Requirements Engineering for Large-Scale Agile System Development: A Tooling Perspective

Master's thesis in Computer science and engineering

MEBRAHTOM GUESH GEBREMICHAEL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

MEBRAHTOM GUESH GEBREMICHAEL

Supervisor: Eric Knauss, Computer Science and Engineering
Examiner: Jan-Philipp Steghöfer, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2019

Abstract

Context: Companies are using the agile methodology for implementing their software-intensive projects. However, different studies have indicated that using the agile methodology in a large-scale context leads to different requirements engineering challenges. According to some studies, tool-support has shown positive results in addressing large-scale agile requirements engineering (RE) challenges. But the current tooling solutions have many limitations in addressing large-scale agile RE challenges.
Objective: This study tries to investigate tool-support related large-scale agile RE challenges and proposes tooling solutions to those challenges by enhancing an existing RE tool called *T-reqs*.
Method: The design science research has been used to conduct this study in three different iterations. Data has been collected using four workshops, two focus groups, one interview and literature. During the workshops, observation and survey questionnaires were used to collect data. Interview data has been analysed using thematic analysis. The functionality assessment results of the artifact which is produced as a result of this study have been analyzed by summarizing and visualizing them using bar-charts. To analyze the usability assessment results of the artifact, the system usability scale (SUS) was applied. For each item of assessment, SUS scores were calculated and interpreted.
Results: The results of this study are a set of tool-support related large-scale agile requirement engineering challenges and their corresponding tooling solutions. The main tool-support related large-scale agile RE challenges identified in this study are difficulty in working with model-based system requirements, difficulty in propagating requirement changes to multiple teams and projects (repositories), difficulty in keeping track of requirement change history, difficulty in balancing the autonomy of agile teams and the need to control requirement documentation, and updating traces in a model and concurrent artifact ID generation. Tooling solutions have also been implemented for those challenges. According to the results of this study, the implemented solutions address the identified challenges.

Keywords: Requirements Engineering (RE), Agile, Large-scale Agile, Tooling, T-reqs, Textual Requirements.

Acknowledgements

I would like to thank Eric Knauss, my supervisor at Chalmers University of Technology, for his guidance and support throughout this study. I also would like to thank my examiner Jan-Philipp Steghöfer at Chalmers University of Technology for his constructive feedback on my thesis work. I am also very grateful for Filip Lange at Ericsson AB, Magnus Ågren and Grischa Liebel at Chalmers University of Technology for their valuable feedback on the technical aspects of the study. Finally, I would like to thank all people who participated in my research.

Mebrahtom Guesh Gebremichael, Gothenburg, December 2019

Contents

1

Introduction

The agile methodology has produced good results in small and medium size software projects. This success of the agile methodology in small and medium size projects has attracted companies that are involved in large-scale software development projects. As a result, many companies are increasingly using agile methods for implementing large-scale software development projects [1], [2]. However, adopting agile methods for large-scale development results in new requirement engineering challenges [1], [2], [3], [4], [5]. This is because the agile methodology was originally created for small co-located team(s) where it is relatively easier for the teams, customers and other stakeholders to discuss the requirements directly [2], [4]. In other words, requirements engineering in agile relies on a face-to-face communication instead of comprehensive documentation [36], [37]. However, face-to-face communication is difficult to achieve in the context of large-scale agile system development. Because large-scale agile projects usually consist of many geographically distributed and interdependent teams; thus it is very difficult if not impossible to achieve the attendance of all teams at the same time. This causes business roles and other stakeholders to be distant from the development teams [1], [38]. The distance between developers and other stakeholders can result in misunderstanding of requirements and loss of customer value [1]. Some literature on large-scale agile suggest hybrid approaches of agile and traditional plan-drive process to scale agile to large-scale environments [40], [39], [41]. However, requirements are defined upfront in traditional plan-driven processes and this does not work well with agile methodologies. According to a study on scaling agile in large organizations, some companies do not follow a clear process to deal with requirements in large-scale agile environment [15]. Therefore, dealing with requirements in the large-scale agile setting is a big concern for companies. This is crucial for the companies because poorly handled requirements may ultimately lead to project failure, result in additional cost or affect the quality of the final product. This calls for better ways of dealing with the requirement engineering activities in large-scale agile environment.

Although some recent studies have identified a significant number of requirement engineering challenges in large-scale agile environment, there is little research on how to address those challenges [2]. According to some studies, companies are looking for tool-support to manage their requirements engineering activities [1], [3]. Some promising results have also been found from using tool support for large-scale agile requirement engineering. One of such tools and which has also become the focus of this study is a tool named *T-reqs*. *T-reqs* was created at `Ericsson AB` with a goal of addressing requirements engineering challenges in large-scale agile system development. The idea of *T-reqs* is to store the system requirements and

tests in *git repositories* together with the code so that development teams will be able to update the requirement documentation. In this way, it is possible to write the system requirements in parallel with the actual implementation so that the requirements can accurately describe the implementation. In an agile methodology which uses continuous integration and delivery (CI/CD) practices, *T-reqs* also solves another important problem: it allows the linking of system requirements and their corresponding implementations by committing them together so that they can be delivered in the same release. This allows stakeholders and suppliers to easily identify which of the requirements have been delivered in a certain release. According to a related study [6], there is a good indication that *T-reqs* addresses some of the problems of scaling agile to large-scale development. For example, it allows teams to be aware of requirement changes at a system level. It also provides them with an easy access to system requirements so that they can be able to propose requirement changes. And this keeps the requirements to be consistent with the implementation. However, *T-reqs* has some limitations and it needs some improvement to make it easy to work with. Not only does it provide limited functionality for some of its features, but also it provides solutions only to limited large-scale agile requirement engineering challenges. For example, it does not support model-based system requirements. It does not also provide a solution to work with system requirements and other artifacts that are shared between multiple repositories.
Inspired by the current limited research in dealing with large-scale agile requirements engineering challenges and the promising results of tool support for large-scale requirement management with *T-reqs*, this study investigates which of the large-scale agile RE challenges reported in literature and the ones discovered during this study can be solved by better tooling and to what extent tooling can address the challenges. This is achieved by extending an open-source variant of the *T-reqs* tool with solutions to the identified large-scale agile RE challenges. The results of the study are common large-scale agile RE challenges related to tooling (meaning RE challenges that can be solved by tool-support), an extended *T-reqs* tool that solves those challenges, a measure of performance that measures to what extent the solutions solve the identified RE challenges and the design theory knowledge used to implement the solutions in the tool. This study will have a contribution in improving the quality of software projects, increasing the speed of development and minimizing the cost of development and/or preventing the failure of software projects by streamlining the requirement engineering activities of a company through tool support.

1.1 Purpose of the study

The purpose of this study is to investigate which of the large-scale agile RE challenges can be addressed by better tooling, to propose viable tooling solutions to the identified RE challenges and to evaluate those solutions. This will be done by extending an existing tool called *T-reqs* [6] which is an open source variant of an in-house developed tool at *Ericsson*. The reason why this tool is selected is that because it was developed to solve large-scale agile RE challenges which makes it inline with the goal of this study. Moreover, an in-house version of *T-reqs* which uses a similar idea is currently being used at *Ericsson* which makes it possible to learn

from its empirical success so far in the industry. Another reason is that there is a need for flexible tooling [9] that can be adjusted to different contexts which *T-reqs* supports by enabling teams to change the template for writing requirements. It is also open source and easily extensible which can be tailored easily to the working ways of a certain company.
New features are added to the tool in three different iterations to solve different set of challenges at each iteration. On every iteration, the tool is assessed against the criteria laid down during the problem identification phase. Finally, the requirement engineering challenges that have been addressed through a better *T-reqs* along with the corresponding solutions are reported. The philosophies or principles used to implement the solutions and the evaluation results of the solutions are discussed as well. This study intends to benefit both researchers and practitioners. Practitioners can use the improved *T-reqs* tool to manage their requirements engineering activities in large-scale agile system development. Researchers can learn about the newly identified large-scale agile requirement engineering challenges and their proposed solutions.

1.2 Problem statement

In large-scale agile projects, many organizations want the system requirements to be written in parallel with the actual implementation so that the requirements can accurately describe the implementation. This needs a support that handles many small and concurrent updates to the requirements from all agile teams. What makes this even more difficult is that requirements frequently change. When requirement change happens in one team, all other dependent teams must be aware of such change as soon as possible to not affect the quality or speed of development. Taking the problem further, requirements change affects other artifacts such as related test cases, quality requirements and source code. This indicates that a miss-communication in requirement change can have a big impact on the software development process which can eventually lead to frustrating the team members, slowing down the development speed or incurring additional cost. A tool named *T-reqs* was created at *Ericsson AB* with a goal of addressing this and other related problems. According to a study which focuses in *T-reqs*, *T-reqs* showed promising results in addressing some of the aforementioned problems. For example, it produced positive results in its support for small and concurrent updates. However, *T-reqs* has some limitations. For example, it has limited support for models. This can be a problem because requirements can also be specified in the form of models. It doesn't also provide a solution for handling requirements that can cross multiple repositories. In the basic version of *T-reqs*, there is no a clear guideline for documenting quality requirements and acceptance tests. The need to document such artifacts is that because system requirements can be related to system requirements and acceptance tests. Its trace-ability functionality is also very limited which make it very slow and difficult to navigate between related artifacts.

1.3 Research questions

This study deals with the following three research questions.
RQ1: Which problems of large-scale agile requirement engineering can be related to tool-support?
RQ2: What possible solutions can be provided based on the *T-reqs* tool?
RQ3: To what extent can the problems (from *RQ1*) be addressed by better tooling?

RQ1 aims to specifically identify RE challenges that relate to tool support. With the basic *T-reqs* tool it is possible to make small concurrent updates to requirements with some limitations such as the lack of model and multi-repository support. Since an in-house version of the tool which uses a similar idea has been in industrial use, it is also worthwhile to investigate if new challenges have arisen or old challenges have changed in priority.
RQ2 aims to propose better or new solutions for the challenges identified in *RQ1* which should be realized by extending the *T-reqs* tool.
RQ3 aims to identify the strengths and weaknesses of the solutions proposed in *RQ2*.

1.4 Scope and Limitations

This study is limited to investigating large-scale agile RE challenges and proposing viable tooling solutions. The proposed solutions are implemented by extending the *T-reqs* tool. The solutions will be also evaluated using workshops, interviews, focus groups and surveys.
T-reqs will be used with *git-based* repositories. Because the *git* version control system has shown positive results with the basic version of *T-reqs*. It also uses *GitLab* as a repository management service and code review tool. This is because the `GitLab groups` feature allows the formulation of consistent trace-links for all members of a group working in the same branch and file. The tool does not deal with requirement elicitation rather it focuses on documenting and managing already elicited system requirements, quality requirements, test cases and acceptance tests.

1.5 Structure and contributions

1.5.1 Structure of the study

Chapter 2 discusses the related work to this study. Chapter 3 provides the background information needed to understand this study. Chapter 4 provides the detailed discussion of the research methodology used and how it was applied in this study. Chapter 5 provides the the findings of the study during all three iterations. Chapter 6 describes the artifact that is implemented as an outcome of this study, i.e., the improved *T-reqs* tool. Chapter 7 is the Discussion section and discusses the results found in this study. Chapter 8 and the concluding chapter is the conclusion section and provides a summary of the study.

1.5.2 Contributions of the study

This study will have a contribution for both the scientific community and software practitioners (individuals and industries).
`Practitioners` can benefit from the following functionalities of the artifact produced as a result of this study (the enhanced *T-reqs* tool):
Managing requirements and related artifacts: Practitioners can use the tool to specify both textual and model-based system requirements in *markdown* and *PlantUML/DOT* files respectively and review them on repository management services such as *GitLab*. *T-reqs* can also be used to manage quality requirements, test cases and acceptance tests. In *T-reqs*, it is possible to establish traces between related artifacts.
Propagating requirement changes to multiple repositories: Practitioners can use the tool to propagate requirement changes to multiple repositories.
Keeping track of requirement change history: Practitioners can use *T-reqs* to keep track of the time a requirement was last changed and who changed it.
`Researchers` can be benefited from the following results of the study:
Investigation of large-scale agile RE challenges: This study has investigated a set of large-scale agile RE challenges that are relevant to tooling. Some of these challenges can be found in Table 1.1.
Tooling solutions for large-scale agile RE challenges: This study has provided possible tooling solutions to some of the identified large-scale agile RE challenges. Some of these solutions can be seen in Table 1.1.

Requirements engineering challenges	**Solution**
In a large-scale agile context, requirements are usually shared between multiple teams and they can quickly change. This makes it difficult to work with model-based system requirements because models are difficult to version control and change.	Model-based system requirements can be specified in PlantUML and the DOT language. Then, *python scripts* are used to generate diagram URLs from the PlantUML/DOT code. Listing 2 shows a code snippet for generating diagram URL from a PlantUML/DOT code. Then, the generated URL is written back to the same file using using a script. Listing 1 shows a code snippet of the *python* script for writing such generated URL to the markdown file. The generated diagram URLs will be rendered as actual diagrams when the file that contains such a URL is viewed on *GitLab*.
When working with requirements that are stored in `git` repositories, it is difficult to propagate requirement changes between multiple repositories.	The *git submodule* feature of the *git* version control system can be used to propagate requirement and other artifact changes between multiple repositories. For this to work, the shared artifacts have to be placed in a separate repository. Then, this repository should be added as a *git submodule* to the other repositories that share the artifacts. This makes it possible to make and fetch changes to the shared artifacts using `git`.

It is difficult to keeping track of requirement change history when requirements are stored in repositories.	The built-in *git* command called *git blame* can be used to know the latest line-based change history of requirements and other artifacts specified in files. The *git log* command can also be used for tracking old change histories.

Table 1.1: Tooling related large-scale agile RE challenges and their solutions

2
Related work

Different studies have recommended different practices, frameworks and tooling for supporting large-scale agile requirements engineering activities. The following reviews are based on some of such studies.

One study on inter-team coordination in large-scale agile system development [10] identified shared mental models, closed-loop communication and trust as important inter-team coordination mechanisms. Although these mechanisms are relevant to large-scale agile development, they are not directly related to tooling.

A systematic review on communication challenges of geographically distributed development teams [11] reported that companies are using different strategies, agile practices and tooling to solve the communication challenges. However, it doesn't discuss which tools are used and how effective they are.

A study on the management of boundary objects (objects that are shared between groups of people) [9] recommends that groups of representatives should be established for each boundary object(shared artifact) to discuss issues and later propagate and store that information in an accessible tool. However, such in person meeting to discuss each boundary object can be time wasting and difficult to manage. Another related paper [2] recommends enabling teams to update requirements, focusing on boundary objects over locally relevant documents and other related practices but there is no much empirical evidence to support those practices.

One study has reported that a negotiation process called "Handshaking with implementation proposals" improved requirements communication during development projects [12]. The technique delegates the detailed requirements engineering to development teams. This enables the teams to easily create their implementation proposals which needs to be negotiated with the customer to reach at agreement. This technique is similar to the philosophy used by *T-reqs* in delegating the detailed requirements to development teams. However, this study doesn't discuss the application of the negotiation process in large-scale agile environment.

A study on requirements for requirements management tools [13] has pointed out different functional requirements that should be supported by requirements management tools. Although this study doesn't deal specifically with requirements for tooling in large-scale agile, the identified requirements can be used in extending and/or assessing the tool to be used in this study. Similarly, another study on RE tools [14] emphasizes the importance of an easy access of requirements to distributed teams and other related functional requirements which can be used for assessing the tool to be developed throughout this thesis work.

Some literature on large-scale agile suggest hybrid approaches of agile and traditional plan-drive process to scale agile to large-scale environments [40], [39], [41]. However,

requirements are specified upfront in the traditional plan-driven approach and this doesn't work well with agile methodologies.
Many large companies are using scaled-agile frameworks such as *SAFe* and *LeSS* [15] to guide their software-intensive projects. But those frameworks don't recognize most of the large-scale RE challenges [2].
Software tools have shown positive results in supporting agile requirements engineering. For example, a study on gamified requirements engineering [16] has shown that user stories and acceptance tests can be elicited with better quality and productivity using gamification techniques. Another study on agile RE [17] uses a tool to empower product owners to create and manage user stories and to enable customers and teams to collaborate on user stories. The combination of intelligent tools such as text-parsing, domain ontology and ontology guidance has also been used to produce high quality user stories [18]. However, all of these studies focus on user stories, not on detailed system requirements. Moreover, most of them focus on requirement elicitation which is not the primary focus of this thesis work.
A case-study with a focus on tooling [19] highlights the capabilities of tools companies are looking for to support their requirements engineering activities and the challenges associated with tooling. According to this case study, companies are looking for RE tool with functional requirements such as data analytics, visualization, effort estimation, integration with upstream tools, dependencies at different levels and stakeholder feedback. However, this case study doesn't provide or recommend concrete tooling solutions for managing requirements.
Another study [6] which is based on the *T-reqs* tool proposes a tool that can be used together with the *git* version control system to manage textual requirements written as *markdown* format. This study is based on the philosophy of bringing requirements to the hands of development teams so that the teams themselves could manage the requirements. The authors of this paper claim that their proposed tool is effective in addressing some of the large-scale agile requirement management challenges [6]. However, it has some limitations. For example, it has limited support for models. This can be a problem because requirements can also be specified in the form of models. It doesn't also provide a solution for handling requirements that can cross multiple repositories. This is also a problem because requirements can be shared by multiple projects in large-scale software development.

3

Background

This chapter presents the theory and background information that is required to understand this study. Section 3.1 describes the basic version of *T-reqs* and the tools used by *T-reqs* for managing different artifacts. This section also describes the different artifacts that can be managed by the basic *T-reqs*. Section 3.2 explains quality requirements and acceptance tests and how they are documented in the improved version of *T-reqs*. Section 3.3 provides an explanation of PlantUML & and the DOT language and their usage in the improved version of *T-reqs*.

3.1 The basic T-reqs

T-reqs is a text-based requirement management tool written in *Python*. "It combines useful conventions, templates and helper scripts with powerful existing solutions from the git ecosystem and provides a working solution to address some known requirements engineering challenges in large-scale agile system development" [6]. *T-reqs* allows agile cross-functional software developers to be aware of system level requirements and to efficiently propose requirement changes [6]. The basic *T-reqs* refers to the version of *T-reqs* before it was improved by this study. The rest of the text in this section describes the different tools used by the basic *T-reqs* and the main artifacts it manages. The artifacts that can be managed by the basic *T-reqs* are system requirements and test cases.

System requirements are detailed descriptions of the services and constraints of a system and are derived from the analysis of user requirements [26]. System requirements are often decomposed in a hierarchical structure into sub-system and component levels of detail. In the basic and improved versions of *T-reqs*, all levels of requirements (system, sub-system and component) are documented in *markdown* files.

System requirements can be related to test cases. " Test case is a set of input values, execution preconditions, expected results, and execution post-conditions developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement" [28]. Test cases are identified by test case IDs. Test cases can have other parameters as well. For example, test case description, expected result, actual result etc. Test cases can be manual or automated. Manual test cases are executed manually by a person without the support of tools whereas automated test cases are executed with the help of software tools and/or scripts. In the basic *T-reqs*, manual test cases are specified in *markdown* whereas the automated test cases are written as *python* scripts.

Markdown is a lightweight markup language with plain text formatting syntax [30]. With the help of markdown, documents can be written in an easy-to-read and easy-to-write format which can then be converted to structurally valid (X)HTML document. Markdown is mainly used for creating web content but it can also be used for producing print-ready documents.
The basic *T-reqs* uses *git* for version controlling system requirements and other artifacts. *Git* is a distributed version control system which is used for tracking source code change history during software development [24]. A distributed version control system enables a developer to work with his/her own copy of the project's history (a clone of the repository). There are two ways of setting up a *git repository* locally: by converting an existing project directory that is not already under *git*'s control into git repository using the *git init* command and by *cloning* a git repository from repository services like *Github* and Git-Lab [24].
In the basic *T-reqs*, it is possible to reference test cases from system requirements by writing the IDs of the respective artifacts as tags in the requirements definition. System requirements can also be referenced from test cases in the same fashion.
The basic *T-reqs* has only one command, i.e., "treqs" which prints out duplicate IDs of artifacts, items without traces, items that are not referenced by other artifacts (missing traces) and items that are referenced by other artifacts but don't exist (missing items).

3.2 Quality requirements & acceptance tests

The improved version of T-reqs manages quality requirements and test cases in addition to system requirements and test cases. System requirements and test cases have been explained in section 3.1. This section discusses quality requirements and acceptance tests. "Quality requirements specify how well the system performs its intended functions" [22]. They are also known by the name non-functional requirements. There are different categories of quality requirements. Usability, performance and maintenance are some examples of quality requirement categories. In the improved version of *T-reqs*, quality requirements are documented in *markdown* files and can be referenced from system requirements, and vice versa.
Acceptance test, which is also known as user acceptance test (UAT) checks if the system meets customer requirements as specified in the contract and/or if the system meets user needs and expectations [28]. In agile software development, acceptance testing is used to check the completeness of a user story. It is common to use frameworks to automate user acceptance tests. *Cucumber* is one of such frameworks and is widely used in industry at the writing of this thesis. *Cucumber* runs automated acceptance tests written in *behaviour-driven development*(BDD) style . *BDD* is a software development style where developers write a set of scenarios to specify the system's behaviour from the user's perspective. The language used to write such scenarios in cucumber is called *Gherkin* [42]. A *Gherkin* file has a *.feature* extension and it contains a set of scenarios that a user story must satisfy in order to be accepted by users of the system. *Cucumber* has also been used in some studies for model-based testing [43].

3.3 PlantUML & the DOT language

In this study, *PlantUML* has been used for documenting model-based system requirements. *PlantUML* is an open-source tool which is used to write UML diagrams using a plain-text language [29]. UML diagrams are commonly used in industry for specifiying model-based system requirements.
UML stands for *unified modeling language* and it is a standard language for specifying, visualizing, constructing and documenting the artifacts of software systems [35]. *UML* is created, standardized and managed by the Object Management Group (OMG). *UML* diagrams can specify both structural and behavioral information of a system. Structural information refer to what must be present in the system being modeled. Component diagrams and class diagrams are examples of structural UML diagrams. Behavioral information refer to what must happen in the system being modeled. Activity diagrams, sequence diagrams and usecase diagrams are examples of UML behavior diagrams.
At the writing of this paper, *PlantUML* supports the following UML diagrams; activity diagram, class diagram, usecase diagram, sequence diagram, component diagram, object diagram, deployment diagram, state diagram and timing diagram. It also supports non-UML diagrams such as gantt diagram, archimate diagram, mindmap diagram etc. Once the diagrams are defined in a simple intuitive language, it is possible to generate diagram images in *PNG, SVG, or LaTeX* format using a locally installed PlantUML tool or the online PlantUML server. The improved T-reqs tool uses the online PlantUMl server to generate UML diagrams from the plain-text PlantUML code.
PlantUML doesn't support some non-UML diagrams. For example, it doesn't support context and data flow diagrams. Therefore, the *DOT* language has been used for documenting non-UML diagrams that are not supported by *PlantUML*. The reason why the *DOT* language was selected for this purpose is that because *PlantUML* uses it internally to draw diagrams and so can be easily integrated in *T-reqs's* model-support feature. DOT is a graph description language. The DOT tool reads attributed graph text files and renders graphs as graph files or in a graphics format such as GIF, PNG, SVG, PDF, or PostScript [31]. The improved T-reqs tool uses the DOT language to write requirement specification diagrams such as flowcharts and context diagrams which are not supported by PlantUML. The DOT tool can be available as part of graph visualization software tools such as *Graphviz*. The improved T-reqs tool uses the online PlantUML server to render graphs that are written in the DOT language. This is possible because PlantUML uses the DOT language internally to draw diagrams.

4

Research Methods

The methodology used in this study is Design Science Research. The following section provides a detailed discussion of the design science research methodology, why it is selected as an appropriate research methodology for this study and how it was applied in conducting it.

4.1 Design Science Research

Design science is the scientific study and creation of artifacts as they are developed and used by people with the goal of solving practical problems of general interest [7]. An important outcome of this research is an artifact that solves a certain domain problem which should be assessed against criteria of value or utility. The design science research involves different activities. The design science research activities used in conducting this study are awareness of problem (problem identification), solution suggestion, solution validation, development (implementation) and evaluation. These activities are depicted in Figure 4.1 along with their relationships.
The design science research methodology was selected as an appropriate methodology because its objective is to develop new technology-based solutions to important and relevant business problems [8] or to improve existing artifacts. This makes it inline with the objective of this study. Because the main goal of this study is to improve the basic version of the T-reqs tool in such a way that it solves the main large-scale agile requirements engineering challenges.
This study has been conducted in three different iterations. Each iteration involves the application of all design science research activities. The following section provides a brief description of the design science activities and a summary of what has been done in this study during those activities.

4.1.1 Awareness of problem

During this stage, the problems that need to be solved during a certain iteration are identified and the importance of solving such problems is justified. The output of this phase is a formal or informal proposal that consists of evidence of the problem and characterization of the external environment.

4.1.2 Solution suggestion

During this activity, solutions are proposed to the problems based on literature, expert recommendation or the researcher's creativity. The output of this phase is a

list of possible solutions.

4.1.3 Solution validation

During this phase, the solutions suggested in the previous phase will be checked for their feasibility. This is usually done together with stakeholders and experts. The solution validation activity is important to avoid time and resource wastage in trying to implement unfeasible and unsuitable solution. If a solution is not valid then another better solution will be suggested. The output of this phase is valid solution/s.

4.1.4 Implementation

During this phase, the validated solution/s is implemented. This can be a model, prototype, method or any other artifact. During the implementation phase, if the solution fails to meet the research requirements, the researcher can return to the *awareness of problem* phase to better understand the problem. The output of this phase is an artifact that solves the identified problems.

4.1.5 Evaluation

During this phase, the implemented solution in the previous phase is evaluated for its capability in solving the identified problems, its quality and other relevant criteria. During this phase, if it was found out that the solution doesn't meet the research requirements, the researcher can go back to the *awareness of problem* phase to better understand the problem. The output of the evaluation phase is a list of performance measures that indicate how well the solution solves the problem in hand. New problems can arise after the evaluation phase and a new cycle should be applied to research those problems.

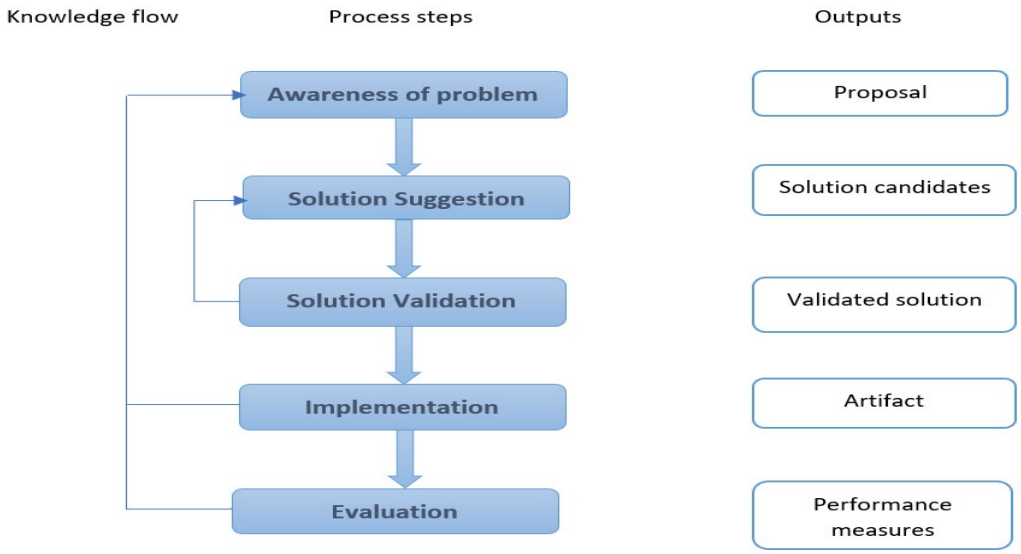


Figure 4.1: Design Science Research Activities

4.2 Data collection

Interviews, focus groups, workshops and surveys are used to gather data during the problem investigation and evaluation phases of the study.
An interview is a face-to-face discussion or communication via some technology between an interviewer and a respondent [20]. There are three types of interviews: (i) *structured*, where the the questions are planned and created in advance and all respondents are asked the same questions in the same order, (II) *Unstructured*, where the interviewer questions are developed as general concerns and interests from the interviewer, (III) *semi-structured*, where the questions are planned in advance but not necessarily asked in the same order as they are listed in the schedule [21]. Interviews can have only one respondent or group of respondents (group interview) [21]. Interview is selected as an appropriate data collection methodology for this study because it allows to get a deep understanding of the experience of the respondents in using the artifact (*T-reqs*) and the problem under research.
Focus groups are a form of group interviews that capitalize on communication between participants to gather data [34]. Focus groups use the group interaction as part of the method. During focus groups, the participants are encouraged to talk to one another. Focus groups are used to explore the knowledge and experiences of research participants.
Workshops have been also conducted to evaluate the *T-reqs* tool. During the workshop, data has been collected through observation. Surveys have been also used after the workshop to get detailed views of the participants. Survey is a method of collecting data from respondents that are considered to represent some population, using an instrument composed of closed structure or open-ended items [20]. The survey questionnaires used in this study assess the functionality and usability of the *T-reqs* tool. The functionality assessment questionnaire contains both open-ended and closed questions. The closed questions are based on the *5-point Likert-scale*. " Likert-scale is a psychometric response scale primarily used in questionnaires to obtain participant's preferences or degree of agreement with a statement or set of statements" [33]. A 5-point Likert-scale ranges from "strongly disagree" to "strongly agree" with "disagree", "neither agree nor disagree" and "agree" consecutively placed in between. The usability assessment data is collected using the system usability scale (SUS) method. System usability scale is a 10-item Likert scale questionnaire that provides a measure of the user's perception of the usability of a system [25]. Table 4.2 shows a list of the data collection methodologies and a list of people who participated during the data collection sessions. The column names in the table are written in short forms. *Int1* stands for interview 1; *WS1*, *WS2*, *WS3*, *WS4* stand for workshops 1, 2, 3 and 4 respectively; *FG1* and *FG2* stand for focus groups 1 and 2 respectively; *I1*, *I2* and *I3* stand for Iterations 1, 2 and 3 respectively and they represent the iteration where each person has participated.

Participant	**Role**	**Experience (in year)**
Person-A	Developer	3
Person-B	Developer	5
Person-C	System Manager	9
Person-D	System Manager Project Manager Software Architect Software Engineer	1 3 9 12
Person-E	Software Engineer	6
Person-F	Software Engineer	1
Person-G	Summer Worker	<1
Person-H	Software Engineer	5
Person-I	System Manager Verification Engineer	1 20
Person-J	System Manager Software Engineer	1 8
Person-K	System Manager Software Engineer	1 18
Person-L	System Manager Function tests Solution Designer Designer	6 11 3 5
Person-M	Master's student	0
Person-N	Master's student	0
Person-O	Master's student	0
Person-P	Master's student	0
Person-Q	Researcher	6
Person-R	Researcher	5
Person-S	Researcher	5
Person-T	Researcher	6

Table 4.1: Profiles of the people who participated in this study

Participant	Int1	WS1	WS2	WS3	WS4	FG1	FG2	I1	I2	I3
Person-A			X					X		
Person-B			X					X		
Person-C		X		X		X	X	X	X	X
Person-D				X					X	
Person-E				X					X	
Person-F				X					X	
Person-G				X					X	
Person-H				X					X	
Person-I				X					X	
Person-J				X					X	
Person-K				X					X	
Person-L				X					X	
Person-M					X					X
Person-N					X					X
Person-O					X					X
Person-P							X			X
Person-Q		X						X		
Person-R						X			X	
Person-S							X			X
Person-T		X						X		

Table 4.2: Data collections methodologies and research participants

4.3 Data analysis

For *Interview 1*, the interview was recorded, transcribed and coded. Then, thematic analysis was applied on the coded transcription to get useful information from the interview. The opinions gathered from focus groups are summarized in text and then interpreted by the author.

The data from the workshops was collected using observations and surveys. During all workshops, the observations were done by the researcher only. The observations have been summarized by taking notes during the workshop. Then, these summarized observations have been interpreted by the author.

The survey questionnaires contain two types of questions; open-ended and closed questions. The responses to the open-ended questions have been simply interpreted by the author. But the responses to the closed questions have been analyzed by summarizing the number of responses for each question in percentage and visualizing this result in a bar chart as explained in the paper by *Dane Bertram* [33]. The Likert-scale based questionnaires assess the functionality and the usability of the *T-reqs* tool. In both functionality and usability assessments, the scales have been converted into numbers for each *Likert-scale item*. That is, *strongly disagree=1; disagree=2; neither agree nor disagree=3; agree=4; strongly agree=5*. The functionality assessment results have been analyzed by drawing bar-charts that show the percentage of *Likert-scale responses* for each *Likert-scale item* in the questionnaire

as it can be seen in Figures 5.1, 5.4, 5.7. Then, the percentage of satisfaction responses (agree & strongly agree) is compared to the percentage of dissatisfaction responses (disagree & strongly disagree) for each item. The usability of the tool have been assessed using the system usability scale. The usability scale results have been interpreted as following:
Step 1: Calculating the score

1. For odd-numbered items: Subtract one from the user's response
2. For even-numbered items: Subtract the user's response from 5
3. For each user, add up the converted responses and multiply the sum by 2.5
4. Then, calculate the average of the results in step 3

Step 2: Interpreting the SUS score
The average SUS score is 68. Table 4.3 provides a rating for each SUS score.

SUS score	**Rating**
>80.3	Excellent
68-80.3	Good
68	Ok
51-68	Poor
<51	Awful

Table 4.3: Interpreting SUS score results

4.4 Iterations

The design science research was applied in three different iterations. In this section, the purpose of the iterations and the methodologies used in each iteration are briefly described. The research questions that have been addressed in each iteration are also described. Generally, each iteration focuses on identifying large-scale agile RE challenges, implementing tooling solutions to those problems and evaluating the implemented solutions. In Table 4.2, columns 3-8 show the data collection methods used in all iterations. In the same table, columns 9, 10 and 11 show the people who participated in Iterations I, II and III respectively.

4.4.1 Iteration I

The purpose of this iteration is to identify initial large-scale agile problems, to implement solutions to those problems and evaluate them.

In this iteration, one interview, one workshop and literature were used to identify large-scale agile RE problems and possible tooling solutions to those problems. To analyze the data collected in this section, the data analysis methods for interview, observation and survey questionnaires which are explained in section 4.3 have been used. This iteration contributes to addressing **RQ1**, **RQ2** and **RQ3**.

4.4.2 Iteration II

The purpose of this iteration is to address the tool's limitations/problems which are identified during the evaluation of *Iteration I*, to identify additional large-scale agile RE challenges and to implement and evaluate tooling solutions to the newly identified challenges. To analyze the data collected in this iteration, the data analysis method for observation, focus group and survey questionnaires which are explained in section 4.3 have been used. This iteration contributes to addressing **RQ1**, **RQ2** and **RQ3**.

4.4.3 Iteration III

This is the last iteration of the design science research. This iteration focuses in addressing the limitations/problems of the *T-reqs* identified in *Iteration II* instead of identifying new large-scale agile RE challenges. Finally, the tool was evaluated using one focus group and one workshop. The workshop was conducted with two software engineering and one computer science master's students. The reason why students were used in this workshop is that because it was difficult to find experts. The tool has also been evaluated with a focus group in the this iteration. One person from academy, another one person from industry and one software engineering master's student have participated in the focus group. The reason why a student was part of the focus group is that because it was difficult to find additional experts. To analyze the data collected in this section, the data analysis method for observation, focus group and survey questionnaires which are explained in section 4.3 have been used. This iteration mainly contributes to addressing **RQ2** and **RQ3**.

5
Findings

5.1 Iteration I

5.1.1 Awareness of problem

In this iteration, large-scale agile RE challenges were identified using one workshop, one semi-structured interview and literature. According to one literature which is based on the basic version of T-reqs [6], lack of model-support in *T-reqs* is identified as one of the main challenges that require solutions. In this iteration, a workshop was conducted to study the idea of *T-reqs* and its functionalities and limitations. Three people from academia and one system developer from *Ericsson AB* had participated in the workshop. During the workshop, the idea of *T-reqs* was presented and the basic version of *T-reqs* was demonstrated to the participants by the industry representative from *Ericsson AB*. Following the discussion, the participants asked different questions regarding the use of *T-reqs* for solving large-scale agile RE challenges. Notes were taken during the workshop to document the opinions of the participants. The main large-scale agile RE challenges identified during the workshop are difficulty in working with model-based system requirements, difficulty in propagating requirement changes to multiple teams and multiple projects (repositories) and difficulty in managing requirement dependencies.
The semi-structured interview is made with one model-based software engineering researcher who also participated in *Workshop I*. He was particularly of interest because of his research experience in Model-Driven Requirements Engineering. The researcher was asked for his comment on adding model-support functionality to T-reqs. He mentioned that it is a good idea to manage model-based system requirements together with the code in a git repository, similar to what is done with the other T-reqs artifacts such as textual system requirements and test cases. But he emphasized that some modeling tools can be limited in their capabilities such as support for lay-outing and this can affect the software development experience:

> "If you would add modeling to T-reqs, the question is where does it stop; how far can you go? ... I would think people like developers, testers and architects need stuff like lay-outing. They need to be able to quickly navigate in the model. Currently, almost in all requirements (in the basic T-reqs version) you have traces; one requirement points to another one. They need to look at those quickly. And the problem is typically those traces can go across models. That means somehow you have to be able to navigate in the model".

He also mentioned another problem related to tracing system requirements to test

executions:

> "Specifically, testers need to link test cases and test executions to requirements. That's extremely important ... for example at Ericsson AB or in any big product as test cases run many times and you have different versions and so on, you might want to know this one test execution worked but this one didn't work and then how is it connected?"

The challenges identified from literature, the workshop and the semi-structured interview are summarized in Table 5.1. Most of the values in the *source* column of the table are written in short forms. *Int1* stands for *Interview I*; *WS1* stands for *Workshop1*; *Literature1* refers to the literature which is based on *T-reqs* [6].

Problem ID	**Problem description**	**Source**
P1.1	System requirements can be documented in a textual format and/or in the form of models. However, models are difficult to version control and modify. This makes model-based system requirements difficult to work with in a large-scale agile context which is characterized by inter-dependent teams and quickly changing requirements.	Int1, WS1, Literature1
P1.2	Writing models in a textual UML tool called PlantUML is the solution suggested for solving P1.1 in this iteration. However, the major repository managers like GitLab and GitHub do not support PlantUML rendering at the writing of this paper. And this makes it difficult to review the model-based system requirements on repository management services like *GitLab*.	WS1
P1.3	System requirements are usually related to other artifacts such as test cases and quality requirements. Therefore, it should be possible to navigate to related artifacts from the model that specifies a certain system requirement. However, most modeling tools do not support traceability functionality in a model. In other words, the models are usually rendered as images and so it is difficult to add trace-links on top of them.	Int1, WS1
P1.4	System requirements can be related to other artifacts in addition to test cases. For example, they can be related to quality requirements and user stories. User stories can also be related to user acceptance tests. However, only system requirements and test cases have been documented in the basic *T-reqs*. In other words, there is no a clear guideline or syntax on how to document the requirement related artifacts such as quality requirements, user stories and acceptance tests.	WS1

P1.5	In a large-scale agile context, high-level system requirements are usually decomposed into low-level system requirements. These low-level system requirements are usually assigned to different but related software components that are implemented in different repositories. In such situations, requirement changes in one repository can affect the related requirements which reside in different repositories. However, it is difficult to propagate requirement changes to multiple teams and multiple repositories (projects).	WS1
P1.6	Requirements can depend on other requirements. However, it is difficult to manage the dependencies between requirements. For example, it is difficult to visualize requirement dependencies and to get updates about changes to related requirements.	WS1
P1.7	Testers are usually interested to know which requirements represent the currently failed test cases. However, *T-reqs* doesn't currently trace requirements to test execution.	Int1

Table 5.1: Problems identified in Iteration I

5.1.2 Solution suggestion

In the first iteration, only the problems *P1.1-P1.4* in Table 5.1 have been prioritized to be solved. This is because these problems are related and it makes sense to solve them in the same iteration.
In *Workshop I*, the participants did not only raise large-scale agile RE challenges but also they suggested possible solutions for some of the challenges. Most of the participants suggested that using textual modeling tools for writing model-based system requirements could provide solutions to the problems *P1.1-P1.3* in Table 5.1. More specifically, PlantUML was mentioned as a possible solution for writing model-based system requirements in *T-reqs*. But, one participant also criticized PlantUML for its limited layout-control functionality (the ability to re-orient the diagram or change the position of the diagram elements). To solve problem *P1.4*, it was suggested to document quality requirements and acceptance tests in *git repositories* together with system requirements and test cases and also to define a trace-ability syntax to link them with related artifacts. Based on the discussions in the workshop and the author's creativity, the suggested solutions are to use *markdown format* and *cucumber feature* files to document quality requirements and acceptance tests respectively. It was suggested to not document user stories in *T-reqs* but to reference them with their IDs. This is because the participants believed that user stories could be better managed using visual story boards such as *scrum/kanban* boards. The suggested solutions to problems *P1.1-1.4* in Table 5.1 are summarized in Table 5.2. The solutions address model-support related problems. In other words, they solve sub problems of the model-support problem and will be used together to solve the whole problem.

Solution ID	Problem ID	Suggested solution
S1.1	P1.1	A textual UML tool called PlantUML is suggested for writing model-based system requirements in *T-reqs*. PlantUML allows development teams or requirement people to write model-based system requirements in a textual format which is easy to version control and modify.
S1.2	P1.2	It is possible to preview a PlantUML diagram by generating a URL from the corresponding textual code and writing it to a markdown file. *Python scripts* will be used for generating the URLs, creating the *markdown* files and writing the URLs to *markdown* files. When the *markdown* files are opened on repository services that support *markdown*, the URLs are rendered as actual diagrams. The diagrams are rendered with the help of the online PlantUML Server (which is available at `http://plantuml.com/`). For previewing PlantUML code locally, a PlantUML previewer extension for Visual Studio Code text editor is suggested.
S1.3	P1.3	Trace-link generation solution for PlantUML diagrams is suggested in order to be able to navigate to traced artifacts by clicking on parts of the model. Trace-links will be generated based on the trace-ability information specified in the corresponding PlantUML code.
S1.4	P1.4	Documenting quality requirements as *markdown* files and storing them in *git repository* together with the system requirements was suggested as a solution for creating a link between system requirements and quality requirements. Storing the quality requirements together with the system requirements does not only allow teams to easily access the quality requirements but also to link them with system requirements by committing both artifacts together or by referencing them with their commit hashes. User acceptance tests are widely used in agile projects. Therefore, storing acceptance tests in *git repository* together with the actual implementation allows the teams to access them easily. *Cucumber feature files* are suggested for documenting user acceptance tests in *T-reqs*. Traces can be created between related artifacts by using *T-req's* trace-ability syntax.

Table 5.2: Suggested solutions in Iteration I

5.1.3 Solution validation

The need to easily version control model-based system requirements using version control systems such as *git* makes textual UML modeling tools preferable solutions for writing model-based system requirements. In *Workshop I*, *PlantUML* was sug-

gested as a solution. Although it is difficult to control the layout of PlantUML diagrams, it was not compared with other similar tools and no further research was made in similar tools that support layout-out control because of time constraints. Therefore, the selection of *PlantUML* is solely based on the expert suggestion from *Workshop I*. As a result, the documentation of the *PlantUML* tool was consulted. Then, sample PlantUML diagrams were written and run on the online PlantUML server in order to try the *PlantUML* tool. Such exercises with the PlantUML tool revealed that it is possible to generate a URL for any syntactically correct PlantUML diagram by encoding the PlantUML code to a simple string of characters that contains only digits, letters, underscore and minus character based on the algorithm used by PlantUML. This makes it easy to communicate the diagrams as URLs. Also, *PlantUML* supported most of the major *UML* and *non-UML* diagrams based on the exercises with the tool. Therefore, PlantUML was accepted as a solution to write model-based system requirements in *T-reqs*. It was also validated by a model-driven requirements engineering researcher in *Interview I* (*Person-A*). In other words, *PlantUML* was accepted as a valid solution because the model-driven researcher in *Interview I* (*Person-A*) accepted it as a solution for writing model-based system requirements and the exercises made by the author by writing sample model-based system requirements in *PlantUML* showed positive results.
The documentation of quality requirements and acceptance tests was validated by trying out sample examples. Sample quality requirements and acceptance tests were written in *markdown format* and *cucumber feature files* respectively. Then, a python script was written to read those artifacts. Documenting quality requirements and acceptance tests in this way produced positive results.
The implementation of the suggested solutions in this iteration are summarized in Table 5.3.

5.1.4 Implementation

Solution ID	**Implementation**
S1.1	Sample model-based system requirements are specified in PlantUML files using the *.puml* file extension. An example of model-based system requirement written in PlantUML can be found in Figure A.7. The use of *PlantUML* in *T-reqs* has been improved in Iteration III by embedding the *PlantUML* code blocks within *markdown* files instead of writing them in *.puml* files. This allows the generation of diagram URLs in the same *markdown* file. The template for embedding *PlantUML/DOT* code can be found in Figure A.2. As with the basic version of *T-reqs*, sample textual system requirements are specified in *markdown* files. The template for specifying textual system requirements can be found in A.1. Examples of textual system requirements written in *markdown* format can be found in Figure A.11.

S1.2	A python script is written to generate the URL of the actual UML diagram from the PlantUML code and to write it to a markdown file. The generated URLs and trace-links for the PlantUML code in Figure A.7 can be seen in Figure A.8. In the same figure, the trace-links can be seen in lines 1-3 and the URL is shown on line 5. When the generated *markdown* file is opened on *GitLab*, it is rendered as a UML diagram along with the trace-links. Figure A.9 shows how the URL and trace-links in Figure A.8 look like when opened on *GitLab*. For local previewing, an extension is implemented for Visual Studio Code text editor. A local preview of the PlantUML code in Figure A.7 can be seen in Figure A.10.
S1.3	Trace-ability information is added to the sample *T-reqs* artifacts within the comment sections of the respective languages (*markdown*, *PlantUML* or *Python*). Then, a *python script* is used to generate trace-links from the specified trace-ability information. Line 2 of the PlantUML code in Figure A.7 shows a trace-ability information indicating the system requirement of ID *SR0003* is traced to the quality requirement of ID *QR0002* and the test cases of IDs *TC0001* and *TC0005*.
S1.4	Sample quality requirements are written in *markdown* format. The template for writing quality requirements can be found in Figure A.3. Examples of quality requirements written in *markdown* can be seen in Figure A.12. Sample manual test cases are written in *markdown* format. The template for writing manual test case can be found in Figure A.4. An example of manual test case can be found in Figure A.13. Sample automated test cases are written as *python* scripts. The template for writing test cases can be found in Figure A.5. An example of automated test case can be found in Figure A.14. Sample user acceptance tests are written in *cucumber* feature files. The template for writing acceptance tests can be seen in Figure A.6. An example of user acceptance test can be found in Figure A.15.

Table 5.3: Implementation details of suggested solutions in Iteration I

5.1.5 Evaluation

During this iteration, a workshop was conducted together with two developers. During the workshop the *T-reqs* tool was demonstrated to them. Then, they were provided with a guideline which contains instructions on how to setup and use the *T-reqs* tool. A set of requirement related tasks which should be executed by the participants were also included in the guideline. In general, the requirement related tasks are related to writing sample model-based system requirements, quality requirements, acceptance tests, test cases and textual system requirements and reviewing them on *GitLab*. During the workshop, observation was made by the author on how they executed the tasks and what challenges they faced. After the workshop, they were asked to fill out a survey to get more detailed views of the participants. The survey contains questions that assess the functionality and usability of the tool. The functionality assessment focuses on the different features of the *T-reqs* tool.

The features were tried out by executing the tasks provided in the workshop guideline. The features of the tool were evaluated based on a *5-point likert scale*. The *likert-scale items* in Figure 5.1 correspond to the tasks the participants were asked to execute in the workshop. The well-known system usability scale (SUS) [25] is used for assessing the usability of the tool. The participants were also asked open-ended questions regarding large-scale agile RE challenges and the *T-reqs* tool in general. The functionality assessment results of *T-reqs* are summarized in Figure 5.1. The corresponding survey questions for this section can be found in Figure A.19. The usability assessment results of *T-reqs* are also summarized in Figures 5.2 and 5.3. The corresponding survey questions can be found in Figure A.21. The developers have also responded to some open-ended questions regarding *T-reqs* limitations and RE challenges in large-scale agile system development. Regarding *T-reqs* limitations, it has been mentioned that the tool does not keep requirements change history.
Regarding large-scale agile RE challenges, one of the developers mentioned that managing dependencies between teams as a challenge in large-scale agile environment. For example, propagating system requirement changes between inter-dependent teams is one of such dependency challenges.

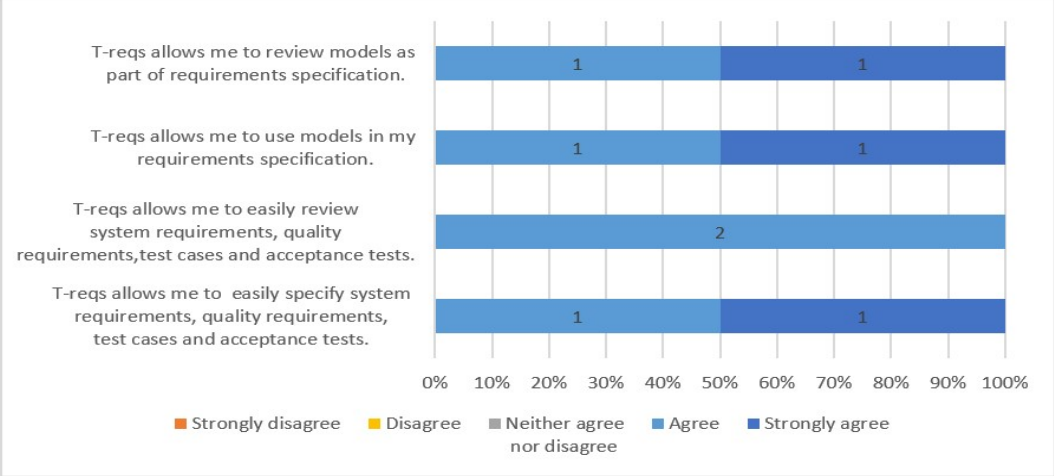


Figure 5.1: Survey result for T-reqs functionality assessment in Iteration I

From Figure 5.1, it can be seen that all respondents either agreed or strongly agreed. Therefore, it can be concluded that the functionalities of the *T-reqs* tool which are indicated in the vertical axis of the same figure are acceptable solutions.

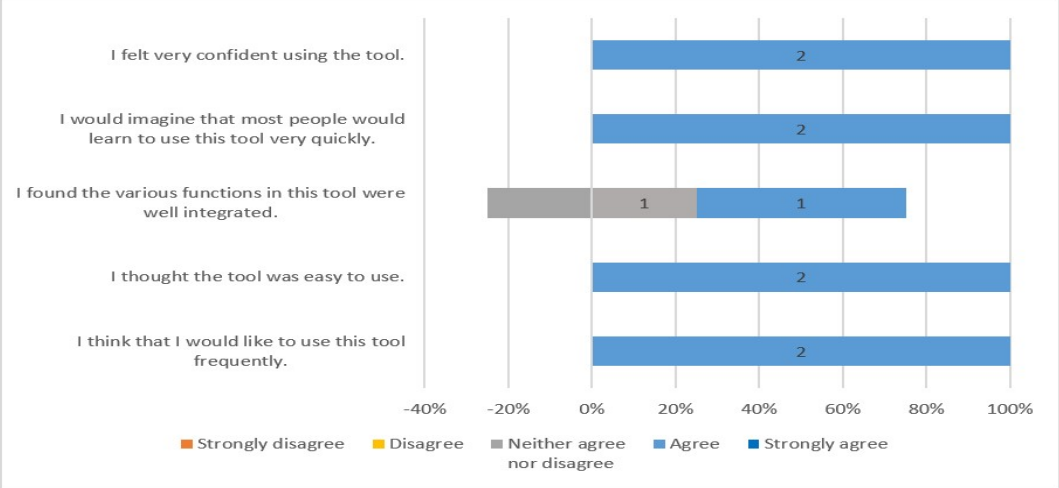


Figure 5.2: Survey results for usability assessment of T-reqs in Iteration I-positive questions

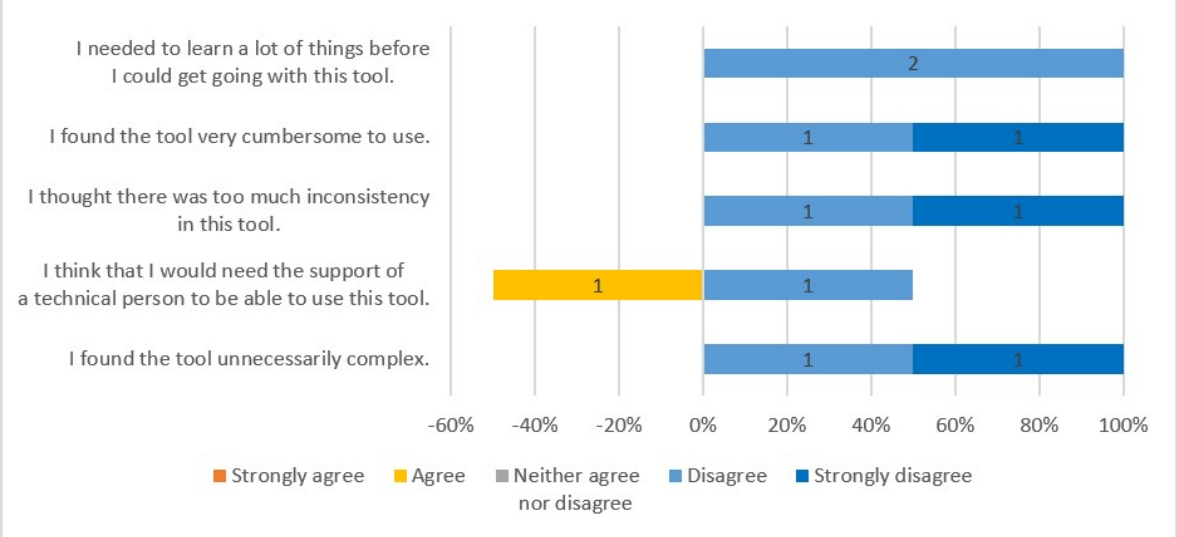


Figure 5.3: Survey results for usability assessment of *T-reqs* in Iteration I-negative questions

Based on the usability assessment data in Figures 5.2 and 5.3, the system usability scale (SUS) score is 75. According to research an SUS score above 68 is considered to be above average and any score below that is considered below average. Therefore, it can be concluded that *T-reqs* has a good usability score.

5.2 Iteration II

5.2.1 Awareness of problem

In this iteration, problem *P1.5* in Table 5.1 and the challenges identified during the evaluation of *Iteration I* will be addressed. A summary of the identified challenges can be found in Table 5.1.

Problem ID	**Problem description**
P2.1	In a large-scale agile context, high-level system requirements are usually decomposed into low-level system requirements. These low-level system requirements are usually assigned to different but related software components that are implemented in different repositories. In such situations, requirement changes in one repository can affect the related requirements which reside in different repositories. However, it is difficult to propagate requirement changes to multiple teams and multiple repositories (projects).
P2.2	It is important to keep a record of requirement changes; what changed, when, why and who changed it. In traditional software development, details about requirement changes are usually stored in databases. However, it is difficult to keeping track of requirement change history when the requirements are stored in repositories.
P2.3	System requirements and other related artifacts should be assigned IDs so that it is possible to uniquely identify them. In *T-reqs*, the artifacts are stored in files under *git repositories* instead of in databases. In such a situation, it is difficult to synchronize the generation of artifact IDs by multiple teams working in parallel.

Table 5.4: Problems to be solved in Iteration II

5.2.2 Solution suggestion

A focus group was conducted together with the supervisor, two people from academia and one industry representative from *Ericsson AB* in order to investigate viable solutions to the problems in Table 5.4. During the discussion, the identified problems were explained to them. Then, they were asked to propose possible solutions for each of the problems. Their opinions were documented using notes. Based on their feedback, different solutions were suggested. The suggested solutions can be found in Table 5.5.

Solution ID	Problem ID	Suggested solution
S2.1	P2.1	1. Using *git submodules* to propagate requirement and other artifact changes to multiple repositories. The shared requirements have to be placed in a separate repository and then this repository has to be included as a *git submodule* to the primary repositories that share the requirements. 2. Using third party tools such as the *Repo* tool from *Google* to share requirements and related artifacts between multiple repositories.
S2.2	P2.2	1. Using built-in *git* features such as *git blame* and *git log* to get the change history of artifacts. 2. Writing scripts to get the change history of artifacts by mining the *git* repositories where the artifacts are stored.
S2.3	P2.3	1. Implementing an algorithm that generates random unique IDs 2. Storing the artifact IDs in a centralized database which can be accessed concurrently by all the teams working on a certain project.

Table 5.5: Suggested solutions in Iteration II

5.2.3 Solution validation

To validate the solution for propagating artifact changes to multiple repositories, some third party tools and built-in *git features* have been tried out. For example, the *Google Repo* tool [1] and another similar tool called *git-subrepo* [2] have been used to propagate artifact changes to multiple repositories. The *git submodules* and *git subtree* features have also been tried out to share requirement and other artifact changes between multiple repositories. Finally, the *git submodule* was selected as a valid solution. Because it was less complex than the rest of the options and it effectively propagated the artifact changes between multiple repositories.
Regarding the tracking of requirement change history, *git blame* and *git log* have been selected over *writing repository mining scripts*. *Git blame* has been selected as a primary solution because it provides a simple line-based change history. Since multiple requirements can be written in a single file in *T-reqs*, a line-based change history solution can be a better option. However, *Git blame* displays only the latest change history. To get older change history of artifacts, *git log* can be used. There was no need to write scripts for mining the repositories because the problem can be easily solved using built-in *git* features.
Regarding unique artifact ID generation, storing the IDs in a centralized cloud-based database is the solution selected over implementing an algorithm that generates unique random IDs. Because the former one allows the users to customize the

[1] https://gerrit.googlesource.com/git-repo
[2] https://github.com/ingydotnet/git-subrepo

patterns of the IDs to be generated.
Finally, the selected solutions were communicated with the supervisor and the other people who were involved in the solution suggestion focus group. And the selected solutions were accepted by all of the participants.

5.2.4 Implementation

The implementation of the solutions to the problems listed in Table 5.4 are described in Table 5.6.

Solution ID	Implementation
S2.1	The requirements and other artifacts that are shared between multiple repositories are stored in a separate repository. Then, this repository is added to the primary repositories as a *git submodule*. Figure A.16 shows how the *shared* repository which contains the shared artifacts between *sample_repo1* and *sample_repo2* can be added as a *git submodule*. In the same figure, the *shared* repository can be seen at the top of the figure hosted on *GitLab*. The repositories *sample_repo1* and *sample_repo2* can be seen on the editor windows to the left and right sides respectively. The *git submodule add* command is used to add the shared repository as a *git submodule*. Changes to the shared artifacts can be pushed to the remote repository by running the *git push* command from the root of the submodule. To pull any remote changes on the shared artifacts, the *git* commands *git submodule update --remote --merge* and *git submodule update --remote --rebase* can be used. Validation of the artifacts and the traces between them is done using python scripts.
S2.2	The system requirements and other related artifacts are written in files. This makes it easy to use a powerful *git* feature called *git blame* to find out who changed a certain line of text for the last time and when that line was changed. Figure 6.18 shows an example of using the *git blame* command. In the figure, the author of the changes, the times of change and the commit hashes have been displayed for each line of the requirements file.
S2.3	The Amazon Web Services Relation Database (AWS RDS) is used to host the database which stores the IDs of *T-reqs* artifacts. Sample database tables for storing system requirement and quality requirement IDs are shown in Figures A.17 (a) and A.17 (b) respectively. The tables for storing other artifacts' IDs also follow a similar structure.

Table 5.6: Implementation details of suggested solutions in Iteration II

5.2.5 Evaluation

The second iteration was evaluated using one workshop which is held at *Ericsson AB* where developers, testers and architects participated to evaluate the implemented solutions in Iterations I & II. In total 10 people were participated in the workshop. The participants were provided with a guideline which contains a list of tasks to be performed in *T-reqs*. The tasks relate to writing and reviewing *T-reqs* artifacts(textual and model-based system requirements, automated and manual test cases, quality requirements and acceptance tests), propagating requirement changes to multiple repositories, tracking the change history of artifacts, customizing the IDs and file name patterns of *T-reqs* artifacts and tracing a specific part of a *PlantUML* diagram to certain test cases and quality requirements. These tasks correspond to the *likert-scale items* in Figure 5.7. Observation is used as the main data collection method during the workshop. They were observed how they performed each task and what challenges they faced. After the workshop, a web-based questionnaire was also sent out to the participants to get more detailed views of the participants. Two experts managed to fill out the questionnaire. Although the response rate is not significant, the two experts gave very detailed feedback on what they have learned from the workshop.
From the observation and the survey results, some problems or limitations of the improved *T-reqs* tool and few large-scale agile RE challenges were identified.
Problems with *T-reqs*: Difficulty in knowing the history of system requirements and other issues that moved between files, difficult to know the history of generated IDs, difficulty in customizing artifacts such as system requirements, test cases etc., lack of traces to a specific UML diagram element, complexity created from saving *PlantUML code* and its diagrammatic representation in different files, dependency on *GitLab* features, and difficulty in setting up and using *T-reqs*.
Large-scale agile RE challenges: The conflict between self empowered teams and the need for a central control of requirement documentation was mentioned as a challenge. Another challenge raised during the workshop is updating the traces to different artifacts and keeping the quality of those traces consistent.
The usability and functionality of the *T-reqs* tool were also assessed using a likert scale based questionnaire. Figure 5.4 shows the results of functional assessment of the *T-reqs* in Iteration II. And the results of the usability assessment can be seen in Figures 5.5 and 5.6. The corresponding survey questions can be found in Figures A.20, A.22 and A.21.

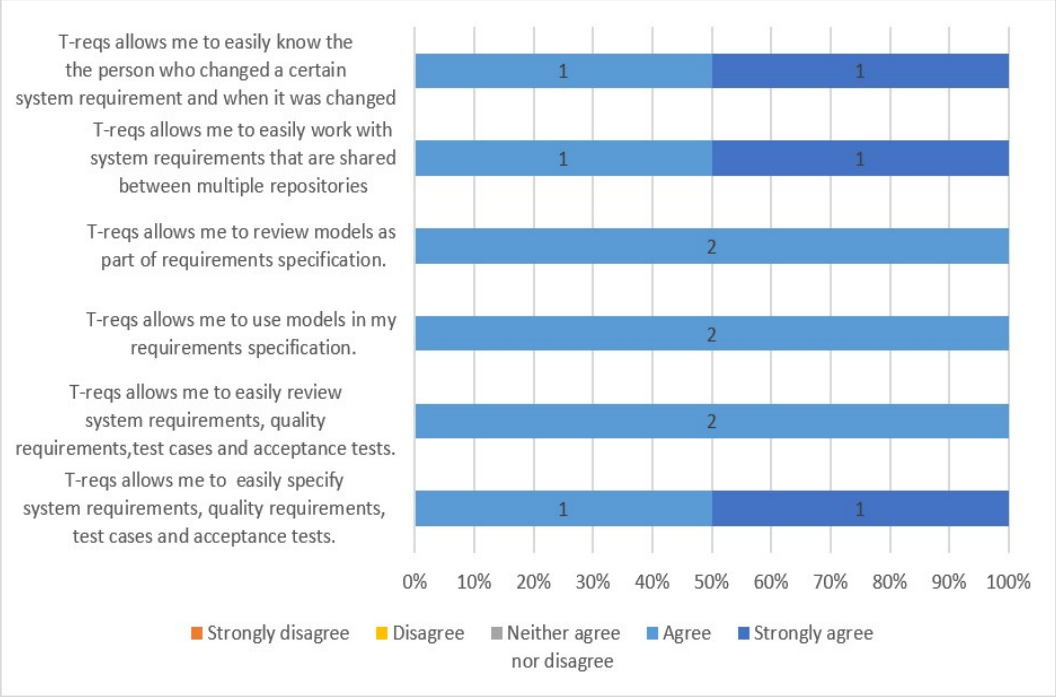


Figure 5.4: Survey results for *T-reqs* functionality assessment in Iteration II

Based on the data in Figure 5.4, it can be seen that all respondents strongly agreed or agreed. Therefore, it can be concluded that the functionalities of the *T-reqs* tool are acceptable solutions.

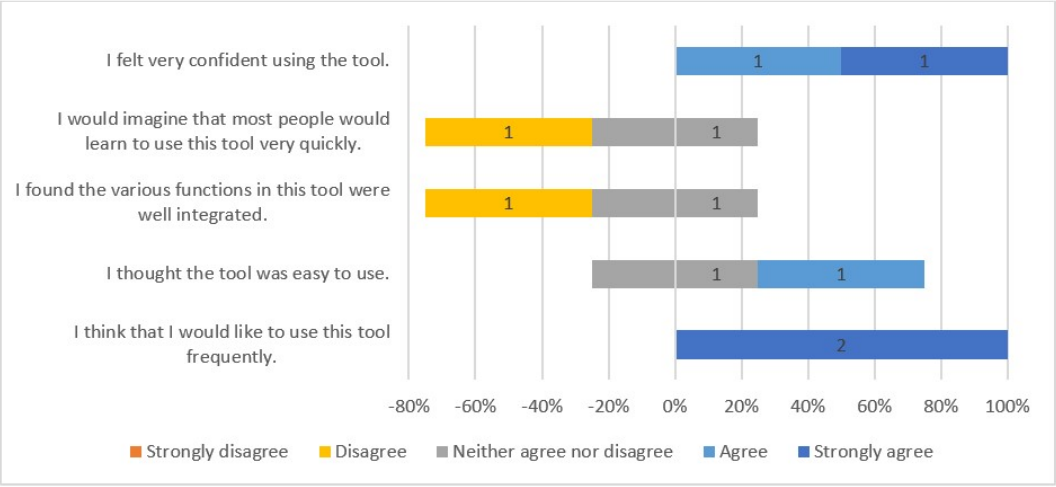


Figure 5.5: Survey results for usability assessment of *T-reqs* in Iteration II-positive questions

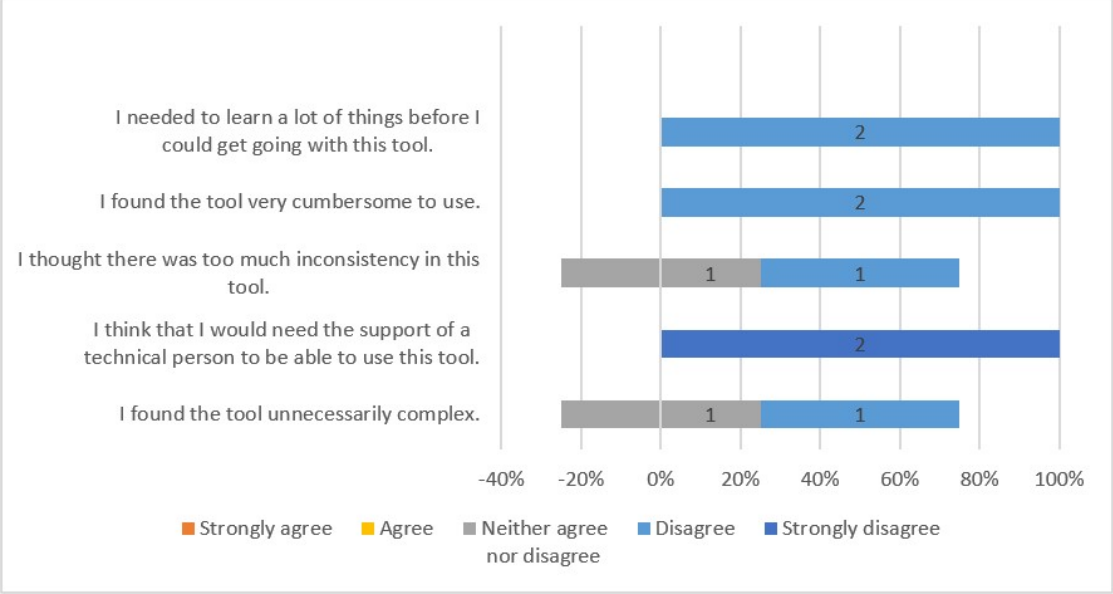


Figure 5.6: Survey results for usability assessment of *T-reqs* in Iteration II-negative questions

Based on the system usability data in Figures 5.5 and 5.6, the SUS score is 70. Therefore, it can be concluded that the *T-reqs* tool has above average usability.

5.3 Iteration III

5.3.1 Awareness of problem

The *T-reqs* problems (limitations) and large-scale agile RE challenges identified during the evaluation of Iteration II were prioritized to be addressed in this iteration. The *T-reqs* problems that will be addressed in this iteration are presented in Table 5.7.

Problem ID	Problem description
P3.1	The ID and filename patterns of *T-reqs* artifacts are difficult to customize.
P3.2	It is difficult to know the history of generated IDs.
P3.3	It is difficult or not possible to create traces to specific parts of a PlantUML diagram.
P3.4	It is difficult to know the history of system requirements that moved between files.
P3.5	Storing the actual PlantUML code and its diagrammatic representation in different files introduces complexity.
P3.6	Difficulty in setting up and using the *T-reqs* tool.

Table 5.7: *T-reqs* problems (limitations) to be addressed in Iteration III

Problem ID	Problem description
P3.7	It is difficult to update traces in a model.
P3.8	The need for a central control of requirement documentation conflicts with the principle of self-empowered teams.

Table 5.8: Large-scale agile RE challenges to be addressed in Iteration III

5.3.2 Solution suggestion

Solutions were proposed to the *T-reqs* problems in Table 5.7 and the RE challenges in Table 5.8 mainly based on the researcher's creativity and based on the knowledge from the already implemented solutions in Iterations I & II. The suggested solutions are presented in Tables 5.9 and 5.10.

Problem ID	Solution ID	Solution description
P3.1	S3.1	Writing ID and filename patterns to a configuration file. In this way, it is possible to customize the artifact IDs and file name patterns within the configuration file. Then, python scripts should be written to read the details written in the configuration file.
P3.2	S3.2	Storing the details of the user who generates artifact IDs in a database together with the IDs. This makes it possible to attach each generated ID to the user who generated it.
P3.3	S3.3	Adding trace-ability information to a specific part of the PlantUML diagram using *PlantUML notes*.
P3.4	S3.4	Using the *git blame options -M and -C* to know change history of artifacts that moved between or within files.
P3.5	S3.5	Writing PlantUML/DOT code in a *markdown* file using *fenced-code blocks*. Then, generating the diagram URL from the *PlantUML* code and writing it to the same markdown file. This makes the *PlantUML* previewer extension which was implemented for visual studio code not important to pursue. Instead, it was suggested to rely on other third party *markdown* previewer extensions.
P3.6	S3.6	Generating templates in the working repository so that the teams can refer to them while writing requirement related artifacts in *T-reqs*. This makes it easy to write *T-reqs* artifacts by following the respective templates. Another additional solution is improving the documentation of the *T-reqs* tool so that *T-reqs* users can easily understand how the tool works.

Table 5.9: Solutions suggested for *T-reqs* problems (limitations) in Iteration II

The RE challenges in Table 5.8 can be addressed using some of the solutions which were already implemented in *Iteration I*. These solutions are provided in Table 5.10.

Problem ID	Solution ID	Solution description
P3.7	S3.7	Writing scripts to automatically update the traces based on *git pre-commit hooks*. In this way, the traces will be updated whenever *git commits* are performed on the file which contains such traces.
P3.8	S3.8	Although it is difficult to decide on how much of the requirements documentation should be done by the development teams, storing the requirements together with the code in *git repositories* provides an easy access to the teams.

Table 5.10: Solutions suggested for large-scale agile RE challenges in Iteration II

5.3.3 Solution validation

The solutions in Table 5.9 were validated mainly by writing simple prototypes and observing the results of those prototypes. For example, some ID and file name patterns were hard coded into a configuration file and then read by a script. This indicated that the artifact details can be stored in a configuration file and used in *T-reqs* scripts. Tutorials were also taken on the languages and tools related to the suggested solutions. This provided a deeper understanding of the suggested solutions. For example, PlantUML examples with PlantUML notes were tried out on the online PlantUML server to see how the PlantUML notes work. This provided a deeper understanding on how PlantUML notes can be used for tracing specific parts of the PlantUML diagram to different test cases and quality requirements.
Finally, the solutions in Tables 5.9 and 5.10 were validated by discussing them with the supervisor and one of the experts who participated in *Workshop II* (Person-D).

5.3.4 Implementation

The implementation of the suggested solutions in Table 5.9 are presented in Table 5.11 below.

Solution ID	Implementation
S3.1	A python script is written to write the ID and filename patterns of artifacts to a configuration file. The details are stored in a JSON format within the configuration file. Then, *python scripts* are written to read those details from the configuration file so that *T-reqs* can work with.
S3.2	The email address of the user who generates IDs is mined from the *git configuration* with the help of the *gitpython* module and stored in a database. The stored email address is then used to check which IDs have been generated by which users.

S3.3	Sample trace-ability information has been specified in PlantUML notes. PlantUML notes are used to add additional information to parts of a PlantUML diagram. Then, a python script is used to generate trace-links and write them below the line where the corresponding trace-ability information is specified in the file.
S3.4	The *git blame* -M and -C options are used to detect lines that moved to different lines within the same file and that moved between files respectively.
S3.5	With the help of *git pre-commit* hooks and the *gitpython* module, the diagram URL is generated and written to the same file below the respective PlantUML/DOT code block automatically while committing the markdown file which contains a PlantUML/DOT code.
S3.6	A python script is written to generate templates in the working repository. This way it is possible to easily refer to the templates while writing *T-reqs* artifacts. The *T-reqs* tool and its features have been also documented in detail.

Table 5.11: Implementation details in Iteration III

The implementations of the large-scale agile RE solutions in Table 5.10 are described in Table 5.12. These solutions have already been implemented as part of the implementation in *Iteration I*.

Solution ID	**Implementation**
S3.7	A *python script* is written to check the existence of trace-links within a *markdown* file which contains *PlantUML/DOT* code whenever any changes in the file are *git committed* (with the help of *git pre-commit hooks*). If such trace-links exist in the file, they are replaced by newly generated trace-links from the latest version of the file.
S3.8	Sample *T-reqs* artifacts are written in files and stored in *git repositories* together with the actual implementation. This allowed the participants in the study to write new requirements and to propose changes to existing ones.

Table 5.12: Implementation of large-scale agile RE challenges in Iteration III

5.3.5 Evaluation

The third iteration was evaluated with one focus group and one workshop.

The focus group was conducted with three experts from industry and academia. During the focus group session, the improved *T-reqs* tool was demonstrated to them. Then, the following questions were asked one after the other to the group:

1. What do you think of the model-support solution in *T-reqs*?
2. What do you think of the multi-repository solution in *T-reqs*?
3. What do you think of the system requirement change history solution in *T-reqs*?
4. What other tool support related large scale agile RE challenges are common in industry?
5. What do you think of the improved *T-reqs* tool in general?

The participants provided answers to the above questions. A number of questions were also asked by the participants. The opinions of the participants were documented using notes.

Some of the points they suggested for improvement are: allowing users to specify their requirement information model; defining artifacts and their relationships; checking for whether IDs follow naming conventions; generating diagram URLs before committing the changes automatically; good idea to no longer pursue the local editor extension for visual studio code editor, but to rely on *markdown* previewers instead (because the tool-chain adds complexity); implementing a *treqshistory* command that replaces iterative usage of the *git blame* command in order to get older history; analysing the requirement types that are changed most often; creating latest changes automatically; providing guideline to help scale long *markdown* documents.

The workshop was conducted with three master's students from the university of *Gothenburg*, Sweden. Two of the students study the software engineering & management program and one of them studies the computer science program. The *T-reqs* tool was demonstrated to them first. Then, they used the tool to execute requirement related tasks for approximately two hours. The tasks relate to writing and reviewing *T-reqs* artifacts (textual and model-based system requirements, automated and manual test cases, quality requirements and acceptance tests), propagating requirement changes to multiple repositories, tracking the change history of artifacts, customizing the IDs and file name patterns of *T-reqs* artifacts and tracing a specific part of a *PlantUML* diagram to certain test cases and quality requirements. These tasks correspond to the *likert-scale items* in Figure 5.7. Finally, they were provided with a web-based questionnaire which contains both closed and open-ended questions. The questionnaire evaluates both the functionalities and usability of *T-reqs*.

The functionality assessment of the *T-reqs* tool in *Iteration III* can be seen in Figure 5.7. And the corresponding usability assessment results can be found in Figures 5.8 and 5.9. The corresponding survey questions can be found in Figures A.20, A.22 and A.21.

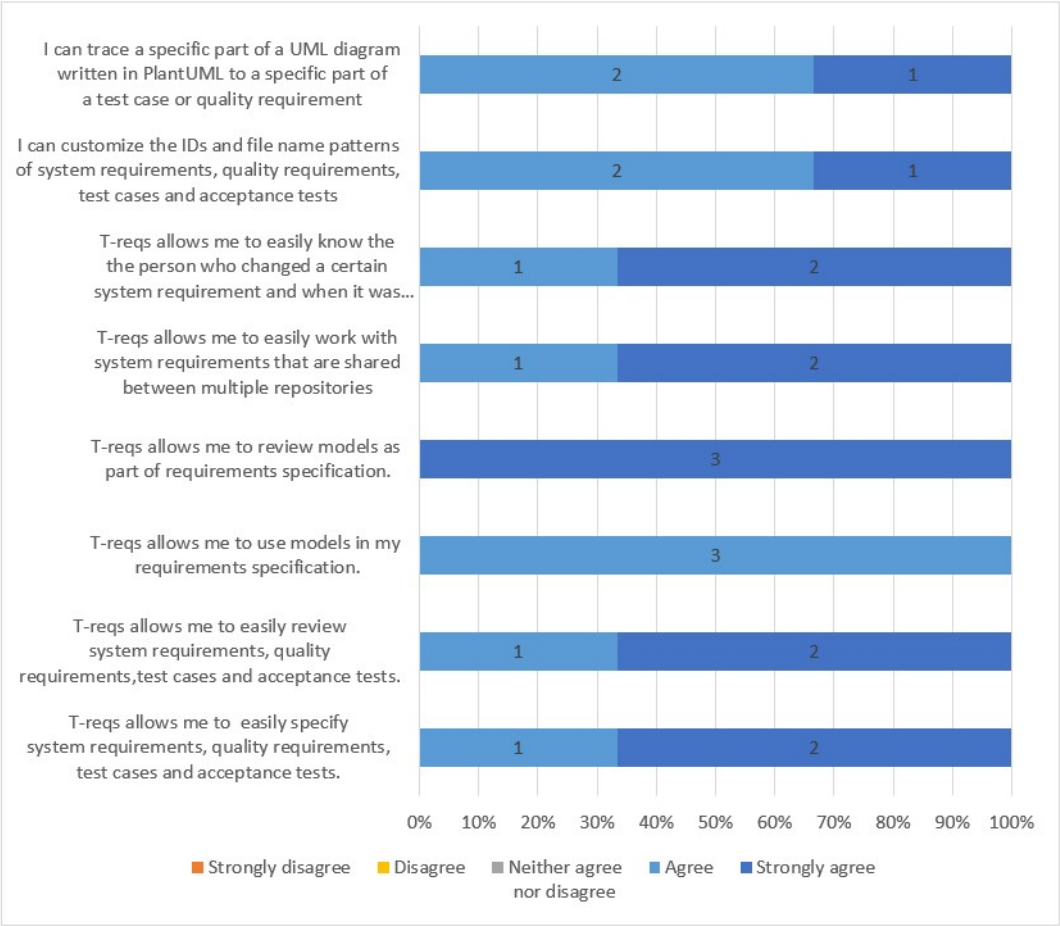


Figure 5.7: Survey results for *T-reqs* functionality assessment in Iteration III

According to Figure 5.7, we can see that all respondents strongly agreed or agreed. Therefore, it can be concluded that the *T-reqs* functionalities indicated on the vertical axis are acceptable solutions.

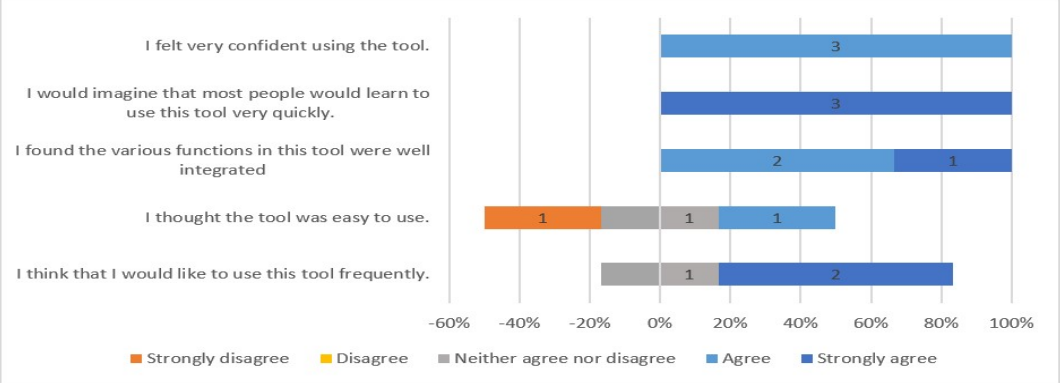


Figure 5.8: Survey results for usability assessment of *T-reqs* in Iteration III-positive questions

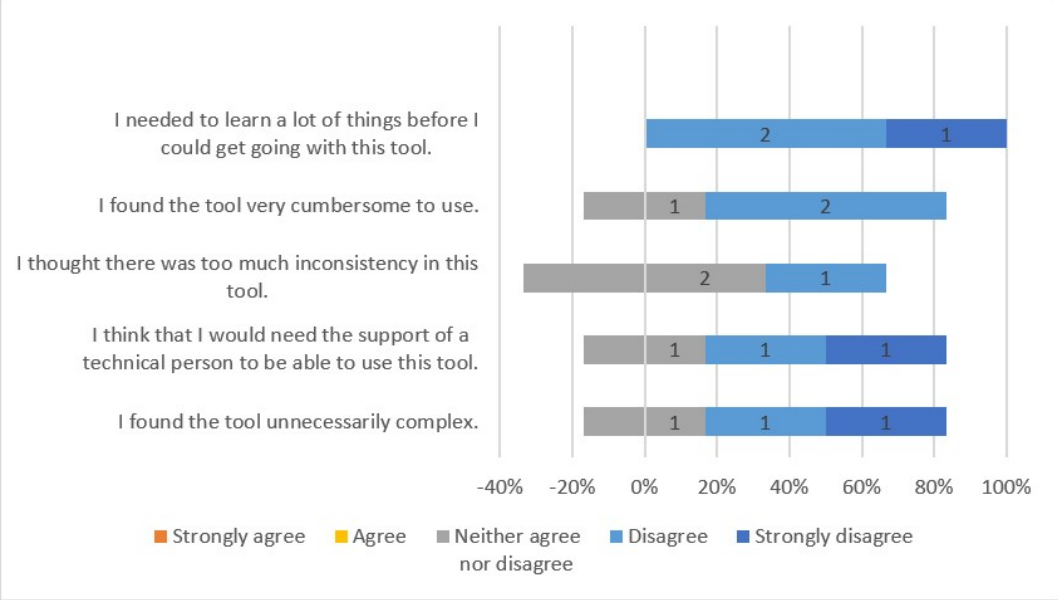


Figure 5.9: Survey results for usability assessment of *T-reqs* in Iteration III-negative questions

Based on the usability assessment data in Figures 5.5 and 5.6, the SUS score is 74. Therefore, it can be concluded that the usability of the *T-reqs* tool is above average.

6

The improved T-reqs tool

This chapter provides a detailed description of the improved *T-reqs* tool.

6.1 Overview

The improved *T-reqs* tool [1] refers to the *T-reqs* version that is improved by this study. The improved version is different from the basic one mainly in its functionalities and the types of artifacts it manages.

In addition to system requirements and test cases, the improved *T-reqs* can manage quality requirements and acceptance tests. System requirements, quality requirements and manual test cases are written in *markdown* files while automated tests are written in *python* scripts. Acceptance tests are written in *cucumber* feature files. Model-based system requirements (requirements that are specified in the form of models) are written using *PlantUML* and the DOT graph description language. UML diagrams are written in *PlantUML* and the non-UML diagrams are written using the DOT language. The *T-reqs* artifacts are stored in a directory under the project repository which contains the actual implementation (source code) of the project. This project refers to any project that uses *T-reqs* to manage system requirements and other requirement-related artifacts.

The improved *T-reqs* consists of four commands, namely *treqsconfig*, *generateid*, *generatedid*, *generatepreview* and *treqs*. Once the *T-reqs* tool is successfully installed locally, these commands should be run using a command line from the path of the project's repository. The *treqsconfig* command is used to generate a configuration file that stores some details about the project repository, *T-reqs* artifacts and database connectivity. Those details are explained in detail in section 6.2.1. The *treqsconfig* command also generates template files that specify the syntax for writing system requirements, quality requirements, test cases and acceptance tests and writes those files to the project repository. The *generateid* command generates IDs for system requirements, quality requirements, tests and acceptance tests and stores those IDs in a cloud-based database. The *generatedid* command reports history of IDs generated by the user issuing the command. The *treqs* command validates the *T-reqs* artifacts. The *treqs* command is available in the basic version of *T-reqs* as well. But in the improved version it also validates quality requirements and acceptance tests. Additionally, it validates the artifact IDs against the specified ID patterns and reports any unused IDs.

[1] https://gitlab.com/mebrahtom/treqs

Furthermore, *T-reqs* makes use of built-in *git* commands to perform some tasks. Thoses *git* commands are *git submodule*, *git blame* and *git log*. The *git submodule* command is used to propagate system requirement changes between multiple repositories while the *git blame* and *git log* commands are used to keep track of the change history of system requirements and other related artifacts.

6.2 T-reqs Features

6.2.1 Configuration and template files generation

This feature is accessed using the *treqsconfig* command. This command prompts the user to enter some details about the project repository, the different *T-reqs* artifacts and the database connectivity. Figure 6.1 shows the different parameters the user is prompted to enter while running the *treqsconfig* command. Most of these parameters have default values specified in brackets. Those default values can be overwritten by entering new values for the respective parameters.

```
mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
$ treqsconfig
The URL of your repository management service(https://gitlab.com):
GitLab group name:treqschalmers
Parent directory of T-reqs artifacts(treqs):
File name pattern for system requirements(SR_.*?\.md):
ID prefix for system requirements(SR):
File name pattern for quality requirements(QR_.*?\.md):
ID prefix for quality requirements(QR):
File name pattern for user stories(US_.*?\.md):
ID prefix for user stories(US):
File name pattern for test cases(TC_.*?(\.py|\.md)$):
ID prefix for test cases (TC):
File name pattern for acceptance tests(AT_.*?\.feature):
ID prefix for acceptance tests(AT):
The endpoint of your database instance on AWS:treqsdb.cwel9elimbjd.eu-central-1.rds.amazonaws.com
Database name:treqsdb
Database username:admin
Database password:treqs123
Database port number:3306
Alias name for your AWS KMS customer master key(CMK):treqsdbapi
------------------------------------------------
You have set up your configuration successfully!
Configuration file path: C:\Users\mebri\Desktop\treqs_chalmers\sample_repo1/treqs/.config.json
Template files path: C:\Users\mebri\Desktop\treqs_chalmers\sample_repo1/treqs/templates

mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
```

Figure 6.1: Parameters of the *treqsconfig* command

The *treqsconfig* command's parameters are described below:
The URL of your repository management service: This parameter refers to the URL of the repository management service where the remote repository is hosted. This parameter is used by *T-reqs* to formulate clickable trace-links on a diagram which can then be clicked to open the traced artifacts on *GitLab*. This parameter is only required when working with model-based system requirements.
GitLab group name: This parameter refers to the group name on *GitLab*. As with the URL of the repository management service, the *GitLab group name* is also

used by *T-reqs* to formulate clickable trace-links on a diagram and is only required when working with model-based system requirements. *GitLab* group groups multiple projects into directories and is used to give multiple team members access to several projects at once. File URLs on *GitLab* consist of a namespace which is usually a username or a group name. If this parameter was a GitLab username, it is difficult to generate consistent trace-links for all members of a project because each user will have a unique username. With GitLab group name it is possible to formulate consistent trace links (trace-links which have the same file URLs for all members of the group on GitLab).

Parent directory of *T-reqs* artifacts: This parameter refers to the parent directory of *T-reqs* artifacts, i.e., system requirements, quality requirements, test cases and acceptance tests. The need for such a parent directory is to separate the *T-reqs* managed artifacts from the project's source code files so that *T-reqs* can quickly find the files that store the requirement related artifacts. *T-reqs* starts searching the requirement related artifacts (T-reqs artifacts) from the *parent directory of the T-reqs artifacts* instead of searching all files in the project repository. *T-reqs* uses the file name patterns stored in the configuration file to search the relevant artifacts within the parent directory of *T-reqs* artifacts.

Filename patterns for *T-reqs* artifacts: This parameter refers to the file name patterns of *T-reqs* artifacts. In this case, *T-reqs* artifacts refers to system requirements, quality requirements, test cases, acceptance tests and user stories. The file name patterns are provided as regular expressions and saved in the configuration file. The file name patterns are used by *T-reqs* to filter out only the relevant files while reading *T-reqs* artifacts. Based on the file name of *T-reqs* artifacts, it is possible to know the type of artifact a file contains. This can help *T-reqs* users access the artifacts quickly.

ID prefix for *T-reqs* artifacts: This parameter refers to the initial characters of *T-reqs* artifact IDs. This allows *T-reqs* users to generate IDs that can be tailored to their own styles. When generating IDs, the ID prefixes are appended by a four digit positive integer number in an incremental fashion starting from 0001.

If the *treqsconfig* command successfully runs, the parameters along with their respective values will be stored in a configuration file named *.config.json* which is automatically created under the parent directory of *T-reqs* artifacts. Figure 6.2 shows the configuration file generated by the *treqsconfig* command when it is provided with the parameter values shown in Figure 6.1 above.

Once the configuration file is generated, the parameter values in the configuration file can be easily modified. This allows the *T-reqs* artifacts to be customized.

Database connectivity details: The user is prompted to enter the database connectivity details upon running the *treqsconfig* command. These details are the endpoint of the database instance on AWS, the port number of the database instance, the name of the database where the tables are created and the username and password of the database. The database username and password will be encrypted using the AWS Key Management Service (KMS) customer master key (CMK) and stored in the configuration file. The username and password are read from the configuration file and decrypted using the same technique when *T-reqs* connects to the database.

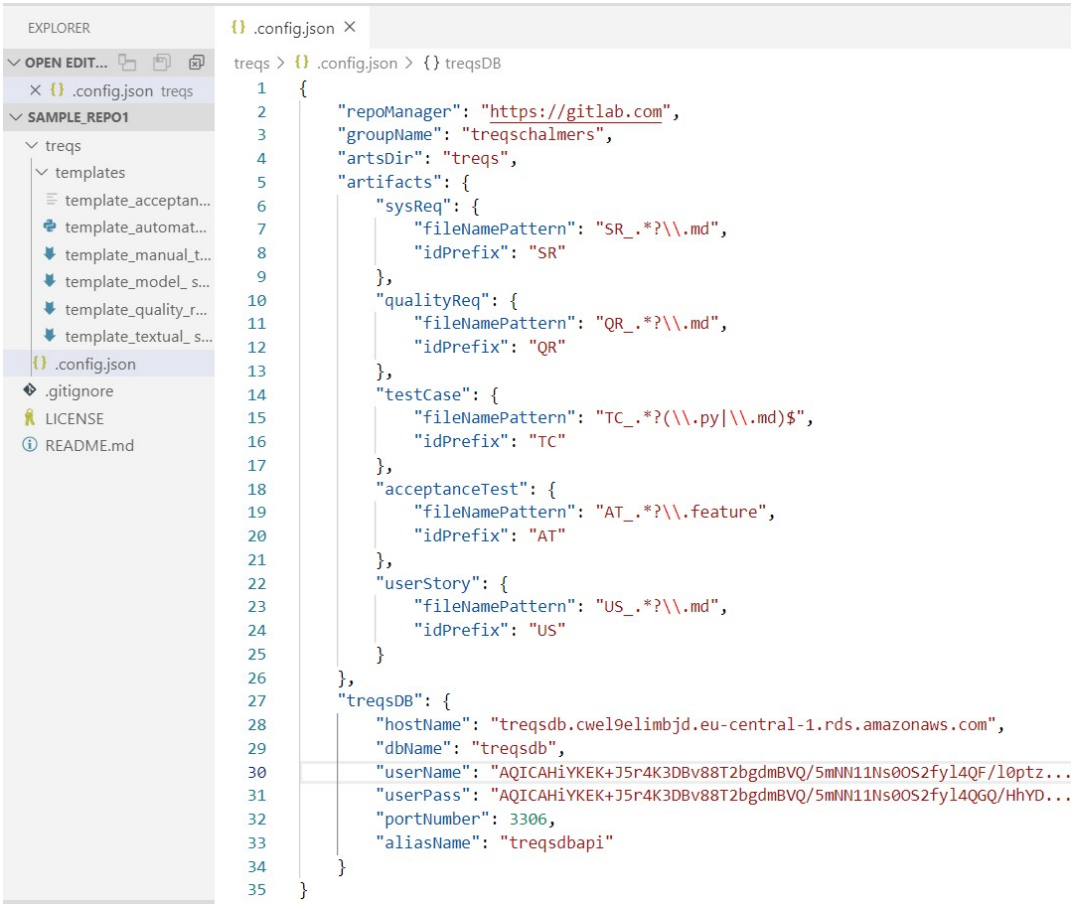


Figure 6.2: Sample repository configuration file generated by T-reqs

6.2.2 ID generation and reporting generated ID history

IDs can be generated for system requirements, user stories, quality requirements, test case and acceptance tests using the *generateid* command. The AWS Relational Database Service (RDs) is used to store the generated IDs. The database and the tables that store the ID details are created when the *generateid* command is run for the first time. The database stores the artifact IDs, the email address of the user (which is automatically mined from *git* given that the user has set an email address for the repository with the *git config* command) who generated the corresponding IDs, and the date when the IDs were generated.
The *generateid* command has two options, namely **-artifact|-a** (which represents the artifact type) and **-number|-n** (which represents the number of IDs to be generated). The **-a** option takes a parameter that represents the artifact type an ID is to be generated for. The possible values of this option are **sr** (system requirements), **qr** (quality requirements), **tc** (test cases), **at** (acceptance tests) and **us** (user stories). The **-n** option takes a positive integer value and it represents the number of IDs to be generated. The defalut value for this option is 1. Figure 6.3 shows how the the *generateid* command can be used to generate system requirement IDs.

```
mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
$ generateid -a sr -n 3
New available IDs:

SR0001
SR0002
SR0003

mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
```

Figure 6.3: Generating system requirement IDs

The *generatedid* command reports the IDs generated on a specific date by the user who is issuing the command. It takes two options; the **-artifact|-a** option (which represents the type of artifact) and the **-date|-d** option (which represents the ID generation date). The possible values of the **-a** option are **sr** (system requriements), **qr** (quality requirements), **tc** (test cases), **at** (acceptance tests) and **us** (user stories). The **-d** option takes a date in the form of YYYY-MM-DD or *all* (to print all IDs). The default value for the **-d** option is *all*. Figure 6.4 shows how the *generatedid* command can be used to get the history of generated IDs.

```
mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
$ generatedid -a sr -d 2019-11-05
You have generated the following IDs on 2019-11-05:
-------------------------
ID        Generation date
-------------------------
SR0001    2019-11-05
SR0002    2019-11-05
SR0003    2019-11-05

mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqs_chalmers/sample_repo1 (dev)
```

Figure 6.4: History of generated system requirement IDs

6.2.3 Model support for requirement specification

System requirements can be specified in textual format or in the form of models. In the improved *T-reqs*, textual requirements are specified in *markdown* language whereas model-based requirements are written in *PlantUML* (for UML diagrams) and in the *DOT* language (for non-UML diagrams). The *PlantUML* code and the *DOT* language are written within *markdown* files using *fenced code blocks*. Traceability information should be added within the PlantUML/DOT code blocks in a comment section. In PlantUML code blocks, the traces are specified using the PlantUML comment syntax (/'...'/). Within DOT code blocks the traces are specified using the DOT language's multi-line comment syntax (/*...*/). Figure 6.5 shows the template for embedding PlantUML/DOT language within markdown files.

template_model_ sys_reqs.md

treqs > templates > template_model_ sys_reqs.md > Markdown Language Features > # Writing

```
1  # Writing model-based system requirements using PlantUML/DOT language
2
3  ## PlantUML diagram
4
5  some text
6
7  ```puml
8  @startuml
9  /'PlantUML code goes here'/
10 note right|left|bottom|top on <element>
11 /'[requirement id=SR0001 test=TC0002,TC0005 quality=QR0010]'/
12 end note
13 @enduml
14 ```
15
16 Some other text
17
18 ## DOT graph
19
20 ```puml
21 @startdot
22 /*DOT language goes here*/
23 digraph <digraph_name>{
24 /*Some code*/
25     subgraph <subgraph_name>{
26     //subgraph code
27     /*[requirement id=SR0002 test=TC0001,TC0003 quality=QR0009]*/
28     //some more code
29   }
30 }
31 @enddot
32 ```
```

Figure 6.5: Template for embedding PlantUML/DOT code in a markdown file

Once the PlantUML code/DOT language has been written within a *markdown* file, the *generatepreview* command can be used to generate a URL of the diagram which will be rendered as a diagram on *GitLab* with the help of the online PlantUML server (which is available at `http://plantuml.com/`). The *generatepreview* command takes the local path to the *markdown* file that contains *PlantUML/DOT* code. Trace-ability information can be added to the PlantUML code using PlantUML notes. PlantUML note is a feature of PlantUML used to add additional information to parts of the diagram. This allows *T-reqs* users to trace specific parts of a PlantUML diagram to different quality requirements and test cases. Figure 6.6 shows a sample PlantUML sequence diagram embedded in a *markdown* file. In the same figure, line 10 shows a trace-ability information which is written using *T-reqs's* trace-ability syntax. The traceability at line 10 indicates that the require-

ment specified as a sequence diagram has an ID of *SR0003* and it is traced to quality requirement *QR0002* and test cases *TC0001* and *TC0005*.

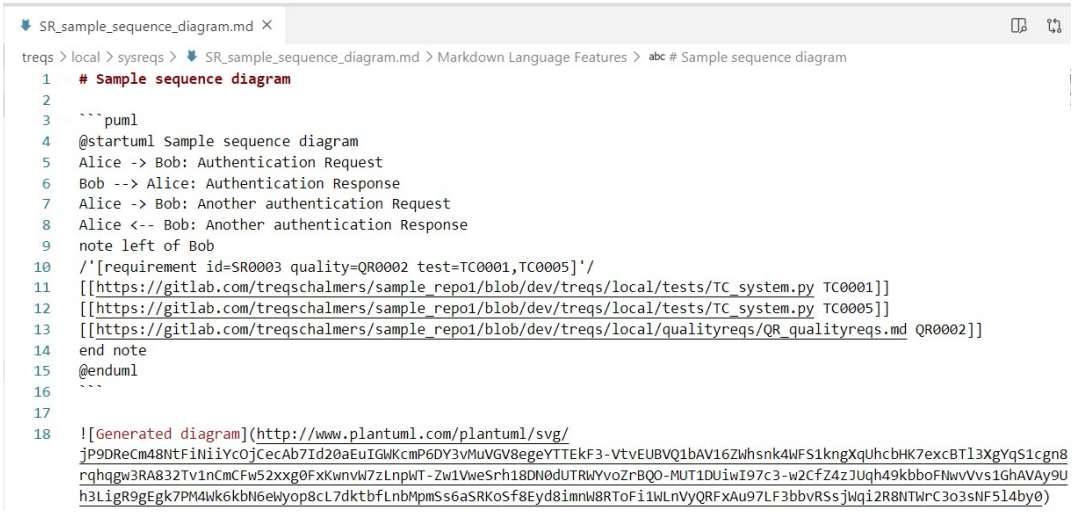
SR_sample_sequence_diagram.md

treqs > local > sysreqs > SR_sample_sequence_diagram.md > Markdown Language Features >

````
1  # Sample sequence diagram
2
3  ```puml
4  @startuml Sample sequence diagram
5  Alice -> Bob: Authentication Request
6  Bob --> Alice: Authentication Response
7  Alice -> Bob: Another authentication Request
8  Alice <-- Bob: Another authentication Response
9  note left of Bob
10 /'[requirement id=SR0003 quality=QR0002 test=TC0001,TC0005]'/
11 end note
12 @enduml
13 ```
````

Figure 6.6: Sample sequence diagram written in PlantUML

The *generatepreview* command generates trace-links based on the provided traceability information and writes them below the line where the corresponding traceability information is specified in the PlantUML code. Then, it generates a URL to the actual diagram by encoding the PlantUML code based on the algorithm used by the online PlantUML server and writes this generated URL to the same file below the respective PlantUML code. The trace-links are displayed on top of the diagram. This makes it possible to navigate to the traced artifacts by clicking on those trace-links.
If there are multiple PlantUML/DOT language code blocks in the same *markdown* file, *T-reqs* generates diagram URLs for each of them below the corresponding PlantUML/DOT code.
Figure 6.7 shows a diagram URL and trace-links which are generated from the file shown in Figure 6.6. Lines 11-13 show the trace-links generated from the traceability syntax in line 10. The URL generated from the PlantUML code in lines 4-12 of Figure 6.6 can be seen in line 18 of Figure 6.7.



SR_sample_sequence_diagram.md

treqs > local > sysreqs > SR_sample_sequence_diagram.md > Markdown Language Features > # Sample sequence diagram

```
1  # Sample sequence diagram
2
3  ```puml
4  @startuml Sample sequence diagram
5  Alice -> Bob: Authentication Request
6  Bob --> Alice: Authentication Response
7  Alice -> Bob: Another authentication Request
8  Alice <-- Bob: Another authentication Response
9  note left of Bob
10 /'[requirement id=SR0003 quality=QR0002 test=TC0001,TC0005]'/
11 [[https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/tests/TC_system.py TC0001]]
12 [[https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/tests/TC_system.py TC0005]]
13 [[https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/qualityreqs/QR_qualityreqs.md QR0002]]
14 end note
15 @enduml
16 ```
17
18 ![Generated diagram](http://www.plantuml.com/plantuml/svg/
   jP9DReCm48NtFiNiiYcOjCecAb7Id20aEuIGWKcmP6DY3vMuVGV8egeYTTEkF3-VtvEUBVQ1bAV16ZWhsnk4WFS1kngXqUhcbHK7excBTl3XgYqS1cgn8
   rqhqgw3RA832Tv1nCmCFw52xxg0FxKwnvW7zLnpWT-Zw1VweSrh18DN0dUTRWYvoZrBQO-MUT1DUiwI97c3-w2CfZ4zJUqh49kbboFNwvVvs1GhAVAy9U
   h3LigR9gEgk7PM4Wk6kbN6eWyop8cL7dktbfLnbMpmSs6aSRKoSf8Eyd8imnW8RToFi1WLnVyQRFxAu97LF3bbvRSsjWqi2R8NTWrC3o3sNF5l4by0)
```

Figure 6.7: Generated trace-links and diagram URL for a sample sequence diagram

When a file that contains model-based system requirements is pushed to *GitLab*, it can be reviewed using *GitLab's* code review feature. Figure 6.8 shows how the code in Figure 6.7 looks like on the *GitLab* code review page. The blocks of code on the left and right sides of the figure represent the old and latest versions of the file respectively. In Line 18 of the same figure, the encoded URL which starts with *http://www.plantuml.com/plantuml/svg* is the URL of the diagram which represents the PlantUML code in the same file. The actual diagram can be viewed by clicking on the 'View file' button on the top right side of the code. Figure 6.9 shows how the code shown in 6.8 looks like when the same file is viewed on *GitLab* by clicking on the 'View file' button.

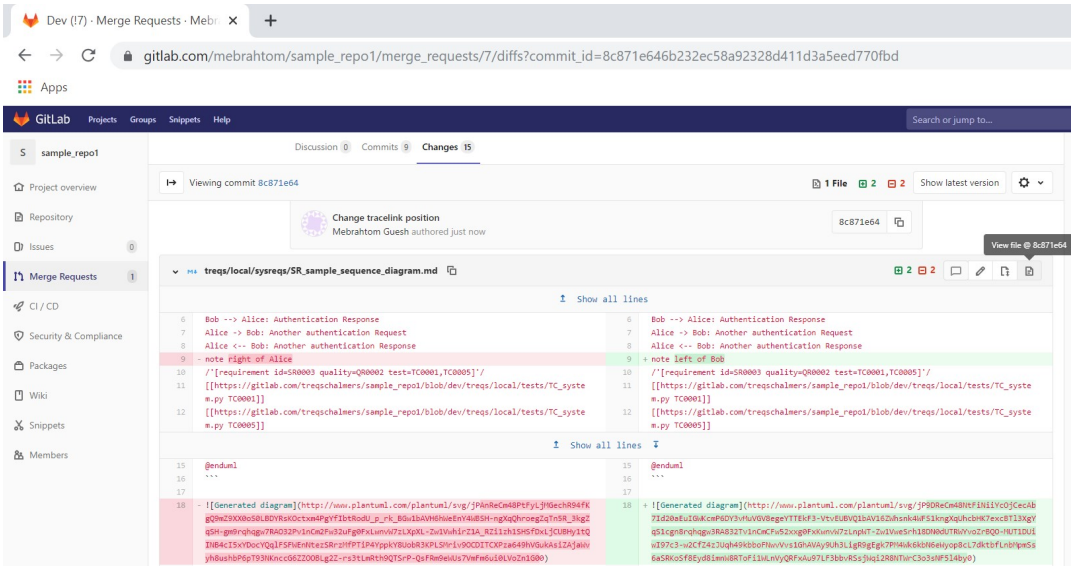


Figure 6.8: Reviewing model-based system requirements on GitLab

Figure 6.9 shows how the code shown in Figure 6.8 looks like when the same file is

viewed on *GitLab* by clicking on the 'View file' button. In the same figure, it can be seen that the corresponding diagram is rendered along with clickable trace-links which are labeled by the IDs of the traced artifacts. Clicking on the trace-links opens the files that contain the traced artifacts. During the code review, the old version of the diagram can be viewed by copying the diagram URL on the left side of Figure 6.8 (Line 18 in the figure) and pasting it on the address bar of any browser. Figure 6.10 shows the diagram rendered by opening this URL in a browser.

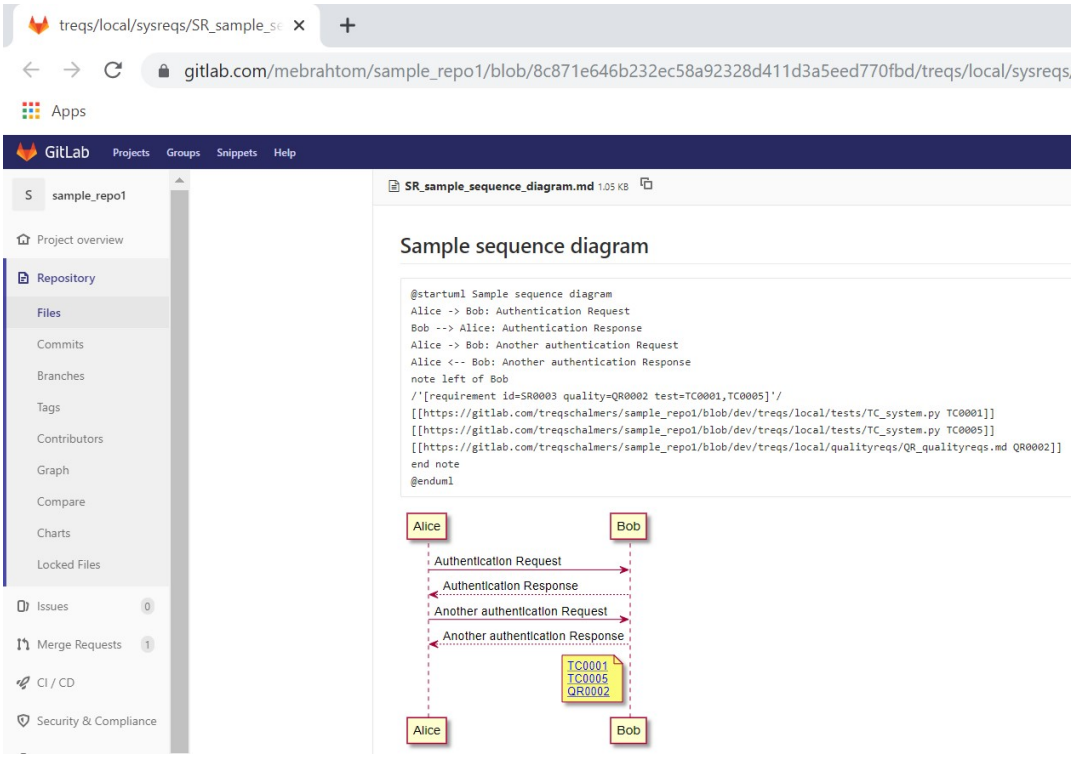


Figure 6.9: Sample plantUML sequence diagram on GitLab

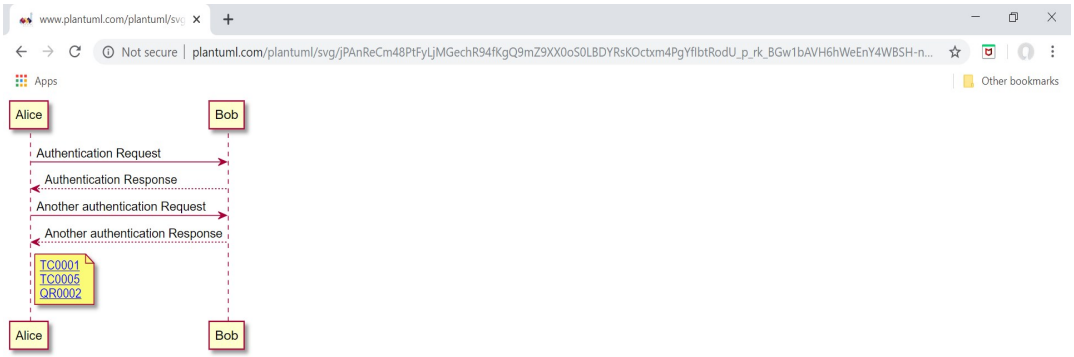


Figure 6.10: Opening a PlantUML diagram URL in a browser

PlantUML is mainly used to write UML diagrams. To write non-UML diagrams, *T-reqs* uses the DOT language using the same syntax used to write PlantUML code.

Figure 6.11 shows a sample context diagram written in the DOT language. Line 15 of this figure shows a traceability information. The tracelink which is generated from this traceability information can be seen on line 16 of the same figure. The DOT language can be surrounded by *@startdot* and *@enddot* or by *@startuml* and *@enduml* tags. When the *generatepreview* command is run on the file shown in Figure 6.11, the URL shown in Figure 6.12 is generated. And when the code shown in Figure 6.12 is pushed to *GitLab*, the actual diagram is rendered as shown in Figure 6.13.

SR_non_UML_diagram.md

treqs > local > sysreqs > SR_non_UML_diagram.md > Markdown Language Features >

````
1  # Sample context diagram
2
3  ```puml
4  @startdot
5  digraph contextDiagram {
6      node[shape="record"]
7      subgraph externalEntities {
8          entity1[label="Customer"];
9          entity2[label="Manager"];
10     }
11     subgraph cluster_System {
12         label="System environment";
13         process1[label="System" shape="ellipse"];
14         graph[style=dotted]
15         /*[requirement id=SR0004 test=TC0001]*/
16     }
17     entity1->process1[label="Request for services"];
18     process1->entity2[label="Reports"];
19 }
20 @enddot
21 ```
````

Figure 6.11: Sample Context diagram written in the DOT language

SR_non_UML_diagram.md

treqs > local > sysreqs > SR_non_UML_diagram.md > Markdown Language Features > # Sample context diagram

```
# Sample context diagram

```puml
@startdot
digraph contextDiagram {
    node[shape="record"]
    subgraph externalEntities {
        entity1[label="Customer"];
        entity2[label="Manager"];
    }
    subgraph cluster_System {
        label="System environment";
        process1[label="System" shape="ellipse"];
        graph[style=dotted]
        /*[requirement id=SR0004 test=TC0001]*/
        TC0001[shape=note,fontcolor = "#0000FF", style = "filled", color = "#A80035", fillcolor="#FBFB77", href="https://gitlab.com/treqschalmers/
        sample_repo1/blob/dev/treqs/local/tests/TC_system.py"]
    }
    entity1->process1[label="Request for services"];
    process1->entity2[label="Reports"];
}
@enddot
```

![Generated diagram](http://www.plantuml.com/plantuml/svg/
NL9DJm8n4BtlhsZApIXT_0X6imH5kNb1RuIGiXtO9jrsxGn4OlplpduWI2_RlddJz_fcap6IZcG29SPkeguAaGTFy4KllEv5D-9uEM3WJaMke9CHiX3D7BH572xQjku1QBNxjMJ9GlODTOBQcW_
c3kz0fV9oHOfb13bul6CSdsWl6clDqV6pt-ftF4D49VlU_wK9q9TTuU3tzaOVCbtowDI5KCEY7zkMh8KtUF0ELiXd7fhHET8UmSP1qLW5dyrTJ6FyBcr4MePOKtsFalJz5OG86KV4zuF5XSgQNTT
WZuGNAuvvZou44KcP8-hwNGgBqKZKqDhwnmOHewaf_iqlRbZgAurS2Pxq-Vfyt38O15XdSc2gC87fJQM-87zF9IAs27cXNQSEIhKPULW6Q4A0xLoOQKCx5gESY7NJjNMKNrCbjX4qw_stQpRCNJJ
l1gT9pdZ8BXLhDakGjnPhXvoF72lHcUJdh6FI0tl9nc3D_Ml-0i0)

Figure 6.12: Generating diagram URL for sample context diagram

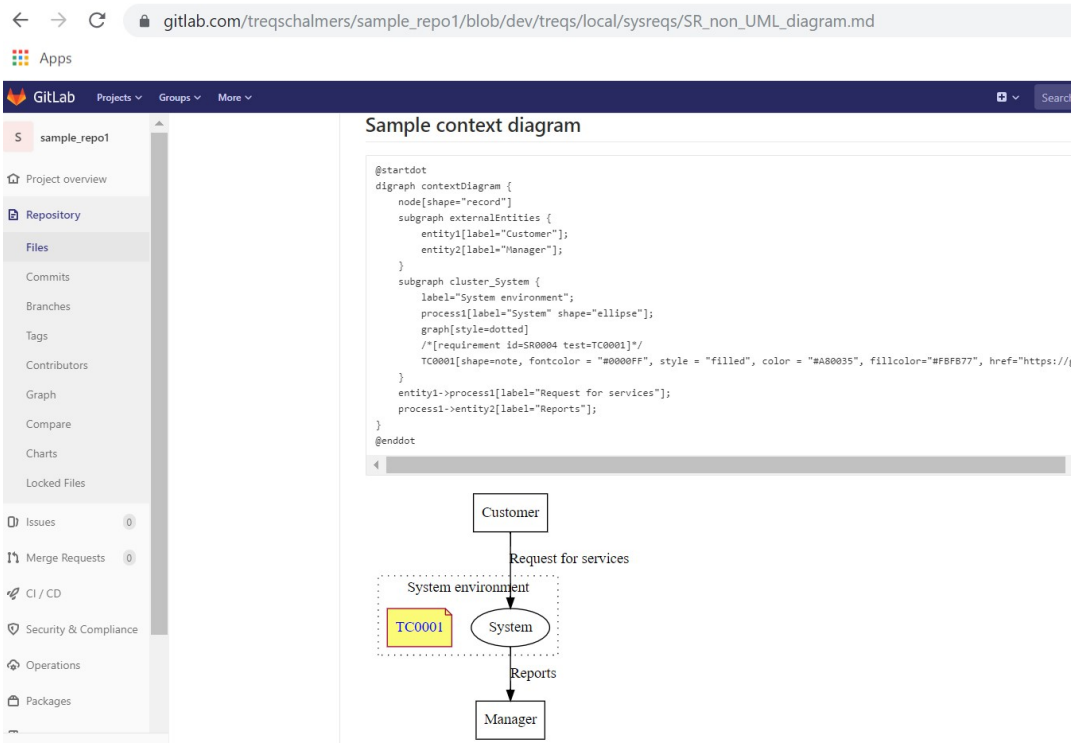


Figure 6.13: Sample context diagram on GitLab

The generation of diagram URLs can be automated with the help of *git pre-commit* hooks. For this to work, the *generatepreview* command should be placed in the *git pre-commit* hook file of the project repository. When files are committed, *T-reqs* scans the *git staging-area* for markdown files that contain embedded PlantUML/-
```

DOT language. If such files are found, it opens those files, extracts the PlantUML/DOT language parts and generates a diagram URL from them. Then, it updates the files with the corresponding diagram URL/s and trace-links and includes those changes in the current commit.

6.2.4 Multi-repository support

High-level system requirements are usually decomposed into low-level system requirements and assigned to software components. In such situations, there can be a need to get an update about the changes in the high-level system requirements from the repositories which implement the related low-level system requirements. Because a change in the high-level system requirements can affect the related low-level system requirements. To enable this, we should somehow be able to share high-level system requirements between multiple repositories so that it is possible to get instant updates when the high-level system requirements change. *T-reqs* uses *git submodules* to enable sharing of system requirements between multiple repositories. The requirements or other artifacts that are shared between multiple repositories are stored in a separate repository. Then, this repository is added as a *git submodule* to each of the repositories that shares those artifacts.
The *git-submodule* is added under the parent directory of *T-reqs* artifacts and it should be configured separately using the *treqsconfig* command. When *T-reqs* users push changes to a shared repository, other users who added this repository as a *git submodule* into their own repositories will be able to pull those changes. The *git submodule add* command is used to add the shared repository as a *git submodule*. Changes to the shared artifacts can be pushed to the remote repository by running the *git push* command from the root of the submodule. To pull any remote changes on the shared artifacts, the *git* commands `git submodule update --remote --merge` and *git submodule update* `--remote --rebase` can be used. Figure 6.14 shows some examples of high-level system requirements stored in a shared repository. This shared repository is added as a *git submodule* to the repositories *sample_repo1* and *sample_repo2* using the *git submodule add* command as it can be seen in Figure 6.15. The system requirement *SR0011* in Figure 6.14 has been changed in *sample_repo1* from "The system must provide an option for paying in card/swish/klarna" to "The system must provide an option for paying in card and swish" and this change is pushed to the *remote shared repository* as shown in Figure 6.16. This change in *sample_repo1* can be fetched from *sample_repo2* using the `git submodule update --remote --merge` command as it can be seen in Figure 6.17.

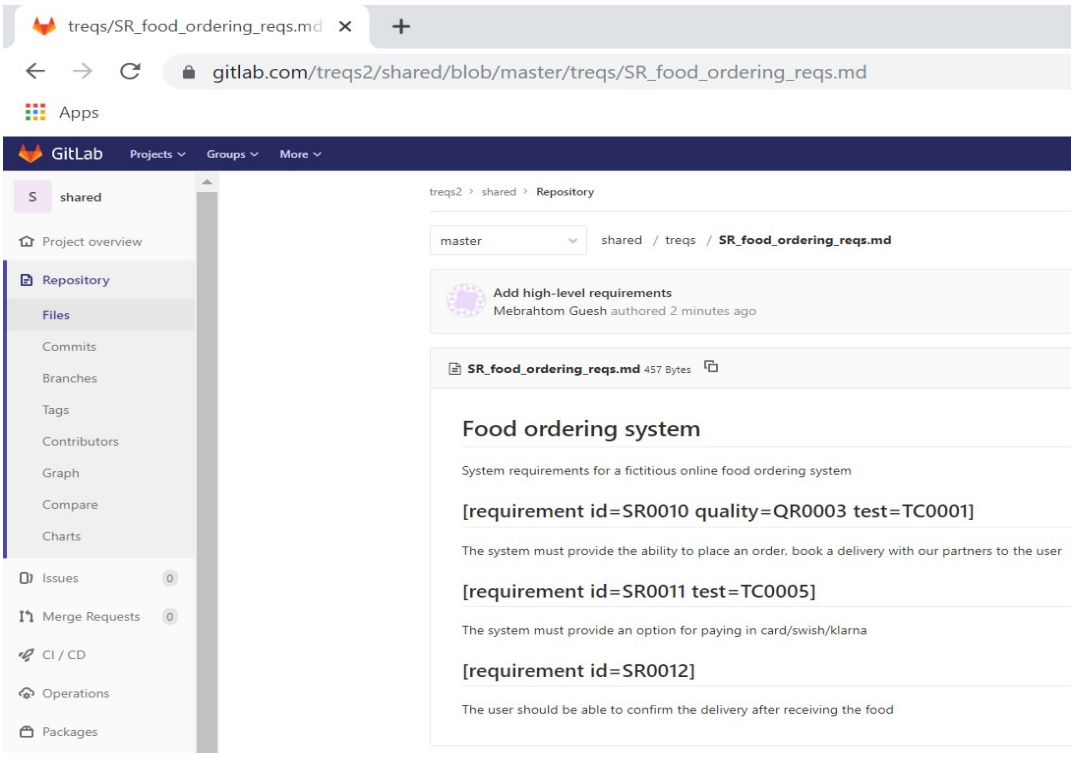


Figure 6.14: A shared repository which contains shared high-level system requirements

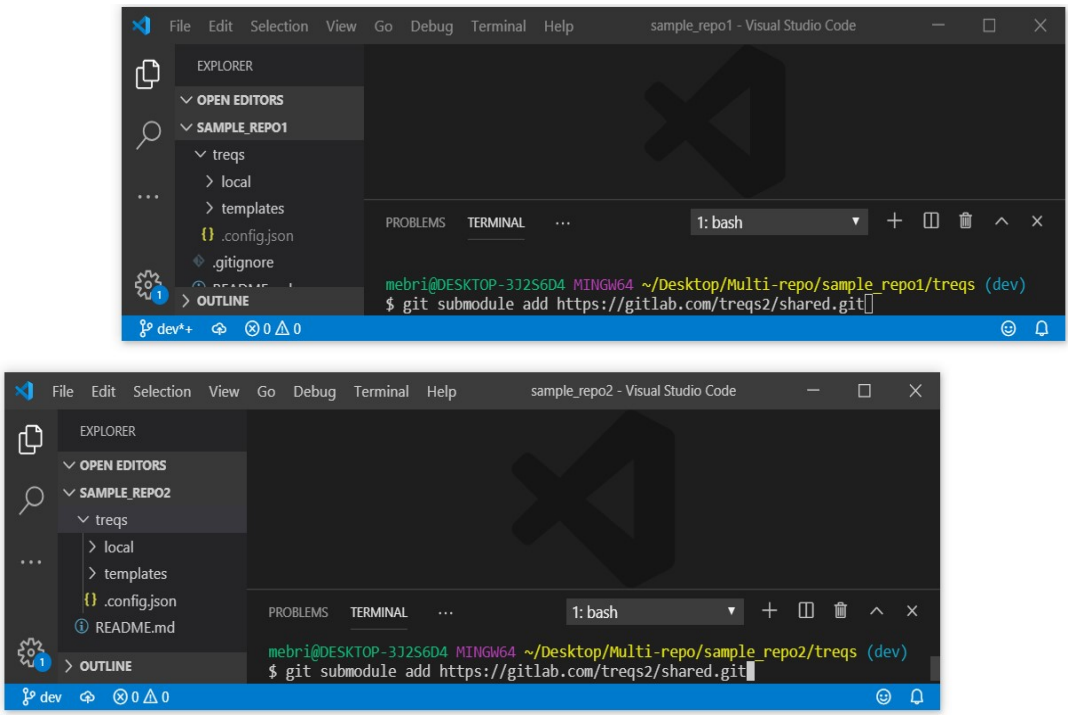


Figure 6.15: Adding a shared repository as a *git submodule*

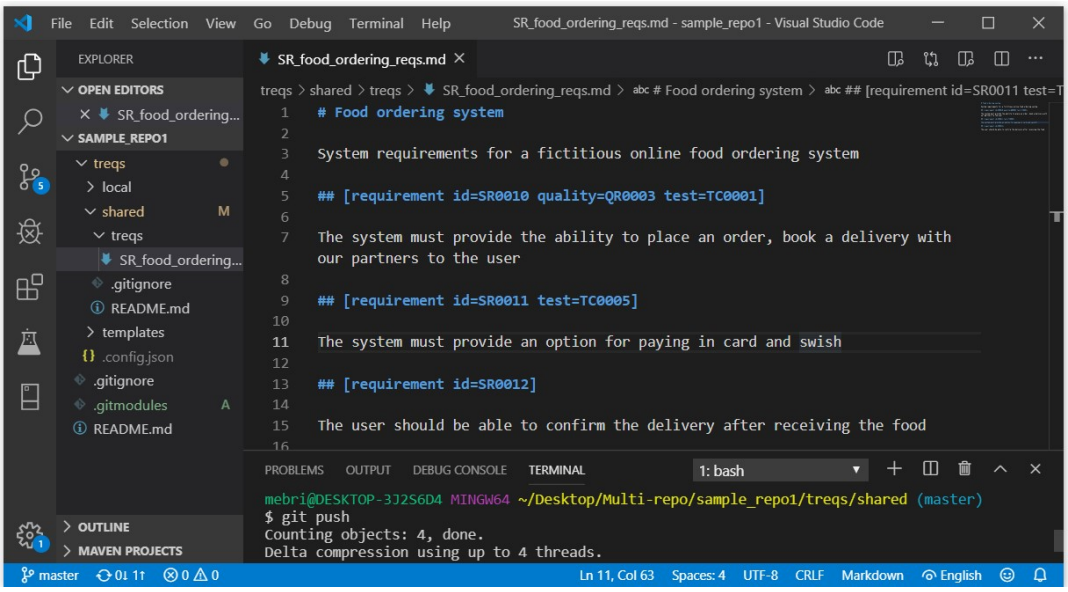


Figure 6.16: Making changes to shared system requirements

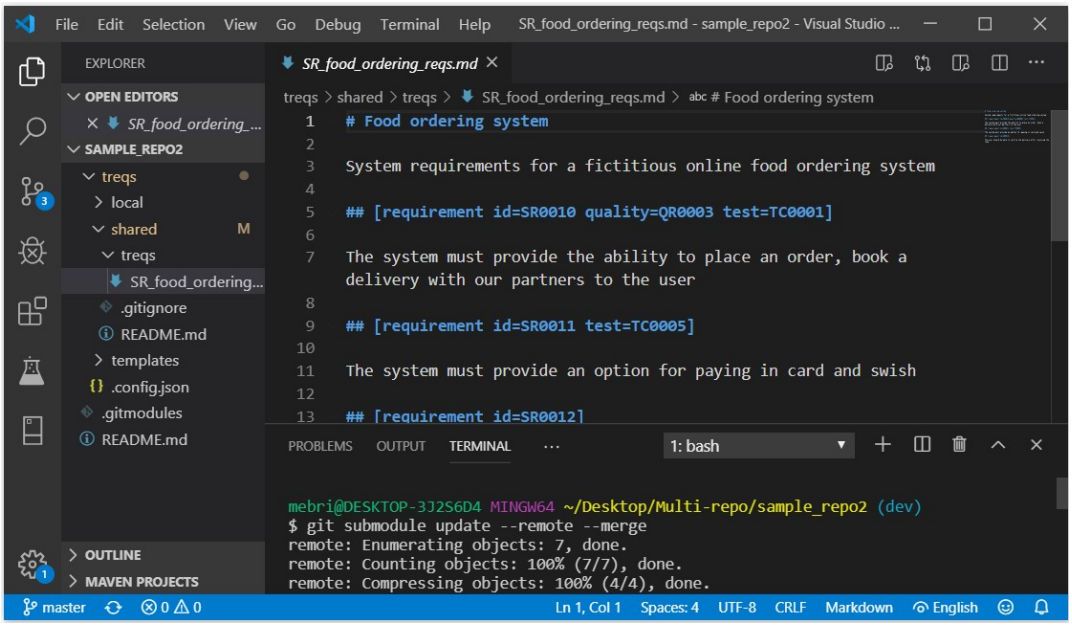


Figure 6.17: Fetching changes to shared system requirements

6.2.5 Change history of requirements

Storing the *T-reqs* artifacts in files makes it easy to use another powerful *git* feature called *git blame* to keep track of line-based change history of the artifacts. With *git blame*, it is possible to know who changed a certain system requirement or another artifact and when that change was happened. Figure 6.18 shows how *gitblame* can be used to know the change history of a specific line in a file.

```
mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqschalmers/sample_repo1 (dev)
$ git blame treqs/local/sysreqs/SR_non_UML_diagram.md
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  1) # Context diagram
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  2)
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  3) ```puml
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  4) @startdot
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  5) digraph contextDiagram {
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  6)     node[shape="record"]
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200  7)     subgraph externalEntities {
1b08cb49 (Mebrahtom Guesh 2019-10-24 10:17:45 +0200  8)         customerEntity[label="Customer"];
1b08cb49 (Mebrahtom Guesh 2019-10-24 10:17:45 +0200  9)         managerEntity[label="Manager"];
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 10)     }
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 11)     subgraph cluster_System {
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 12)         label="System environment";
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 13)         process1[label="System" shape="ellipse"];
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 14)         graph[style=dotted]
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 15)     }
1b08cb49 (Mebrahtom Guesh 2019-10-24 10:17:45 +0200 16)     customerEntity->process1[label="Request for services"];
1b08cb49 (Mebrahtom Guesh 2019-10-24 10:17:45 +0200 17)     process1->managerEntity[label="Reports"];
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 18) }
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 19) @enddot
e3b980e0 (Mebrahtom Guesh 2019-10-23 11:47:06 +0200 20) ```

mebri@DESKTOP-3J2S6D4 MINGW64 ~/Desktop/treqschalmers/sample_repo1 (dev)
```

Figure 6.18: Using *gitblame* to get the change history of system requirements

From Figure 6.18 above, it can be seen that the *commit hash*, *the author* and the date and time of change of each line have been displayed on the console. The *gitblame* command has some options that can provide more details about change history of the lines in a file. For example, the *-M* option detects moved or copied lines within in the same file. This option reports the original author of the lines instead of the last author that moved or copied the lines. The *-C* option detects lines that were moved or copied from other files. This option also reports the original author of the lines instead of the last author that moved or copied the lines.
Git blame displays the last author who modified a line. Therefore, it can be difficult to use *git blame* to know when a line was originally added. In such situations, the *git log* command can be used together with the *-S* option. The *-S* option takes a string and shows only those commits that changed the number of occurrences of that string.

6.2.6 Validation report

T-reqs can report some feedback regarding the validation of *T-reqs* artifacts. In the basic *T-reqs*, the *treqs* command reports duplicate IDs of artifacts, items without traces, items that are not referenced by other artifacts and items that are referenced by other artifacts but don't exist. It does this only for system requirements and test cases. In the improved version of *T-reqs*, the *treqs* command does the same for quality requirements and acceptance tests. It also reports IDs that have been generated but not used by the user who is issuing the command. Additionally, it validates the artifact IDs against the specified ID patterns in the *T-reqs* configuration file.

6.3 Work flow

The work-flow of using the *T-reqs* tool can be roughly described as following:

6.3.1 Setting up a database for storing artifact IDs

1. Create an Amazon Web Services (AWs) Account. For this case, the AWS account allows you to get access to the AWS database services.
2. Create a *MySQL* database instance on *AWS*. If you create a database for storing IDs while creating the database instance (note that in this case a database instance refers to a database server and can contain multiple databases), then you have to provide this name to the *database name* parameter while running the *treqsconfig* command. Otherwise, you can provide any name to the *database name* parameter of the *treqsconfig* command and it will create a database with that name in your database instance.
3. Once the database instance is created, you have to remember the *endpoint* and *port number* of the database instance and the *username* and *password* of your database. You will have to provide these details while configuring your repository with the *treqsconfig* command.
4. Install the AWS CLI (Amazon Web Services Command Line Interface) in your system. The AWS CLI allows you to store your AWS account's credentials locally in your system. *T-reqs* uses these credentials to encrypt the database *username* and *password* before storing them in the *T-reqs* configuration file.
5. Setup your *AWS CLI* installation using the *aws configure* command. The *aws configure* prompts you to enter the *AWS access key ID*, *AWS secret access key*, *default region name* and *default output format* for your AWS account.

6.3.2 Working with *T-reqs*

1. Clone the *T-reqs* tool from `https://gitlab.com/mebrahtom/treqs`
2. Change directory to the root of the *T-reqs* repository and install it in your system by running the following command on your terminal.
   `pip install .`
3. Once the *T-reqs* tool is successfully installed in your system, then you have to configure your project's repository using the *treqsconfig* command.
4. Write any of the *T-reqs* artifacts (system requirements, quality requirements, test cases and acceptance tests) by following the respective templates which can be found under the *parent directory of T-reqs artifacts* in the project repository.
5. Generate IDs using the *generateid* command and assign them to your artifacts. If you have forgotten which IDs you have previously generated, you can use the *generatedid* command to get the IDs you have already generated.
6. Write the *generatepreview* command to the *git pre-commit* hook file of your project repository. This makes the *generatepreview* command to automatically get executed whenever new changes on model-based system requirements are committed.

7. If you have high-level system requirements or other artifacts that are shared between multiple repositories, then place the shared artifacts in their own repository. Then, add this repository as a *git submodule* to the other repositories that share the shared artifacts. This allows *T-reqs* users to get updated about the changes in the shared repository by `git pulling` them from the shared repository. They can also *git push* changes to the shared repository.
8. Use the *git blame* and/or *git log* commands to keep track of change history of the *T-reqs* artifacts.

6.4 Implementation

The *T-reqs* tool is implemented in `Python3`. *T-reqs* also uses third party and built-in python modules to implement different functionalities. The *T-reqs* configuration file stores the different details about the *T-reqs* artifacts, project repository and database connectivity in a `JSON` format. The MySQL database is used to store the artifact IDs. The MySQL database instance is created on *Amazon Web Services Relational Database Service (AWS RDS)*.
In *T-reqs*, text-based system requirements are specified in *markdown* format while model-based system requirements are specified using PlantUML code/-DOT language. Quality requirements and manual test cases are also specified in *markdown* format. Automated test cases are written as python scripts.

7
Discussion

In this section, the results of this study will be discussed and compared with other literature.

7.1 Revisiting the research questions

7.1.1 Research Question 1

Which problems of large-scale agile requirement engineering are related to tool support?
To answer this research question, a set of large-scale agile RE challenges have been identified in three different iterations. Then, tooling solutions were proposed, implemented and evaluated. If the evaluation results are positive, it can be concluded that the corresponding large-scale agile RE challenges are tool-support (tooling) related, i.e., the RE challenges can be addressed using tooling. Based on the evaluation results of the three different iterations, the tool-support related large-scale agile RE challenges are summarized in Table 7.1 below:

Problem ID	**Problem description**
P1	System requirements can be documented in a textual format and/or in the form of models. However, models are difficult to version control and modify. This makes model-based system requirements difficult to work with in a large-scale agile context which is characterized by inter-dependent teams and quickly changing requirements.
P2	Writing models in a textual UML tool called PlantUML is the solution suggested for solving P1.1 in this iteration. However, the major repository managers like GitLab and GitHub do not support PlantUML rendering at the writing of this paper. And this makes it difficult to review the model-based system requirements on repository management services like *GitLab*.
P3	System requirements are usually related to other artifacts such as test cases and quality requirements. Therefore, it should be possible to navigate to related artifacts from the model that specifies a certain system requirement. However, most modeling tools do not support traceability functionality in a model. In other words, the models are usually rendered as images and so it is difficult to add trace-links on top of them.

P4	System requirements can be related to other artifacts in addition to test cases. For example, they can be related to quality requirements and user stories. User stories can also be related to user acceptance tests. However, only system requirements and test cases have been documented in the basic *T-reqs*. In other words, there is no a clear guideline or syntax on how to document the requirement related artifacts such as quality requirements, user stories and acceptance tests.
P5	In a large-scale agile context, high-level system requirements are usually decomposed into low-level system requirements. These low-level system requirements are usually assigned to different but related software components that are implemented in different repositories. In such situations, requirement changes in one repository can affect the related requirements which reside in different repositories. However, it is difficult to propagate requirement changes to multiple teams and multiple repositories (projects).
P6	It is important to keep a record of requirement changes; what changed, when, why and who changed it. In traditional software development, details about requirement changes are usually stored in databases. However, it is difficult to keeping track of requirement change history when the requirements are stored in repositories.
P7	System requirements and other related artifacts should be assigned IDs so that it is possible to uniquely identify them. In *T-reqs*, the artifacts are stored in files under *git repositories*. With multiple teams working with those artifacts in parallel, it is difficult to synchronize the generation of artifact IDs.
P8	It is difficult to update traces in a model.
P9	The need for a central control of requirement documentation conflicts with the principle of self-empowered teams.

Table 7.1: Tool support related large-scale agile RE challenges

The problems *P1-P4* and *P8* in Table 7.1 are related to difficulties in updating requirement documentation in large-scale agile software development. This kind of problem is a well-known problem that has been mentioned in a number of literature [1], [2], [4], [9]. The need for propagating requirement changes between interdependent teams and bridging the gap between development teams and other stakeholders have been discussed in some literature [1], [2]. Theses challenges are related to the problems *P5* and *P9* in Table 7.1 respectively. The difficulty to achieve inter-team coordination is a known challenge in a large-scale agile software development [10]. The problem of synchronizing artifact ID generation (problem P7 in Table 7.1) can be considered as an example of synchronization problem between development team members. Addressing the challenge of tracking requirement change history (problem P6 in Table 7.1) allows the traceability of system requirements to customers and other stakeholders and it ensures that development teams are working on the correct version of a system requirement. Therefore, this study contributes in solving well-known large-scale agile RE challenges.

The difficulty in managing requirement dependency and in tracing system requirements to test execution (Problems P1.6 and P1.7 in Table 5.1) have not been addressed in this study because of time limitation.

7.1.2 Research Question 2

What possible solutions can be provided based on the T-reqs tool?
The solutions to the identified large-scale agile RE challenges are implemented by extending the *T-reqs* tool. *T-reqs* focuses on bringing system requirements and other related artifacts closer to development teams. The solutions provided for the tool-support related problems in Table 7.2 are provided in Table 7.2:

Problem ID	**Solution ID**	**Solution description**
P1	S1	PlantUML and the DOT language can be used to write model-based system requirements within a markdown file. This way it is possible to write the requirements in a textual format which is easy to version control and modify.
P2	S2	The PlantUML/DOT code blocks should be embedded within the *markdown* file using *fenced code blocks* as shown in the template in Figure A.2. Then, a python script is written to generate an encoded URL from the PlantUML/DOT code and write it to the same markdown file. Listings 2 and 1 show code snippets for generating and writing a URL to a file respectively. With the help of a web-based PlantUML server, the actual diagrams can be viewed on repository management services like *GitLab*. In Figure 6.7, line 18 shows a generated URL for a sample sequence diagram. And Figure 6.9 shows how that generated URL looks like on GitLab.
P3	S3	*T-reqs* has a trace-ability syntax which can be used for defining traces between related artifacts. Python scripts are written to formulate trace-links in the model based on the specified traces. Such trace-links can be used to navigate from the model to related artifacts. In *T-reqs*, the generated trace-links are automatically updated when ever changes are *git committed*.

P4	S4	Quality requirements can be documented as *markdown* files and stored in *git repository* together with the system requirements. This does not only allow teams to easily access the quality requirements but also to link them with system requirements by committing both artifacts together or by referencing them with their commit hashes. Acceptance tests can be stored in a *git repository* as well. *Cucumber feature files* can be used for documenting user acceptance tests in *T-reqs*. Traces can be created between related artifacts by using *T-req's* trace-ability syntax.
P5	S5	The *git submodule* can be used to propagate high-level requirement changes to multiple repositories. By storing the high-level system requirements in a separate repository and then letting teams add this repository as a *git submodule* into their own repositories, they can *git pull* the changes to the *git submodule*. They can also *git push* changes to the shared repository.
P6	S6	The *git blame* command can be used to keep track of artifact change history. To get older change histories, the *git log* command can be used.
P7	S7	The artifact IDs can be stored in a could-based database. This allows to generate new IDs using the "UNIQUE" SQL constraint.
P8	S8	Writing scripts to automatically update the traces based on *git pre-commit hooks*. In this way, the traces will be updated whenever *git commits* are performed on the file which contains such traces.
P9	S9	Although it is difficult to decide on how much of the requirements documentation should be done by the development teams, storing the requirements together with the code in *git repositories* provides an easy access to the teams.

Table 7.2: Tooling solutions to large-scale agile RE challenges

The solutions implemented in this study have not been implemented in the basic *T-reqs* tool [6]. So, this study provides new tooling solutions to some important RE challenges within the context of the *T-reqs* tool.

7.1.3 Research Question 3

To what extent can the problems identified from RQ1 be addressed by the proposed solutions in RQ2?
According to current results, all of the participants who evaluated the tool have strongly agreed and agreed based on the *5-point likert scale* in all iterations. This

indicates that the proposed solutions work for the problems listed in Table 7.1. The usability assessment results of the tool are 75, 70 and 74 in *Iterations* I, II, and II respectively. These results are above average based on the system usability scale (SUS) [25]. Therefore, it can be concluded that the solutions are acceptable with above average usability.

7.2 Threats of validity

In this section, the different threats to the validity of this study and the strategies used to mitigate those threats will be discussed. The categories of validity threats that will be discussed are based on the classification by Runeson and Höst [21].

7.2.1 Construct validity

To avoid misunderstanding of the survey questionnaires, the questionnaire items were explained to the participants in the workshops before they fill out the survey. The functionality assessment items are likert-scale based closed questions. This can prevent the users from sharing their detailed views regarding the different functionalities of the tool. The usability of the tool was also assessed only using the system usability scale. This can be a threat to the construct validity of the usability assessment because the system usability scale does not provide accurate information about the weaknesses of the tool. To mitigate these problems, an open-ended question was included where they were asked to write any limitations/problems they have observed in the tool. In the third iteration (*Focus group II*), a student, a developer and another person from academia were used to evaluate the tool. Such a mix of professionals and students may have an impact on the construct validity. Because, the students, developers and academicians represent different populations and so this makes it difficult to give conclusions about a specific population. To mitigate this problem, the tool was also evaluated using a workshop where only students had participated.

7.2.2 Internal validity

Some of the people who participated in the evaluation of the tool had knowledge about the basic version of *T-reqs* and so their previous experience with the tool can be a source of internal validity threat. Furthermore, some people had participated in multiple iterations. This can be also a threat to the internal validity of the results because they can learn the tool from iteration to iteration. The fact that the tool was also evaluated using students and other people who had no previous knowledge of the tool can partially mitigate this problem. During the workshops, some of the participants had asked for help while using the tool. This can be a threat to the validity of the study as it helps them to learn it on the way.

7.2.3 External validity

The *T-reqs* tool was evaluated at one company only. And this can have a negative effect in the generalization of the results. To mitigate this problem, different roles were used during the evaluation of the tool. For example, software architect, system managers, researchers, developers and students are some of the people who took part in the evaluation and problem identification phases of the different iterations. There were a small number of participants in each iteration because it was difficult to get large-number of participants. This can affect the external validity of the study. To mitigate this problem, the solutions which were implemented in the previous iterations were evaluated again in the next consecutive iterations.

7.2.4 Reliability

The results from the focus groups and observations were summarized and interpreted directly instead of applying formal data analysis techniques. This can pose a threat of validity because different researchers can interpret the summarized text in different ways. Furthermore, there was only one author who observed the participants and took notes during the workshop and focus group sessions. This can also be threat to the validity as it can be difficult for one person to observe and grasp all participants' activities. To mitigate these problems, survey questionnaires were used during all workshops in addition to observing the participants' activities. To help other researchers replicate this study, all the survey questionnaires and the focus group questions are published in the appendix section of this study.

8

Conclusions and future work

This study intends to investigate tool-support related RE challenges in large-scale agile system development and tries to solve the identified challenges by extending an existing RE tool called *T-reqs*.
As results of this study, a set of tool-support related large-scale agile RE challenges have been identified. These challenges can be found in Table 7.1. And tooling solutions have been implemented for the identified challenges withing the *T-reqs* tool. These solutions can be found in Table 7.2. The implemented solutions have been also evaluated using workshops, interviews, focus groups and surveys in three different iterations. Based on the results of this study, it can be concluded that a set of large-scale agile RE challenges have been addressed using *T-reqs*.
The philosophy used by *T-reqs* is to manage system requirements and other requirement related artifacts by bringing them closer to development teams. As a result, system requirements, test cases, quality requirements and acceptance tests can be written in files and stored in *git* repositories together with the actual implementation. This way of managing system requirements is made easier to work with by improving the *T-reqs* tool in such a way that it addresses a number of additional large-scale agile RE challenges.
Using the improved version of *T-reqs*, system requirements can be written in parallel with the actual implementation instead of using hybrid approaches of traditional and agile methods or adopting scaled-agile frameworks as recommended by some studies [40], [39], [41], [15]. Because those approaches introduce other challenges or address very limited challenges.
The *T-reqs* tool can be further extended to solve additional large-scale agile RE challenges. Some of the RE challenges that have been identified but not solved in this study are managing requirement dependencies, tracing system requirements to test executions, integrating analytics functionality to provide insights about the requirement types that are changed often and defining an information model that can be used by teams to specify their artifacts. Further research should be done on how to address these and other related challenges within the context of the *T-reqs* tool.

Bibliography

[1] R. Kasauli, G. Liebel, E. Knauss, S. Gopakumar, and B. Kanagwa, "Requirements Engineering Challenges in Large-Scale Agile System Development," 2017 IEEE 25th International Requirements Engineering Conference (RE), 2017.

[2] E. Knauss, "The Missing Requirements Perspective in Large-Scale Agile System Development," IEEE Software, vol. 36, no. 3, pp. 9–13, 2019.

[3] D. Fucci, A. Falkner, G. Schenner, F. Brasca, T. Männistö, A. Felfernig, W. Maalej, C. Palomares, X. Franch, D. Costal, M. Raatikainen, M. Stettinger, Z. Kurtanovic, T. Kojo, and L. Koenig, "Needs and challenges for a platform to support large-scale requirements engineering," Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM 18, 2018.

[4] I. Inayat, S. S. Salim, S. Marczak, M. Daneva, and S. Shamshirband, "A systematic literature review on agile requirements engineering practices and challenges," Computers in Human Behavior, vol. 51, pp. 915–929, 2015.

[5] O. Uludag, M. Kleehaus, C. Caprano, and F. Matthes, "Identifying and Structuring Challenges in Large-Scale Agile Development Based on a Structured Literature Review," 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), 2018.

[6] E. Knauss, G. Liebel, J. Horkoff, R. Wohlrab, R. Kasauli, F. Lange, and P. Gildert, "T-Reqs: Tool Support for Managing Requirements in Large-Scale Agile System Development," 2018 IEEE 26th International Requirements Engineering Conference (RE), 2018.

[7] P. Johannesson and E. Perjons, "Introduction," An Introduction to Design Science, pp. 1–19, 2014.

[8] V.Vijay and K.William, "Design science research in information systems,", 2004.

[9] Wohlrab, R., Pelliccione, P., Knauss, E., & Larsson, M. (2018, May). Boundary objects in agile practices: Continuous management of systems engineering artifacts in the automotive domain. In Proceedings of the 2018 International Conference on Software and System Process (pp. 31-40). ACM.

[10] F. O. Bjørnson, J. Wijnmaalen, C. J. Stettina, and T. Dingsøyr, "Inter-team Coordination in Large-Scale Agile Development: A Case Study of Three Enabling Mechanisms," Lecture Notes in Business Information Processing Agile Processes in Software Engineering and Extreme Programming, pp. 216–231, 2018.

[11] Y. I. Alzoubi, A. Q. Gill, and A. Al-Ani, "Empirical studies of geographically distributed agile development communication challenges: A systematic review," Information & Management, vol. 53, no. 1, pp. 22–37, 2016.

[12] S. Fricker, T. Gorschek, C. Byman, and A. Schmidle, "Handshaking with Implementation Proposals: Negotiating Requirements Understanding," IEEE Software, vol. 27, no. 2, pp. 72–80, 2010.

[13] M. Hoffmann, N. Kuhn, M. Bittner, and M. Weber, "Requirements for requirements management tools," Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.

[14] J. M. C. D. Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno, "Requirements engineering tools: Capabilities, survey and assessment," Information and Software Technology, vol. 54, no. 10, pp. 1142–1157, 2012.

[15] M. Kalenda, P. Hyna, and B. Rossi, "Scaling agile in large organizations: Practices, challenges, and success factors," Journal of Software: Evolution and Process, vol. 30, no. 10, 2018.

[16] P. Lombriser, F. Dalpiaz, G. Lucassen, and S. Brinkkemper, "Gamified Requirements Engineering: Model and Experimentation," Requirements Engineering: Foundation for Software Quality Lecture Notes in Computer Science, pp. 171–187, 2016.

[17] V. Gaikwad, P. Joeg, and S. Joshi, "AgileRE: Agile Requirements Management Tool," Advances in Intelligent Systems and Computing Cybernetics Approaches in Intelligent Systems, pp. 236–249, Jun. 2017.

[18] T. Avdeenko and M. Murtazina, "Intelligent Support of Requirements Management in Agile Environment," Service Orientation in Holonic and Multi-Agent Manufacturing Studies in Computational Intelligence, pp. 97–108, Dec. 2018.

[19] D. Fucci, A. Falkner, G. Schenner, F. Brasca, T. Männistö, A. Felfernig, W. Maalej, C. Palomares, X. Franch, D. Costal, M. Raatikainen, M. Stettinger, Z. Kurtanovic, T. Kojo, and L. Koenig, "Needs and challenges for a platform to support large-scale requirements engineering," Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM 18, 2018.

[20] K. Singh, "Quantitative Social Research Methods," 2007.

[21] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering, vol. 14, no. 2, pp. 131–164, 2008.

[22] S. Lauesen, Software requirements: styles and techniques. Harlow: Addison-Wesley, 2002.

[23] A. V. Lamsweerde, Requirements engineering: from system goals to UML models and software specifications. Chichester: Wiley, 2010.

[24] S. Chacon and B. Straub, Pro Git. Berkeley, CA: Apress, 2014.

[25] B.John, "System usability scale (SUS): a quick-and-dirty method of system evaluation user information," Reading, UK: Digital Equipment Co Ltd, vol. 43, 1986.

[26] P. A. Laplante, Requirements engineering for software and systems. Boca Raton: CRC Press, 2018.

[27] B. Meyer, Agile!: the good, the hype and the ugly. Zurich: Springer International Publishing, 2014.
[28] A. Spillner, T. Linz, and H. Schaefer, Software Testing Foundations: a study guide for the certified tester exam. Santa Barbara, CA: Rocky Nook, 2014.
[29] "Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams.," PlantUML.com. [Online]. Available: http://plantuml.com/. [Accessed: 02-Dec-2019].
[30] A. Herrero, Instant Markdown: learn how to efficiently manage your content and use different services with Markdown. Birmingham, UK: Packt Publishing, 2013.
[31] Gansner, E., Koutsofios, E., & North, S. (2006). Drawing graphs with dot. E.R. Gansner and E.Koutsofios and S.North, "Drawing graphs with dot," 2015.
[32] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kahkonen, "Agile software development in large organizations," Computer, vol. 37, no. 12, pp. 26–34, 2004.
[33] B. Daniel, "Likert scales," Retrieved November, vol. 2, 2007.
[34] J. Kitzinger, "Qualitative Research: Introducing focus groups," Bmj, vol. 311, no. 7000, pp. 299–302, 1995.
[35] J. Rumbaugh and I. Jacobson, Unified modeling language reference manual. Reading, Mass.: Addison-Wesley, 1998.
[36] B. Ramesh, L. Cao, and R. Baskerville, "Agile requirements engineering practices and challenges: an empirical study," Information Systems Journal, vol. 20, no. 5, pp. 449–480, 2007.
[37] E. Bjarnason, K. Wnuk, and B. Regnell, "A case study on benefits and side-effects of agile practices in large-scale requirements engineering," Proceedings of the 1st Workshop on Agile Requirements Engineering - AREW 11, 2011.
[38] M. Paasivaara, S. Durasiewicz, and C. Lassenius, "Using scrum in a globally distributed project: a case study," Software Process: Improvement and Practice, vol. 13, no. 6, pp. 527–544, 2008.
[39] J. B. Barlow, J. S. Giboney, M. J. Keith, D. W. Wilson, R. M. Schuetzler, P. B. Lowry, and A. Vance, "Overview and Guidance on Agile Development in Large Organizations," Communications of the Association for Information Systems, vol. 29, 2011.
[40] D. Badampudi, S. A. Fricker, and A. M. Moreno, "Perspectives on Productivity and Delays in Large-Scale Agile Projects," Lecture Notes in Business Information Processing Agile Processes in Software Engineering and Extreme Programming, pp. 180–194, 2013.
[41] D. Batra, W. Xia, D. Vandermeer, and K. Dutta, "Balancing Agile and Structured Development Approaches to Successfully Manage Large Distributed Software Projects: A Case Study from the Cruise Line Industry," Communications of the Association for Information Systems, vol. 27, 2010.
[42] W. Matt, H. Aslak and T. Steve, The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf, 2017.
[43] L. Nan, E. Anthony, K. Tariq, "2016 IEEE International Conference on Software Testing, Verification and Validation (ICST),". IEEE, pp. 393–400, 2016.

A

Appendix

A.1 Implementation

The following figures show some of the implementation details used in the improved version of *T-reqs*.

template_textual_ sys_reqs.md

template_textual_ sys_reqs.md > abc # Document title > abc ## Another subsection > abc ### [requirement id=sys_req_id quality=

```
1  # Document title
2
3  Document description
4
5  ## Subsection
6
7  Section description
8
9  ### [requirement id=sys_req_id test=test_case_id quality=quality_req_id]
10
11 Requirement text
12
13 ## Another subsection
14
15 Section description
16
17 ### [requirement id=sys_req_id  quality=quality_req_id test=test_case_id1,test_case_id2]
18
19 Requirement text
```

Figure A.1: Template for specifying textual system requirements in *markdown*

template_model_ sys_reqs.md

template_model_ sys_reqs.md > Markdown Language Features > abc # Writing model-based system requirements using PlantUML,

```
# Writing model-based system requirements using PlantUML/DOT language

## PlantUML diagram

some text

```puml
@startuml
/'PlantUML code goes here'/
note right|left|bottom|top on <element>
/'[requirement id=sys_req_id test=test_case_id1,test_case_id2 quality=quality_req_id]'/
end note
@enduml
```

DOT graph

Some text
```puml
@startdot
/*DOT language goes here*/
digraph <digraph_name>{
/*Some code*/
    subgraph <subgraph_name>{
    //subgraph code
    /*[requirement id=sys_req_id test=test_case_id1,test_case_id2 quality=quality_req_id]*/
    //some more code
  }
}
@enddot
```
```

Figure A.2: Template for specifying model-based system requirements in *PlantUML/DOT language*

template_quality_req.md

template_quality_req.md > abc # Document title > abc ## Another subsection > abc ### [quality id=

```
# Document title

Document description

## Subsection

Section description

### [quality id=quality_req_id requirement=sys_req_id]

Quality requirement text

## Another subsection

Section description

### [quality id=quality_req_id requirement=sys_req_id]

Quality requirement text
```

Figure A.3: Template for specifying quality requirements in *markdown*

template_manual_test_case.md

template_manual_test_case.md > abc # Document title > abc ## Test result

```
1   # Document title
2
3   Document description
4
5   ## [testcase id=test_case_id requirement=sys_req_id]
6
7   Purpose: Purpose of the test case without line break.
8
9   ## Setup
10
11  Describe any steps that must be done before performing the test.
12
13  ### Scenario / Steps
14
15  ### Expected outcome
16
17  ### Tear down
18
19  Describe what to do after the test
20
21  ## Test result
22
23  Protocol of the result of executing this test, latest on top
```

Figure A.4: Template for specifying manual test cases in *markdown*

template_automated_test_case.py

```python
import unittest


class test_case_name(unittest.TestCase):
    """
    [testcase id=test_case_id  requirement=sys_req_id]

    <Purpose of the test case>
    """

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)


if __name__ == '__main__':
    unittest.main()
```

Figure A.5: Template for specifying automated test cases as *python scripts*

template_acceptance_test.feature

```
# [acceptancetest id=acceptance_test_id story=user_story_id]
Feature: <Feature_title>

    Scenario: <Scenario_title1>
        When:<Some_action>
        Then:<Expected_result>

    Scenario: <Scenario_title2>
        Given:<Some_precondition>
        When:<Some_action>
        Then:<Expected_result>
```

Figure A.6: Template for specifying acceptance tests in *cucumber feature files*

The following figures show sample artifacts written in *T-reqs*.

SR_sample_sequence_diagram.puml

treqs > local > sysreqs > SR_sample_sequence_diagram.puml > ...

```
@startuml Sample sequence diagram
/'[requirement id=SR0003 quality=QR0002 test=TC0001,TC0005]'/
Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response
Alice -> Bob: Another authentication Request
Alice <-- Bob: Another authentication Response
@enduml
```

Figure A.7: Sample sequence diagram written in PlantUML

SR_sample_sequence_diagram.md

treqs > local > SR_sample_sequence_diagram.md

```
[TC0001](https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/tests/TC_system.py)
[TC0005](https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/tests/TC_system.py)
[QR0002](https://gitlab.com/treqschalmers/sample_repo1/blob/dev/treqs/local/qualityreqs/QR_qualityreqs.md)

![Generated diagram](http://www.plantuml.com/plantuml/svg/
XS_13S8m38NXUwPu2sG1X53R4M02qreGAN5AxEoF2vnGfHwjz-cN70OrQjPBnetAcXdAhyuoCvP4ZqO5OSp9ptZ1LEydZDsUB9PcibG5rush1TyGtNpmZjArYl9_Iwgp1jflVl4vnWFzQmyiYpyI
tW00)
```

Figure A.8: Generated URL and trace-links for sample PlantUML diagram

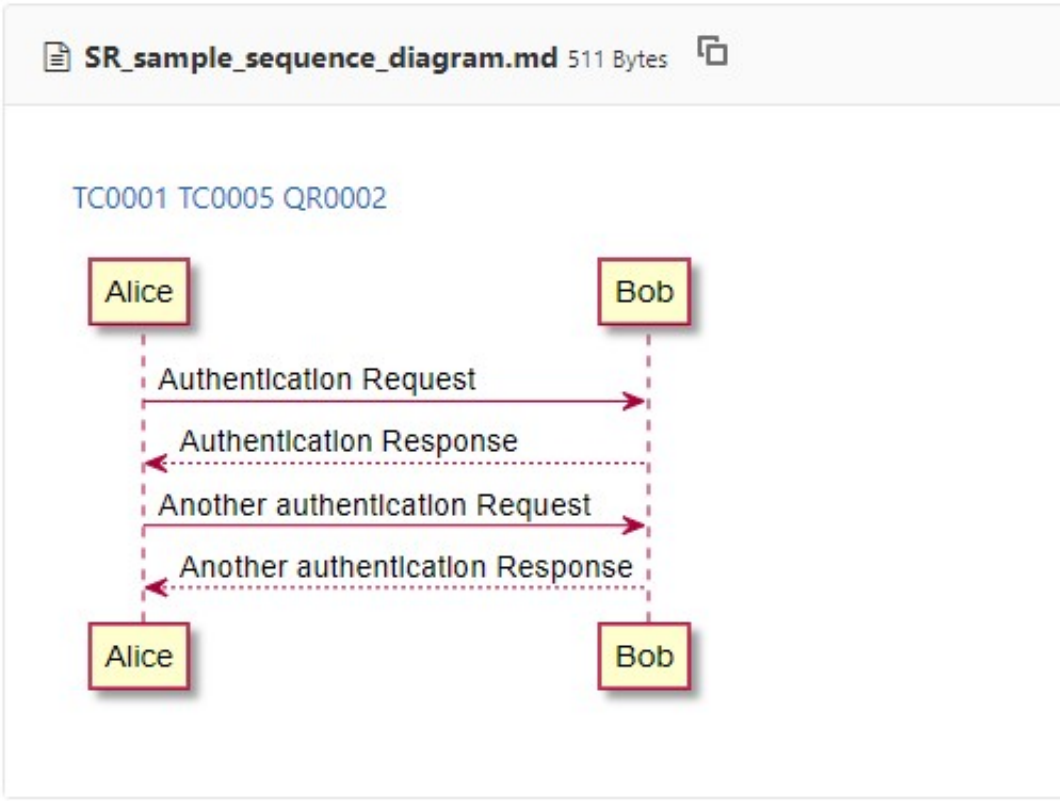


Figure A.9: Previewing PlantUML diagram URLs and trace-links on *GitLab*

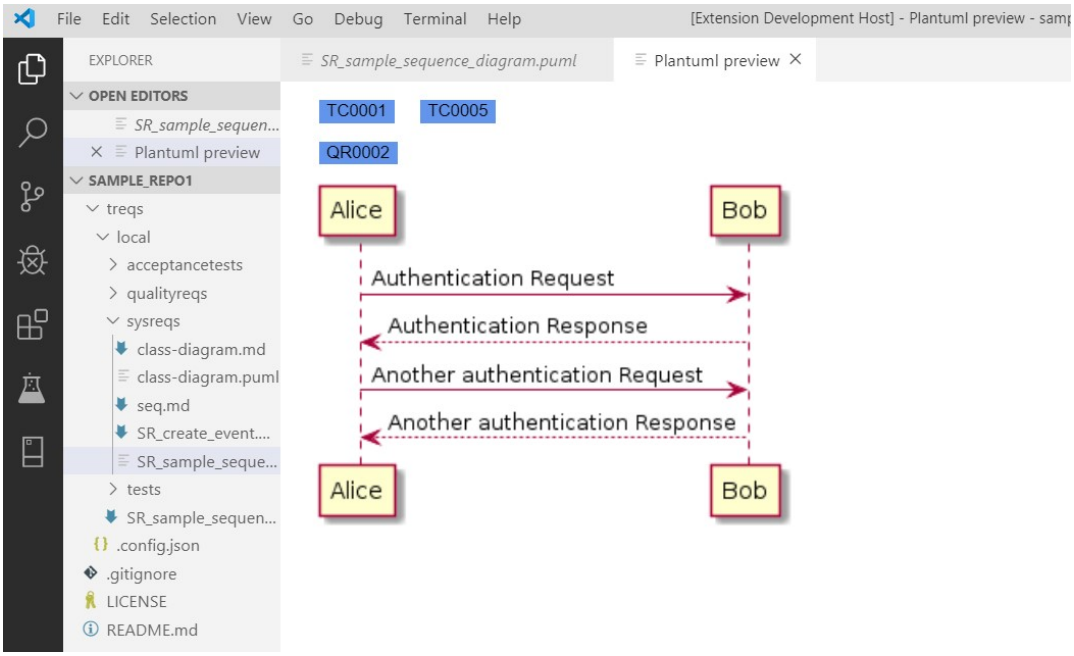


Figure A.10: Previewing PlantUML code locally with the help of *T-reqs* extension for VSCODE

SR_food_ordering_reqs.md

treqs > local > sysreqs > SR_food_ordering_reqs.md > # Food ordering system > ## [requirement id=SR0012]

```
# Food ordering system

System requirements for a fictitious online food ordering system

## [requirement id=SR0010 quality=QR0003 test=TC0001]

The system must provide the ability to place an order, book a delivery with our partners to the user

## [requirement id=SR0011 test=TC0005]

The system must provide an option for paying in card/swish/klarna

## [requirement id=SR0012]

The user should be able to confirm the delivery after receiving the food
```

Figure A.11: Examples of textual system requirements written in *markdown* format

QR_food_ordering_qualities.md

treqs > local > qualityreqs > QR_food_ordering_qualities.md > # Quality requirements for a fictitious online food ordering system >

```
# Quality requirements for a fictitious online food ordering system

## [quality id=QR0003 requirement=SR0010]

The system should be able to place an order successfully in less than 30 seconds

## [quality id=QR0004 requirement=SR0010]

For two or more users ordering food within a range of 5 seconds,
priority should be given to the user who has ordered the most in the system so far
```

Figure A.12: Sample quality requirements written in *markdown* format

TC_manual.md

treqs > local > tests > TC_manual.md > # Sample manual test case > ## [testcase id=TC0001 requirement=SR0010] >

```
# Sample manual test case

## [testcase id=TC0001 requirement=SR0010]

Purpose: Test that an order is placed for food delivery

### Setup

### Scenario / Steps

1. Click the "order food" button
2. Select type of food
3. Select delivery time
4. Click "ok" button

### Expected outcome

1. The food type and time of delivery are saved
2. A courier is assigned to the order
```

Figure A.13: Sample manual test case written in *markdown* format

TC_sample_automated_test.py

treqs > local > tests > TC_sample_automated_test.py > ...

```python
import os
import unittest
from git import Repo


class TestSystem(unittest.TestCase):
    def test_config_file_exists(self):
        """
        [testcase id=TC0007 requirement=SR0020]

        Ensure that a configuration file is created.
        """
        repo = Repo('.', search_parent_directories=True)
        repoPath = repo.working_tree_dir
        repoPath = os.path.normpath(repoPath)
        configFileExists = False
        for root, _, files in os.walk(repoPath):
            for file in files:
                filePath = os.path.join(root, file)
                if file == ".config.json" and os.path.dirname(os.path.dirname(filePath)) == repoPath:
                    configFileExists = True
                    break
        self.assertTrue(configFileExists)


if __name__ == '__main__':
    unittest.main()
```

Figure A.14: Sample automated test case written in *Python* script

AT_sample_acceptance_test.feature

treqs > local > acceptancetests > AT_sample_acceptance_test.feature

```
# [acceptancetest id=AT0001 story=US0005]
Feature: Order food

    # Food order scenario
    Scenario: Order food
        Given the user has successfully logged in
        When the user clicks the "order food" button
        Then the user is presented with food selection menu

    # Payment scenario
    Scenario: Pay for your food order
        Given the user has successfully logged in
        And the user selects the type of food
        When the user clicks "pay" button
        And  the user selects payment method
        Then the system deducts the amount of the selected food's cost from his bank account
```

Figure A.15: Sample acceptance tests written in *cucumber* feature file

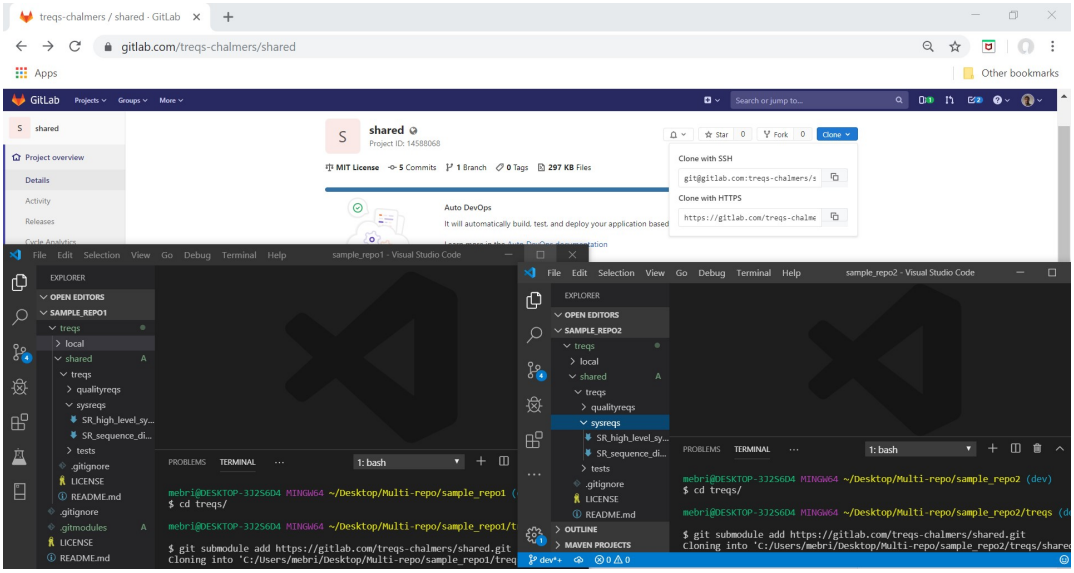


Figure A.16: Implementation of artifact sharing between multiple repositories using *git submodules*

	id	sysreqid	user_email	date_generated
▶	17	SR0001	guesh@student.chalmers.se	2019-11-05
	18	SR0002	guesh@student.chalmers.se	2019-11-05
	19	SR0003	guesh@student.chalmers.se	2019-11-05

(a) Database table for storing system requirement IDs

	id	qualityreqid	user_email	date_generated
▶	1	QR0001	guesh@student.chalmers.se	2019-10-29
	2	QR0002	guesh@student.chalmers.se	2019-10-29

(b) Database table for storing quality requirement IDs

Figure A.17: Sample database tables for storing artifact IDs in *T-reqs*

```
def generate_preview():
    """Generates a URL for PlantUML diagram.
    """
    md_files = set()
    if len(sys.argv) > 1:
        for file in sys.argv[1:]:
            md_files.add(file)
    else:
        stagedFiles = repo.index.diff("HEAD")
        for file in stagedFiles:
            filePath = file.a_path
            if filePath.endswith(".md"):
                md_files.add(filePath)
    for md_file in md_files:

        with open(md_file, "r", encoding="utf-8") as file:
            file_content = file.read()
            # Remove old trace links from PlantUML code
            file_content = re.sub(
                r'\s+' + re.escape('[[') + '.*?[0-9]{4}' +
                ↪  re.escape(']]'), "", file_content, re.DOTALL)
            # Remove old trace links from DOT language
            file_content = re.sub(
                r'\s+' + '.*?[0-9]{4}' + re.escape('[') + '.*?' +
                ↪  re.escape(']'), "", file_content, re.DOTALL)
            # Remove old generated diagram URL
            file_content = re.sub(r'\s+' + re.escape('![') +
            ↪  'Generated diagram' + re.escape(
                ']') + '.*?' + re.escape(')'), "", file_content,
                ↪  re.DOTALL)
            file_content = write_trace_links(file_content)
            # Extract PlantUML code blocks
            puml_content = re.findall(
                r'```puml(.*?)```', file_content, re.DOTALL)
            # Generate the digram URL for each PlantUML code block
        for puml in puml_content:
            diagram_url = get_url(puml)
            puml_with_url = '```puml'+puml + '```\n' +\
                "\n![Generated diagram]({})".format(diagram_url)
            file_content = file_content.replace(
                '```puml' + puml + '```', puml_with_url)

        with open(md_file, "w") as file:
            file.write(file_content)
        repo.index.add(([md_file]))
```

Listing 1: Code snippet for writing generated trace-links and diagram URLs to a file

X

```
 #: Default plantuml service url
SERVER_URL = 'http://www.plantuml.com/plantuml/svg/'

def deflate_and_encode(plantuml_text):
    """zlib compress the plantuml text and encode it for the plantuml server.
    """
    zlibbed_str = zlib.compress(plantuml_text.encode('utf-8'))
    compressed_string = zlibbed_str[2:-4]
    return encode(compressed_string.decode('latin-1'))

def encode(data):
    """encode the plantuml data which may be compresses in the proper
    encoding for the plantuml server
    """
    res = ""
    for i in range(0, len(data), 3):
        if (i+2 == len(data)):
            res += _encode3bytes(ord(data[i]), ord(data[i+1]), 0)
        elif (i+1 == len(data)):
            res += _encode3bytes(ord(data[i]), 0, 0)
        else:
            res += _encode3bytes(ord(data[i]), ord(data[i+1]), ord(data[i+2]))
    return res

def _encode3bytes(b1, b2, b3):
    c1 = b1 >> 2
    c2 = ((b1 & 0x3) << 4) | (b2 >> 4)
    c3 = ((b2 & 0xF) << 2) | (b3 >> 6)
    c4 = b3 & 0x3F
    res = ""
    res += _encode6bit(c1 & 0x3F)
    res += _encode6bit(c2 & 0x3F)
    res += _encode6bit(c3 & 0x3F)
    res += _encode6bit(c4 & 0x3F)
    return res

def _encode6bit(b):
    if b < 10:
        return chr(48 + b)
    b -= 10
    if b < 26:
        return chr(65 + b)
    b -= 26
    if b < 26:
        return chr(97 + b)
    b -= 26
    if b == 0:
        return '-'
    if b == 1:
        return '_'
    return '?'

def get_url(plantuml_text):
    """Return the server URL for the diagram.

    :param str plantuml_text: The plantuml markup to render
    :returns: the plantuml server diagram URL
    """
    return SERVER_URL + deflate_and_encode(plantuml_text)
```


Listing 2: Code snippet for generating diagram URL for PlantUML/DOT language

A.2 Data collection

The following figures show the interview and focus group questions and the survey questionnaires used in this study.

Semi-structured interview questions – problem identification in iteration I

Broad questions

1. Tell me about your experience in using modeling for requirements engineering
2. Tell me anything you know about the T-reqs tool

In-depth questions

1. Do you think modeling in T-reqs is a problem?
2. If you think modeling in T-reqs is a problem, could you explain why it is a problem?
3. Do you think textual UML tools like PlantUML can be well integrated into T-reqs?
4. What other techniques do you think can be promising for integrating modeling in the T-reqs tool?
5. What stakeholders do you think can be of interest for implementing modeling support in the T-reqs tool?
6. What do you think are some of the concerns/interests of the stakeholders in the modeling support of T-reqs?
7. What other limitations/problems of the T-reqs tool do you know?

Figure A.18: Interview questions for identifying problems in Iteration I

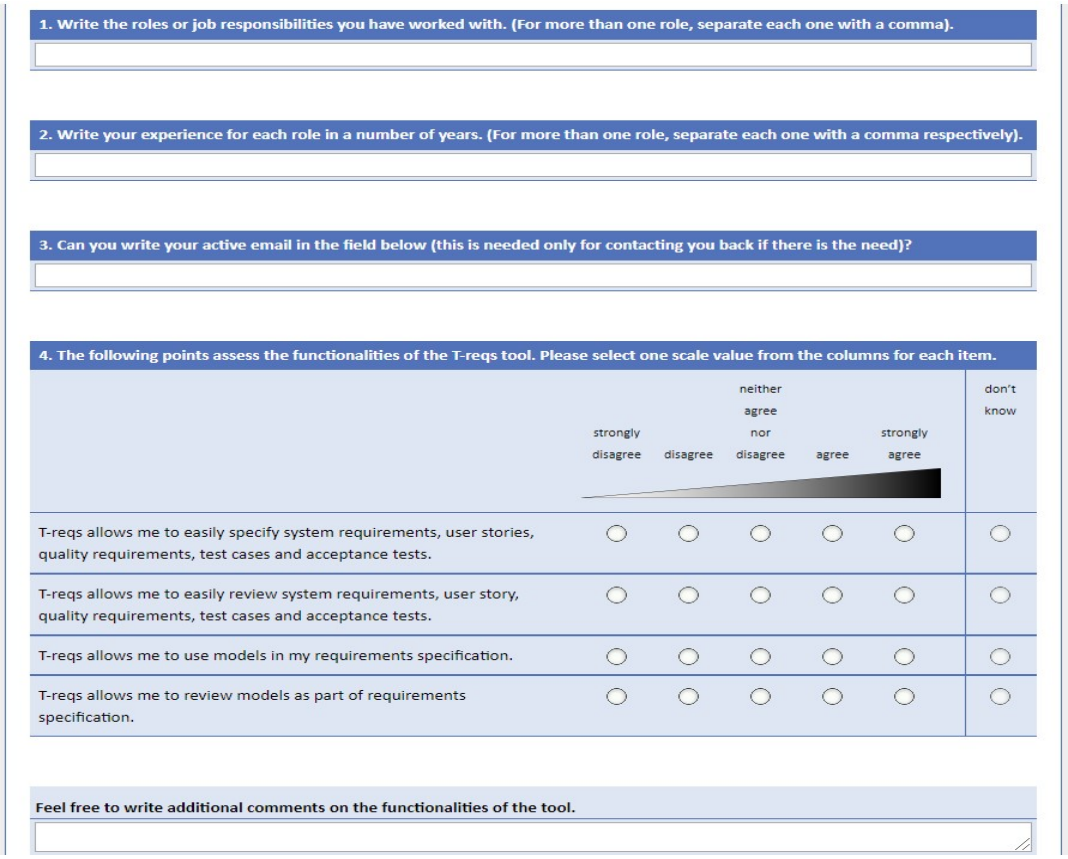
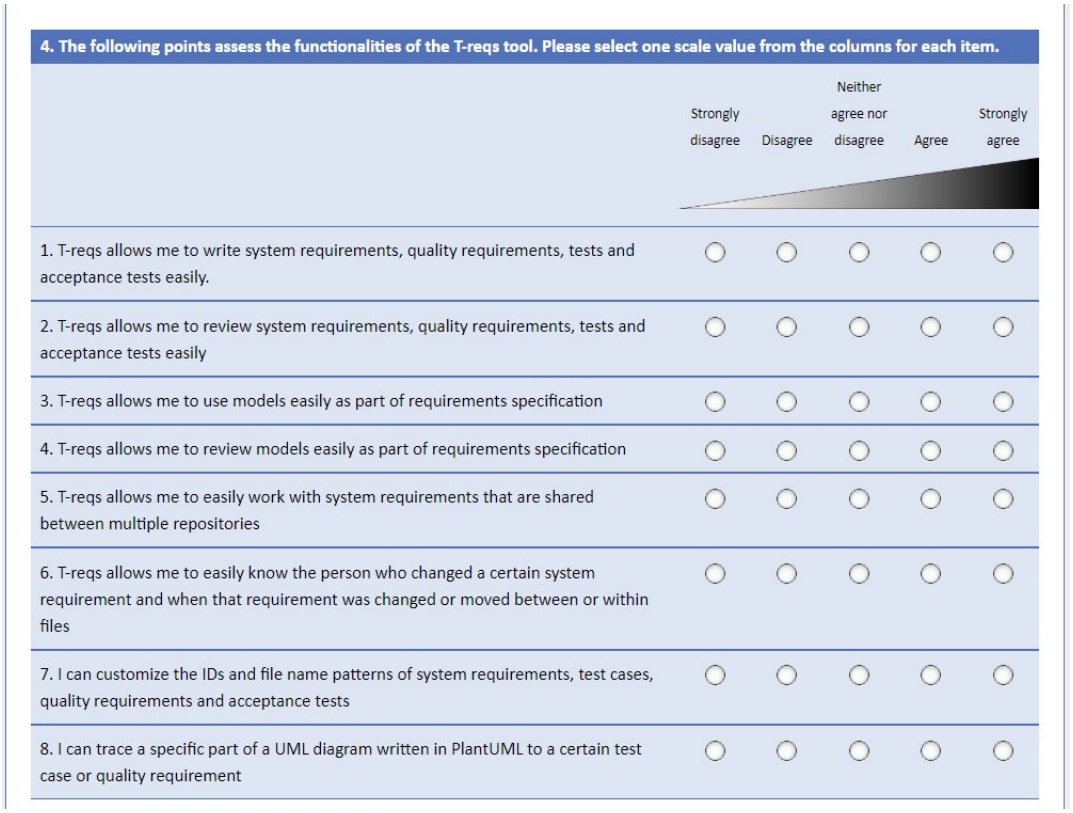
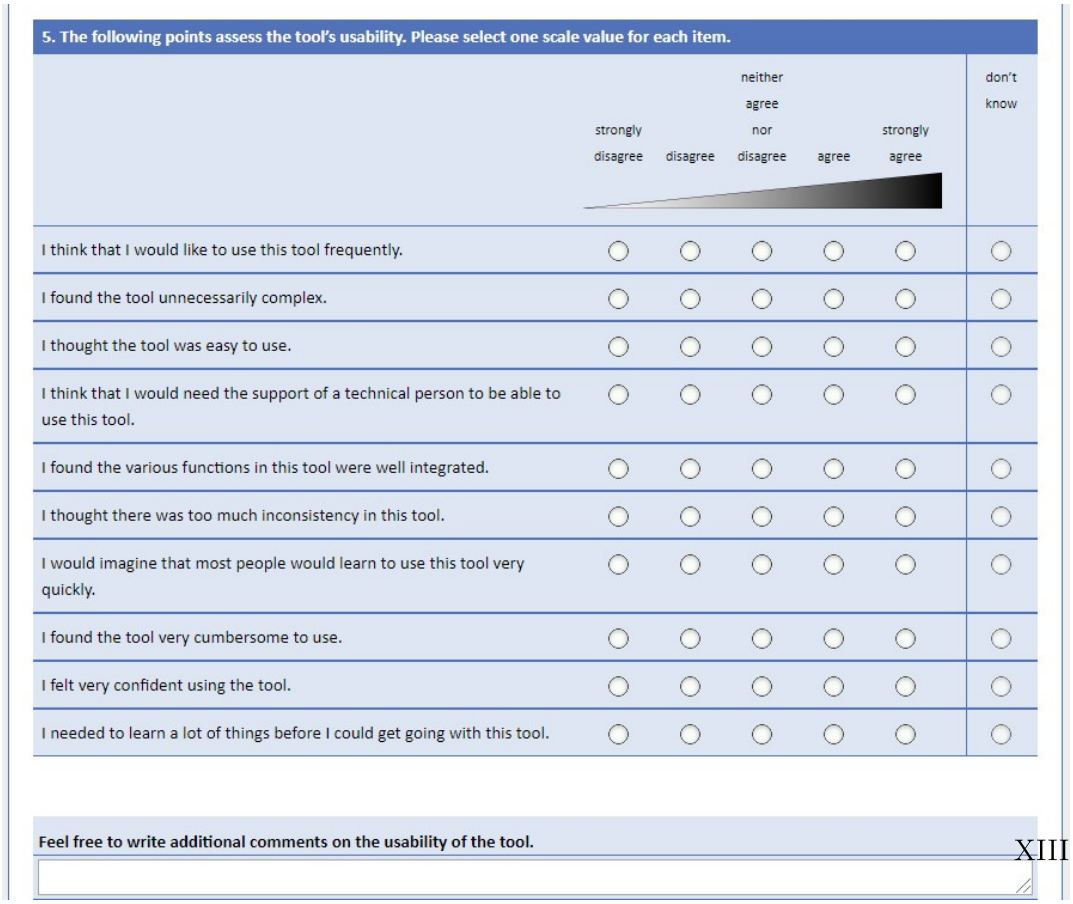


Figure A.19: Iteration I survey questions - T-reqs functionality assessment



4. The following points assess the functionalities of the T-reqs tool. Please select one scale value from the columns for each item.	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
1. T-reqs allows me to write system requirements, quality requirements, tests and acceptance tests easily.	○	○	○	○	○
2. T-reqs allows me to review system requirements, quality requirements, tests and acceptance tests easily	○	○	○	○	○
3. T-reqs allows me to use models easily as part of requirements specification	○	○	○	○	○
4. T-reqs allows me to review models easily as part of requirements specification	○	○	○	○	○
5. T-reqs allows me to easily work with system requirements that are shared between multiple repositories	○	○	○	○	○
6. T-reqs allows me to easily know the person who changed a certain system requirement and when that requirement was changed or moved between or within files	○	○	○	○	○
7. I can customize the IDs and file name patterns of system requirements, test cases, quality requirements and acceptance tests	○	○	○	○	○
8. I can trace a specific part of a UML diagram written in PlantUML to a certain test case or quality requirement	○	○	○	○	○

Figure A.20: Iteration I & II survery questions-T-reqs functionality assessment



5. The following points assess the tool's usability. Please select one scale value for each item.	strongly disagree	disagree	neither agree nor disagree	agree	strongly agree	don't know
I think that I would like to use this tool frequently.	○	○	○	○	○	○
I found the tool unnecessarily complex.	○	○	○	○	○	○
I thought the tool was easy to use.	○	○	○	○	○	○
I think that I would need the support of a technical person to be able to use this tool.	○	○	○	○	○	○
I found the various functions in this tool were well integrated.	○	○	○	○	○	○
I thought there was too much inconsistency in this tool.	○	○	○	○	○	○
I would imagine that most people would learn to use this tool very quickly.	○	○	○	○	○	○
I found the tool very cumbersome to use.	○	○	○	○	○	○
I felt very confident using the tool.	○	○	○	○	○	○
I needed to learn a lot of things before I could get going with this tool.	○	○	○	○	○	○

Feel free to write additional comments on the usability of the tool.

Figure A.21: Survey questions - T-reqs usability assessment

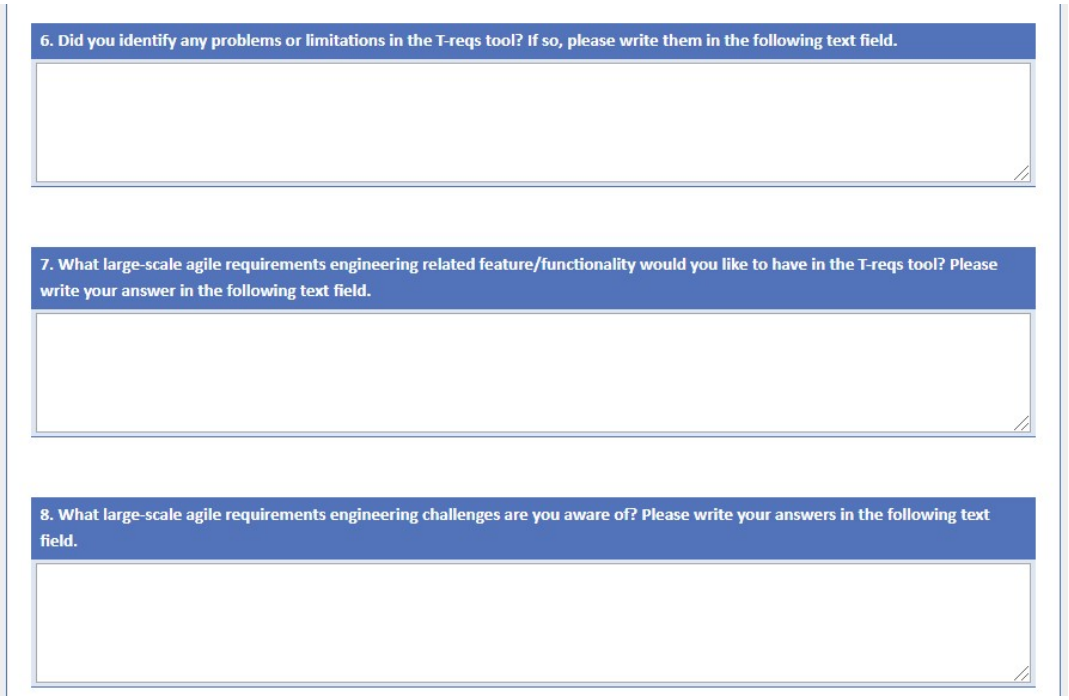


Figure A.22: Survey questions - General comments about T-reqs

Focus Group 1: Questions

1. What tooling solutions can be used to propagate requirement changes to different repositories?
2. How can be the change history of requirements be tracked in T-reqs given that the artifacts are stored in files under git repositories?
3. How can unique IDs be concurrently generated in T-reqs?

Figure A.23: Focus group 1 questions

Focus Group 2: Questions

1. What do you think of the model-support solution in T-reqs?
2. What do you think of the multi-repository solution in T-reqs?
3. What do you think of the system requirements change history solution in T-reqs?

Figure A.24: Focus group 2 questions