Automated Testing Toolkit Service
Software Requirements Specification

CHRISTOFFER STJERNLÖF
LUKAS LEVIN
Abstract

Frequent automated testing of software services is vital to speed up the development cycle and ensure upgrades do not break existing features. With a centralised testing service, it is also possible to catch errors at customer sites before they become severe enough that the customers (or in the end – regular people) start suffering from them. It also gives the customers an insight into how well their services are working at a predictable cost. When developing a larger software system such as an automated testing service toolkit, a requirements specification can drastically cut development costs at the expense of a larger up-front investment. We discover some of the immediately important requirements for the first version of such an automated testing toolkit.
Sammanfattning

Statement of Collaboration

Both authors have been involved in all activities that were part of this project. However, the following divisions of responsibility are worthy of mention.

**Lukas** has been responsible for the contact with the stakeholders and formulating the requirements.

**Christoffer** has been responsible for reviewing the available scientific literature, planning the method and editing the report.
## Contents

1. Introduction  
   1.1. Background ........................................... 1  
   1.2. Problem ............................................... 2  
   1.3. Purpose ............................................... 3  
   1.4. Goals ................................................. 3  
   1.5. Method ............................................... 3  
   1.6. Scope ................................................. 4  
   1.7. Risks ................................................. 4  
   1.8. Ethics ................................................. 5  
      1.8.1. Non-disclosure .................................. 5  
      1.8.2. Interviews ...................................... 6  
      1.8.3. Document Analysis .............................. 6  
   1.9. Sustainable Development ............................. 6  
   1.10. Report Outline ..................................... 7  
   1.11. Second Revision .................................... 8  

2. Theory  
   2.1. Requirements Specification ........................... 10  
      2.1.1. Requirements Elicitation ........................ 11  
         2.1.1.1. Interviews .................................. 12  
         2.1.1.2. Document Analysis .......................... 12  
   2.2. Network Services Testing ............................. 13  
      2.2.1. Test Automation .................................. 13  
      2.2.2. Testing as a Service ............................ 14  
      2.2.3. Quality of Service, Feature, and Legacy Testing 14  
      2.2.4. Continuous Integration and agile Ways-of-Working 15  

3. Method  
   3.1. General Considerations ............................... 18  
      3.1.1. Ambiguity ....................................... 18  
      3.1.2. Evaluation for Inclusion ........................ 19  
   3.2. Interviews ........................................... 19  
      3.2.1. Interviewees ..................................... 19  
      3.2.2. Ericsson Customers ............................... 20  
      3.2.3. Interviews ....................................... 21  
   3.3. Document Analysis .................................... 23
## Contents

3.4. Examples ................................................................. 24  
3.4.1. Example: Conflicting Requirements ........................... 24  
3.4.2. Example: Compromise in Requirements ...................... 24  
3.4.3. Example: Obviously Unimportant Requirement .............. 25  
3.4.4. Example: Implicit Requirement Description ................ 25  
3.4.5. Example: Modular Requirements ............................... 26  
3.4.6. Details: Requirement 2.4 ..................................... 26  
3.4.7. Details: Requirements 3.3 and 3.11 ......................... 27  
3.4.8. Details: Requirement 5.1 ..................................... 28  
3.4.9. Details: Requirement 3.1 ..................................... 28

4. Results ................................................................. 31

5. Retrospective ........................................................... 43  
5.1. Conclusions .......................................................... 43  
5.2. Reflection ........................................................... 43

A. Interview Questions .................................................. 49  
A.1. Core questions ...................................................... 49  
A.2. User stories .......................................................... 50  
A.3. Purpose of the EAT-tool ........................................... 51  
A.4. Customers and stakeholder ........................................ 51  
A.5. Delimitations of the tool .......................................... 51  
A.6. Facts and presumptions ............................................ 51  
A.7. Requirements According to IEEE 830-1998 .................... 52  
A.8. Risk and uncertainties ............................................. 53  
A.9. Economics and budget ............................................. 53  
A.10. User documentation and training ................................. 53  
A.11. Add-ons/waiting room ............................................ 54
Abbreviations

**ATT** automated testing toolkit.

**CI** continuous integration.

**E2E** end-to-end.

**EAT** end-to-end automated testing.

**GUI** graphical user interface.

**IEEE** the Institute of Electrical and Electronics Engineers.

**IETF** the Internet Engineering Task Force.

**IMS** IP Multimedia Subsystem.

**JCAT** Java Common Auto Tester.

**NDA** non-disclosure agreement.

**OC** Original Concept.

**PDU** Product Development Unit.

**PM** Program Manager.

**QOS** Quality of Service.

**RD** Remote Developer.

**SI** Systems Integration.

**SRS** system requirements specification.

**SUT** System Under Test.

**TBD** to be determined.
Abbreviations

**TLS**  Transport Layer Security.

**TW**  Test Writer.

**WOW**  Way of Working.
Chapter 1.
Introduction

This chapter provides an introduction to our work, the problem it solves and how the problem is solved. Furthermore, it gives some idea of the value of solving the problem and the delimitations on the scope of the solution. A short discussion on ethics and sustainable development closes the chapter along with an overview of the disposition of this document.

1.1. Background

Testing of services is used both to ensure legacy features are not broken by new updates, as well as to find faults in new features. This means that after every software update (where virtualisation makes it possible to count a large proportion of hardware updates as software updates), testing needs to be performed, ideally in the same environment as the software will be used. Most systems are patched frequently which triggers the need for rigorous testing often. Automation of testing procedures greatly increases the rate at which testing can be performed, and removes a bottleneck in the release cycle.

End-to-end (E2E) testing is a necessity in many of the network services of today. Exhaustively testing each component of an entire network is infeasible, not to speak of reliably testing the integration of the components into the full toolkit.

Ericsson is a large Swedish company involved in producing telecom and network solutions for operators active all over the world. They are no longer producing end-user equipment, but their systems can be found along all other parts of the network.

The end-to-end automated testing (EAT)-tool was created at Ericsson in 2016 as an internal experiment to automate E2E testing of networks. It has been continuously tested on a live network product, and despite being an experimental idea, it has already shown great results. Estimated time savings when the tool reached its first stage of maturity was about 90 percent of the previous test cycles. The E2E
verification testing team was able to verify patches for release much faster, and Ericsson’s network product customers could correspondingly receive them earlier, which means a better service level.

1.2. Problem

What complicated developing a testing service for Ericsson were two things:

1. the Ericsson products and IP Multimedia Subsystem (IMS) services can be run under many valid configurations, and for Ericsson to test all combinations of parameters is unfeasible; and

2. some customers do not exclusively use network hardware or IMS services from Ericsson, but are mixing them with hardware manufactured by competitors.

For these two reasons, customers would like to perform the testing Ericsson does internally, but in their own environment, to ensure the integration of new hardware or software updates has not introduced unknown defects.

Since the EAT-tool was originally developed for internal use only, and is limited in the number of scenarios to which it can be applied, it is expensive from a maintenance, security and business strategy perspective to deploy it externally in the customer environment right away. In light of this, the Ericsson Product Development Unit (PDU) was looking to take the EAT-tool a step further by offering it as a complete service toolkit product to customers, who would then be able to use it in their own networks.

To be able to offer this automated testing toolkit (ATT), it needed to be built around a service oriented architecture, along with the improvements that entails, such as improved security, availability, report generation and more.

The EAT-tool was originally not planned to be a product in the Ericsson portfolio, but rather grew organically in order to streamline the internal testing at Ericsson. This means that the EAT-tool has gone through some radical changes in its structure, including reversions back to earlier states of the project. A predetermined plan of development could have saved the time expended doing exploratory programming.

As we have learned in the Guide to the Software Engineering Body of Knowledge [1], there are several reasons for establishing an unambiguous, complete and verifiable system requirements specification (SRS) for the ATT:

- it limits the scope and increases the speed of development,
- it informs Ericsson customers of what they are buying,
1.3. Purpose

- it assists in discovering flaws in the software,
- it serves as a basis to evaluate e.g. security certifications, and
- it facilitates transfer and deployment in new environments.

Such an SRS does not exist for the ATT, which was the problem we addressed with our work.

1.3. Purpose

In order to address this need, we worked on producing an SRS for the proposed ATT, using established practises and basing the specification on information collected from within the Ericsson organisation. This way, we intended for our SRS to be both accurate (i.e. closely match the expectations from Ericsson) and precise (i.e. and good SRS in and of itself; see chapter 2 where we discuss what constitutes a good SRS further.)

1.4. Goals

Given a specification, Ericsson should be able to develop an ATT which matches their expectations. They should also be able to verify that it indeed does match their expectations in a cost-effective way. It was our hopes that, since Ericsson relayed their customers interests to us, our specification would also satisfy the customers who get the option to purchase the final product based on it.

Some of the other goals linked to establishing an SRS are mentioned in sections 1.2 and 1.3, such as increasing the transparency of the product and its development, as well as making it easier to audit the product from a security standpoint, which increases user confidence over the long run.

1.5. Method

We searched the available literature related to both constructing requirement specifications and automated network services testing as a service, attempting to get more than one perspective on each subject.

When creating the SRS, we primarily used two established requirement elicitation techniques: interviews and document analysis (also known as domain analysis).
The interviews were held with Ericsson employees with varied positions within the company, in order to get as close to a full picture as possible (including being able to relay customer expressed desires). There were eight rounds of interviews, with a progression from wide-scope, few-details surveying with few people, into the final rounds which were held with many people discussing details rather than painting broad strokes.

The document analysis, performed mostly on emails and chat logs, provided some additional perspective from other users as well as a more candid insight into the day-to-day communication about the software.

Condensing all this information into actual requirements is a fraught process where each statement can easily be misinterpreted. We relied heavily on confirming our assumptions and understanding with our interview subjects in order to avoid these problems as well as possible, and tried to err on the side of having too many interviews rather than too few. Conflicts in the requirement set were sorted out by the product owner at Ericsson.

### 1.6. Scope

The limited time-frame in which this thesis work was to be completed did, by necessity, also limit the scope of what could be accomplished within the borders of this thesis work. The original idea that spawned this thesis work was to fully implement the software components of the ATT, but this very ambitious idea was quickly refined to simply defining the characteristics of such a toolkit – a formidable task in and of itself!

Additionally, the scope of the SRS we have produced was limited in order to preserve options for implementation later on. In line with the recommendation from the Institute of Electrical and Electronics Engineers (IEEE) [2], an SRS should limit the range of designs to those that fulfill the needs of the users, but it should never impose a specific design in any aspect of the implementation.

In the SRS produced, we have elected to address functionality, interfaces, performance, attributes and design constraints related to things such as portability, correctness, security, data integrity, resource limits and operating environments; all of which are core components of an SRS according to IEEE.

### 1.7. Risks

There are very few risks involved in developing an SRS for the proposed ATT, all stakeholders considered. Failure to extract an SRS entirely would, of course, count
as unnecessary expenses for Ericsson, but it would not affect any other party, since Ericsson did not externally communicate the plans to create an ATT. That said, customers of Ericsson who became involved in this thesis work may get a false impression of an SRS being promised.

Failing to extract a good SRS does carry a larger risk, both for Ericsson, its customers and, in the worst case, society at large. If the produced SRS is ambiguous, incomplete or impossible to verify, the ATT developed based on the SRS may contain flaws in places where the SRS was either silent, impossible to satisfy or impossible to verify. Flaws in that system are at best a source of inconvenience but will likely cause an unnecessary expenditure of money and time. In the worst case, a flawed ATT may instill a false sense of false security, under which patches with critical errors are silently accepted and labeled by the system as fully working. For a sufficiently important system (which Ericsson indeed do develop), this may cause harm to society and people.

1.8. Ethics

There are no ethical concerns involved directly with the SRS itself. However, the methods we used to gather information about the desired requirements are linked to some ethical considerations.

Regardless of method used to gather information, analysis had to be done to make sure this report and SRS does not contain sensitive information, neither about their customers nor themselves that would then be publicly available. That also applies to information about systems and software code protected under non-disclosure agreement (NDA). This is also to protect us, the authors of this report, from liability.

1.8.1. Non-disclosure

A large portion of the technology and concepts we have been introduced to and worked with during this project are Ericsson trade secrets. This includes some of the parts of the SRS we have produced. In this report, these trade secrets have been either excluded or discussed in very general terms. As a consequence, the SRS reproduced within this report is incomplete, and the full version exists as an Ericsson internal resource only.
1.8.2. Interviews

Participants in interviews were notified that their answers were recorded and in some manner traceable back to them. They were informed that the recording had the purpose of being a memory aid, and the answers needed to be linked to the individual in order to weigh the needs of different stakeholders.

Participants were informed that they could terminate the interview at any point if they felt uncomfortable with sharing more details about their work. We began all interviews by instructing the participants that we want to use as much as possible of what they say, but that they have the option of asking us to not share some details. Whenever we were unsure, we made sure to ask.

1.8.3. Document Analysis

Documents analysed as part of this study were obtained with the express permission to perform document analysis to gather requirement suggestions. They were kept locally on the participants computers and never uploaded publicly.

1.9. Sustainable Development

Sustainability (the ability to persist, divided into the environmental, economic, social, individual and technical dimensions) [3] is an emerging concern when writing an SRS today [4].

Chitchyan, Becker et al. discovered in their research about the state of practice in requirements engineering [3] how the subject of reflecting sustainability concerns in the SRS stage of software design is poorly understood. Interviewed practitioners report that sustainability is, in their experience, put aside as a tertiary concern. Sustainability is only taken into account after other functional and non-functional requirements; oftentimes it is not considered at all. Some practitioners even misunderstand sustainability as being limited to conservation of natural resources, and thus completely separate from software engineering. Another view is that sustainability necessarily increases the development cost and clients are not willing to fund this additional expense.

The Karlskrona Manifesto outlines how there are sustainability concerns in all types of software development; explaining that when the software is used as part of a system, that system will adapt to the software, which in turn will have effects on the current sustainability status of the system. [5] Put slightly more humorously, “An axe can easily be made from recyclable steel, but it will still have a negative environmental impact if used to clear-cut a forest.” This highlights the importance
of considering sustainability as early as possible during the design cycle, when changes to the design are cheap [6] [7].

Since sustainability is a non-functional requirement, it remains slightly outside the scope of our work, which is primarily focused on functional requirements. Some sustainability concerns (extendability of code base, cross-culturally readable visualisations in the interface, ability of local engineers to alter the product according to their needs and so on) overlap strongly with other functional requirements which were discovered throughout the process. All other requirements have been considered for their sustainability impact, but no requirements have been introduced or dropped on account of sustainability.

It has been said that automation in general promotes an unsustainable market environment [8], and the architecture which we aim to define is no exception to this. However, establishing an SRS – which is the true aim of this thesis work – can only aid in analysing the impact of the product on the market environment, and as such should not be able to make things worse.

1.10. Report Outline

This thesis is organised as follows:

Chapter 1 contains information on the underlying problem, as well as what this thesis attempts to do to address the problem. It also includes information on delimitations on this work, and considerations in terms of ethics and sustainable development.

Chapter 2 provides information on the prior art as well as existing research, primarily related to requirements elicitation techniques, but also to some degree automated testing and testing services.

Chapter 3 outlines the process which was used to establish requirements in general terms. This includes both how information was gathered from suppliers and customers, and how this information was later distilled into a concrete requirements specification and solution proposal.

Chapter 4 contains a brief version of the full SRS, reproduced as faithfully as information security policies would allow.

Chapter 5 sums up the work that was done and provides a reflection on the process and quality of the work, as well as ideas for further developments and how the process could be improved until the next time.
1.11. Second Revision

This is the second revision of this thesis report, which is substantially different enough to warrant a summary of the included modifications. Compared to the previous version, we have been much more detailed in chapter 3, and made section 3.4 a subsection of that chapter, rather than its own chapter. This version of the report also has a much more finalised version of the srs in chapter 4. Other than that, there have only been minor grammatical and typographical fixes.
Chapter 2.

Theory

This chapter will summarise the existing research and published material surrounding both requirements specifications as well as network service systems testing. These two parts to this thesis work that are worth considering separately.

First of all, the purpose of this thesis work was to produce an SRS – and in pursuit of that there were several established methods to gather requirements as well as protocols for how the final specification should be arranged.

Additionally, there is a separate purpose of the ATT described by the SRS: end-to-end testing within the field of network services. There are already a number of testing procedures in existence within the network services field; the suitability of these and their properties were briefly examined by looking at both the publicly accessible literature as well as patents and marketing material.

When it comes to researching the literature about testing, it is easy to cast a far too wide net, accumulate a multitude of articles with only passing relevance, and never recover from the backlog of reading. To prevent this, we have decided to research only the most relevant and popular published material, where published simply refers to it being available to the public, either online or in print form. We decided not to base our literature search on internal Ericsson material.

We applied the "relevant" and "popular" criteria in rather informal ways. Popularity was decided based on both number of citations in academic texts and number of references found online. Relevancy was determined by attempting to subjectively determine whether or not the work would contribute to our study based primarily on title and abstract.

The informal way we reasoned about popularity and relevancy introduces a risk for bias toward confirming our own assumptions. To combat this, we attempted to incorporate multiple perspectives where possible, i.e. if one prominent resource made a certain claim, we tried to find a credible (published in high-impact journal, many citations, many corroborating alternative sources, and/or well-reasoned arguments) source with the opposite claim and evaluate their respective arguments, in order to get a better understanding of the subject as a whole.
We also received literature suggestions from Ericsson employees connected to this project, and all of those suggestions were evaluated the same way as our search results and not given any priority.

2.1. Requirements Specification

From the IEEE Recommended Practise for Software Requirements Specifications [2], we have learned that an SRS is concerned with

**Functionality**

The specific workflows the software is supposed to support;

**External interfaces**

The ways through which the software interacts with people and other systems (software as well as underlying hardware); and

**Performance**

Factors such as speed, availability, response time, recovery time of demanding functionality and other measures of how well the software supports its functionality.

It is also explained that a requirement normally exists because it is a part of the nature of the required work-flows, but it may in some cases exist because of some limitation of the project in which the software is developed. However, an SRS should not specify implementation details and should not impose additional constraints on the implementation. In the words of the document [2], “an SRS limits the range of valid designs, it never imposes a specific design on any aspect of the implementation.”

Any ambiguous terms – ambiguous either for the user of the software or the developer – used as part of a requirement must be unambiguously defined in a glossary. Having a third party review the SRS for any ambiguity is recommended.

All requirements in an SRS must have the following three properties.

**Completeness**

If a requirement is not complete (because it is to be determined (TBD) or any similar reasons), it must specify why it cannot be determined at the moment, when it will be determined, and what must be done by whom until then to make it possible to determine it.

**Verifiability**

All requirements must have a cost-effective way to check if it is fulfilled. They cannot include phrases like “works well”, “good human interface” or “shall usually happen”, unless those terms (“well”, “good” and, “usually”) are unambiguously defined in terms of measureable quantities.
2.1. Requirements Specification

**Traceability**

All requirements must be traceable both backward and forward. In other words, the source of each requirement must be specified, and each requirement shall also have a unique identifier that can be referred to in future requirements.

The SRS should normally not be concerned with design aspects related to modularisation. In other words, it should not specify partitioning the system into modules, allocate functions to modules or describe the flow of information between modules. However, these sorts of requirements *may* be necessary in some cases (related to physical, capacity, safety, etc. constraints).

It is recommended by IEEE as well that the SRS is produced by a combination of representatives from the supplier as well as potential consumers.

The IEEE also specifies a few unambiguous terms to express varying degrees of optionality of requirement conformity in their standards board operations manual [9]. In our SRS, we have opted to instead use the terminology defined by the Internet Engineering Task Force (IETF) in RFC2119 [10], simply because we were more familiar with this set of terms.

2.1.1. Requirements Elicitation

Several techniques for requirements elicitation exist. Examples include [11]

1. interviews,
2. workshops,
3. brainstorming,
4. document analysis,
5. surveys,
6. observation, and
7. interface analysis.

It is worth mentioning for full transparency that some unstructured brainstorming was hard to avoid when we sat down and discussed the results of the other elicitation attempts, but it was not performed formally enough to justify a section here.
2. Theory

2.1.1. Interviews

Zowghi and Coulin give a survey of requirements elicitation techniques\cite{zowghi}, in which they describe interviews as allowing a requirements engineer to collect large amounts of data quickly. Since they are based on human interaction, interviews can be held at various levels of formality, or structure:

- **Structured**
  - A fully structured interview is performed off a set of predetermined questions, with little or no deviation.

- **Unstructured**
  - The opposite of a structured interview, where there is little or no pre-written material and the entire interview is very conversational in nature.

- **Semi-structured**
  - A semi-structured interview is grounded in a smaller set of pre-written, important core questions, and takes a conversational turn to fill in surrounding information and clarify details.

The informal and conversational nature of the semi-structured interview combined with the pre-written core questions provides the best of both worlds: interviewees are given an opportunity to voice questions that haven’t been considered before, but they are unlikely to get stuck in details when there’s no clear big picture question to move on to\cite{zowghi}.

However, according to experience recorded by Hove and Anda\cite{hove}, it is a mistake to think that the semi-structured interview does not need to be prepared just as much as a structured interview. Crucial for conducting an efficient semi-structured interview is proper preparation, in order to avoid common errors related to poor interaction between interviewer and interviewee. As an example, prepared user-stories can provide a launching point for further discussion.

The efficiency of interviews is further supported in another overview of requirements elicitation techniques\cite{masters}, where Masters concludes that interviews are the most effective technique for requirements elicitation. To minimise the risk for misunderstanding, the interviewee should preferably get access to the interview notes. This also provides an opportunity for the interviewee to include more information.

2.1.1.2. Document Analysis

Again in reference to the Zowghi and Coulin survey of requirements elicitation techniques\cite{zowghi}, where they explain document analysis, also known as domain analysis, as a requirements elicitation technique that is particularly useful when the project is supposed to partly replace or augment an existing system. Using this
2.2. Network Services Testing

Existing documentation surrounding the current and future system is examined in detail; existing documentation includes instruction manuals, email conversations, design documents, and any other pre-written material relevant to the functionality of the project. Other forms of domain related documentation that may be examined is—perhaps surprisingly—marketing material for competing or alternative products.

In their overview of elicitation techniques [15], Masters goes into further detail and concludes that document analysis by itself will generally not find enough requirements to satisfy customer demands.

### 2.2. Network Services Testing

Since the ultimate goal of this project was producing an SRS for a test automation service that was intended to target the endpoints of network services, there was some interest in learning what the prior art looks like when it comes to end-to-end network services testing.

#### 2.2.1. Test Automation

Test automation is generally believed to be favourable over manual testing. Testing manually will, according to this idea, result in more faults and a more expensive development effort. Besides experimental evidence showing this [16], there is also the more intuitive reasoning that the larger up-front cost of automated testing pales in comparison with the savings over a potentially infinite product lifetime [17].

However, analysis by Ramler and Wolfmaier [17] reveals that assuming an infinite (or even very long) product lifetime with the same fixed set of test cases is not a realistic model. If one instead considers a fixed testing time budget, automating a manual test scenario will reduce the amount of manual testing that is possible to perform within that time budget. There exists some optimum amount of conversion from manual to automated testing where no important manual test scenarios are lost in the total reduction of test scenarios.

Thus, automation of testing needs to be considered carefully and not taken as good in and of itself. The toolkit we have specified is defined with this in mind; since it supports a wide range of test scenarios, the choice of which scenarios to automate is left to the test engineers closer to the problem at hand.
2.2.2. Testing as a Service

Testing as a service has several apparent benefits shared with other “as a service” domains: the maintenance burden on the users decreases, resources can be used more efficiently when pooled, knowledge (including security improvements) can be shared across environments, and the total cost of ownership is explicitly known to the user at all times [18].

When providing testing as a service, there are some problems that need to be managed to be able to provide an efficient service. One of the more pressing issues is that of capacity during peak loads. For automation of testing to be worth it, test suites have to be run faster than it would be possible to run them manually; a sufficiently overloaded service is no better than manual testing. Two approaches to solving this problem are [19] related to

- **scheduling** which attempts to both run similar tests using the same environment, and evaluate which test sets are more urgent or likely to uncover important errors; and

- **elasticity** by designing the service in a horizontally scalable way and allowing additional hardware resources to be allocated quickly during peak usage.

The most commonly known testing as a service application on the market today is the Oracle solution called, predictably, “Oracle Testing as a Service”. However, this is primarily focused on web applications and does not fulfill all of the requirements of the toolkit to be developed [20].

2.2.3. Quality of Service, Feature, and Legacy Testing

When testing, it is important to have a good understanding of what you are supposed to test. Fortunately, most network protocols and services follow some sort of protocol specification.

Some things that are of interest when testing network services are

1. **Setup time** - the initial handshake when two endpoints connect when communicating should not take too long.

2. **Transmission integrity** – the accuracy and precision of the transmitted data should be reasonable given the application.

3. **Features** - the features promised by the service or protocol must work. Some service providers limit the number of features, and this must be reflected in the test suites that are run for that provider.
2.2. Network Services Testing

4. Legacy support – some services must support different versions of the same protocol at the same time, and sometimes even different protocols which both fulfill roughly the same need.

5. Total Quality of Service (QoS) testing – not all quality issues can be defined as being a property of one of the above issues alone. The above examples must also integrate seamlessly enough together to uphold the expected quality of the service as a whole.

2.2.4. Continuous Integration and agile Ways-of-Working

Major changes has happened over the decades within the software business as to how software is developed. In the first decades software was written using the waterfall method, which was adopted from the manufacturing and construction industries. [21] [22] This is a non-iterative Way of Working (WoW) which makes sense when building a house since it has to have no construction errors which could cause accidents. The same method isn’t as good a fit for software development [23]; this could pass by unnoticed when most end users didn’t have fast networks and couldn’t receive timely updates. Most users would have to obtain some form of physical media if the software was updated.

In 2001, the number of people who had access to fast networks was growing rapidly, and a new way of writing software was needed. A group of developers released The Manifesto for Agile Software Development [24] which have twelve principles:

1. Customer satisfaction by early and continuous delivery of valuable software
2. Welcome changing requirements, even in late development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity: the art of maximizing the amount of work not done is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

The development cycles of most Ericsson network products consist of short development cycles, also known as continuous integration (CI) loops. What CI means in a practical manner is frequently releasing software updates, features from requirements and bug fixes for the multitude of components that each network product is made of. Each component has its own CI loop and will have its own set of tests to make sure that the specific component is working properly before a release [25].

Then the product will have to be tested as a whole (E2E), usually by a separate team. They will have to verify that all components work as intended or to a specific standard and then report back to the different teams if any faults were encountered, this is called regression testing. Then the updates have to be verified by the customer and if found to be working properly deployed into their Ericsson products, this is called acceptance testing. These kinds of tests are done for each iteration of the CI loop, and each iteration is usually what’s known in agile wow as a sprint. Each sprint lasts for about 3 to 4 weeks so in a year of continuous deployment, this equates to a lot of repeated testing. And this is where the importance of automated testing really makes itself known. In reference to the previously mentioned principles, this address principle one, three and to some extent number eight.
Chapter 3.

Method

We evaluated the seven techniques for requirements elicitation listed in section 2.1.1 for their applicability to our project. We decided to exclude three based on their up-front cost alone. These were,

workshops because they require too much time from too many key stakeholders,
surveys because they require too large a sample size to be statistically meaningful, and
observation which would require the construction of a prototype.

Further, we suspected one additional method would not provide enough new useful information to legitimize its cost. This was

interface analysis which is of limited usefulness because of the limited user interface available in the predecessor software and, as before, the up-front cost of creating a prototype of the new software.

We decided to attempt this elicitation technique in a limited scale if there were resources available for it, to see if it could quickly unearth some new information that was not found using the remaining three methods. However, it was not part of our primary focus. We decided to primarily focus on the remaining three elicitation techniques:

interviews where we speak to a few key stakeholders,

brainstorming in collaboration with key stakeholders, and

document analysis where we look at anything written about the predecessor technology as well as this project

In this chapter, we will describe in more detail how we approached these techniques, and what specific actions were taken.
3. Method

3.1. General Considerations

There were generally two bigger pressing problems with this work. First and foremost, there is no exact recipe for eliciting requirements and compiling them into a specification, so some amount of improvisation and a huge amount of intuitive reasoning was required.

The second problem we experienced was that of resources. After the information gathering process and conversion of audio to text, we ended up with just under 72 kB of text, spanning over 22 files, excluding system logs and documentation from the EAT-tool. This is probably not enough to capture all the requirements both past and future, and therefore cannot give a reliable picture of the full system. This problem was slightly mitigated by one of the authors’ previous involvement in the development of the predecessor system.

3.1.1. Ambiguity

In order to get an accurate SRS which closely matches the actual desires of the stakeholders, we had to be meticulous when going from (e.g.) an interview response to a concrete requirement. A large portion of the raw information we collected fell under what we came to call “ambiguous phrasing”. It is important to note that this is not only a concern for information presented with natural languages, but also things such as code fragments and commit messages.

To be able to measure this tendency toward ambiguity, we crudely split the information we gathered into units. For natural language, a unit was a sentence. For code fragments, a unit was a procedure. Commit messages counted as individual units.

All files containing collected information, but excluding log files and documentation, were split by hand into units, yielding a total of 839 units, distributed normally with an average of 40 per file, and a standard deviation of around 10. Out of these 839 units, a massive 814 (making up 97%) was phrased such that we could not with absolute certainty determine an unambiguous meaning. The 25 units that were truly unambiguous were either commit messages (17) or extraordinarily clear interview responses (8).

We used a very ad hoc process to extract clear requirements from these ambiguous statements, relying entirely on verifying the requirements with the interviewees (and preferring to verify with the source of the statement where possible) to ensure we were not completely off base.
3.1.2. Evaluation for Inclusion

There is also the problem of deciding which parts of the information gathered are of a high enough priority to warrant inclusion in the requirements specification.

When multiple conflicting requirement suggestions were encountered (for example, see section 3.4.1), only one of the requirements can be kept as-is, or a compromise must be reached (for example, see section 3.4.2). In our project, we resolved these conflicts in discussion with the owner of the project, described under section 3.2.1 as oc. Requirement suggestions that were obviously unimportant were excluded immediately (for example, see section 3.4.3), and suggestions that seemed unimportant but were not obviously so were reviewed by the owner of the project.

In other words, the owner of the project took the role of final arbiter whenever there were conflicts or insecurities in the requirements. We intentionally assigned this role to a single person in order to expedite the process at the cost of some accuracy [26], since the project was limited in time. We picked the owner of the project because he had the most experience both in the field and in talking to customers about what they want to get out of the system.

This arbitration happened in three different ways: as part of interviews, in private conversation and by digital correspondence.

3.2. Interviews

All interviews were recorded using a dictaphone app on a smartphone, in order to have an accurate copy of what was said during the interview; this was especially important since many of the interviews were conducted one-on-one, without the benefit of a third party taking notes.

A listing of the talking points that formed the basis of the semi-structured interview is provided in appendix A. These were not shared with the interviewees prior to holding the interviews. Notes from the interview were shared to the interviewed party after the interviews, however, and the interviewees were welcome to return feedback. This is a good step to avoid biasing the results too much due to the evaluator effect [27].

3.2.1. Interviewees

Interviews were conducted with five people who were all Ericsson employees. Based on a privacy request, we have protected the identities of all five, instead choosing
to refer to them by a symbolic name representing their contribution toward the ATT project and us. They have the following roles.

Original Concept (oc)

oc is a Software Architect at Ericsson, and the primary stakeholder in the ATT project as a consequence of having the original idea and being the current owner of the project. oc is an important interview subject because he has the most total knowledge of the project. He is also our contact person through which we reach Ericsson customers.

Systems Integration (si)

si is a Senior Test Automation Specialist at Ericsson. The EAT-tool is stand-alone software, but the planned ATT is meant to work as a part of the Ericsson test automation portfolio. si has good knowledge about what the Ericsson test automation ecosystem looks like, and was able to provide us with useful feedback on interoperability aspects.

Program Manager (pm)

pm is a Senior Technology Specialist at Ericsson and Program Manager of the overarching development program under which the ATT project resides at Ericsson. Any important strategic directions the ATT project should align itself with will be known by pm, which makes interviewing him a requirement from Ericsson.

Remote Developer (rd)

rd is a Senior Software Developer at Ericsson, based in Asia in order to provide perspective from the Asian Ericsson customers. rd works with developing the developing the EAT-tool from this perspective. For this reason, rd could provide valuable feedback to us as well, both about technical solutions and customer expectations.

Test Writer (tw)

tw is a Verification Engineer at Ericsson. As a person who writes EAT test cases, tw can provide valuable feedback on the usability of the system from an advanced users’ perspective.

All interviewees except tw preferred for the interview to be conducted in English, a request we honored. We had assumed in advance that would be the case for most interviewees, since we knew ahead of time that oc and rd do not speak Swedish. This was not a problem, since both authors are fluent in English.

3.2.2. Ericsson Customers

In order not to reveal too much about the project to Ericsson customers, we were not allowed to speak directly to Ericsson customers. All customer requests and
questions to customers were directed through oc.

This may be a problem, because the external Ericsson customers will ultimately be the ones to benefit from this work, and with that position, they should be primary targets for eliciting requirements that may get misinterpreted through the eyes of Ericsson in the role of supplier. We voiced this concern but were assured by oc that he would try to remain neutral in transmitting the wishes of the customers.

We also received some customer requirements from pm who, through the Ericsson Customer Unit, stays up to date with what the customers need in order to make correct strategic decisions.

3.2.3. Interviews

We conducted 8 rounds of interviews, with the intent as described below.

**Initial probing of feature set**

**Interview with oc**

When this interview was held, we had already had some conversations with oc about the project and where it’s heading. We had tried to make sense of the general idea of the system and proposed some basic talking points regarding what the system was about. The reason we held this interview was to formally consolidate our beliefs as well as to get the answers recorded to prevent fickle memory problems affect the final result.

**Groundwork for integration with Ericsson services**

**Interview with si**

We learned during the previous interview that the att was meant to integrate well with other tools in the Ericsson testing portfolio – notably the Java Common Auto Tester (jcat) library. In order to get this correct we held an interview where we learned about the jcat library and the ways integration happens with it, as well as what other possibilities that it gives in terms of control and reporting. In this interview, we also learned about a lot of functionality that exists in other Ericsson testing libraries, some of which made for useful requirement ideas.

**Confirmation that rough draft aligns with strategic goals**

**Interview with oc and pm**

After having drafted a rough idea of a requirements specification – a draft which can probably be more accurately described as a list of requirement ideas – we held an interview with both oc and pm in the same room. The goal of this interview was to verify that the requirements we had gathered
so far were in line with the long term strategic goals and that the product described for them was of a kind which was likely to receive funding.

At this point, we discovered that we had the wrong idea of the end purpose of the product, but the real purpose was not too far off, and we could refocus our efforts.

**Working out details**

**Interview with oc**

After having refocused onto the real purpose of the ATT, we interviewed oc again to iron out some of the finer details of the system, given his long experience with developing the predecessor. We were able to gather surprisingly many requirement ideas by asking about specifics rather than generalisations.

**Attempt at getting a full view of the system**

**Interview with oc**

We interviewed oc again with a more complete draft ready since the previous interview, and got some additional pointers as well as corrected some simple misunderstandings.

**Interview with PM**

Immediately after, we interviewed PM to confirm the new detailed draft with future plans. PM was very enthusiastic about our progress and as we have noticed during this work, enthusiasm brings on a lot of requirements ideas – most of them completely out of scope of the first version of the product!

**Interview with SI and RD**

We then talked to both SI and RD in a conference call, where we had the opportunity to check with an external perspective whether we had the right sort of idea with what we were doing. The dialog between SI and RD was very enlightening, and put the project in a different perspective when we got to listen to two peers arguing about it, rather than us being told how things work.

**Confirmation of specifics**

**Interview with oc**

After having polished the draft in accordance with a lot of this new information, we had the next interview with oc. Again, a few misunderstandings were ironed out, but this interview was mostly useful because it let us work together with oc to find reasonable compromises for requirements that probably otherwise wouldn’t make the cut.
Additional perspective

**Interview with TW**

We held a short interview with TW to get an additional external perspective from someone involved in the product, but not directly in the work of creating a requirements specification.

**Finalisation, inclusion of additional perspective**

**Interview with OC and SI**

The last interview we had was one with both OC and SI. We verified that the new requirements suggestions from TW were fairly considered, and then we made sure all the last feedback was communicated to us.

After the last interview, a final draft was created and sent by email to the interviewees for review. At this point, the only feedback we received was on minor details such as spelling and technical details relating to the specific systems that are expected to be tested using the system described by the requirements.

### 3.3. Document Analysis

The readily available body of documentation discovered in project consisted almost exclusively of emails and chat logs. We also had access to parts of the codebase for the predecessor tool, as well as the commit messages relating to it. In total, this made up 21 kB out of the 72 kB of information we have collected, proportionally divided as 68% emails, 24% chat logs, and 8% commit messages.

These were translated into specific requirements suggestions by reading them and through intuitive reasoning writing down what seemed like a feature the author of the text either

- had created;
- reported a bug, or created a bug fix for; or
- described, either implicitly or explicitly.

The last item is worth expanding slightly upon, which we do in section 3.4.4.

After we had written this down, the original authors (where possible) or the project owner (where original author was not reachable) were asked if the feature request we assumed was there seemed correct. Sharing the original content of the information we gathered would be in violation of the security policies in place, which is the reason that is not included in the appendix.

We also gathered some system logs and documentation from the predecessor technology, the EAT-tool. These were useful as brainstorming inspiration, but for the
most part not sufficient for our work. We were not allowed to discuss these further in our report.

3.4. Examples

In order to provide a more visceral feel for how this process was applied concretely to specific requirements, we have included some examples of the process of arriving at a particular requirement.

3.4.1. Example: Conflicting Requirements

When compiling requirement desires, we discovered that while one person had expressed that “the system should never end up in a state where there are connectivity problems”, someone else had, later in the process, explained that “the system must be able to perform a connectivity test”. These are conflicting because a system that never has connectivity problems will always return “no problems” when the connectivity test is performed.

While having a faux connectivity test is not a problem in and of itself (and may even be beneficial, if future development expects the connectivity test to become useful), we decided to bring this up for review by oc.

The way this was interpreted by oc (who later verified it with the author of the first statement), was that the first statement (“never any connectivity problems”) was more of a figure of speech expressing the desire for high stability and robustness of connections, and not an absolute requirement. Thus, this requirement was overridden by the later statement about the connectivity test (which is a prerequisite to maintain connectivity stability).

3.4.2. Example: Compromise in Requirements

This is a more interesting example, where both conflicting requirements ended up in the specification. A strategic decision was laid down from higher up in the Ericsson hierarchy by email, and it was clear about the fact that “only technicians should be able to configure the hardware”.

This was received by email. When we attempted to confirm it by talking to tw, we learned that “some regular users must be able to configure the hardware”.

After further discussion with oc, it turned out that the reason some regular users must be able to configure the hardware is because the tool is planned to be used
for experimentation internally in Ericsson. This wider internal use is predicted to result in more hardware failures, which in turn will require regular users to configure the hardware.

The way this was resolved, then, was by allowing technicians to grant hardware configuration privilege to regular users.

### 3.4.3. Example: Obviously Unimportant Requirement

We wanted to bring up an example of an obviously unimportant requirement, to offer a sense of what kind of material that got thrown away without further consideration. The following statement was communicated in an instant message.

“Just imagine if the system analysed when your boss was interacting with it, so if you screw something up, you’ll know when to barge into their office to prevent them from seeing it!”

This was most likely meant as a joke, and is in any case not a good way to spend developer time.

There is a grain of a useful requirement in there: usage statistics in general are not obviously unimportant. However, since not one piece of information we collected mentions user statistics in any other capacity, we have simply decided to view it as unimportant.

In fact, it may be a bit of a misnomer to call this category *obviously* unimportant, both because scarcely anything is really unimportant if you read into it deeply enough, and because what’s obvious to one person may not be to the next. However, we decided to use that terminology anyway because it best fits the general process of intentional ignorance that went into that phase of discarding requirements.

### 3.4.4. Example: Implicit Requirement Description

As an example of what is meant by implicit description of a requirement, one chat message contained the phrase “the system appears to detect when I connect a component, but it doesn’t want to make use of it unless I restart the whole process”. This contains two implicit feature descriptions (“the system detects and makes use of components that are plugged in when the system is started” and “the system should detect and make use of components that are plugged in while the system is running”).
These specific requirements were both later dropped because the features they provide was considered to be an implementation detail of the hardware controller this system interfaces with, and not an interface concern for this system.

### 3.4.5. Example: Modular Requirements

Requirements 1.1 and 1.2 are very similar. They are stated as “All requests must be authenticated.” and “Authentication must use standard asymmetric cryptography algorithms with at least the security level of Transport Layer Security (TLS) 1.2.” respectively. They could be merged into one requirement stating that “All requests must be authenticated with standard asymmetric cryptographic algorithms with at least the security level of TLS 1.2.”

The reason these are split into two separate requirements is in order to make it easier to update the requirements specification whenever stronger crypto algorithms are required. The fewer things that change in the specification, the easier it will be to re-check it for consistency and completeness. Thus, by isolating requirements that are expected to change frequently, future workload is reduced.

This was performed based on a request from Ericsson security specialists.

### 3.4.6. Details: Requirement 2.4

Requirement 2.4 states that “Test cases created by a user of class Manager or lower must belong to the same organisation as the creating user at the point of creation”. This requirement means that users can create their own test cases for the software to run.

This requirement stems from one of the initial core requirements set forth by oc. One of the business goals driving the project is to have a unified platform for several different forms of testing. While it is possible to imagine a platform that has all these different forms of testing built in, it was judged inefficient in terms of manpower.

Thus, it was decided that the testing platform should be agnostic to the types of test being performed, and furthermore which specific test cases are run. The types of tests and specific test cases are meant to be created and configured by the individual verification engineers responsible for those tests. This also decreases the test feedback cycle when complications arise, because the verification engineers doing the testing can also debug their own test cases (requirement 2.9) and run variants of them (requirement 5.4).
3.4. Examples

It was discovered later in the process that the space of types of tests desired was larger than initially envisioned by us; we had approached this project with the mindset of testing only a specific class of hardware, but the ATT is meant to be a general testing service for any type of centrally-controlled testing. This initial mis-prediction of the purpose of the project meant that some of the initial research was spent delving into the details of that specific class of hardware – knowledge which may very well prove to be useful later, but is not integrated in this report.

There was no further misunderstanding or conflict after we had confirmed this requirement with the owner of the project.

3.4.7. Details: Requirements 3.3 and 3.11

Requirement 3.3 states that “When the system is unable to keep up with its current workload, the failure mode should not be such that the test cases fail if they wouldn’t under normal circumstances.”

Requirement 3.11 states that “Toolkit execution policies must be in place to prevent the test controller from starting too many parallel executions, defined as when test cases stop running at 100 % of their execution speed, or when the results are directly affected by the system being overloaded.”

These requirements are related and have overlapping context. They arose during document analysis when we studied log files from tests performed with the existing EAT-tool to get a sense of the kind of work the tool is expected to perform.

The logs were collected during a week long period, which was described by the verification engineers as “ordinary”. These log files were dense with details, which we avoided by doing basic statistical summaries of what we considered to be key metrics: number of concurrent tests, time taken per test, number of positives and negatives, both across time and space domains (i.e. “which single test case has failed the greatest number of times”, in contrast with “which single test run has the greatest number of test failures?”).

In this process, it was discovered that it is part of common test suites to test the continuous operation of some aspect of a system, and these tests monitor the status of the system for anywhere between a few minutes and a full hour. Upon reading the documentation for the EAT-tool, we could confirm that the systems under test are not necessarily designed for concurrent access, which means the current EAT-tool prevents further testing to be performed on that system until the currently running test has finished.
Since the ATT centralised service should be able to act as a controller for several – disparate but identical – systems under test, it is of significance that this limitation of the EAT-tool is not replicated in the ATT system.

It also became clear to us that there are some long-running test cases which do not test continuous operation, but rather a specific series of commands repeated with high frequency. These tests will typically run for an hour, while exercising some specific part of a system hundreds of times in rapid sequence. This kind of test can be run on multiple systems at the same time, and the ATT must have the capacity to issue those commands and collect the resulting data without causing test failures in itself.

### 3.4.8. Details: Requirement 5.1

Requirement 5.1 in this version of the specification states that “The toolkit must provide a graphical control panel interface for all configuration options and settings related to the toolkit.

This is a very clear and marked contrast from the final draft of the requirements specification, which had a requirement stating that “The toolkit should not provide the user with a graphical user interface (GUI) in its first implementation”.

Originall, stakeholders had rejected the idea of a GUI because the toolkit was primarily focused on internal Ericsson use, and in that capacity, functionality was prioritised over usability. When the opportunity arose to be able to sell this toolkit to customers sooner than anticipated, the focus of the project shifted slightly to provide robust foundational functionality that is easy to use. This is used in addition to the internal control mechanisms (requirement 4.1).

However, the original idea of constructing a modular results reporting GUI which simply reads a standardised log format is still retained (requirement 4.2). Test result reporting, statistical analysis and so on is a complex problem which benefits from separate resources being dedicated to it.

### 3.4.9. Details: Requirement 3.1

Requirement 3.1 states that “The software suite should run a diagnostic self-test before a scheduled test run to detect any potential equipment errors before the test run.”

This requirement was developed using personal experience from one of the developers of the EAT-tool. There had been occurrences where an entire scheduled nightly test suite had failed due to some faulty setting or hardware malfunction.
If these errors are reproducible, a quick diagnostic of the most important features would catch these errors before the testers leave the office for the day.

This would provide an opportunity to temporarily disable some of the affected test cases if the errors are not fixable before the nightly test suite is run. These disabled test cases would still come up as failed but the test suite would be able to complete faster or other test cases could be temporarily added in its place. This would not be a part of the core functionality, but as a complement. It was still desirable to include this requirement due to the potential efficiency when testing that this could provide but it wasn’t important enough to have it as mandatory.
Chapter 4.

Results

Through the requirements elicitation techniques we used we collected a massive amount of information about the prospective system. This information included both overt requirements requests and what we call requirement suggestions, i.e. sentiments that hid the wish for some underlying requirement on the system. From this, we evaluated each potential requirement individually as well as in conjunction with all other potential requirements.

This effort resulted in a final SRS for the ATT. Part of this SRS is reproduced below, to the extent that is possible without exposing Ericsson trade secrets.

As explained in chapter 2, to indicate the degree of optionality of conformance to specific requirements we use the terms must, must not, should, should not and may. These are used in accordance with their IETF definitions [10].

<table>
<thead>
<tr>
<th></th>
<th>Authentication, Authorization and Users</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>1.1</td>
<td>All requests <strong>must</strong> be authenticated.</td>
</tr>
</tbody>
</table>
### 4. Results

<table>
<thead>
<tr>
<th></th>
<th>Authentication <strong>must</strong> use standard asymmetric cryptography algorithms with at least the security level of <strong>TLS 1.2</strong>.</th>
<th>Using standard algorithms is important for user trust and ease of deployment. Asymmetric cryptography increases flexibility over symmetric alternatives. As a consequence of this and 1.1, there is no support for unencrypted communication.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td><strong>Authentication</strong> <strong>must</strong> use standard asymmetric cryptography algorithms with at least the security level of <strong>TLS 1.2</strong>.</td>
<td><strong>Authentication</strong> <strong>must</strong> identify a user as belonging to exactly one of the following classes of users:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A Tester is a user who creates test cases; corresponds to a Verification Engineer at Ericsson</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A Manager is a user who schedules test cases; corresponds to a Test Coordinator at Ericsson</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A Technician is a user responsible for managing the UE performing the actual tests and the connection from the service to the hardware</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• An Administrator is a user with full administrative privileges over the entire service (this role is likely to be limited to a few service engineers)</td>
</tr>
</tbody>
</table>

- Tester
- Manager
- Technician
- Administrator
1.4 The user classes **must** form a hierarchy, where
- Manager has all privileges of Tester in addition to their own
- Technician has all privileges of Tester and Manager in addition to their own
- Administrator has all privileges of Tester, Manager and Technician in addition to their own

Hierarchically structured user classes corresponds well with user expectations, decreasing misunderstandings (which are important to avoid especially relating to user privileges).

1.5 Authentication **may** identify a user as belonging to an organisation.

The concept of an organisation allows sharing of test equipment and test suites among multiple users.

1.6 Any user with a lower class than Administrator **must not** be able to add an organisation.

1.7 Any user with a lower class than Manager **must not** be able to add a user.

1.8 When a Manager adds a user, the new user **must** have the same organisation as the Manager.

If the user is an Administrator, they can add the user to any organisation.

1.9 A user with a lower class than Administrator **must not** be able to view or otherwise access resources associated with an organisation different than their own.

<table>
<thead>
<tr>
<th>2</th>
<th>Test Cases and Test Suites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Rationale</td>
</tr>
</tbody>
</table>

33
### 4. Results

<table>
<thead>
<tr>
<th>Test Cases and Test Suites</th>
<th>A test suite represents a collection of test cases.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>must</strong> have an identifier that is unique across the whole system (over all organisations).</td>
<td><strong>must</strong> be associated with at least one organisation.</td>
</tr>
<tr>
<td><strong>may</strong> allow test cases and test suites to be associated with more than one organisation.</td>
<td>Sharing tests between organisations reduces duplication of both effort and storage.</td>
</tr>
<tr>
<td><strong>must</strong> belong to the same organisation as the creating user at the point of creation.</td>
<td><strong>must not</strong> be able to create new test suites.</td>
</tr>
<tr>
<td><strong>must not</strong> be able to mark a test case for inclusion in a test suite.</td>
<td>If the inclusion is requested by the author of the test case, this can be referred to as accepting the test case.</td>
</tr>
<tr>
<td><strong>must not</strong> be able to remove a test case from a test suite.</td>
<td>This can be also referred to as submitting the test case for review.</td>
</tr>
<tr>
<td><strong>must not</strong> be able to request inclusion of the test cases they have authored into one or more test suites.</td>
<td>Test cases can only be modified by the original author or a privileged user. Test cases must carry an association to the user who created them.</td>
</tr>
<tr>
<td>2.10</td>
<td>If a test case is modified, it <strong>must</strong> be removed from all suites and then automatically submitted for review.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Operational Behaviour</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>3.1</td>
<td>The software suite <strong>should</strong> run a diagnostic self-test before a scheduled test run to detect any potential equipment errors before the test run.</td>
</tr>
<tr>
<td>3.2</td>
<td>The diagnostic self-test <strong>must</strong> run to completion in no more than 600 seconds.</td>
</tr>
<tr>
<td>3.3</td>
<td>When the system is unable to keep up with its current workload, the failure mode <strong>should not</strong> be such that the test cases fail if they wouldn't under normal circumstances.</td>
</tr>
<tr>
<td>3.4</td>
<td>The central service controller <strong>must</strong> be able to handle concurrent executions of test cases.</td>
</tr>
<tr>
<td>3.5</td>
<td>The test controller and hardware controller <strong>must</strong> be able to run on the same machine.</td>
</tr>
<tr>
<td>3.6</td>
<td>The test controller and hardware controller <strong>must</strong> be able to run on different machines.</td>
</tr>
<tr>
<td>3.7</td>
<td>Test controller and hardware controller <strong>must</strong> be able to communicate with each other as long as the network allows it.</td>
</tr>
<tr>
<td>3.8</td>
<td>The software <strong>must</strong> have measures in place accepted by Ericsson security specialists to protect confidential and <strong>NDA</strong> covered information from unauthorised access.</td>
</tr>
<tr>
<td>3.9</td>
<td>The system <strong>must</strong> have a component management system capable of detecting when test cases freezing due to improper component configuration states.</td>
</tr>
<tr>
<td>3.10</td>
<td>The component management system <strong>must</strong> be capable of resetting components to known configurations.</td>
</tr>
<tr>
<td>3.11</td>
<td><strong>Toolkit execution policies</strong> <strong>must</strong> be in place to prevent the test controller from starting too many parallel executions, defined as when test cases stop running at 100% of its execution speed, or when results are directly affected by the system being overloaded.</td>
</tr>
<tr>
<td>3.12</td>
<td><strong>Toolkit execution policies</strong> <strong>must</strong> be in place to prevent the user from starting too many parallel executions, defined as in 3.11</td>
</tr>
</tbody>
</table>
### 3.14 Inbound connections

**must** be load balanced to avoid exhausting network resources.

The expectation, rather than the exemption, is many connections. Load balancing increases toolkit relative robustness. (Robustness relative to itself, ie the same toolkit with load balancing is considered to be more robust than a version without it.)

### 3.15 Customer environment and Ericsson internal testing environment

**must** be isolated from each other.

To ensure sensitive information belonging to Ericsson is protected by not allowing customer executions to read anything from the Ericsson execution.

### 4 Interfaces

<table>
<thead>
<tr>
<th>Description</th>
<th>Rationale</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 The system <strong>must</strong> be able to start running tests when asked to do so over the <strong>JCAT</strong> interface.</td>
<td>This is both for backwards compatibility and to make it possible to merge different Ericsson products into one toolkit. <strong>JCAT</strong> is the Ericsson standard Java library for automated testing.</td>
<td>3.8</td>
</tr>
<tr>
<td>4.2 The system <strong>must</strong> call methods of the <strong>JCAT Log-Writer</strong> interface to report its results.</td>
<td>As above, this is to increase the potential for integration with other Ericsson tools.</td>
<td></td>
</tr>
<tr>
<td>4.3 The communication between test coordinator and hardware controller <strong>must</strong> be implemented exactly according to its separate specification.</td>
<td>This allows separately developed hardware controllers to be used in conjunction with the system. The hardware controller protocol is specified separately because it is expected to evolve quickly, following the rate of advancements of the systems under test.</td>
<td></td>
</tr>
</tbody>
</table>

### 5 Control Panel and Reporting

<table>
<thead>
<tr>
<th>Description</th>
<th>Rationale</th>
<th>Parent</th>
</tr>
</thead>
</table>

37
4. Results

| 5.1 | The toolkit **must** provide an graphical control panel interface for all configuration options and settings related to the toolkit. |
| 5.2 | All parameters of the toolkit **must** be configurable. |
| 5.3 | All parameters **must** be accessible outside of program in an standardised XML format. |
| 5.4 | Users with a class of at least Tester **must** be able to start a test case at the press of a button in the control panel. | One press execution is one of the primary selling points of this toolkit. |
| 5.5 | Users with a class of at least Manager **must** be able to schedule single executions of tests in the control panel. |
| 5.6 | Users with a class of at least Manager **must** be able to schedule recurring executions of tests in the control panel. |
| 5.7 | When a test suite is running the system **must** defer any new executions until the test suite is finished. | This is an initial solution that ensures no scheduled test suites are accidentally skipped. It is suboptimal and may improve in the future. |
| 5.8 | Users **should** be able to run a diagnostic analysis at the press of a button in the control panel |
| 5.9   | Any user **must** have an accessible view of the passed and or failed test cases of the test runs of an interval of user defined length from the current date to a minimum of 3 and a maximum of 12 months back in time from their organisation. | To get quick feedback that is easy to understand is vital for efficiency. |
| 5.10  | The software **should** provide a way to produce a report where relevant information is included according to a user-controlled filter. | I.e. a tester might not want debug (low-level) information in their results. |
| 5.11  | The user **must** have the option to download reports in the PDF format. | The reports are commonly used by higher-level employees who may not have the technical understanding to work with a more technical, machine-processable format. |
| 5.12  | The software **must** be able to run a test suite at the press of a single button by the user. | “One-press execution” is one of the baseline requirements for a successful system of this kind. |
| 5.13  | Any user **should** be able to see statistics represented as either number or graphs or both over the last nightlies for either the entire day, week, month or year. | This was desired to be able to see trends in the results, 6 months was defined as a usable maximum length. |
4. Results

| 5.14 | Statistics tracked **must** include the following measures, all averages per week:  
  - Number of tests run  
  - Number of tests passed  
  - Fraction of tests passed | Average per week is desired because test suites often repeat in a weekly cycle, with different sets of functionality being tested on different days. | 5.13 |

| 5.15 | The software **must** be able to automatically mark failed test cases as false positives. | Being able to filter out which tests indicate a problem with the System Under Test (SUT) and which tests failed for other reasons is vital for quick feedback and troubleshooting. This requirement deliberately leaves out the conditions under which test cases are marked as false positives, because that is expected to be an area of quick progress. As a consequence, a user must not rely on the system to mark anything in particular as a false positive. |

| 5.16 | 95% of test cases marked as a false positive **must** have failed due to connectivity issues or an error with the testing hardware. | For automatic false positive detection to be useful, it must have high specificity. (Under the assumption of correctly written test cases, the sensitivity is also high.) |

| 5.17 | The tool **should** give examples of what could have gone wrong based on what errors are generated. | Having the software suggest documented solutions to common problems decreases onboarding time for new testers. |
5.18 Users **must** be provided with a wiki including the following documentation:

- Installation guides for all software and hardware components required to run this tool
- User guide with instructions on how to run a test suite
- Instructions on how to write new test cases
- Troubleshooting guide for the most common errors

Having the instructions embedded within the system decreases the risk of forgetting to distribute instructions separately.

<table>
<thead>
<tr>
<th>6</th>
<th>Hardware and configuration</th>
<th>Rationale</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>The toolkit <strong>must</strong> be able to run in a containerized format capable of execution on a local machine or in a cloud environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>All hardware <strong>must</strong> be supported as long as they can be automated to perform all or subsets of the required functions used to test the network.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>The hardware controller <strong>must</strong> support controlling multiple hardware of different manufacturers, models and software versions simultaneously.</td>
<td>6.2</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Technicians <strong>must</strong> be able to modify or change the hardware. This allows technicians to replace faulty components or expand the range of operations supported by the software.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Technicians <strong>must</strong> be able to do any necessary hardware parameter adjustments to the software. The software is configured for the specific hardware setup. When the hardware changes, the associated configuration changes need to be made.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Technicians <strong>must</strong> be able to grant access to hardware and hardware parameter adjustments to non-technician users. For internal Ericsson testing, it must be possible for all users to tinker with the hardware in order to test new features.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>Non-technician users <strong>must not</strong> be able to access and/or modify hardware parameters unless granted that right by a technician. The hardware is property owned and fully managed by Ericsson. It contains sensitive components and failure to handle it correctly may result in faulty operation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>The diagnostic self-test <strong>must</strong> include connectivity checks to the required hardware, reporting services as well as probing the configuration state of internal components. If a component is incorrectly configured, this gives the opportunity to reset the component to a known state before the nightly tests are run.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5.

5 Retrospective

5.1. Conclusions

Testing as a service is an emerging area with very little research, as we discovered when doing background research for this project. We imagine the reason is that the entire as-a-service concept is relatively novel (enabled by the recent widespread access to fast and reliable internet connections), and testing appears not to be one of the first areas where the paradigm is explored. Key players in the domain tend to be large corporate organisations with little interest in sharing how they accomplish their goals. Apparent issues when designing such a system is performance, clear presentation of test results and information security.

Semi-structured interviews with key stakeholders are a cost-effective way of bringing out many requirements before building a software software. Document analysis is hard to do methodically, and misunderstandings are easy to make when freely interpreting someone else's written text with no other context. It is also hard to gather fleeting written evidence (e.g. email messages) because access to it is so divided and there are many suspicions raised regarding the safeguarding of the material.

Despite this, with an iterative, agile development process, even the SRS can be made somewhat fluid, and the first version only really needs to support the initial prototype, while keeping options open for future versions.

5.2. Reflection

We believe our resulting SRS is accurate: in the very least, no part of it should be incorrect. We have not yet had the opportunity to put it to test to see if it is also sufficiently well-defined to produce clear answers for all questions that may appear during development.
There are good guidelines readily available for how an SRS should be constructed, as well as plenty of literature on requirements elicitation; what was not obvious to us as we went into this project was how much effort is required up-front to produce a good SRS. Most of the decision making that goes into development without an SRS is moved to “specification time” rather than “implementation time” when constructing an SRS ahead of the initial development efforts.

For other people who plan on doing something similar to our project, we strongly recommend thinking about the following improvements to the process:

- Onboard enough stakeholders early on, who willing to be interviewed and review the SRS.
- Have a good way of organising all suggestions that are supposed to turn into requirements.
- Do not assume a document will be easy to get access to just because it exists digitally; private correspondence is still considered private by most.
- Be very firm in not considering details of usage before the basic overarching goals of the project have been established.
- Have a process in place for quickly synchronising knowledge between members of the project.

All in all, we are content with the results from this 10-week study, but if we could have extended it we would have done so to increase our confidence in the results.
References


Appendix A.

Interview Questions

These interview questions formed the basis of the semi-structured interviews that were held. The answers are not reproduced in this document for two reasons:

1. The sheer volume of text would be overwhelming, and
2. Some parts of what has been said falls under the Ericsson security policy and can’t be shared publically.

All mentions of “tool” or “the tool” in this appendix refers to the EAT-tool. Similarly, “system” or “the system” refers to the proposed ATT. User classes used below include both:

**Tester**  
*A user who has the authority to run test suites and create test cases. Official Ericsson nomenclature is Verification Engineer, and*

**Test Manager**  
*A user who is ultimately responsible for the testing performed by testers, and communicates the testing activity to other divisions.*

A.1. Core questions

1. Why was the tool needed?
2. Has the goals for the tool changed since its creation?
3. Who are the main stakeholders for the system? (Owner?)
4. Are there other stakeholders? Who are they?
5. Who will use the system?
6. What classes of user will interact with the system? (Testers, administrators, integration engineers, executives, developers?)
A. Interview Questions

7. What do these users expect to get from the system?

8. What existing external systems will the system interact with? Through what protocols?

A.2. User stories

• As a tester, I want to be able to see a quickly understandable overview of the results of last night’s test suite

• As a test manager, I want to be notified of the nightly results by e-mail

• As a test manager, I want to be able to generate graphical reports from a specified interval of versions to show my managers

• As a tester, I want to be able to write new test cases

• As a tester, I want to be able to submit new test cases to one or more nightlies

• As a tester, I want to be able to configure the hardware configurations of the suite

• As a tester, I want the suite to do a system diagnosis every day to prevent nightlies from failing

• As a tester, I want to be able to run a suite at will

• As a tester, I want to be able to see statistics over the last nightlies for the entire:
  1. Days
  2. Weeks
  3. Months
  4. Year

• As a tester, I want to be able to troubleshoot failing test cases quickly

• As a tester, I want to be able to tell quickly if a failed test case is due to:
  – Hardware state
  – EAT tool instability
  – Network error/instability
  – Connection failures

• As a tester, I want to have a wiki where I can search for troubleshooting, user/installation guides and more
A.3. Purpose of the EAT-tool

1. Why was this created in-house and not outsourced?
2. What are the overlying goals with this tool?
3. What impact has this tool had for the department? (Company?)

A.4. Customers and stakeholder

1. Who is the main target group for this tool?
2. Who will handle the support of this tool?

A.5. Delimitations of the tool

1. Who are the tool designed for? (Ericsson internal/customers?)
2. Who will implement this tool?
3. What license will this tool run under? (Apache 2.0/GNU/GPL/closed source)
4. What architecture will this tool have? (IPO-model, SOA)
   a) How will the tool interact with these systems?
   b) What API:s will this tool provide?
   c) What API:s will this tool use?

A.6. Facts and presumptions

1. What external factors can affect this system?
   a) Economic?
   b) Time?
   c) Others

1. External Interfaces/data flow
   a) What is the transmitted data called?
   b) Why is it relevant?
   c) Where does it come from and where does it go? Unicast? Multicast? Anycast?
   d) Does the transmission use a standard protocol?
   e) What is the valid range of values? What is the expected accuracy? What are the units of measure?
   f) How fast is the output produced? How fast is the input consumed? What is the maximum allowed transmission time?
   g) Is the transmission sensitive/secret?
   h) Is the transmission critical? What should happen if it fails?

2. Functions (fundamental actions taken by the system in response to timing or input events)
   a) What validity checks are performed? What is the expected result if the check fails?
   b) How is the result/output affected by the operational state?
   c) How is the result/output affected by the input?

3. Information stored permanently
   a) What is the transmitted data called?
   b) Why is it relevant?
   c) Where does it come from?
   d) What is the valid range of values? What is the expected accuracy? What are the units of measure?
   e) Is the information sensitive/secret?
   f) How frequently is the information read?
   g) How frequently is the information written?
   h) Is the information critical? What are the data retention requirements?

4. User classes
A.8. Risk and uncertainties

a) Does the functionality differ for different classes of users? Which classes of users exist in the system?
b) (Remember to specify for all requirements what class of user it concerns.)

A.8. Risk and uncertainties

1. Dependencies
   a) What other system does the tool rely on to operate?
   b) Can the failure of those system cause the tool to not function?

2. Stability
   a) How sensible is the tool?
   b) If a part of the tool isn’t functioning properly, should the tool still be able to run tests?
   c) What parts of the tool are critical for the normal operation of the tool?

3. Performance

4. Other identified risks

A.9. Economics and budget

1. How many man-hours is estimated to implement this service?
2. If there more time is needed, can it be allocated?
3. Will there need to be support for the current implementation of the tool?
4. If there are hardware requirements, estimate our budget for that.

A.10. User documentation and training

1. What documentation should be available for the customers and/or the internal users?
2. How should they be able to access that information?
A.11. Add-ons/waiting room

1. What features can be added that isn’t required for a first release?