TRACEABILITY OF REQUIREMENTS IN SCRUM SOFTWARE DEVELOPMENT PROCESS

Manvisha Kodali
mki14001@student.mdh.se

Examiner: Mikael Sjödin
Mälardalen University, Västerås, Sweden

IDT supervisor: Alessio Bucaioni
Mälardalen University, Västerås, Sweden

Company supervisor: Peter Johansson
Westermo, Västerås, Sweden

June 9, 2015
Acknowledgements

I would like to express my sincere gratitude to my supervisors Peter Johansson and Alessio Bucaiioni for their continuous support and extraordinary motivation all through my master’s thesis. They are great mentors and had provided me with guidance, encouragement and clarity to work on my research. I thank them for allowing me to explore the challenging world of Software Engineering and for always finding the time to discuss the difficulties along the way.

I am grateful for the opportunity to work on the thesis in cooperation with the industrial partner Westermo. This helped me to observe the application of the knowledge, gained throughout my studies in a real industrial environment, which have certainly enriched my professional experience. I would like to thank Westermo employees and consultants especially Jon-Olov Vatn and Jonas Nylander for their guidance and support throughout my work. My sincere thanks to Per Strandberg for helping me think through specific problems in the right way. I am fortunate to have numerous technical discussions with him from which I have benefited enormously. I also acknowledge and thank MDH for providing me scholarship in the form of full tuition fee waiver.

I would like to thank Václav Struhár, my fellow class mate for his help and valuable suggestions, Bahodir Karshibayev, Valerio Lucantonio, Sudhangathan BS, Nandu Sreevalsan, Thanos Stratis and Slobodanka Cenevska for sharing their pearls of wisdom with me during my masters program. I would like to extend my thanks to every friend and associate who has helped me in doing this thesis and achieving a major milestone in my academic career.

I dedicate this thesis to my parents, grand-parents and to the rest of my family for their wonderful guidance always. My sincere thank you to my boy-friend Bajibabu Bollepalli, who has always supported and believed in me.
Abstract

Incomplete and incorrect requirements might lead to sub-optimal software products, which might not satisfy customers’ needs and expectations. Software verification and validation is one way to ensure that the software products meets the customers’ expectations while delivering the correct functionality. In this direction, the establishment and the maintenance of traceability links between requirements and test cases have been appointed as promising technique towards a more efficient software verification and validation.

Through the last decades, several methodologies supporting traceability have been proposed, where most of them realize traceability by implicitly exploiting existing documents and relations.

Nevertheless, parts of the industry is reluctant to implement traceability within software development processes due to the intrinsic overhead it brings. This is especially true for all those light-weight, code-centric software development processes, such as scrum, which focus on the coding activities, trying to minimizing the administrative overhead. In fact, the lack of documentation finishes to hamper the establishment of those trace links which are the means by which traceability is realized.

In this thesis, we propose a methodology which integrates traceability within a scrum development process minimizing the development effort and administrative overhead. More precisely we i) investigate the state-of-the-art of traceability in a scrum development process, ii) propose a methodology for supporting traceability in scrum and iii) evaluate such a methodology upon an industrial case study provided by Westermo.

**Keywords:** Scrum Development, Requirements Traceability, Requirements, Test Cases.
# List of Figures

1. Software Development Life cycle ........................................... 10
2. Water-fall SDP ................................................................. 11
3. Scrum SDP ................................................................. 13
4. Burndown chart .............................................................. 15
5. Research method ............................................................. 16
6. A traceability information model for a basic agile project [1] .............. 19
8. Proposed method .............................................................. 22
9. Final RTM with all relations and criteria along with different colours .... 27
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Product backlog</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Sprint backlog</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Quality Checklist</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>RT techniques in agile or Scrum with their references</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Planned requirement IDs (R1, R2, R3, R4) and test case IDs (T1, T2, T3, T4) and their relations</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Developers tag the code with requirement IDs (R2, R3, R4, R5) and test case IDs (T1, T2, T3, T4, T5) and their relations</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Test results and their test case IDs (T1, T2, T3, T5)</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>RTM with all requirement IDs (R1, R2, R3, R4, R5) and testcase IDs (T1, T2, T3, T4) from requirement’s list and test case source code</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>RTM with all requirement IDs (R1, R2, R3, R4, R5) and test case IDs (T1, T2, T3, T4, T5) from the requirements list and test case source code</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>Sample requirement IDs and their short description for VRRP-SNMP-MIB project</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>Planned autotest IDs (T1, T2, T3, T4) and their description</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>Planned requirement IDs (R1, R2, R3, R4, R5, R6) and testcase IDs (T1, T2, T3, T4, T5) and their relation</td>
<td>26</td>
</tr>
<tr>
<td>13</td>
<td>Test cases results</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>Quality checklist</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Algorithm for tagging test case ID and requirements IDs in test case(s) source code 26
2. Algorithm for python script ................................................. 27
List of Acronyms

CoEST  Center of Excellence for Software Traceability
DSML  Domain-Specific Modelling Language
ID  Integrating Documents
IR  Information Retrieval
JITT  Just-in-time Traceability
LSI  Latent Semantic Indexing
MDE  Model Driven Engineering
MIB  Management Information Base
NFR  Non-Functional Requirements
PM  Probabilistic Model
RAD  Rapid Action Development
ROMPL  Requirement-Oriented Modeling and Programming Language
RQ  Research Question
RT  Requirement Traceability
RTM  Requirement Traceability Matrix
RUP  Rational Unified Process
SDD  Story-Driven Development
SDLC  Software Development Life Cycle
SDP  Software Development Process
SNMP  Simple Network Management Protocol
SRS  Structured Requirements Specification
TDD  Test-Driven Development
TIM  Traceability Information Model
TmM  Traceability meta-model
TPM  Traceability Process Model
VRRP  Virtual Router Redundancy Protocol
VSM  Vector Space Model
# Table of Contents

1 Introduction .................................................. 8
   1.1 Problem Formulation ........................................ 8
   1.2 Thesis contribution ........................................ 8
   1.3 Thesis outline ............................................ 9

2 Background .................................................. 10
   2.1 SDP ......................................................... 10
      2.1.1 Linear SDPs ........................................... 11
      2.1.2 Iterative SDPs .......................................... 12
   2.2 Agile SDPs ................................................ 12
      2.2.1 Scrum ................................................ 12
   2.3 RT ......................................................... 14

3 Research Method ............................................. 16

4 Systematic Review ........................................... 18
   4.1 RTM ......................................................... 18
   4.2 Cross-reference ........................................... 18
   4.3 Domain specific modeling .................................. 18
   4.4 Tagging ..................................................... 19
   4.5 Integrating Documents(ID) ................................. 19
   4.6 Information Retrieval(IR) ................................ 20
   4.7 Other techniques .......................................... 20

5 Proposed Method ............................................. 22

6 Case study .................................................... 25

7 Discussions .................................................. 28

8 Conclusion and Future Work ................................. 29

References ...................................................... 32

Appendix A Quality checklist .................................. 33
1 Introduction

Software plays a major role in this era. In our society, the usage of software is rapidly growing as our daily life relies on and is affected by software. Software is used in several sectors (e.g., industries, government organizations, educational systems) within different kinds of projects (e.g., small-size, medium-size and large-size). The software is intangible (i.e., we can’t touch or perceive it) and it doesn’t hold any physical properties (e.g., mass, strength, form, etc.) [3]. Therefore the development of a software product is very different from any other product manufacturing [4].

In the past decades, several software development processes (SDPs) have been defined (i.e., linear/plan driven SDPs, incremental/iterative SDPs) where each of them is characterized by different phases and outcomes and tailored for different needs. Within the linear SDPs, as the waterfall SDP, the development phases are sequential and documentation is created at the completion of each and every phase. In contrast, some iterative SDPs, like agile SDP, are code-centric, meaning that they mostly focus on the software implementation minimizing the related documentation [5].

Besides which SDP is elicited, a close linkage among software development phases has been proven to be a key factor in achieving successful software development [6]. The ability to trace an artifact through the SDP, i.e., traceability, represents one of the key challenges towards such goal. More precisely, according to [7], Requirement Traceability (RT) is the ability to describe and follow the life of a requirement in both forward and backward direction within the whole software life-cycle. RT is appealing for companies as it facilitates the requirements validation, by tracing or linking the requirements to test cases [8].

RT is commonly achieved using formally Structured Requirements Specification (SRS) documents. SRS documents act as a glue artifacts, explicitly relating requirements to other software artifacts [6]. Depending on which SDP is chosen, RT might be easier or harder to achieve with respect to the document production. For instance, in a waterfall SDP, SRS documents are created before starting the software design phase, since all the requirements are clearly elicited in the early stages [9]. In contrast, in an agile SDPs, like scrum SDP, due to requirements changes, documentation is not created prior to the software implementation phase [10].

Such a lack of documentation, affects the following activities such as change management, impact analysis and project estimation [11]. Thus, RT is a major problem for the scrum SDP [7].

1.1 Problem Formulation

Within a software project, customer satisfaction depends upon how well the intended requirements are implemented in a project. Thus, verification and validation (V&V) are crucial activities for delivering high quality software products. RT is one way to improve software verification and validation activities by relating requirements to test cases and, more generally, to other software artifacts.

RT is particularly hard to achieve in all those SDPs which are light-weight and code-centric, as scrum [4].

In this respect, it is particularly important to integrate the RT within the scrum SDP. There are several literature studies on how to achieve RT within a SDP [12], where most of them claim that RT is not extensively used due to it’s intrinsic overhead. In this thesis, we investigate how to achieve RT within a scrum SDP, minimizing the related development effort, administrative overhead and maintenance.

Two research questions (RQs) are identified for this thesis work.

\textbf{RQ1} : How can we integrate RT within the scrum SDP, minimizing the related development effort, administrative overhead and maintenance?

\textbf{RQ2} : Is the scrum SDP resulting from RQ1 improving the current industrial practices?

1.2 Thesis contribution

In this thesis work, with respect to the aforesaid RQs, we :

- Investigate the state-of-the-art of how RT can be implemented within a scrum SDP.
• Improve the existing scrum SDP where RT is integrated.
• Evaluate the resulting scrum SDP upon an industrial case study.

1.3 Thesis outline

The rest of this thesis report is structured as follows.
Section 2 describes the background of this research work by reviewing fundamental concepts of SDP, agile SDP and RT. Section 3 presents the research method adopted for this work. Section 4 shows the review of the relevant techniques. Section 5 describes the approach proposed to integrate RT within scrum SDP. Section 6 illustrates the case study. Section 7 presents the discussions on the results and the threats to validity. Finally, the last Section 8 is devoted to give the conclusion future works to this thesis work.
2 Background

In this section, we present the background for this thesis work.

2.1 SDP

SDP is the method of developing and maintaining the software product. SDP is a critical activity that has to be carefully studied, understood, improved and supported [8]. SDP is unique from one software product to another. Software Development Life cycle (SDLC) is a framework comprising of the phases characterizing a general SDP. Figure 1 depicts a generic SDLC together with its main phases, i.e., Requirements Analysis and Definition, Software Design, Implementation and Unit Testing, Integration and Testing, Operation and Maintenance [13].

![Software Development Life cycle](image)

**Requirements Analysis and Definition.** This is the initial phase. Requirements (both functional and non-functional) are collected from users, validated, defined and prioritized. The output is the requirements specification [13].

**Software Design.** Starting from requirement specification, the software architecture is identified and described. Eventually, algorithms are identified and described too. The output is the software architecture [13].

**Implementation and Unit Testing.** Software architecture is the input for this phase. In this phase, a non-empty set of units, implementing the software architecture is developed. Each unit is tested and verified to meet its specification during unit testing. The output are the program units [13].

**Integration and Testing.** The input is the program units. These are integrated and tested as a whole software product to make sure that they meet the software requirements. The output is the verdict which informs whether or not the tests are passed. When all the tests are passed, then the final outcome is the software product [13].

**Operation and Maintenance.** The input is the software product. In this phase, the software product is delivered to the customer. Errors that are not detected during the testing phase before delivery are corrected, if an error is discovered that is critical enough to produce a new
release. Also, the software product is improved based on the customer’ usage feedback. Possibly, the output is the discovery of new requirements to enhance the software product [13].

SDPs are also characterized by roles (people involved in SDP) and artifacts (outcome of the phase). Consequently, the performance of a certain SDP is affected by the roles and behavior of individuals and organizations [13].

Different SDPs can be created by changing the flow of SDLC phases. Considering such a flow, SDPs can be divided into linear and iterative. Linear SDPs steadily follow the SDLC phases sequentially and they do not iterate. In contrast, iterative SDPs develop the software product in different iterations, where each iteration does not necessarily follow the phases as they were initially planned. The main difference between linear SDP (e.g., water-fall model) and iterative SDP (e.g., scrum) is that the testing phase is performed at different stages during SDLC. In water-fall model, testing phase is a separate phase that is executed after the completion of the implementation phase. Conversely, in scrum, testing phase is usually performed in parallel with the implementation phase.

2.1.1 Linear SDPs

Linear SDPs, also known as plan-driven SDPs, are SDPs where the development phases are followed sequentially (i.e., a phase can not start before the previous phase ends). For each phase, deadlines and budgets are estimated prior to the start of the phase. At the completion of each phase one or more documents are produced. These documents are used as input to the next phase.

Figure 2 shows the diagrammatic representation of a linear SDP, namely water-fall. Accordingly the Design Specification Document is created at the end of the Design phase and it is used as input for the Coding phase. These documents facilitate the management and the control of the development process. Nevertheless, their production is time-consuming and costly. Due to the phases flow, changes in the software product can not be addressed along the development without starting from the top again.

To sum up, in linear SDPs, the process is visible and easily understandable. For the aforesaid reasons, linear SDPs must not be used in projects where requirements are not known at the beginning of the project. Also, given the production of documents for each phase, the SDPs are not suited for low budget and small-size projects.

![Water-fall SDP Diagram](image-url)
2.1.2 Iterative SDPs

Iterative SDPs, also known as incremental SDPs, are SDPs, where each phase can occur multiple times and overlaps other phases. A series of different versions of a software product is produced and delivered to the customer. These version are also used for internal validation of the software product. Improvements and changes detected in a certain iteration, are addressed within the next release of the software. This process occurs continuously until the customer is satisfied with the software product [13].

Unlike linear SDPs, documents are not produced after completion of each phase. Therefore, considering the lack of document production, iterative SDPs might be hard to manage and control [15].

Iterative SDPs can be categorized in heavy-weight (e.g., spiral SDP) and light-weight (e.g., agile SDP). Spiral SDPs follow rapidly evolving iterative SDPs by managing risks to reduce in future. More precisely, based on the unique risk patterns of a given project, spiral SDP guides a team to adopt elements of one or more SDPs. Consequently, spiral SDPs might be hard to plan and follow. Spiral SDPs are suitable for large team sizes and for non-critical systems [13].

In agile SDPs, the requirements and the solutions evolve through collaboration between self-organizing and cross functional teams. Indeed, agile SDPs accelerates adaptive planning, evolutionary development, early delivery, continuous improvement and flexible response to change [14]. Agile SDPs are explained in detail in 2.2. Some other examples of iterative SDPs are prototyping, Rapid Action Development (RAD), Rational Unified Process (RUP).

Summing up, iterative SDPs are tailored for all those software projects which are prone to changes, as they can be easily recognized and addressed within the next process iteration. Regular software deliverables are used to measure the progress of the projects. In contrast, as many of the phases are run in parallel, big projects can be hard to monitor.

2.2 Agile SDPs

Agile SDPs are meant to support fast software projects. This SDP provides the customer with the view of the product before and as each complete functionality is delivered. In 2001, the agile manifesto, a document collecting the fundamental principles of this SDP, was released [14]. The agile SDP is organized in iterations ending with a code release. Agile SDPs were thought for the development of high quality software, which required low cost and short development time [15]. Agile SDPs adoption has several benefits mainly related to the ability of these SDPs to easily address requirement and environment changes during the software development.

Nonetheless, there are certain limitations in using agile SDPs [15]. For instance, it is hard to arrange meetings for globally distributed development environments. Also, there is no support for subcontracting, it is hard to build reusable artifacts and it has limited support for large teams and complex software.

There are several examples of agile SDPs each of which is characterized by its own practices and emphasizes different issues [16]. Versionone\(^1\) surveyed different agile SDPs. The results reveal that 73% of the software projects are realized by means of scrum SDP (or/and scrum variants). Also, it shows that scrum is the most popular SDP as it is used in the 55% of the software projects [17].

2.2.1 Scrum

Scrum is an agile SDP formalized by Jeff Sutherland and Ken Schwaber [18] in 1995 and it is particularly suited for small and low budget software projects. As all the agile SDPs is based on iterations. In scrum one iteration typically lasts for 1-4 weeks. Also, scrum is characterized by roles, and artifacts. Figure 3 shows roles, artifacts, ceremonies and their relation within an iteration.

Roles

\(^1\)http://www.versionone.com/.
**Team:** A team can be composed of, up to, 8 members. A member of the team is called developer and has specific priorities and a single focus. They select the sprint goals and commit all necessary tasks during the sprint.

**Scrum master:** The scrum master has to ensure that the cooperation within the team and product owner is proficient and that the team follows the process as established.

**Product owner:** The product owner sets features, release dates, contents and priorities for each sprint. Product owner accepts or rejects the software product and is responsible for the profitability of the product.

---

**Ceremonies**

**Sprint planning:** In this ceremony, the detailed plan for the upcoming iteration is developed. The team is responsible for deciding the load of work for each iteration. The work is defined in terms of features to be realized; these features are taken from the product backlog. The resulting artifact is a sprint backlog.

**Daily Scrum meeting:** As the name suggest, such a ceremony is a daily meeting which aims in understanding the status of the project. The meeting can not exceed 15 minutes and is usually driven by pre-defined questions like: "what did I do yesterday?", "what do I plan to do today?" and "are there any obstacles?".

**Sprint review:** Within this ceremony, the team shows the implemented functionalities to the product owner. Each accepted functionality is removed from the product backlog. Eventually, during the sprint review, priorities and efforts are tuned based on the available funding.

**Sprint retrospective:** Within this ceremony, the team and the scrum master analyzes the sprint for eliciting possible improvements.

---

**Artifacts**

**Product backlog:** Product backlog is a sorted list of features, describing all functionalities of the software product and has an effort value assigned by the team. It also contains bugs to be fixed, knowledge acquisition and technical work. The product owner will add, remove or re-prioritize them. Table 1 depicts a product backlog of WeOS\(^3\) product at Westermo\(^4\). The leftmost column indicates the unique ids of all the rows. Column ‘A’ represents the high-level requirements, column ‘B’ indicates the list of all low-level requirements that belongs to each high-level requirement. The columns ‘C’, ‘D’ and ‘E’ represents the estimated time\(^5\) for completing each low-level requirement.

---

\(^2\)https://javamaster.wordpress.com/2009/07/24/scrum-based-task-management/

\(^3\)WeOS is an Operating System developed by Westermo that runs on many of Westermo’s network products.

\(^4\)http://westermo.so/

\(^5\)It is measured in ideal working days where a working day is composed of 8 hours.
**Sprint backlog:** Sprint backlog is the resulting artifact of the sprint planning ceremony, which contains the tasks, from the product backlog which have to be completed within the sprint. Table 2 depicts the sprint backlog, which contains some low-level requirements of the product backlog in Table 1. The leftmost column indicates the unique ids of all the rows. Column ‘A’ indicates the sprint number and low-level requirements. Column ‘B’ indicates the effort (i.e., the number of man days) required for completing the related requirement. Column ‘C’ contains the name of the developer responsible.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>02Sep2014</td>
<td>26Sep2014</td>
<td>02Oct2014</td>
</tr>
<tr>
<td>2</td>
<td>RFIR-119</td>
<td>AC GUI(CLI, WEB, SNMP)</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>Follow up</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Bootloader</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>HW support(proof of concept)</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Design description</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Test list</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Test specification</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Bootloader audit</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>CLI/WEB/SNMP changes</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>WeOS config of Boot settings</td>
<td>2</td>
<td>2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 1: Product backlog

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Sprint 97</strong></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HW support(proof of concept)</td>
<td>Mattias, Tobbe</td>
</tr>
<tr>
<td>3</td>
<td>Bootloader audit</td>
<td>Jocke, Tobbe</td>
</tr>
<tr>
<td>4</td>
<td>CLI/WEB/SNMP changes</td>
<td>Tobbe, Johan</td>
</tr>
<tr>
<td>5</td>
<td>WeOS config of Boot settings</td>
<td>Tobbe, Johan</td>
</tr>
</tbody>
</table>

Table 2: Sprint backlog

**Burn-down chart:** The Burn-down chart shows the remaining work in the sprint. As the sprint goes on, the requirements in sprint backlog reduce or burn down. If the sprint backlog is zero, then the sprint is completed. In this context, burn-down chart helps in keeping track of the sprint backlog progression [19]. In Figure 4, the remaining effort is plotted on y-axis and on x-axis with the sprint number. The small bar graphs on the x-axis represents the completed tasks. The blue line shows the remaining time for each sprint. The yellow line denotes the ideal velocity. It is calculated by taking the estimated amount of work at the first sprint and dividing it by the number of sprints before the deadline. Remaining tasks that must be completed in that sprint. The red line is the average velocity and is recalculated after each sprint.

In Figure 3, the initial phase is the product backlog prioritization and the final phase is the shippable product. Between these phases, the sprint review and retrospective takes place. The product backlog, sprint backlog and also the backlog items are discussed in the sprint planning meeting. With the on-going sprints, the feedback is collected and so changes can be adapted [20]. The product is delivered as a group of artifacts for each sprint. In the final sprint, all produced artifacts are wrapped up and delivered as the final product.

### 2.3 RT

Requirements are considered as the core artifact in the whole SDP. Therefore, collecting and controlling the requirements are significant activities within a SDP. The need to initiate and maintain
traceability links to other artifacts arises as soon as the requirements are identified, and the SDPs are chosen. In this new emerging and fast growing technology, it is necessary to accept changes in the requirements. The quality and success of a software product depends on the fulfillment of all the requirements within defined cost and time [21]. To this end, RT provides an essential support to produce secure and high-quality software systems [6].

RT is defined as the ability to describe and follow the life of a requirement in both a forwards and backwards direction within the whole SDP i.e, from requirements, design, implementation, to testing and maintenance [7].

The importance of RT is stressed in [22]. One of the key benefits of RT is that the correlation of requirements with other artifacts is visible. It is possible to make sure that all the requirements are implemented and it ensures better project management [23]. RT can also be used to analyze the impact of change and to estimate the effort required to implement it.

There are challenges in achieving RT. These challenges are summarized by the Center of Excellence for Software Traceability (CoEST) [24]. Some of them are:

1. requirements must be traceable to verify coverage, to prevent redundancies or to assess impact of change,
2. RT has to be maintained as the system evolves,
3. RT must be available and accessible at run-time,
4. RT has to be visible at different levels of abstraction [25].

Examples of RT techniques include: cross referencing schemes, RT matrices, graph-based representation, key phrase dependencies, hypertext and Integration documents. There are also commercial tools such as IBM Rational Rose XDE, IBM Rational RequisitePro and IBM Telelogic DOORS that can assist to some extent, however often requires a change of the way of working.
3 Research Method

Figure 5 depicts the research method used for this thesis work. The research method comprises of the following five activities: problem formulation, identification of RQ(s), creation of solution(s), implementation and solution(s) validation.

![Figure 5: Research method](image)

The research started with a survey of the background aiming at identifying a clear research problem. In the next phase, the elicited problem has been defined in terms of two RQs discussed in Section 1.1. In order to answer RQ1, we conducted a systematic review of the state-of-the-art of RT in scrum with the following protocol.

1. Starting from RQ1, we derived the following research string: (("requirements" OR "user requirements" OR "system requirements" OR "functional requirements" OR "non functional requirements") AND ("traceability" OR "traceability links") AND ("lightweight" OR "agile" OR "scrum")).

2. With the aforesaid search string, we exercised the following databases:
   - ACM Digital Library
   - IEEE Xplore Digital Library
   - Google Scholar
   - Springer Link

3. We applied the inclusion criteria listed below:
   - The articles are written in English.
   - The articles are longer than two pages.
   - The articles contain at least one search item in the metadata.
   - The articles refer to the RT in the title or in the abstract.

4. Eventually, the quality checklist showed in Table 3 has been used to strengthen and complement the aforesaid inclusion criteria [27].
Table 3: Quality Checklist

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Does the study consider a specific RT technique?</td>
</tr>
<tr>
<td>2</td>
<td>If the study refers to an RT technique, is it industrial or academic?</td>
</tr>
<tr>
<td>3</td>
<td>Does the study use a case study?</td>
</tr>
<tr>
<td>4</td>
<td>Does the study consider any tool?</td>
</tr>
<tr>
<td>5</td>
<td>Does the study consider threats in using RT tool or technique?</td>
</tr>
</tbody>
</table>

Appendix A shows the list of the identified works together with the quality checks for each of these works.

The findings of the systematic review have been used for driving the development of a methodology for integrating RT within the scrum SDP minimizing administrative overhead. Finally, the proposed methodology has been implemented and tested within an industrial project, namely Virtual Router Redundancy Protocol (VRRP)-Management Information Base (MIB)\(^7\) project, provided by Westermo. That project enhances the existing VRRP functionality in WeOS with the option to read some data using Simple Network Management Protocol (SNMP)\(^8\).


\(^8\)SNMP is used to collect information from, and configure network devices.
4 Systematic Review

In this section we show the results of the systematic review which aims at answering to the RQ1. To this end, we identified the RT techniques which are currently used within a scrum, or agile SDP. Table 4 shows the RT techniques and the reference to the paper(s).

<table>
<thead>
<tr>
<th>RT technique</th>
<th>Reference for the paper(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement Traceability Matrix (RTM)</td>
<td>[28]</td>
</tr>
<tr>
<td>Cross-reference</td>
<td>[1, 29]</td>
</tr>
<tr>
<td>Tagging</td>
<td>[30]</td>
</tr>
<tr>
<td>Domain specific modelling</td>
<td>[6, 31]</td>
</tr>
<tr>
<td>Integrating documents</td>
<td>[2]</td>
</tr>
<tr>
<td>Information retrieval</td>
<td>[2]</td>
</tr>
<tr>
<td>Other techniques</td>
<td>[1, 32, 33]</td>
</tr>
</tbody>
</table>

Table 4: RT techniques in agile or scrum with their references

4.1 RTM

RTM is used to map requirements to other artifacts. It is a two dimensional grid that shows artifacts in rows and columns. RT links are represented as marks, or values, on the cell(s). More precisely, if a cell $c_{i,j}$ is marked, it means that the requirement of column $j$ and the artifact of row $i$ are linked.

According to our research, RTM represents the most common technique for realizing traceability [34]. RTM is simple to create and easy to understand when kept to a manageable size and used to associate two kinds of artifacts [22].

Duraisamy et al. [28], introduce a RTM technique to create RT links of requirements between product backlog and sprint backlog where the information are retrieved by using keyword searching functionality.

4.2 Cross-reference

According to this technique, each artifact is listed using natural language and it is provided with a list of links to other, related, artifact(s) [35]. A cross reference is often denoted as: See also$^9$.

Cleland-Huang et al. [1] described Traceability Information Model (TIM) shown in Figure 6. It shows a basic RT technique for agile projects which establishes traces between acceptance tests and user stories$^{10}$. RT is build by inserting a cross-reference to one or more user story into each acceptance test. when the test cases are executed and passed, RT links are automatically created between the source code and test cases, implicitly. However, in this RT method, the code is treated as a single high-level target artifact, and there is no information regarding which code fragment is related to which test cases. This technique is currently supported by Rally software$^{11}$.

Ratanotayanon et al. [29], designed a tool, namely Zelda [29], to work with agile SDP which captures and maintains links between high-level information and source code. This tool is an Eclipse Modeling Framework$^{12}$ plug-in that helps developers in creating links from user stories and tasks to source code, test cases and various text-based files.

4.3 Domain specific modeling

Some researchers attempted to investigate the cross play of Model Driven Engineering (MDE) and requirements engineering. In this direction, Badreddin et al. [25] proposed a Requirement-Oriented Modeling and Programming Language (ROMPL). In this technique, the authors provide a

---


$^{10}$Here user stories is same as requirements.


$^{12}$http://www.eclipse.org
textual syntax for requirements concepts and integrate them within model-oriented code. ROMPL support the automatic generation and maintenance of the RT links between requirements, models, and code.

Taromirad et al. [31] proposed an RT approach in MDE through a domain-specific modeling. The author suggested to use Domain-Specific Modelling Language (DSML) to build a traceability scheme for a specific domain or project, incrementally. The idea here is to describe a domain-specific RT meta-model and represent the structures, behaviors and features of the target domain with respect to the project RT goals. Then, MDE tooling is used to create RT models and help to evolve models and meta-models to project-specific.

Espinoza et al. [6] introduce an approach with higher degree of automation namely, traceability meta-model (TmM). This extend’s Ramesh’s previous work on metamodels [36]. Espinoza advocates creating RT fit for purpose through allowing user-defined RT links, well defined roles and linkage rules which specify what kinds of RT links can and must be created. Practically, it is adapted to the agile SDPs like Test-Driven Development (TDD) and Story-Driven Development (SDD). From the best of our knowledge, the TmM model has not been validated yet for scrum software development process.

Other, worth to mention, researches using domain-specific modeling techniques are [37, 38].

4.4 Tagging

Tag\(^{13}\) is a non-hierarchical keyword or term assigned to a piece of information. Each element in a artifact must have a unique identifier. In RT tagging however, those identifiers are used as tags in the subsequent artifacts to identify backwards tracing to the predecessor artifact [39].

Jakobsson et al. [30] conducted a series of interviews with different people and scrum roles. Based on the interview results, he identified several kind of RT links. For instance, for RT link between requirements and tests, he proposed to mark a test with a requirements id or adding a test id to a requirement (and use both for a bi-directional tracing connection).

4.5 Integrating Documents(ID)

Banka et al. [2] collected data by means of a survey focused on software companies who undertakes different roles and responsibilities in testing field. Based on their survey and research findings, they concluded to merge requirements and test cases in one document. To this end, they have developed a template and described the step-by-step procedure used for creating that template, which can be used for the requirements and testing in an agile SDP. Figure 7 depicts the improved template for requirements and test cases.

4.6 Information Retrieval (IR)

IR techniques such as Latent Semantic Indexing (LSI) [40], Probabilistic Model (PM) and Vector Space Model (VSM) [41] index the documents in a document space as well as the queries by extracting information about the occurrences of terms within them. Often, the output is described by a term-by-document matrix. The entry $a_{i,j}$ of this matrix denotes a measure of the weight of the $i$th term in the $j$th document [42].

Cleland-Huang et al. [43], use the information retrieval technique to automatically create candidate RT links between user stories and code. Just-in-time Traceability (JITT) tool provides an interactive environment for retrieving potentially relevant code. The tool returns a list of candidate classes and shows their likelihood that the retrieved class is related to the user story.

4.7 Other techniques

Cleland-Huang et al. [1] introduced an approach to extract RT information from the check-in transactions of version control systems. This approach establish more finely grained RT links from requirements and test cases to code.
Ghazarian et al. [32] introduced traceability patterns to achieve RT through source code. This traceability patterns use a combination of natural language, pseudo-code, string patterns, and code snippets written in a Java-like programming language.

Appleton et al. [33] described several additional techniques for lightweight tracing. For instance, one technique uses existing configuration management tools in order to capture RT links. The requirements, design, code and test cases would all be a part of the same task, and could be tracked to a single transaction ID in the change management system. Thus, RT links are automatically produced into the code at the granularity of the check-in units.

Arbain et al. [44] proposed a Traceability Process Model (TPM) to trace Non-Functional Requirements (NFR) with special focus on those expressing security and performance. This model combines TIM and TmM.

Some other, worth to mention technique, is [45].
5 Proposed Method

The method combines four already existing techniques, namely RTM [28], tagging [30], IR [43] and ID [2]. Each adopted technique is tailored to ensure a smooth interoperability with the other techniques. In particular, the tagging technique is used for assigning IDs to requirements and test cases. IR uses the created tags to generate all the potential links between requirements and test cases. ID is used for merging requirements and test cases IDs. Finally, the RTM is populated accordingly. Also, we identified precise relations between the task composing the methodology and the scrum roles who should perform these tasks. To the best of our knowledge, the method discussed in this work represents the first attempt in merging different RT techniques with the aim of providing a more extensive RT support within a scrum SDP. Figure 8 depicts the proposed method.

Figure 8: Proposed method

It comprises of 5 steps, namely identify requirements, test cases and assign IDs to them (marked as 1 in Figure 8), tag test case code with IDs of requirements and test cases (marked as 2 in Figure 8), collect test results with different criteria (marked as 3 in Figure 8), retrieve all IDs of requirements, test cases and create RTM (marked as 4 in Figure 8) and populate RTM with relations and criteria (marked as 5 in Figure 8).

**Step 1:** In this step, the scrum master has to i) identify the requirements and the test cases for the given project, ii) assign unique IDs and iii) store this information in a list. The assignment of IDs can be performed either automatically, using scripts and already existing tools, or manually. The list can be stored either in a file or in a database. Table 5 shows an example of such a list for a software project composed of four requirements and three test cases. Please note that, the relation between a requirement and a test case can be n:m

<table>
<thead>
<tr>
<th>Planned requirements</th>
<th>Planned test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>T2, T3</td>
</tr>
<tr>
<td>R2</td>
<td>T1</td>
</tr>
<tr>
<td>R3</td>
<td>T4, T2</td>
</tr>
<tr>
<td>R4</td>
<td>T1, T4</td>
</tr>
</tbody>
</table>

Table 5: Planned requirement IDs (R1, R2, R3, R4) and test case IDs (T1, T2, T3, T4) and their relations

**Step 2:** Once the implementation phase has started, the developers have to relate the code implementing a specific test case, to the related test case ID and the ID(s) of the related requirement(s).
Please note that in this phase, new requirements and test cases can be defined. For instance, it can be noted that the test case T5 is associated to a newly added requirement, namely R5, see 6. Also, it can be noted that a new test case, T5, has been added.

<table>
<thead>
<tr>
<th>Requirement IDs</th>
<th>Implemented in test case(s) IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>T3</td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>T2, T4</td>
</tr>
<tr>
<td>R4</td>
<td>T3, T4</td>
</tr>
<tr>
<td>R5</td>
<td>T5</td>
</tr>
</tbody>
</table>

Table 6: Developers tag the code with requirement IDs (R2, R3, R4, R5) and test case IDs (T1, T2, T3, T4, T5) and their relations.

**Step 3:** At this point, test cases are run and test case results are collected. More precisely, each test case can be associated with the following results: *pass, fail, blocked* and *not run*. The test case results are associated to the test cases and this information is stored in a list (Figure 7).

<table>
<thead>
<tr>
<th>Test results</th>
<th>Test case IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass</td>
<td>T1, T2</td>
</tr>
<tr>
<td>fail</td>
<td>T3</td>
</tr>
<tr>
<td>blocked</td>
<td>T4</td>
</tr>
<tr>
<td>not run</td>
<td>T5</td>
</tr>
</tbody>
</table>

Table 7: Test results and their test case IDs (T1, T2, T3, T5)

**Step 4:** In this step all requirements and test cases’ IDs are retrieved with the aim of creating the RTM. To this end, all the IDs in the list created during the first step are considered. Also, the IDs in this list are compared with the IDs specified in the code (step 2) in order to detect new requirements and/or test cases. Lets consider that a new requirement (R5) is introduced together with a new test case (T5) during the implementation. Therefore Figure 8, depicts the correspondent RTM.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5*</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: RTM with all requirement IDs (R1, R2, R3, R4, R5) and testcase IDs (T1, T2, T3, T4) from requirement’s list and test case source code

**Step 5:** Finally, the RTM is populated. To this end, we identified six values for describing the relationships along with their criteria between requirements and test cases, which are: *missing, pass, fail, block, not run* and *(new).* The missing values indicates that a requirement ID is missing in the source code of the related test case(s). For instance, requirement R1 in Table 5 is associated to test cases T2 and T3. Nonetheless, during the implementation of T2 and T3, the requirement ID R1 was not related to the code of the aforesaid test cases (Table 6). The value pass indicates that the test case was run and it passed. Similarly, the value fail indicates that the test case was run and it failed. The value block indicates that the execution of the test case was blocked due to
for instance lack of resources (e.g., time-out). The value not-run indicates that the test case could not be executed. Finally, the value *(new) indicates that the test case and requirement was added during the implementation of the test cases and not contained in the list created during step 1. Table 9 shows the RTM populated with the aforesaid values.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5*</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>missing</td>
<td></td>
<td>pass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>missing</td>
<td>fail</td>
<td>fail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>block</td>
<td>block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>not run</td>
</tr>
</tbody>
</table>

Table 9: RTM with all requirement IDs (R1, R2, R3, R4, R5) and test case IDs (T1, T2, T3, T4, T5) from the requirements list and test case source code.
To assess the validity of our proposed method, we conducted a case study at Westermo on the VRRP-MIB project. That project enhances the existing VRRP functionality in WeOS with the option to read some data using SNMP\textsuperscript{14}. Specifically, MIB is based on network management and an extension to use SNMP. In particular, it defines objects for configuring, monitoring, and controlling routers that employ the VRRP. Due to its verbosity and complexity, we will exploit a simplified version of VRRP-MIB project consisting of a limited number of requirements and test cases. Table 10 shows the (subset of) initial requirements.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Support &quot;read-only&quot; for applicable VRRPv2 (RFC2787) OIDs in the &quot;vrrpOperations&quot; sub-tree</td>
</tr>
<tr>
<td>R1.1</td>
<td>Only enabled instances of VRRP will be shown in the vrrpOperations table</td>
</tr>
<tr>
<td>R1.2</td>
<td>vrrpOperVrld: Must report the associated Vrid for all VRRPv2 instance</td>
</tr>
<tr>
<td>R1.3</td>
<td>vrrpOperVirtualMacAddr: Must return the virtual MAC address of the VRRP instance</td>
</tr>
<tr>
<td>R1.4</td>
<td>vrrpOperState: Shall reflect current state of the router (master, backup, init)</td>
</tr>
<tr>
<td>R1.5</td>
<td>vrrpOperAdminState: Must only list enabled vrrp instances listed and up(1)</td>
</tr>
<tr>
<td>R2</td>
<td>Support &quot;read-only&quot; for applicable VRRPv3 (RFC6527) OIDs in the &quot;vrrpv3Operations&quot; sub-tree</td>
</tr>
<tr>
<td>R2.1</td>
<td>Only enabled instances of VRRP will be shown in the vrrpOperations table</td>
</tr>
<tr>
<td>R2.2</td>
<td>vrrpv3OperationsVrld: Must report the associated Vrid for all VRRPv3 instance</td>
</tr>
<tr>
<td>R2.8</td>
<td>vrrpv3OperationsPriority: The effective priority of the VRRP instance will be returned</td>
</tr>
<tr>
<td>R2.5</td>
<td>vrrpv3OperationsPrimaryIpAddr: No answer</td>
</tr>
</tbody>
</table>

Table 10: Sample requirement IDs and their short description for VRRP-SNMP-MIB project

More precisely, in the initial sprint, we consider two requirements, R1 and R2, together with nine sub-requirements (i.e., R1.1, R1.2, R1.3, R1.4, R1.5, R2.1, R2.2, R2.8, R2.5). The left-most column shows their IDs while the right-most column shows a brief description of the requirements.

In the considered case study, all of the product’s functionality (source code and test cases) is implemented in c and python. The system is large and there might be four different kind of tests, i.e, manual tests, autotests and nightly build tests, for a total of 1400 tests. The software product is tested with new tests and also existing tests (regression testing) to ensure the right level of quality continuously. In the simplified version of the case study, we consider the subset of test cases collected in Table 11.

According to the proposed methodology, the first step is to relate each requirement with the related test case(s) and store this information. Thus, the scrum master identifies the requirements and the test cases during the sprint backlog. The IDs to requirements and test cases are assigned manually and stored in a list (Table 12).

As a preparation to the second step of the methodology, the developers have to relate the test case source code with the IDs of the test cases together with the IDs of related requirement(s)
<table>
<thead>
<tr>
<th>Autotest ID</th>
<th>Autotest name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>vrrp advanced</td>
</tr>
<tr>
<td>T2</td>
<td>vrrp fast failover</td>
</tr>
<tr>
<td>T3</td>
<td>vrrp auth</td>
</tr>
<tr>
<td>T4</td>
<td>vrrp cli_test</td>
</tr>
<tr>
<td>T5</td>
<td>vrrp basic</td>
</tr>
<tr>
<td>T6</td>
<td>vrrp preempt_delay</td>
</tr>
</tbody>
</table>

Table 11: Planned autotest IDs (T1, T2, T3, T4) and their description

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Test case ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1.1</td>
<td>T1, T2,</td>
</tr>
<tr>
<td>R1.2</td>
<td>T3</td>
</tr>
<tr>
<td>R1.4</td>
<td>T4</td>
</tr>
<tr>
<td>R2.2</td>
<td>T1</td>
</tr>
<tr>
<td>R2.8</td>
<td>T5</td>
</tr>
<tr>
<td>R2.5</td>
<td>T4, T5</td>
</tr>
</tbody>
</table>

Table 12: Planned requirement IDs (R1, R2, R3, R4, R5, R6) and testcase IDs (T1, T2, T3, T4, T5) and their relation

manually. In our case study, this step has been realized by simple annotations of the Python code composing the implementation of the test case(s) (Algorithm 1).

Data: Test case(s)
Result: Tagging Requirement ID, Test case ID
while Test case do
  if class then
    Tag test IDs;
    Tag list of Req IDs;
  else
    Do nothing;
  end
end

Algorithm 1: Algorithm for tagging test case ID and requirements IDs in test case(s) source code

In order to collect the test cases results (step three in the proposed methodology), we relies on log files. Sequentially, we manually created the list containing the test case(s) IDs and their results. Such a list is depicted in Table 13.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Test case ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass</td>
<td>T1, T3, T6</td>
</tr>
<tr>
<td>fail</td>
<td>T2</td>
</tr>
<tr>
<td>blocked</td>
<td>T4</td>
</tr>
<tr>
<td>not run</td>
<td>T5</td>
</tr>
</tbody>
</table>

Table 13: Test cases results

In our case study, the last two steps are performed by means of a python script which takes as input the list created in the first step and the actual source code of the test cases. Algorithm 2 shows the meta-code of the aforesaid script.
Data: Planned requirement IDs and test case IDs, Test cases code with tagging, Test results

Result: Requirement Traceability Matrix

compare Req.IDs, test.IDs and create RTM;

while Req.ID do
    | if Req.ID in test.ID then
    | Fill RTM;
    else
    | check for missing Req.ID in test.ID;
    | Fill RTM;
end

Algorithm 2: Algorithm for python script

Consequently, all the retrieved requirements and test cases IDs are used for creating the RTM. More precisely, the requirements IDs are used for labeling the columns while the test cases IDs are used for labeling the rows. Figure 9 shows the final RTM populated with the actual relationships between requirements and test cases explained in Section 5.

![Final RTM with all relations and criteria along with different colours](image)

Figure 9: Final RTM with all relations and criteria along with different colours
7 Discussions

The purpose of this thesis was to investigate and evaluate the integrity of RT in scrum, with the focus on requirements and test cases also considering minimizing administrative overhead and maintenance. The following section discusses about the analysis of results and threats that might affect the validity of our results.

The qualitative analysis of the systematic review results highlights the positive influence of considering RT in a scrum SDP. The results allow us to answer positively to RQ1. In particular, we were able to identify different methodologies supporting RT within a scrum SDP, minimizing development effort, administrative overhead and maintenance. Specifically, in the context of our study, we proposed a method which makes use of a more accurate RT links semantic between requirements, test cases and test results. Such a methodology has been realized by combining four already existed techniques namely, IR, ID, tagging and RTM. The proposed methodology implements a bidirectional RT. Also, it provides means for an easy detection of missing requirements and test cases.

One may argue that assign test case ID as a first step could be infeasible as in some projects test cases are not developed until the testing phase. While this may appear as a valid concern, it should be noted that the proposed methodology is still applicable. In fact, the test cases IDs can be inserted directly in the source code of the test case (step 3 of the proposed methodology). The method proposed in this thesis exploits the relations between requirements and test cases. In our industrial case study, we derived this information by using tags in the source code of test cases. Nevertheless, there might be situations where the proposed method is unable to retrieve correct RT links. For instance, if the developers do not add tags in the test case source code.

In order to answer to RQ2, we made use of an industrial case study provided by Westermo. Due to its verbosity and complexity, and considering the given thesis deadlines, we exploited a simplified version on the VRRP-MIB project. However, the project context allowed reasonable insights into improving the current industrial practices.

Within Westermo, software teams work according to a scrum SDP. To understand their scrum work process, we observed and joined the daily scrum meetings. Also we reviewed the documentation describing the product development model to understand their development process. The generated RT links between requirements and test cases shows that RT information between these two artifacts are bidirectional.

The case study results validate the proposed methodology as it succeeded in generating RT links as well as it enabled a better software V&V. In fact, one can easily determine whether or not the product increment produced during the sprint is verified and validated by checking the produced RTM which has all the necessary information.

Also, the case study results confirmed that the proposed method is actually applicable in an industrial setting without bringing a significant overhead to the already established SDP. In fact, the proposed methodology makes use of already existing information (e.g., requirements and test cases IDs) and does not add any relevant activity to any of the people involved in the project. Moreover, most of the phases of the methodology can be automated by means of scripts. One may argue that the development of these scripts is indeed an overhead. Although this might be a valid argument, the overhead introduced by the development of the scripts would affect only the first project developed with the proposed methodology. In fact, once developed, the scripts can be re-used without any further effort. It is worth to note, that our methodology can be easily adapted to other SDPs. Additionally, most of the steps in the proposed method can be fully automated especially if a test-driven development process is adopted. In this direction, in the case study, we provided semi-automation by means of Python scripts. It should be noted that our methodology is not bound to any specific programming language as the scripts can be implemented in any available programming language.
8 Conclusion and Future Work

Through this thesis, we proposed a novel methodology for achieving RT within a scrum SDP focusing on minimizing its overhead. In particular, we conducted a systematic research with the aim of identifying existing approaches and understanding how they can be combined for a more extensive RT support within a scrum SDP. Based on the systematic research, we have proposed a new methodology comprising of five steps.

The proposed methodology makes use of four different already existing techniques, which are RTM, IR, ID and tagging. In particular, the four techniques are refined and linked together. To this end, the semantic of the RTM has been extended with six relations which can be defined between requirements and test cases. The proposed methodology has been validated on an industrial case study provided by the Westermo company.

Further work could be devoted to i) provide hyperlinks (cross references) to associate requirement IDs and test case IDs and traverse the links interactively, ii) replicate the proposed method on other systems by combining version control information and the requirements allocation information and to iii) investigate how the proposed method can be extended to other SDPs.
References


### A Quality checklist

The quality checklist is as shown in the Table 14.

<table>
<thead>
<tr>
<th>Title/ Quality identifiers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duraisamy et al. (2013) [28]</td>
<td>Keyword searching, RTM</td>
<td>Industrial</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Banka et al. (2015) [2]</td>
<td>Integrating documents</td>
<td>Academic</td>
<td>No</td>
<td>No</td>
<td>Reliability threats</td>
</tr>
<tr>
<td>Jane celand-Huang et al. [1]</td>
<td>Cross reference</td>
<td>Both</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Espinoza et al. (2011) [6]</td>
<td>Domain-specific modelling (Power-type patterns)</td>
<td>Academic</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Jakobsson (2009) [30]</td>
<td>Tagging</td>
<td>Both</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Taromirad et al. (2012) [31]</td>
<td>Domain-specific modelling</td>
<td>Academic</td>
<td>No</td>
<td>Epsilon [46]</td>
<td>No</td>
</tr>
<tr>
<td>Badreddinet al. (2014) [25]</td>
<td>Domain-specific modelling</td>
<td>Academic</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ratanotayanon et al. (2009) [29]</td>
<td>Cross-reference</td>
<td>Academic</td>
<td>No</td>
<td>Xplanner [47]</td>
<td>Yes, lost links and does not handle large changes</td>
</tr>
</tbody>
</table>

Table 14: Quality checklist