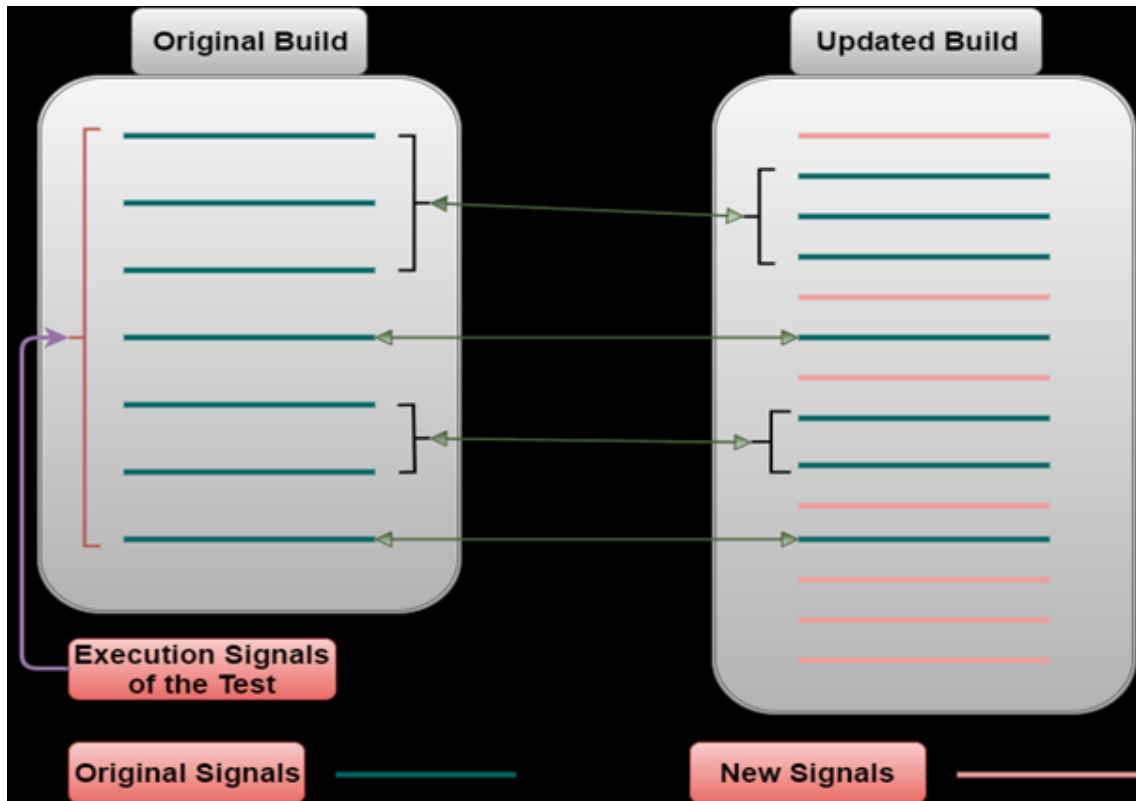


Figure 7. Comparing the Execution Signals between Software Builds

The next step is to match the execution sequence of the original version of the software against that of the updated version. According to the definition of *execution*

based backwards compatibility, an update that is introduced into the system's software should not change the behavior of that software, however, it can expand on it. That means that execution sequence that was recorded for the test case in the original build of the software must exist in the exact same order, within the execution sequence of the updated software. Thus, the sequence that was recorded while testing the original software must be a subset of the sequence that was recorded after the update is introduced; the matching of signals can be seen in Figure (7) (denoted by a green arrow).

The execution sequence of the updated software might contain more signals representing the updated parts of the software, this is depicted on the right hand side of Figure (7); however those are of no interest, since the main point of focus is checking if the software continues to perform its original task as it did. Thus, if all the captured signals that represent the execution sequence of the original software exist in the captured signals of the updated software, then we can say that the software's update is *execution based backwards compatible*.

4.1.7 Prototyping the Algorithm

In order to establish the validity of the algorithm, a prototype was created for testing it. The first step of the prototype is capturing the execution signals of the test scenarios of the component; this is done using the trace analysis tool wireshark. The second step is the extraction of packets based on the patterns that were identified for the component that is being tested. This is done using the terminal version of wireshark, which is tshark. The patterns are inputted into *tshark* using its filtering feature which result in a list of packets that share the same communication protocols. Tshark then extracts the filtered packets and stores them in a text file in the exact order they were found. This procedure is done for two test cases of the software, one before the update and one after the update.

Next, using bash scripting, a program checks whether the execution signals of a given test scenario of the original software exist in the updated version of the software. The program returns a percentage which represents how much of the execution signals from the original build exist in the updated ones. This percentage represents the backwards compatibility level of the updated software [PSS10, Sta12]. Based on recommendations from Ericsson, classification of the percentage is done in a manner that helps to assess the level of backwards compatibility:

- **95% or More**

This indicates that the updated version is backwards compatible and can be introduced into the system.

- **70% - 95%**

The updated version is semi compatible and must be further examined in order to increase its compatibility.

- **0% - 70%**

The updated version is not compatible and must undergo thorough examination in order to identify all the compatibility issues.

The script also returns all the signals that exist in the original build and do not exist in the updated version; this can be used for identifying the problems that the updated versions have.

4.1.8 Testing of the Prototype

The prototype of the algorithm has been thoroughly tested to ensure its validity. The testing of the prototype has been on different versions of the software of the EPS component. The component goes through two types of tests:

- *Test cases*

Test cases are the simplest of ways to test a component's behavior. Each test case simulates some task the component is able to execute. Test cases are usually small and take relatively short time to execute.

- *Test Suites*

Test suites are collections of test cases that simulate the performance of groups of tasks by the component. Most test suites are very large and take a long time to be fully tested.

In order to test the prototype of the algorithm on the system and produce verifiable results of backwards compatibility checks, two types of proofs are created [HS02]. The proofs help verifying the validity of the algorithm as well as understanding the results of the prototype [HS02]. The proofs are tested on both test cases as well as test suites of the component and they are as follows:

- *Validating the Prototype - Simple Proof of Concept*

This proof of concept involves testing a software build against itself. The logic behind it is to prove that the prototype behaves as expected when testing software against itself [HS02]. The proof is done as follows:

- If the prototype is ran on two identical runs of a test scenario of the software in the same build (i.e. test case and a re-run of the same test case), the prototype should result in 100% compatibility level.
- If the prototype is ran on a test scenario of the build against a re-run of the test, however, the second test is interrupted half way through, the algorithm should result in between 50% to 70% compatibility level.

- *Comparing Builds*

This proof of concept involves testing a software build against an updated

version of the software [HS02]. This proof is the actual verification of execution based backwards compatibility between different versions of the software. The proof is done as follows:

- If the prototype is ran on a test scenario of the original build against itself it should result in 100% compatibility level.
- If the prototype is ran on a test scenario of the updated version of the software against that same updated version, it should result in 100% compatibility level.
- If the prototype is ran on a test scenario of the original build against a test scenario of the updated build, it should return over 95% compatibility level in order for the updated version to be backwards compatible.
- If the prototype is ran on a test scenario of the original build against a test scenario of the updated build and returns between 70% to 95% compatibility level then it is semi compatible and should be further analyzed.

The results of the tests that were performed on the prototype of this algorithm can be seen in Chapter 5 which contains the results of this thesis, section (5.2.1 Signal Based Algorithms Results).

4.2 Events Based Algorithms

The second implementation cycle of this thesis also focuses on verifying execution based backwards compatibility, however, the goal is to develop an algorithm that is more focused on the details of what the component of a system is doing. The reason behind choosing this type of algorithm is the following:

- Acquire key details of execution of a component's task.
- It is an ideal algorithm for following the execution of a program at a low level.

The algorithm is useful for developers and testers that are interested in maintaining backwards compatibility of software during an update. The algorithm uses a number of measurements in order to evaluate the performance of the system. Thus, it is important to understand what those measurement methods are and how they work.

4.2.1 Counters and Events

The SGSN-MME utilizes a variety of methods and techniques for monitoring and maintaining the behaviour and performance of the system's components while they are executing their tasks. The algorithm utilizes two of these methods which are responsible for monitoring and registering the performance and behaviour of the component's tasks. These two methods are:

- *Counters*

A counter is a non-negative integer value that is used for monitoring the performance and changes in the component during the execution of a particular task. Counters are incremented and may not be decremented. When the value of a counter reaches the maximum limit of an integer, it is reset to zero, thus a counter can be seen as the trip counter in a car. The value of the counter can be read, and reading that value does not alter it.

The value of a counter is incremented every time a change that is related to the counter occurs during the execution of a task. This means that the counter's value can be incremented multiple times during the execution of a task. For example, a counter is used to measure the number of successful attaches of a mobile unit into the system during a period of time. Another counter is used to measure the number of failing attaches. Thus, there is a very large number of counters that are used in the system, all of which are kept in a database and are updated constantly whenever new changes occur in the system while it executes its various tasks.

- *Event Based Monitors*

Event based monitors are used for logging information about the behaviour of the system during the execution of a task, in particular the success or failure of the events of the task. An event is an indication of an activity change within the component which is triggered during the execution of a task, such as user connecting and disconnecting from a service, running traffic, changing type of a service, or when a failure occurs. Contrary to counters, events include a number of parameters that provide details for each subscriber in the system. The parameters include the type of device and service being used, the time at which the event was registered, the duration of the event, whether the event was successful or not, and much more.

The *key performance indicators* of the system (KPI) make use of the counters and events, as well as other measurements in a number of complex equations in order to compute the performance of each component in the system. However for the purpose of creating a prototype for this algorithm, we will only use counters and events as those are the only ones that we have access to and are constantly updated during each execution of the component's tasks.

4.2.2 Capturing Counters

All counters in the system are stored in a database and they are updated periodically with increments that result from the execution of tasks in the system. Counters are queried either by names or by getting the entire list of counters from the database. However, we are only interested in counters that were updated during the execution of specific tasks, or in particular during a specific test scenario of these tasks.

- *Updated Counter Value*

In order to get the updated counters that pertain to a specific execution of a task a method is established by which we can extract the names of the updated counters during the execution of one of these tasks. Then the counters are queried, using their extracted names, from the system's logs. This results in a list of the updated counters during the execution of a task and their newly updated values.

- *Delta of an Updated Counter*

Since the counters are always only incremented, getting the updated value of the counter does not provide enough information, because the counter's value is a different number at any given time during the execution. It is therefore more useful to extract the delta by which the counter has been updated during the execution of a task. This is done by getting the value of the counter before and after the execution and calculating the delta of the update.

For example, during the execution of some task in the component, the value of some counter (*A*) has changed from 87 to 103, resulting in a delta of 16. It is important to note that when executing a task and re-executing it, the delta by which the counter has been updated is always the same, which makes it a reliable way of testing backwards compatibility.

4.2.3 Capturing Events

The *event based monitors* capture all events that result during the execution of the various tasks of the system. These monitors can be queried for the new events that resulted during the execution of some task, which in turn return all registered events. As mentioned before, however, each event contains a large list of parameters most of which are not interesting to us for verifying backwards compatibility. Thus, only the parameters that are of interest are extracted, which include the *name* parameter which gives the name of the event and the *status* parameter that indicates whether the event was a success or a fail.

The execution of some task in the system always results in the same group of events, and should also result in the same status of succeeding or failing. Therefore, for the purpose of the algorithm, the aforementioned parameters are the only two parameters we need for verifying backwards compatibility.

4.2.4 Verifying Backwards Compatibility

The counters and the deltas by which they are updated, as well as the events and their statuses will be used for verifying whether the component's software update is *execution based backwards compatible* or not. The algorithm, which is illustrated in Figure (8), works as follows:

- The algorithm begins with capturing all the counters and their respective

deltas that result from the execution of one of the component's tasks before an update.

- The algorithm then captures all the events and their respective statuses which result from the execution of the task before an update.
- The two above steps are repeated for the execution of the component's same task, however, after the software update is introduced into the component.
- At this stage, the counters and their deltas for the execution of the task of both builds are obtained. Likewise, all the events and their respective statuses from the execution of the task in both builds are obtained.

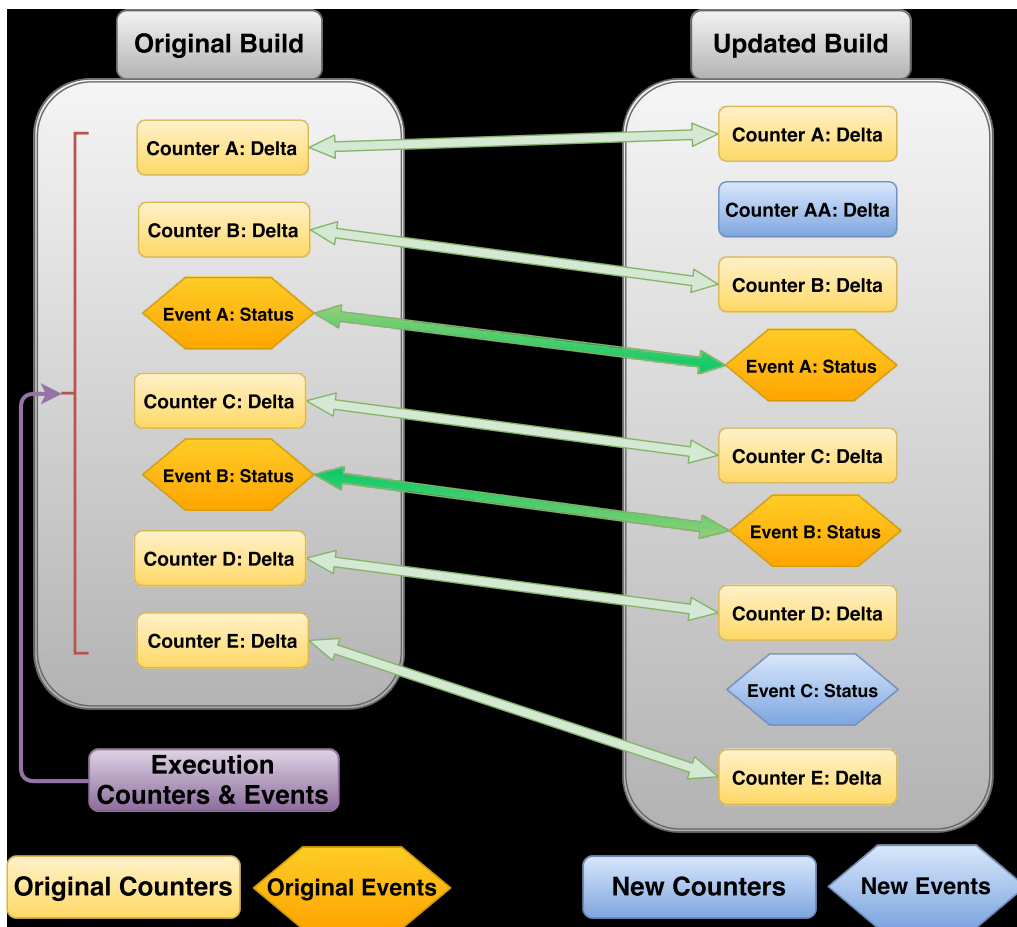


Figure 8. Comparing Counters and Events between Software builds

The next step of the algorithm is to match all the extracted information before the update is introduced to the component against all the extracted information after the update. The algorithm begins by matching the counters and deltas of the two builds:

- Each counter and its delta that resulted from the execution of the task before the update is matched against the counter and delta that resulted from the execution of the same task in the updated build as depicted in Figure (9).

- In order for the updated software to be backwards compatible, all the counters that were changed as a result of the execution of a task before an update, must exist with the exact same deltas by which they were updated by, in the captured counters of the task that was ran after the test case.
- If a counter or its delta does not match against the one in the updated version, it is filtered out as a faulty counter in order to be further analysed by the testers and developers. This is depicted in Figure (9), where the *Delta* of the 2nd counter in the original build is not equal to *Delta'* of the 2nd counter in the updated build.

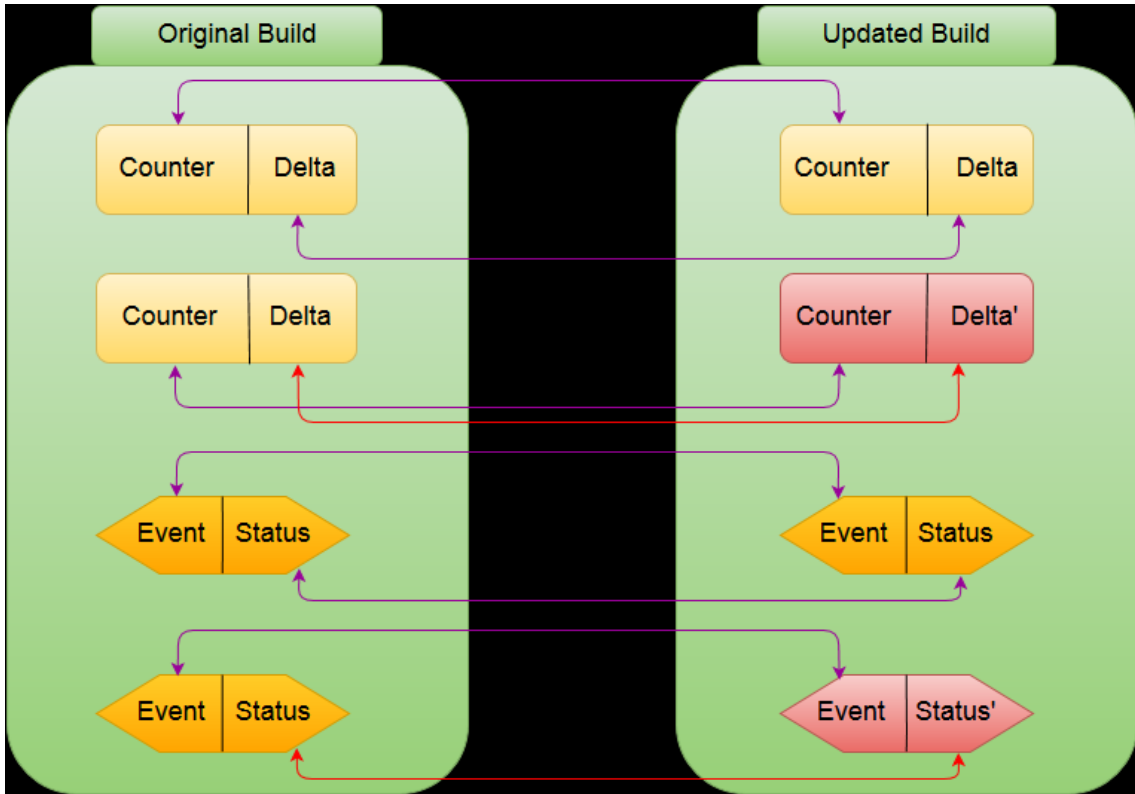


Figure 9. Matching Counters, Deltas, Events and Statuses

Next the algorithm matches the events and their statuses from the two builds:

- Each event and its status that resulted from the execution of the task before the update is matched against the event and status that resulted from the execution of the same task in the updated build as depicted in Figure (9).
- As in the case with the counters, in order for the updated software to be backwards compatible, all the events and their statuses must exist exactly the same, in the captured events of the updated build.
- If an event or its status does not match against the one in the updated version, it is filtered out as a fault counter in order to be further analysed by the testers and developers. This is depicted in Figure (9), where the *Status* of the 2nd

event in the original build is not equal to *Status*' of the 2nd event in the updated build.

New updates will most likely contain new counters and events that are added as part of the tasks of the updated software as can be seen in Figure (8) (i.e. Counter AA, Event C). Those are not taken into consideration when checking for backwards compatibility, as we are only interested in checking if the software continues to perform its original task as it did.

4.2.5 Prototyping the Algorithm

The prototype of the algorithm is created in order to establish the validity of the algorithm by testing its prototype on the system. The prototype is divided into the following two parts:

- *Collecting Counters and Events*

The counters and the deltas were collected from the system using an *Erlang* module. A test scenario that represent a task of system's component is specified, the module then collects all the counters and their values before running the test. After the test scenario is ran, the module collects all the counters and their values again. The module then compares the counters that were collected before and after the running of the test based on their values. A counter whose value has incremented after the running of the test is stored in a file, along with the delta by which the counter's value has changed. The same procedure is repeated for the same test scenario, however on an updated build, thus there will be two files containing counters and their deltas, one created before software update, and one after the update.

The events are collected from their event based monitors using an *Erlang* module as well. In this case, after the execution of test scenario, the event based monitors have all the new events that resulted from the test. The module requests those events, and then filters out the unnecessary parameters, since we are only interested in the name of the event and its status. The module would then store all the events and their respective statuses into a file. The procedure is repeated for the same test scenario on an updated build, thus, two files are created, one that contains the events of the test before the software update and one that contains the events for after the software update.

- *Backwards Compatibility Check*

The second step involves matching the counters and events of a test scenario that were collected before a software update, against the counters and events of an updated software as depicted in Figure (8). The check is performed using a *Bash* script which reads the two counter files (counters before an update, and counters after an update). The script matches the counters and their respective deltas that were recorded before a software update against the counters and deltas after a software update as depicted in Figure (9). If some

counter that was recorded before the software update, does not exist in the set of counters after the software update, then the software update is reported to be incompatible and the mismatching counter is printed out. If a counter does exist in both files, however, the delta by which the counter was changed does not match in both files, then the software build is reported incompatible and the mismatching counters and their respective deltas from both files are printed out.

The script then reads the events of both files (events before an update, and events after the update). The script matches the events and their respective statuses that were collected before the software update against the events that were recorded after the software update Figure (9). If an event that was collected before the software update does not exist in the set of events that were collected after the software update, the software is reported incompatible and the missing event is displayed to the terminal. If the status of event which indicates whether the event was successful or not, does not match in both files, then the software update is reported incompatible, and the mismatching events along with its statuses is displayed in the terminal. The displayed counters and events can be used later on by the developers and testers in order to trace the source of the incompatibility and resolve the problem.

4.2.6 Testing of the Prototype

Following the same pattern of testing that the signal based algorithm underwent, the prototype of the algorithm was subjected to two types of proofs [HS02]. The proofs help verifying the validity of the algorithm as well as understanding the results of the prototype [HS02]. It is important to note however, that the testing of this prototype was done only on test cases and not test suites. This is because the goal of the algorithm is to evaluate the execution of separate tasks of the component, such that the results of the tests do not affect each other. The two proofs of the algorithms are as follows:

- *Validating the Prototype*

Similarly to the previous algorithm's proof of concept, this proof involves testing a software build against itself in order to prove that the prototype behaves as expected when testing software against itself [HS02]. The proof is done as follows:

- If the prototype is ran on two identical runs of a test scenario of the software in the same build (i.e. test case and a re-run of the same test case), the prototype should report that the software is compatible.
- If the prototype is ran on a test scenario of the build against a re-run of the test, however, the second test is interrupted half way through, the prototype should result that the software is incompatible and display between (50% to 70%) of the counters and events, which are missing.

- *Comparing Builds*

Similarly to the previous algorithm's testing procedure, this proof of concept involves testing a software build against an updated version of the software as depicted in Figure (8). This proof is the actual verification of backwards compatibility between different versions of the software. The proof is done as follows:

- If the prototype is ran on a test scenario of the original build against itself it should result that the software is compatible.
- If the prototype is ran on a test scenario of the original build against a test scenario of the updated build, and all events and counters of the previous build are present in the new one, then prototype should report that the software update is backwards compatible.
- If the prototype is ran on a test scenario of the original build against a test scenario of the updated build, and a counter or its delta is mismatching, then prototype should report that the software update is incompatible and display the mismatching counters and their respective deltas.
- If the prototype is ran on a test scenario of the original build against a test scenario of the updated build, and an event or its status is mismatching, then prototype should report that the software update is incompatible and display the mismatching events and their respective deltas.

The results of the tests that were performed on the prototype of this algorithm can be seen in Chapter 5 which contains the results of this thesis, section (5.2.2 Events Based Algorithms Results).

4.3 Unified Algorithm

The *signal based verification algorithm* provides an abstract method for verifying execution based backwards compatibility. It does not require an understanding of the details of what the component of the system is doing. And it provides a method of checking the external communication signals between the nodes of the component and other components during its execution. On the other hand the *events based verification algorithm* provides an in-depth method of verifying execution based backwards compatibility, which focuses on the details of what the component is doing, without any focus on the external communication of the component's nodes.

Due to the different focus of the two algorithms, it is expected that they may result in some differences in the level of backwards compatibility of a software update and the information about the problems that cause the backwards incompatibility. Thus in order to get more accurate results in terms of backwards compatibility, and for the purpose of answering the third research question *RQ 3*, a unified algorithm is established that merges the characteristics of the two algorithms.

4.3.1 Merging the Algorithms

The unified algorithm incorporates the steps of both the signal based verification algorithm and the events based verification algorithm. The algorithm begins with a data collection phase which is as follow:

begins with collecting the necessary information for the signal based verification algorithm, then collecting the information for the events

- *Recording Execution Signals*

The unified algorithm begins by collecting the information that are needed for performing the signal based verification algorithm. The information are represented in the trace of execution signals of the component during the execution of one of its tasks. The execution signals represent the sequence of communication between the nodes of that component during the execution of the task. The procedure of capturing the trace of signals is done as explained in sections (4.1.4, 4.1.5 and 4.1.6). This step is performed first because the collection of the execution signals is done during the execution of the task and not after it is done. By the end of this step, a full trace of the execution signals are recorded into a file.

- *Collecting Counter and Delta*

The next step is to collect all the counters that were updated during the execution of the component's task, and the delta by which the counters have been updated. This step is done after collecting the trace of execution signals, because counters and their deltas can only be collected after the execution of the task is finished. The capturing of the counters and their deltas is done following the procedure that is explained in section (4.2.2).

- *Collecting Event and Status*

The events based verification algorithm also requires information about the events that occurred during the execution of the component's task along with the status of each event of whether it was successful or a failure. Thus the next step is collecting all events and their statuses that result from the execution of a task. As in the case of counters and their respective deltas, the events are collected at the end of the execution of the task. The capturing of the task's events and their respective statuses is explained in section (4.2.3).

The above process is first done for the execution of the task before an update is introduced into the component and then once again after the update. At the end of this part of the algorithm, all the necessary information for performing both algorithms must be available and stored in files that will be accessed during the verification part of the algorithm. The algorithm performs the verification checks as follows:

- *Checking Signals*

The next part of the algorithm is to perform the signal based backwards compatibility check on the captured trace of execution signals. The procedure of checking the signals of the original build against an updated build is explained in section (4.1.6). It is important to note that for the purpose of the unified algorithm, the results of the signal based algorithm are only considered backwards compatible if the resulting compatibility percentage is above 95%. This means that any results that are below this value are considered incompatible and the mismatching signals are displayed for thorough examination by the development team. The rationale behind this decision is that the unified algorithm should capture as many of the incompatibility faults as possible.

- *Checking Counters and Events*

The algorithm then checks that all the counters that were captured before an update exist after the update is introduced. Then the algorithm checks the delta by which each counter has been changed during the execution of a task before an update, against the deltas that result from the task after the update.

The algorithm then checks that all the events that resulted from the execution of the task before the update against the events of the updated build. Then the algorithm matches the statuses of each event from before the update against the statuses of the events after the update. The algorithm reports that the update is backwards compatible if all the counters, deltas, event and statuses match, otherwise it displays the mismatching data. The procedure is explained in details in section (4.2.5).

4.3.2 Verifying Backwards Compatibility

The unified algorithm combines the verification of both signal based and events based algorithms. Thus based on the results of the previous checks, the algorithm makes a decision on the backwards compatibility of the software update as follows:

- The algorithm reports that the software update is backwards compatible if and only if the results of checking the signals is above or equal to 95% and the result of checking counters and events is backwards compatible.
- The algorithm reports that the software update is not backwards compatible if the results of checking the signals is below 95%. The algorithm displays the missing or mismatching signals between the two software builds for further analysis.
- The algorithm reports that the software update is not backwards compatible if the results of checking the counters yield missing counters between the original build and the updated one, or if the delta of any of the counters in the two builds do not match. The algorithm displays all the missing counters and mismatching deltas between the two software builds for further analysis.

- The algorithm reports that the software update is not backwards compatible if the results of checking the events yield missing events between the original build and the updated one, or if the status of any of the events in the two builds do not match. The algorithm displays all the missing events and mismatching statuses between the two software builds for further analysis.

4.4 Performance Based Algorithms

Due to the complexity of the system and time limitations of the study, we were unable to develop a complete algorithm for verifying performance based backwards compatibility. The following are the methods that were attempted unsuccessfully:

- *Comparing CPU Usage*

The idea behind this algorithm is to get accurate readings of the CPU usage during the execution of the component's tasks before a software update and compare them to the CPU usage after the software update is introduced. However, getting accurate readings of the CPU usage that are related to the execution of the task alone proved very difficult. Most of the CPU readings obtained fluctuated greatly, even after averaging them over a period of time. It was also noted that the CPU usage was fluctuating even when the component is not executing any tasks; thus, it was concluded that prototyping this algorithm as a viable manner of testing is not possible.

- *Comparing Process Usage*

An alternative idea to the CPU usage was an attempt at monitoring the number of registered processes that are spawned during the execution of the component's task and comparing their amount and other characteristics before and after the update. However, performed tests on the component have shown that processes are spawned and terminated rather randomly, or in a way that is hard to follow. It was also observed that processes were being spawned and terminated even when the component was not executing any tasks. Thus, this algorithm, much like the previous one, was deemed unreliable for prototyping.

- *Comparing Memory Usage*

Similarly to the CPU usage algorithm, this algorithm involves comparing how much is the memory usage of the component during the execution of a task before and after the update. The method could not be applied on the system's component because the readings were not stable at all. The memory usage varied even when the component is not executing a task; thus, prototyping this method was not possible.

- *Comparing KPI*

The main purpose of key performance indicators (KPI) is keeping track of the degradation of the system's performance in general. This is done through

a number of complex equations using specific measurement methods, such as counters and events. The basis behind the algorithm is to compare the values of the key performance indicators that were affected by the execution of a component's task before and after an update. Now, since KPI's use predefined mathematical formulas, this means that to measure the change in terms of performance of a component, the same counters and events as in the legacy version must exist in the updated version. However, if this algorithm was to be prototyped, this would mean going beyond functional testing and would require more time and effort to understand how the system works and in what manner can the KPI's be used for verifying backwards compatibility.

5

Results

IN this section we present the results of the thesis, which includes the results of the study phase as well as the results of the algorithms and their prototypes.

5.1 Study Results

To answer the first research question *RQ1*, a number of algorithms and strategies from both academia and the industry have been identified. Most of the algorithms that were found within the industry are the ones which are recommended by Ericsson.

5.1.1 Literature Review Results

- *Semantics Check using Boogie*

This algorithm targets object oriented libraries, such as Java [WH12]. By monitoring the *stack/heap* (in terms of computer memory), the authors were able to distinguish which parts belong to the library and which parts belong to the program context [WH12]. Due to the fact that for certain libraries, there are only a finite number of types, therefore, the traces that the library leaves on the *stack/heap* are limited as well [WH12]. What this allows, in turn, is by monitoring the traces of the library in newer implementations, it is possible to determine whether the updated library will be backwards compatible or not by monitoring its behavior in the *stack/heap* and monitor the control flow, whether it passes from the library to program context or viceversa [WH12]. This algorithm works only for object oriented programming languages and the paper's main focus is *Java* programming language when testing the framework, whilst the algorithm itself belongs to *execution based backwards compatibility*.

- *Checking for Deleted Binary Symbols*

This algorithm's focus lies on automatic checking of binary symbols in a newly released library in comparison to the previous version of the library [PR12]. This is done by extracting public symbols from two versions of the library and

comparing them [PR12]. There is a number of tools that can perform checks at this level, such as “dpkg-gensymbols, cmpdylib”. This particular type of algorithms correspond to *binary backwards compatibility verification* type.

- *Targeted Unit Tests*

This strategy focuses on creating unit tests that are targeting backwards compatibility directly [PR12]. What this means is that unit tests are made to be run on newer version of the software, and these unit tests will identify changes that can lead the newly implemented software to be non-backward-compatible. According to [PR12], there do exist tools that can be used for automated test creation; for the full list of tools please refer to paper by [PR12]. However, in majority of cases, a minor drawback is that the unit tests have to be written manually, which will result in a very large amount of tests very quickly.

- *Versioning*

This strategy ensures *binary backwards compatibility* by stacking multiple versions of a library or component into the same binary file, thereby ensuring that all the functionalities of the system, old or new, can use the functions they require [PR12]. While this algorithm may ensure *binary backwards compatibility*, there is a drawback, since the library can contain multiple functionalities of a module pertaining to a specific application, the size of the library is increased, and this is accompanied by an increase in the complexity of the library module. The main advantage of Static linking to ensure backwards compatibility is that it requires no external dependence on the system library. This results in minimal or almost none backwards compatibility issues, however, there is a certain drawback [PR12]. On the one hand, as the library is increased in size, so is the application. On the other hand, when performing an update on the library, recompilation of the application is required as well.

- *Compatibility Layers*

This is a strategy where *binary backwards compatibility* is ensured by recovering binary interface of a library from a preceding version, to function with the newly implemented library. In the paper, the ReBA approach proposes a completely automated solution for creating compatibility layers [PR12]. It is based on the analysis of log files, containing all the changes in the code whilst developing the new version of the library. Although, there is a drawback to this algorithm, namely, reduction in performance and increase in memory use.

- *ABI-Compliance Checker*

This is an automated algorithm that is capable of identifying all types of changes in libraries that prevent the new implementations being *binary backwards compatible* to the older version of the library. The tool is known as *abi-compliance-checker* [PR12]. The main working principle behind is to take *header* and *binary .so* files of a library, and compare the type definitions and function signatures recovered from intermediate representation in the structure

of the abstract source code syntax tree[PR12]. This tool is open-source; any modifications in it require knowledge of C/C++. Therefore, this algorithm addresses *binary backwards compatibility*.

- *MicroFormal*

This algorithm is aimed at verification of *microcode level backwards compatibility* in a formal and fully automated manner. This tool uses a term denoted as Intermediate Representation Language (IRL) which is generic enough to be used on low-level languages [AEF05]. IRL uses templates, where the template is made of IRL code that is equivalent to a microinstruction in terms of operations to execute. This algorithm is extremely low-level, as it uses actual processor instructions [AEF05]. It is hard to specify which programming language is used, as the authors do not explicitly state it.

5.1.2 Empirical Results

The algorithms and strategies that were recommended by Ericsson, as well as some other algorithms and tools that were found within other companies and from people with experience in the field of backwards compatibility can be seen below:

- *Regulating Mechanism*

This idea is meant to be applied in the system for features that have already been proven to be functionally backwards compatible in terms of *execution based backwards compatibility*, i.e. it works without failing itself or the other sub-systems. If the added feature results in a decrease of the performance of the system, there should be a mechanism for disabling that feature. The idea here is to have two versions of the software, one with the feature and one without, and the system should be able to automatically switch between the versions in order to maintain the performance and expected behavior [Hut12]. Thus the idea requires information about the behavior of the feature and its effect on the system's performance.

- *Compare Logs Between Legacy and Updated Version*

The idea behind this algorithm is to compare the logs of the two versions of a certain functionality in order to check if it is *execution based backwards compatible* or not. This requires finding certain patterns in the system's run-time information which could lead to verification of the *execution based backwards compatibility* of a feature. However this is based on the assumption that the logs follow a generic pattern that is reliable for this form of testing.

- *Compare Variables-Counters-Stack/Heap*

This algorithm can be used for monitoring the behavior of the nodes, sub-system, and network in general. It involves keeping track of all the variables usage of the system and then comparing them for the various runs. The idea

behind this algorithm is by using *Stack/Heap* (in terms of memory) to check how much of the *stack* and *heap* has been used per function in order to verify if the function is *performance based backwards compatible* or not. Counters are used to keep track of the many of the indicators of the system whether it's functional, non-functional or performance based.

- *Comparing KPI*

Key Performance Indicators (KPI) are variables that are used to measure the performance of the system through the use of some pre-defined mathematical models and formulas [SNM15]. This is a much more detailed algorithm in comparison to the previous one (*Compare Variables-Counters-Stack/Heap*) because it uses counters that are built specifically for measuring the various performance indicators of a function. Therefore, the KPIs can be utilized for the creation of an algorithm that can verify *performance based backwards compatibility* of a software update [SNM15].

- *Using Flags for Enabling New Features*

This is a strategy where the feature should have already been tested both for functionality and *execution and performance based backwards compatibility*. If the feature has a high risk of affecting the system as a whole then the feature is introduced with a flag that is set to “off”, which can be enabled when needed [Hut12]. If the feature has a low risk of affecting the system, it will come with a disabling flag that is set to “on”, which can be disabled if necessary [Hut12]. If the feature has no risks of affecting the system, it does not need any flags.

- *Network Traffic Monitoring*

There is a number of tools that are used for network analysis, which can capture all packet information and store them in a “.cap” file. The suggested idea is to run the legacy version of component and capture all its network communication information and then run the updated version and capture the same information. Then using special differentiation tools, the two “.cap” files will be compared in order to establish whether or not the new feature is backwards compatible in terms of *execution based backwards compatibility*. The Signal Based Verification Algorithm that was developed by the authors, used this idea as a starting point for the design of the algorithm, however the resulting algorithm has a completely different design than the proposed idea.

- *Regression Testing for Verifying Backwards Compatibility*

The main working principle of this strategy is to re-use the test cases and test suites of the legacy version of the software on the updated version [WHL97]. If the software update is *execution based backwards compatible*, then in principle it should be able to pass the tests of the legacy version of the software. However, this comes with the assumption that the test cases are generic enough to handle only the results of what is being tested and not deal with the entirety of the content.

- *Run-Time Tests*

This strategy involves the creation of test suites and test cases that are specifically made for checking *execution based backwards compatibility* of the system. While this may be a very reliable way of verifying backwards compatibility, it is not an automatic verification method, as it requires the manual creation of tests for the various components of the system as well as the various circumstances that could arise from the development of such a feature. Thus, it will require a team that is dedicated for this type of testing.

- *SigTest*

This tool is a bundle of Oracle's commercial tools [WWW11]. It is used to compare different API's and measure their test coverage. Additionally, this tool can assist in the creation of test suites. The tool is mostly used for *Java* programming language applications. The main advantage of this tool is that it can compare signatures of two different API's; performs checking whether old version can be replaced by the new one without adversely affecting existing clients of the API. API coverage tool can be used to estimate test coverage a test suite provides for an implementation of a specified API. This tool is aimed at verifying *source level backwards compatibility*.



Figure 10. Taxonomy of Algorithms & Strategies

The aforementioned algorithms and strategies were studied and classified into four categories which include performance, execution, syntax as well as system level categories. This helps in simplifying the process of choosing an algorithm that is suitable to a particular class of problems. The classification of the algorithms is mapped into a taxonomy which is illustrated in Figure (10). The taxonomy contains an overview of the identified algorithms for verifying backwards compatibility in different areas of software checks and components of systems.

5.2 Results of Implementation Phase

In this section the results of the implementation phase are presented. These results represent the answers to research questions *RQ2* and *RQ3*.

5.2.1 Results of Applying Signal Based Algorithm

The algorithm that was developed during the first cycle of the implementation phase has been very successful in verifying *execution based backwards compatibility*. The results of testing the algorithm's prototype can be seen below:

- *Results of the Proof of Concept*

The prototype has been tested following the design of this proof of concept on 500 test cases. Each of these test cases simulates a certain task that the component executes in real environment. Not a lot of information can be provided on the behavior of the test cases other than the fact that they test the component's simulated functionalities. This is the case due to confidentiality agreement that is signed with the company.

The prototype was successful in passing the designed proof of concept for all the test scenarios that it was applied to. This means that when the prototype was ran on a test case and a re-run of that test case, the prototype resulted in 100% backwards compatibility for all the test cases (denoted as *Fully Executed* in the figure), thereby passing the proof of concept as expected, this can be seen in Figure (11). When the prototype is ran on test cases that were fully executed against the same test cases, however, interrupted half way through their execution, the prototype resulted in an average compatibility level that is above 50% and below 70% (denoted as *Interrupted Half Way* in the figure). This is because a number of signals obtained from fully executed test cases did not exist in the test cases that were interrupted half way through their execution.

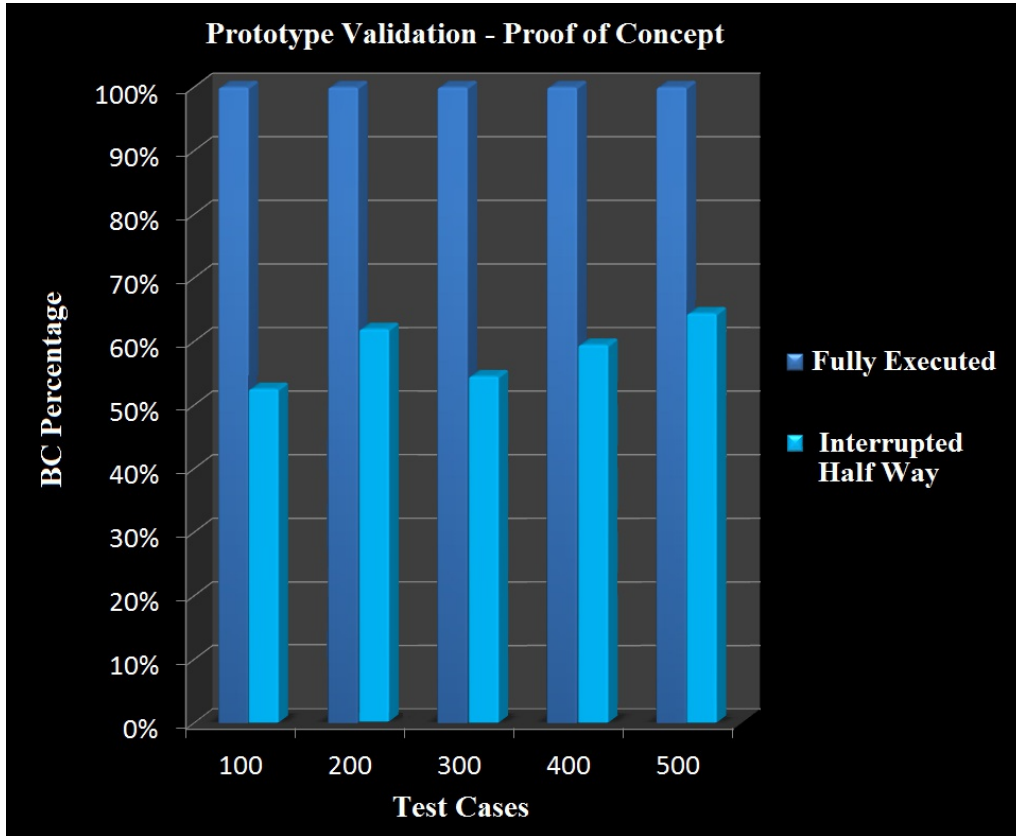


Figure 11. Signal Based Algorithm's Prototype Validation

The values that resulted from this proof of concept conform to the theoretical predictions of the algorithm, thus proving the validity of the prototype.

- *Results of Comparing Builds*

The prototype has been used for comparing the execution signals of an original build of the component's software against its updated version. The update includes a feature for enhancing *power saving mode* of the component. The parts of the components that were affected by the update were tested against the original version of the program to verify the *execution based backwards compatibility* of the update. This was done by using certain test suites that test the behavior of the updated parts. Following the algorithm's design, two versions of the signal execution were extracted, before and after the update.

The prototype checked whether the execution signals of the original build were a subset of that of the update build. Then a percentage was computed which represents the backwards compatibility level of the updated software. The prototype was run on three major test suites that test various tasks of the component which include the use of power saving mode. According to the designers and the developers of the feature in this software update, the feature should be *execution based backwards compatible* since it has not caused any problems to the system upon its introduction. As can be seen in Figure (12),

the prototype resulted in an average percentage that is over 95% for all three test suites. This confirms that the new feature is actually *execution based backwards compatible* within the levels of backwards compatibility that were set during the algorithm's designing phase.

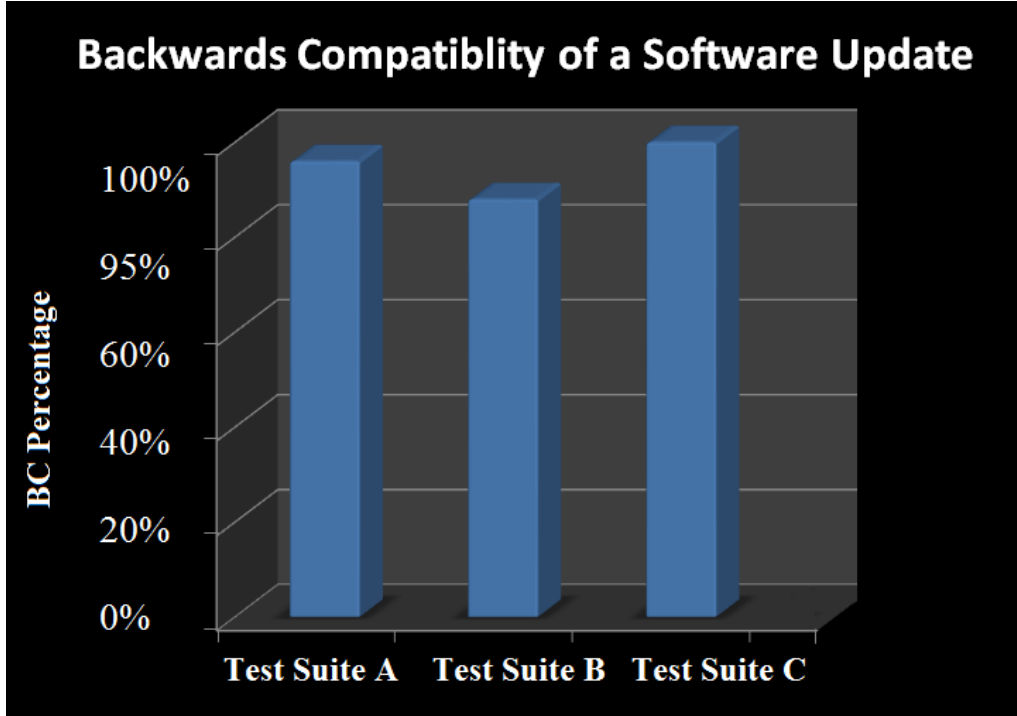


Figure 12. Signal Based Backwards Compatibility of a Software Update

The results of the prototype show a lot of promise for an algorithm that with more work can be developed into a fully automated system that can be used for verifying *execution based backwards compatibility* in distributed real-time systems.

5.2.2 Results of Applying Events Based Algorithm

The algorithm that is about to be described was developed during the second cycle of the implementation phase. The prototype of this algorithm has undergone thorough testing following both types of proofs in order to establish the algorithm's validity in verifying *execution based backwards compatibility*. The results of testing the algorithm's prototype can be seen below:

- *Results of Prototype Validation*

The validation of the algorithm was done following the designed proof for testing the prototype on 120 test cases in the system. The test cases simulate the various tasks that the component must execute in a real environment. Since the number of counters and events vary between each test scenario, and

in order to make it easier to visualise the results in diagrams, the number of counters and events for each test scenario was summed into a total value (N). Then for each test scenario, a percentage is showed that is based on the total number of counters and events (N). The results of testing this prototype are depicted in Figure (13).

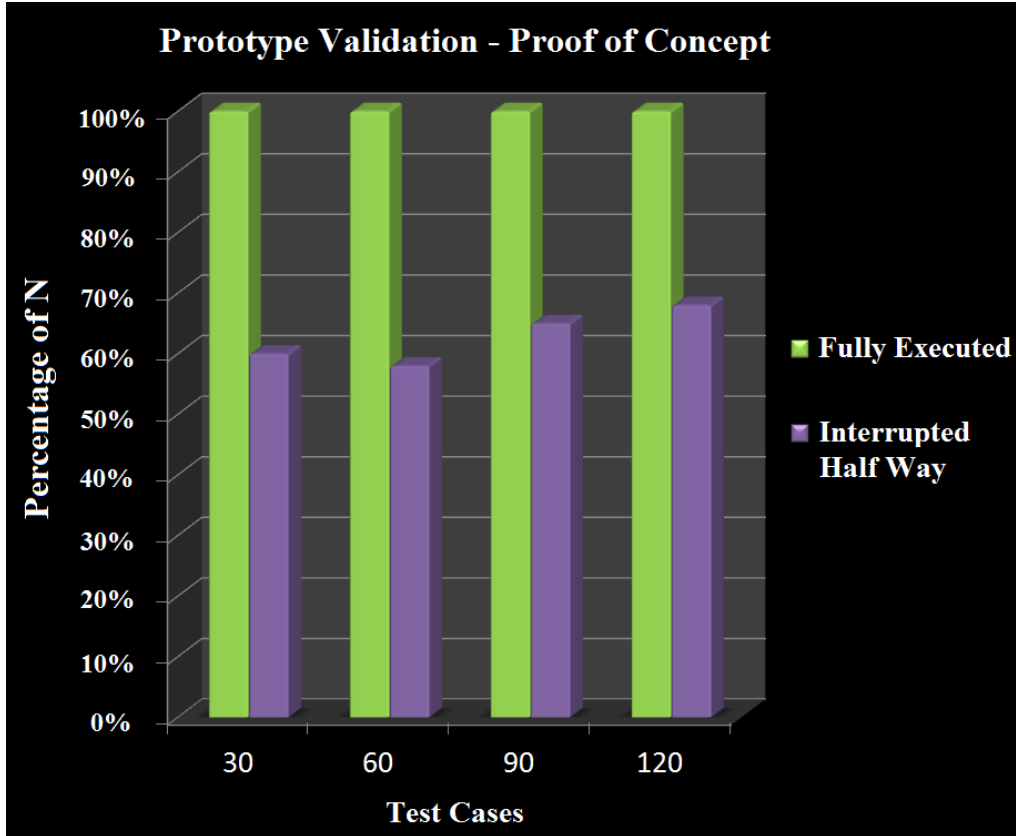


Figure 13. Events Based Algorithm's Prototype Validation

The algorithm was successful in passing the designed proof of concept for all the test scenarios that it was applied to. In the case of matching test scenarios of the component's tasks against full re-runs of the tests, the prototype resulted in a 100% match of the counters and events between the two runs of each test. In the case of matching test scenarios against re-runs of the tests that are interrupted half way through their execution, the prototype resulted between (50 to 70%) match of the counters and events between the two runs for each test.

This is very close to the expected values of the proof; the prototype also displays all the missing and mismatching events and counters. Thus, the values that resulted from this proof of concept conform very closely to the theoretical predictions of the algorithm, which proves the validity of the prototype.

- *Results of Comparing Builds*

Following the validation of the prototype, the prototype has been used for

comparing the counters and events of an original build against its updated version. The parts of the components that were affected by the update were tested against the original version of the program to verify the *execution based backwards compatibility* of the update. The testing of the component was done using test cases that simulate the various tasks of the component in a real environment. The update for which the algorithm was tested on, is the same update for which the signal based algorithm was tested on, which is a power saving mode feature for the component.

The prototype was ran on the results of three test suites that simulate the various tasks of the component. The prototype reports the update is backwards compatible, if all counters and events from the original build exist in the updated build. The prototype displays all counters and events that are missing or contain mismatching values when compared to the updated build. As can be seen in Figure (14), the prototype reported 94% of the tests in test suite (A) to have matching results, 97% of the tests in test suite (B) to have matching results, and 98% of the tests in test suite (C) to have matching results.

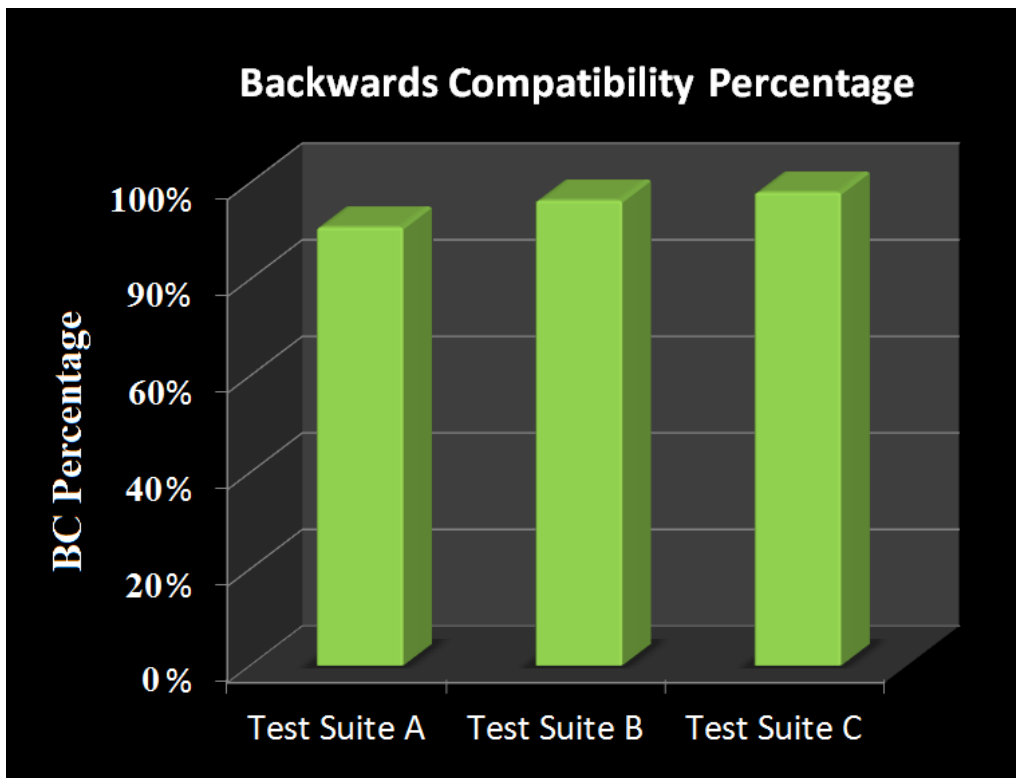


Figure 14. Events Based Backwards Compatibility of a Software Update

This indicates that there is a high level of compatibility in the update with its original build. The prototype displayed all missing counters and events between the original and the updated build, as well as the respective mismatching values, for all the failing tests.

Looking at Figure (14) we can see that the results of this algorithms are fairly close to the results that were reported from the signal based verification algorithm. However, the prototype of this algorithm seems to detect slightly more errors in terms of compatibility. The incompatibility results of the previous algorithm were mostly related to missing communication signals within the execution of the component, or to an incorrect order in the execution signals. However, the incompatibility results of this algorithm are related to missing counters or events due to the update, or mismatching values of the details of the counters or the statuses of the events. Hence, the two algorithms provide independent results in terms of the levels of backwards compatibility.

5.2.3 Results of Applying the Unified Algorithm

Similarly to the previous two algorithms, the prototype of the unified algorithm was tested on the system for verifying the *execution based backwards compatibility* of a software update. Since this algorithm is a combination of the signal based verification algorithm and the events based verification algorithm, there was no need to perform validation on the prototype, since the validation step was already done for the prototype of the two algorithms separately (see sections 5.2.1 and 5.2.2). Thus, the prototype of the algorithm was used for comparing the signals, counters and events of an original build against the updated version. In order to be able to evaluate the results of this algorithm, the prototype was tested on the same three test suites that were used in the previous two algorithms, which test the functionalities of the power saving mode feature of the component.

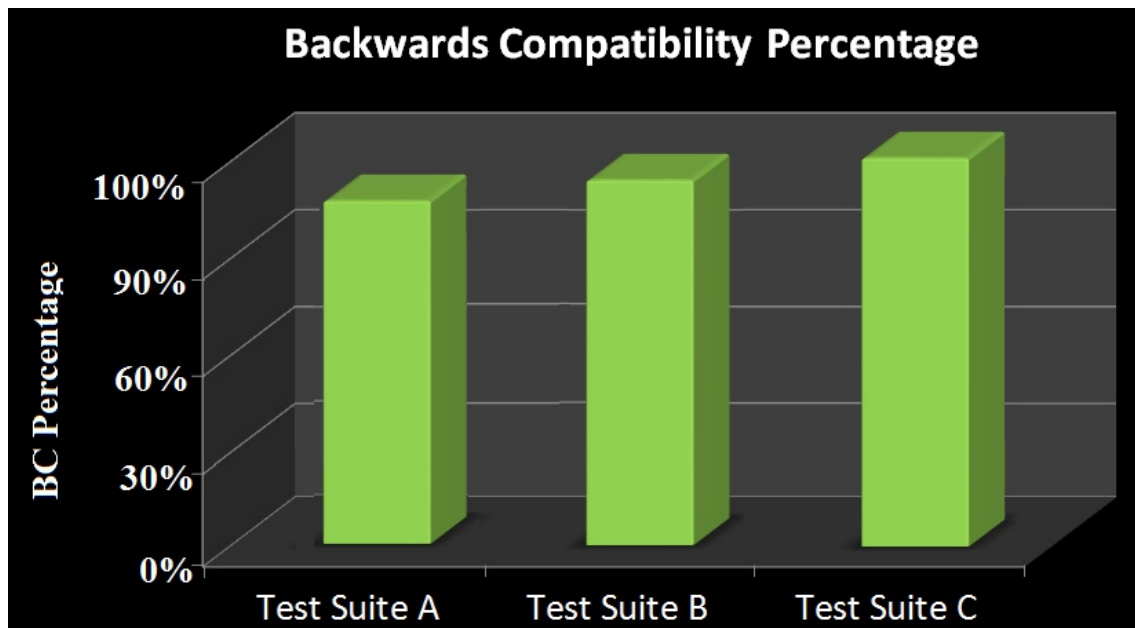


Figure 15. Unified Algorithm's Results

The prototype reports that the software update is *execution based backwards compatible* if all the signals, counters and events of the original build match against those of the updated build. The prototype should otherwise display all missing signals, counters or events or the mismatching deltas of counters or statuses of events. The results of testing this algorithm on the three test suites can be seen in Figure (15). As can be seen in the figure, the prototype has reported 91% of the tests in test suite (A), 94% of the tests in test suite (B), and 98% of the tests in test suite (C) to have matching signals, counters and events.

The unified algorithm's result is a combination of the results of the signal based and events based verification algorithms. While the two algorithms reported close results, there were some minor differences in the number of tests that had mismatching results. For instance, in the signal based algorithm, test suite (A) had 97% of the tests having matching results, while in the events based algorithm, test suite (A) had 94% of the tests with matching results between the original and updated build. This is to be expected since the two algorithms work at different levels of the software, i.e. the signal based algorithm focuses on communication signals, while events based focuses on the behaviour of the node in the component. However, it is possible that some of the tests that fail due to mismatching results between the original and updated builds, can be the same for both algorithms, however, this is a rare case. Since the unified algorithm combines the characteristics of both algorithms, it is only natural that the prototype of the algorithm would detect even more tests with mismatching results, which is the case in, for example, test suite (A) with matching tests of 91%.

The results of this algorithm provide an answer for the third research question *RQ3*, where the two previous algorithms were combined together into formulating a concrete method for verifying execution based backwards compatibility.

5.2.4 Impact of the Results

The prototypes were all tested on an updated component that was provided for us as a proof of concept. According to the developers of the feature, the prototype should be backwards compatible and have no expected problems. However, testing the prototypes of the developed algorithms on this updated component have reported a small percentage of mismatches before and after the update, namely 8%. These anomalies were represented in missing signals that were detected by the signal based algorithm, and mismatching counter deltas that were detected by the events based algorithm. These anomalies were reported to the company as part of the delivery of the prototype, and will be further investigated by the developers of the prototype in order to verify that they are not affecting backwards compatibility of the updated software.

6

Threats to Validity

IN the case of the signal based verification algorithm, the identified signal might be an erroneous one. Depicted in Table 2 are the classification of signals and the corresponding information whether the signal is erroneous or not, where the signals and events can either match or mismatch between an original build and an updated one.

	Match	Mismatch
Correct Signal/Event	TP	FN
Erroneous Signal/Event	FP	TN

Table 2. Signal Identification Fallacy

Hence, the point of focus is whether the erroneous signal is identified as a matching signal incorrectly (False Positive; FP) and is added to the list of all the correct signals that are identified as matching signals without any errors (True Positive; TP). However, since the filter (explained in section 4.1.4) in the wireshark is explicit, namely, any packet that is recorded via wireshark that does not satisfy the criteria, for instance, packets that do not correspond to *gtpv2* protocol, will be dropped on the listening interface. Only the packets that fulfil the criteria of the filter are collected. Therefore, the threats of identifying erroneous signals as matching signals are considered under control. Similarly, when it comes to comparing the execution sequence via bash scripting, the threats of identifying signals incorrectly is also considered under control. This is because before the script was integrated into the algorithm, it was extensively tested via many simulated scenarios during the validation phase that is explained in (section 4.1.8).

It is worth noting that the scenario of collecting wrong data for the events based verification algorithm can never happen, as the data that are collected from the sources for the counters and events are always updated appropriately. It is highly unlikely for the wrong counter to be updated, or the wrong event to be triggered as a result of the execution of a task since the SGSN-MME system has been thoroughly tested and verified. However, in order to ensure that the algorithm behaves correctly when it comes to comparing counters and events before and after the update, the algorithm was also extensively tested on many scenarios through the validation step that is explained in (section 4.2.6), thus, making sure that false positives can never occur.

7

Ethical Considerations

At the beginning of this thesis, several ethical considerations had to be made. It is also a vital matter that the thesis and its outcome has to be built, first and foremost, on mutual trust between the collaborating parties, in this case, the authors and the company [SV02]. Furthermore, it is also the case that since the thesis is being performed at the premises of Ericsson; the study will deal with confidential information in an organization [SV02]. The thesis involves technical interviews performed with the employees of the company, where participation of the employees interviewed had to be voluntary. Hence, it is hard to say which information should or should not be published, however, the key ethical issues to consider are:

- Informed consent.
- Confidentiality.
- Sensitive information.
- Feedback.
- Supervising manager's approval.

Receiving feedback from the supervisor and other employees who participated in the technical interviews was extremely helpful. Moreover, by receiving feedback in a positive manner, for instance, on where should the work be done (which subsystem), it helps to maintain trust and validity of the thesis [SV02]. During the interviews, since they were of a technical manner, no transcripts were being made as they are not a part of the main goal of the thesis. However, the suggestions and understanding obtained from these interviews regarding the heading of the thesis or explanations regarding the functioning principle of specific components were important.

8

Conclusion

BACKWARDS compatibility is an important goal for companies to achieve in order to reduce the cost of constant deployment of updates to their products. In this study we have explored and developed algorithms for verifying backwards compatibility for distributed real-time systems. The study started by exploring the existing definitions of backwards compatibility in literature. Additionally to the existing backwards compatibility definitions in the literature, two new definitions were introduced which are *execution based backwards compatibility*, and *performance based backwards compatibility*. Several algorithms and strategies were identified through academia and the industry in order to verify backwards compatibility. The identified algorithms and strategies which are presented in Section 5 were classified into 4 main categories which are execution check, performance check, system check and syntax check.

Three algorithms were designed, prototyped, and tested on the SGSN-MME system at Ericsson. The algorithms pertain to the execution check category, which means that the component must continue executing its tasks as it did before the update. The first algorithm is the *Signal Based Verification Algorithm*, which is based on the network traffic monitoring idea proposed by the company. It verifies that the sequence of communication signals between the nodes of a component match before and after the update. The second algorithm is the *Events Based Verification Algorithm*, which makes use of a number of measurements within the system and verifies that those measurements do not change when an update is introduced to the system. Lastly, the third is a *unified algorithm* which combines the characteristics of both algorithms.

Testing the prototypes of the algorithms on the system for comparing an original build and its updated one revealed a small percentage of non-backward compatible anomalies, which were represented in mismatching signals, counters and events. These anomalies were reported as part of the prototypes that were delivered to the company. Based on the testing and evaluation of the prototype, the signal based verification algorithm can be used during unit, component and functional testing of a software. However, the events based verification algorithm and therefore the unified algorithm, can only be used during unit testing of a software. In order to get concrete results and discover compatibility flaws as early as possible during the software development, it is important that each algorithm is used during the recommended test phase.

Due to the complexity of the system and limited time of the study, and in spite of the time that was put in evaluating some of the proposed methods, we were unable to develop a viable algorithm for verifying performance based backwards compatibility. It is clear that there is a lot more work that can be done from this point forward. Future work for this thesis includes studying more of the identified algorithms and creating prototypes to prove their capabilities in detecting compatibility flaws. Furthermore, development of each of the prototypes into full automated tools for verifying backwards compatibility is required. Lastly, exploring more methods and techniques in order to develop a viable algorithm for verifying performance based backwards compatibility is also needed.

Bibliography

Bibliography

- [AEF05] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, L. D. Zuck. Formal Verification of Backwards Compatibility of Microcode. In *International Conference on Computer Aided Verification*, volume 3576, pages 185-198, Edinburgh, Scotland, July 2005.
- [ALM99] D. Avison, F. Lau, M.D. Myers, and P. A. Nielson. Action Research. *Communications of the ACM*, Vol. 42(1):94-97, January 1999.
- [BG99] T.F. Bresnahan, and G. Shane. Technological Competition and the Structure of the Computer Industry. *The Journal of Industrial Economics*, 47(1):1-40, March 1999.
- [CJB04] A. Collins, D. Joseph, and K. Bielaczyc. Design research: Theoretical and methodological issues. *Journal of the Learning Sciences*, 13(1):15-42, January 2004.
- [GMB15] R. T. Gretz, M. Kim, S. Basuroy. Backwards Compatibility in Two-Sided Markets. October (2015).
- [HS02] B. Hailpern, P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4-12, 2002.
- [JIH02] B. Jarvis, L. Izatt, M. Hinckley. Method of implementing a forward compatibility network directory syntax. July 23, 2002.
- [KP96] B. Kitchenham, S.L. Pfleeger. Software Quality: The Elusive Target. *IEEE Computer Society*, 13(1):12-21, January 1996.
- [LI00] J. W. S. Liu. Real-Time Systems, Prentice Hall PTR Upper Saddle River, NJ, USA. January 2000.
- [MBN12] A. Maedche, A. Botzenhardt, and L. Nee. *Software for People: Fundamentals, Trends and Best Practices*. Springer 2012.
- [Hut12] M., Hüttermann. DevOps for Developers (Expert's Voice in Web Development). Apress Edition, page 59, September, 2012.
- [PR12] A. Ponomarenko and V. Rubanov. Backwards Compatibility of Software Interfaces: Steps towards Automatic Verification. *Programming and Computer Software*, volume 38, issue 5, pages 257 – 267, September 2012.
- [PSS10] K. Pandazo, A. Shollo, M. Staron, and W. Meding. Presenting Software Metrics Indicators - A Case Study. *Proceedings of the 20th International Conference on Software Product and Process Measurement (MENSURA)*, volume 20, issue 1, 2010.
- [SNM15] M. Staron, K. Niesel, and W. Meding. Selecting the Right Visualization of Indicators and Measures - Dashboard Selection Model. *International Conference on Software Measurement (Mensura)*, volume 230, page 130-143, 2015.
- [Sta12] M. Staron. Critical role of measures in decision processes: Managerial and technical measures in the context of large software development organizations. *Information and Software Technology*, volume 54, issue 8, Pages 887–899, August 2012.
- [ST88] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation system. *IEEE Computer Society*, 21(10):10-19, October 1988.
- [SV02] J. Singer, NG. Vinson. Ethical issues in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 28(12):1171-1180, 2002.
- [TS14] A.S. Tanenbaum, M. Steen. Distributed systems: principles and paradigms. *Pearson Education Limited*, 2014.
- [VK04] Vaishnavi, V. and Kuechler, W. Design Science Research in Information Systems. January 20, 2004; last updated: November 15, 2015. URL: <http://www.desrist.org/design-research-in-information-systems>
- [VO08] C. Vogt. Six/one router: a scalable and backwards compatible solution for provider independent addressing. *Proceedings of the 3rd international workshop on Mobility in the evolving internet architecture*, pages 13-18, August 2008.
- [WHL97] W. Wong, J. R. Horgan, S. London, H.A. Bellcore. A Study of Effective Regression Testing in Practice. *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, page 264, 1997.
- [WH11] Y. Welsch, A.P. Heffter. A fully abstract trace-based semantics for reasoning about backwards compatibility of class libraries. *Science of Computer Programming*, 92:129-161, October 2014.
- [WH12] Y. Welsch, A.P. Heffter. Verifying backwards compatibility of object-oriented libraries using Boogie. *Proceedings of the 14th Workshop on formal techniques for java-like programs*, pages 35-41, June 2012.
- [WWW1] Ericsson Incorporate <http://www.ericsson.com/> [Date of Access: 17/01/2016].

Bibliography

- [WWW2] Ericsson's Serving GPRS Support Node - Mobility Management Entity (SGSN-MME) [ONLINE] http://www.ericsson.com/ourportfolio/products/sgsnmme?nav=productcategory004%7Cfcb_101_256 [Date of Access: 17/01/2016].
- [WWW3] DSniff Network Auditing And Penetration Testing [ONLINE] <https://www.monkey.org/~dugsong/dsniff/> [Date of Access: 21/02/2016].
- [WWW4] ETHERape Network Monitoring Tool [ONLINE] <http://etherape.sourceforge.net/> [Date of Access: 21/02/2016].
- [WWW5] NetSniff Networking Toolkit [ONLINE] <http://netsniff-ng.org/> [Date of Access: 21/02/2016].
- [WWW6] Packet Analyzer Analyzing Tool [ONLINE] <http://www.solarwinds.com/topics/packet-analyzer.aspx/> [Date of Access: 21/02/2016].
- [WWW7] Packet Square Network Protocol Testing Tool [ONLINE] <https://code.google.com/archive/p/packetsquare-capedit/> [Date of Access: 21/02/2016].
- [WWW8] Scapy Packet Manipulation Tool [ONLINE] <http://www.secdev.org/projects/scapy/> [Date of Access: 21/02/2016].
- [WWW9] TcpDump Packet Analyzing Tool [ONLINE] <http://www.tcpdump.org/> [Date of Access: 21/02/2016].
- [WWW10] Wireshark Packet Analyzing Tool [ONLINE] <https://www.wireshark.org/> [Date of Access: 21/02/2016].
- [WWW11] Sigtest [ONLINE] <https://wiki.openjdk.java.net/display/CodeTools/sigtest> [Date of Access: 22/02/2016].