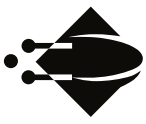
Traceability in a heterogeneous software development lifecycle

Sofus Albertsen

Advisor: Thomas Hildebrandt
Submitted: June 2016



IT University
of Copenhagen

Abstract

The need for efficient traceability in a distributed software development lifecycle (SDLC) is growing with the size of the product under construction. As the lifecycle gets more and more complex, the traceability suffers, making creation and alternation of a pipeline difficult. By using an event-based technique that supports dynamic trace generation stored in a graph database and leveraging the ideas of linked data, we can make a cross tool link of the pipeline, giving the much-wanted traceability in the SDLC. This thesis will present a data format for creating a graph structure, a DSL for manipulating the target tool data, as well as a framework for extracting the relevant information from the tools included in a distributed SDLC.

Contents

List of Figures

Chapter 1

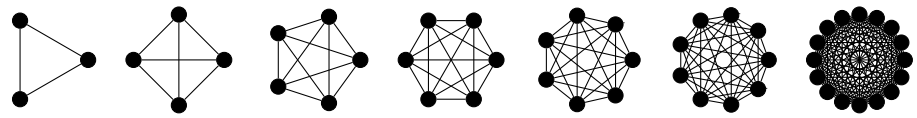
Introduction

1.1 Problem description

A typical software development lifecycle (SDLC) goes from requirements to design, construction, testing, debugging, deployment, and lastly maintenance. With increased complexity and trends like continuous integration/delivery in an SDLC, the traceability between the different artifacts gets equally harder to obtain efficiently. Some commercial tools like IBM's Jazz platform, Atlassian's tool suite, and Microsoft's Team Foundation Server maintain a full development stack and can, therefore, maintain a trace between all steps. The downside to these solutions is both the narrow tool selection, the steep learning curve, and license cost. This vendor lock-in prevents the companies from choosing the best-of-breed tools for their SDLC.

The enterprise ecosystem is very diverse. Its tool stack is heterogeneous, making a "fixed set of tools" approach impossible. Each tool is producing its artifacts independently, and usually only with integration to the neighboring tools in the stack. Lastly, companies have different tools on each of the steps in the SDLC and therefore different artifacts. The idea of having this heterogeneous collection of tools maintaining an integration to each other is too much work to be feasible, as illustrated in figure 1.1.

At the same time a lot of industries have regulations and laws permitting them to have full traceability from requirements to the final product; Automotive directives [15], FDA regulatory [33], Solvency II [22] and the like. There is a need for a framework that can adapt to certain regulatory workflows as well as a diverse set of tools involved in the



$n = 3$ $i = 3$ | $n = 4$ $i = 6$ | $n = 5$ $i = 10$ | $n = 6$ $i = 15$ | $n = 7$ $i = 21$ | $n = 9$ $i = 36$ | $n = 16$ $i = 120$

Figure 1.1: Exponential increase in integration points i for the number of tools available in the toolstack n.

lifecycle while still being flexible enough to plug into any given SDLC. The given framework must not obstruct the target software applications and only set a minimum of requirements in order for the framework to be as broadly integrable as possible.

1.2 Abbreviations

COTS Commercial of the shelf software

SDLC Software development lifecycle

DSDLC Distributed software development lifecycle

ALM Application lifecycle management

DSL Domain specific language

GPL General purpose language

IS Information System

1.3 Organization of this thesis

This thesis is organized into the following chapters:

Introduction, where this section also resides, talks about the research methodology this thesis is working on. It also gives a definition of the core terminology used.

The Analysis chapter contains the literature review where past approaches to solve the problem are analyzed. Thereafter it describes the delimitation of the thesis, the analysis of the domain under research, and the requirements from the companies interviewed. The result here is a list of capabilities the framework must possess, along with some verifiable questions that can be executed on the framework in order to validate its suitability to solve the problems.

The Framework design chapter takes the key points from Analysis and transforms it into the technologies chosen for the framework. Furthermore it gives a detailed view of how the framework is structured. It describes the architecture of the framework and points out some examples of the implementation.

The Framework validation and Limitations chapters validate and describe the limitations of the framework. They display the test scenarios that will act as the evaluation criteria for the framework. Lastly they discuss the limitations of the framework on the basis of the validation performed and the findings from the Analysis chapter.

The Future research chapter lists areas of interest for future research and development based uppon the project conducted in this thesis.

The Conclusion chapter concludes on the work produced in this thesis. It synthesizes the reason for the research, the results generated and talks about the significance of the study.

The target audience of this thesis is people in the software engineering industry as well as students interested in traceability. Basic knowledge about programming, graph databases, domain specific languages, configuration management and traceability is presumed.

1.4 Terminology definitions

Terms can oftentimes be ambiguous, even in a specialized field of research. This section will highlight some of the most important terms and make a definition on the meaning of the term in the context of this thesis. The section is divided into subsections, each with its own term or family of terms.

1.4.1 Software development life cycle

Describes the chain of phases a given piece of software is going through during its life cycle. A waterfall SDLC consists of the following phases [1] shown in 1.2.

```
1    Requirements - Design - Construction
2    - Testing - Debugging - Deployment - Maintenance
```

Figure 1.2: SDLC phases, performed in the waterfall methodology

Phases can be merged into one another and looped in different methodologies giving a different cycle than the one shown in 1.2. An example is the Test-driven development approach for constructing software[27]. Here you design your tests, construct your tests, construct the functionality, and execute the tests to verify the correctness of the implemented.

All methodologies generally have the same activities, but gives them different importance and placement in the SDLC.

Distributed SDLC puts a geographical aspect into software development. The main problem with distributed software development is both formal and informal communication[39]. Studies show that the amount and quality of the communication across sites are drastically reduced compared to co-located teams.

Application lifecycle management is a broader view that focuses on the whole enterprise around software development. ALM is about managing the entire application lifecycle, both business and operations. SDLC is one of its focus areas.

1.4.2 Traceability

Literature on traceability stems from the study of requirements specifications. One definition often cited is [37], which defines requirements traceability as:

> The ability to describe and follow the life of a requirement, in both a forward and backward direction

Forward traceability is the trace from requirements through task definition all through deployment and use. Backward is the same but in the opposite direction i.e. from deployment to requirements. The benefits of traceability are now recognized across all phases of software engineering. IEEE[23] has the following two general definitions of traceability:

> (1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.
>
> (2) The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

Another way to look at traceability as a definition is by the following two factors: the scope and the type of traceability.

Scope The scope of a traceability effort is defined as being either vertical or horizontal[52, 41, 54, 43].

Vertical traceability is the tracing done inside a single phase with similar types of artifacts. One example of this is one code component that is required for another component to work, or a task that is broken down into subtasks. As these vertical traces are single phased, they are also normally single tooled, making traceability easy to manage.

Horizontal traceability is the cross-phased tracing of artifacts as they evolve. An example of horizontal traceability is the flow of an action propagating though the system, e.g. a code commit that resolves a task, gets compiled, tested and released to artifact management. The extent of the horizontal scope varies greatly, depending on the tool chosen for the traceability.

This thesis is going to focus on the horizontal traceability across phases, also defined as developmental relations in [52, p6], referencing [36]

> Dependency relations are called developmental relations and are used to describe the logical structure of development and

> provide tracing requirements through the artifacts generated during the other phases of the software development lifecycle.

1.4.3 Trace links

Trace links (or just links in this context) is the relation between artifacts in an SDLC. [30] has a quite formal definition of (traceability) links as:

> A link $Link(a, a\prime)$ represents an explicit relationship defined between two artifacts a and $a\prime$. If a and $a\prime$ are directly linked as in $a \rightarrow a\prime$, where $\rightarrow$ indicates a link, then the link is said to be direct, and if $a\prime$ and $a\prime\prime$ are indirectly linked through one or more intermediate artifacts, as in $a \rightarrow a\prime \rightarrow a\prime\prime$, then the link is said to be indirect. Let $a => a\prime\prime$ represent an indirect link from a to $a\prime\prime$. An artifact (such as $a\prime$) through which an indirect link is established is
>
> said to be an intermediate artifact. The direction of the link indicates that the link is established from the artifact on the left-hand side (LHS) to the one on the right-hand side (RHS), such that the LHS artifact exhibits a dependency upon the RHS artifact.

When talking about trace links, we define them as directed binary relation markers between two artifacts. Even though it is directed, the marker needs to be traversable both ways in order for us to achieve both forward and backward traceability. A given artifact can have an unlimited amount of relations to other artifacts.

1.4.4 Artifacts and events

Each phase in the SDLC produces different kinds of artifacts. For example, requirements phase works with functional and non-functional requirements as artifacts, where Construction phase works with tasks, repositories and code changes as its artifacts. An artifact is in [30] defined as:

> a piece of information produced or modified as part of the software engineering process [49]. Artifacts take a variety of forms including models, documents, source code, test cases,

and executables. Such artifacts, in whole or in part, form the traceable objects of the system. Let A $=\{\alpha 1, \alpha 2, \alpha 3, ..., \alpha n\}$ be the set of all identified artifacts in the system. Let R ={r 1 ; r 2 ; r 3 ; . . . ; r n} $\subset$ A be the subset of artifacts that are requirements.

We define artifacts as any kind of content produced or generated either manually or automatically during the SDLC.

Events are defined as isolated activities that transform or manipulate artifacts produced in the SDLC.

1.4.5 Continuous integration

Continuous integration is a software development practice with the purpose of ending the so-called "integration hell" often found in large non-agile projects[5]. It was made popular by the advocators of Extreme Programming[7]. It advocates for one main source repository for the whole team, and only a few, short-lived branches from the mainline. Every developer shall commit code to the common repository often, making the change sets small. The idea behind it is to fail fast and control the quality of the code through tests (another practice in extreme programming). As you integrate often, your tests and builds must be automated.

1.4.6 Continuous delivery

is an extension of continuous integration that span all the way out to the release of a product[4]. The biggest distinction is the notion that everything needs to be tested and ready for a potential deployment with every commit. This requires that all users, both humans and machines, are ready and able to integrate with the new version of the tool at commit time. The focus here is not only on the components in isolation, but the whole ecosystem around it.

1.5 Research methodology

As stated in the abstract, a software framework is to be developed in order to test whether the assumptions stated in the Framework design

chapter can be validated against the problems described in the Analysis chapter.

The research methodology conducted in this thesis is done through a multi-methodological approach to information systems (IS) research, described by [46]. The paper explains how building software can be the best way to validate a hypothesis:

> Concepts alone do not ensure a system's survival. Systems must be developed in order to test and measure the underlying concepts. Systems development is, therefore, a key element of IS research.

The paper describes a set of methodologies that can be used when conducting software engineering in research.

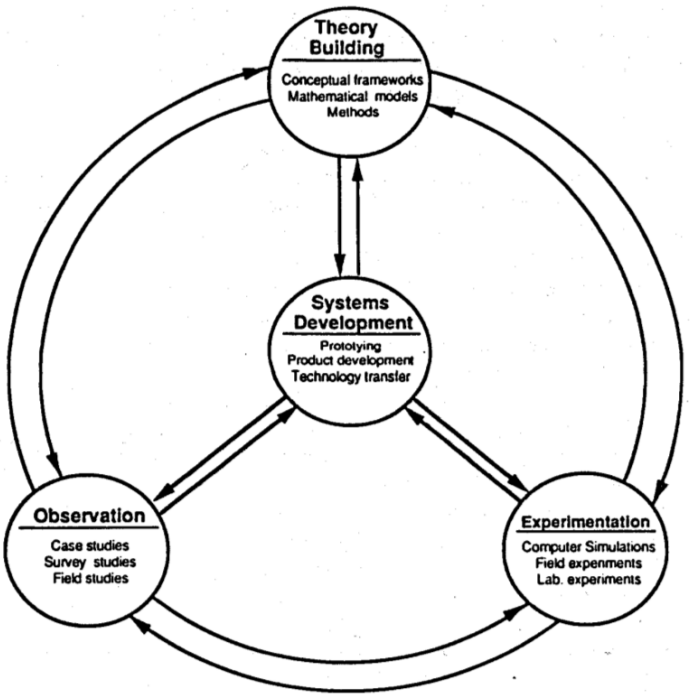


Figure 1.3: Multi-methodological approach to information systems research by [46]

.

The overall methodology has four stages shown in figure 1.3; Theory building, Experimentation, Observation and Systems Development.

According to [46], the research starts with the notion of a new model, concept or construction of conceptual framework in the theory building

process. One must then conduct experiments and make observations in order to gain knowledge about the domain under research. Here, the researcher also learns how to generalize their hypotheses/models in order to make their research relevant to a broader field. The system development phase is where the knowledge gained from the other phases is put to use when constructing the proof of concept/prototype that enables the researcher to validate the theory.

By using the stages described in the paper, it is easier to divide the research conducted in this thesis into chapters.
Our theory building phase is described in Problem description where we have defined a problem domain, and found a gap in the research to cover that problem. The Experimentation and Observation phases are covered with interviews with industry companies, narrowing problem domain and requirements. It is also through these observations that we can begin to sketch a framework for solving the problem.
The systems development phase is covered in the Framework design chapter where we describe the overall architecture of the framework constructed as well as discuss some of the experiments that were created along the way in order for us to refine our framework. With the Framework validation chapter, we test the effectiveness of our framework to manifest our theory building phase.

We deem this approach to be the best suited for the area under research. Without a prototype framework, it is hard to uncover the practical problems that the theoretical research lacks.

Chapter 2

Analysis

With the overall description and terminology of the domain now defined, we will analyze the domain further in order to extract the requirements our framework must possess. We will study the current literature on efforts to solve the problem and discuss their differences and limitations. That leads to defining the delimitation of the thesis. The delimitation makes some assumptions and scopes on the problem making it tangible to design the framework later on. When the scope of the solution is defined, we will look at the specific requirements for the framework through industry interviews.

At the end, we will have a list of requirements that the framework must be able to fulfill in order to validate its ability to solve the problem under research.

2.1 Literature review

Traceability is widely recognized as a key success factor for companies involved in software development[41, 51, 52, 31].

In empirical studies like [41], different companies note different potential benefits from having traceability; from verification against third party certification, and increase in process efficiency, to the transparency of the development process.

These benefits are very diverse and rarely coherent. A possible explanation to this is that the term traceability is not without ambiguity [51], as also stated in the Terminology definitions section. This ambiguity collects many different activities under the same term, giving a

blurred picture on which benefits are to be gained. A lot of research has been conducted into the different aspects of traceability.

In order to gain traceability, one must collect the trace links from one artifact or event to the next. The production of trace links can either be done manually or automatically. Fully manual traceability without aid is unfeasible if not mandated e.g. by a regulation due to the amount of resources needed for the activity and the high risk of errors in the repetitive job [26]. Automatically generated traceability is prone to low accuracy in identifying the traces, as well as having a large portion of false positives[29, 38].

With automatic traceability, one can either try to create the traceability as part of the development process or as a post-development effort. [25] calls these two methods *in situ* or *ex post facto*. [54] uses the terms off-line and on-line traceability and describes them as follows:

> Regarding requirements traceability, this can be done either on-line, in which case traces are stored automatically by a tool as a by-product of the development activity. Or it can be done off-line, which means that traces are recorded automatically or manually after the actual development activity has been finished

Both studies point to the fact that online traceability is better than offline recreation of the trace links, specially if the tools provided can help the users create the wanted traceability.

Traceability classification types There are many different types of traceability relations discussed in the literature [52, 54, 51]. [54] divides traceability into two categories; functional and non-functional: Functional traces are described as follows:

> These are created by transforming one artifact into another using a defined rule set. The transformation is not required to be performed automatically, but it has to follow unambiguous procedures, such as transcribing an audio recording of an interview.

Non-functional traces are described in this way:

> They refer to traces of informal nature. These traces result from more or less creative process, such as semantically analyzing and extracting customer requirements from a set of

meeting minutes. In the technical category, non-functional traces could, for example, exist on the parts of the code which are affected by quality requirements.

This thesis focuses on the functional, transformatory process of traceability throughout the SDLC.

When reading the literature on traceability in SDLC, the main focus has been on finding a superset of relationship types in order to describe the traceability in full detail. [52] divides it into eight different main groups of relations; dependency, generalization/refinement, evolution, satisfaction, overlap, conflicting, rationalization, and contribution.

dependency When artifact A's existence relies on artifact B's existence, there is a dependency relation between A and B.

generalization/refinement as stated in [52]. Generalization/Refinement relations are used to represent logical entities at different levels of abstraction.
An example of this in the agile methodology could be an epic broken down to multiple tasks, where the information about the work being performed becomes more and more refined in the breakdown structure.

evolution is the trace of an artifact that evolves. It represents the history of the artifact by maintaining a schemaless versioning system of the artifacts.

satisfaction is the link between a requirement and some part of the product that has been developed if the part satisfies the given requirement.

overlap relations represent relationships between document entities that represent the same logical entity. Two artifacts referring to the same feature of a system constitutes an overlap relation.

conflicting relations is when two artifacts cannot both be implemented. This can either be as requirements or design guidelines etc. This conflict can be resolved when one of the artifacts either gets deprecated/deleted or modified.

rationalization represents the *why* of a given feature or task that points to a use case, requirement or design rationale.

contribution This relation type is defining who is behind a given artifact whether the artifact is a use case or product release. As an example, all developers that push change sets to a particular class or method would be added as contributors to that class.

Other literature has different types of traceability relationships. In another study [51] nine traceability relationships were defined in their Traceability Reference Schema (TRS).

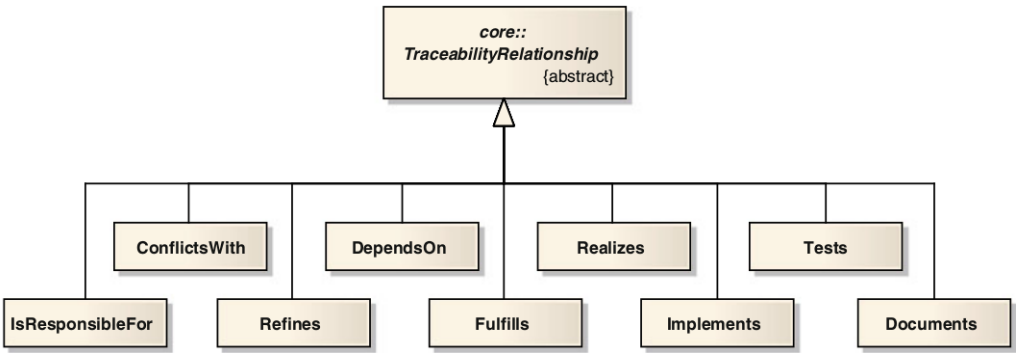


Figure 2.1: Traceability Reference Schema from [51]

The two definitions have some similar types such as conflicting, and ConflictsWith, but others does not have a direct equivalent like TRS's IsResponsibleFor type. This leads to the conclusion that there does not exist a fixed set of relationship types when dealing with traceability.

Most of the literature referenced has focused on traceability of human made artifacts. Examples of that are requirements, use cases, tasks, and code. They either neglect the building phase of a program, or do not focus on that part of the traceability. With continuous delivery and integration having an increasingly bigger impact on the industry with time, this part of the SDLC has gained increased importance as well. Especially the focus on tracing the artifacts that are created in a modern SDLC after a commit of code to a repository. We have gone from having manual release processes to defining it all in an automated release pipeline. It is the trace of artifacts that is illustrated on figure 2.2.

```
1 work item -> commit -> build -> test -> release/promotion
```

Figure 2.2: Trace of artifacts in a SDLC

Each step of the figure can happen several times in a lifecycle, e.g. a commit can be build and tested multiple times.

All of the studied literature except [30, 52, 31] are targeting artifacts as their base of traceability. [30] is describing the hurdles of maintaining the artifact-based traceability. They are describing another approach where they trace the events instead of the artifacts. They see an artifact evolution as a series of events based on changes. So if all changes create an event, then the evolution of an artifact can be described as the sum of all changes. They try to solve the maintenance problem described as one of the three hurdles of traceability in [54]. The three hurdles of traceability are creation, maintenance and usage of traceability artifacts. By making the events immutable, the maintenance of the versioning is built into the event approach; every event acts as an indicator for a new version of the underlying artifact.

The Swedish telco Ericsson has in its unpublished paper [31] presented an online event-based traceability framework following this idea. They describe the framework as follows:

> This paper identifies traceability as a key challenge in achieving continuous integration and delivery and documents an industry developed framework — Eiffel — designed to provide real-time traceability in continuous integration and delivery.

It has a fixed set of relationship types as well as the capability of controlling the software pipeline by having the clients trigger tools on given events making it not only focused on traceability but also on orchestration of the SDLC pipeline.

2.2 Thesis delimitation

As stated in the Literature review, traceability is a broad subject of research. In this section, we try to narrow down the scope of research conducted in this thesis, while also stating some assumptions going forward.

We are not looking into the human aspects of SDLC or traceability. If a human constructs a wrong tracelink, then that tracelink is blindly incorporated in our framework. We are assuming that the tracelinks are created, whether or not it is from manual or automated activities. For

example, when a developer resolves an issue with a commit in source control, we assume that there is guidance to do that.

What we are aiming at is to collect and visualize all the tracelinks in a uniform way. We are focusing on the horizontal traceability that spans across phases in the SDLC. We are not trying to trace the artifacts themselves but rather the corresponding events. The framework strives to be method agnostic and applicable to all SDLCs with emphasis on the ones using continuous integration and delivery. This is due to the fact that much more importance is laid on configuring the build pipeline in order to produce the wanted software components.

Instead of defining a new set of relationship types up front, or use an existing, the presented framework of this thesis will try on an undefined type system. It will work with one event supertype where the type is not restricted to any given set but can be defined by the users of the framework from instance to instance. That way we keep the framework as lightweight as possible, making it adaptable to any given methodology or process used.

Proper tool support and visualization is deemed important in order to raise the quality of traceability[44]. Many of the papers presented in the review do not talk about how they visualize the given traceability they are trying to reach [51, 50, 24]. Even though visualization of the traced is deemed important, it is not directly part of this thesis. Focus is on creating the horizontal tracelinks and using the built-in tools in the database to visualize the data stored. The creation of an effective GUI would require mockups and user experience research and is therefore thought of as a study on its own.

[48] Ramesh et al. describe a traceability dimensions table that represents the dimensions of traceability information. Their usage of the table is tightly coupled with their research on reference models for requirements traceability and therefore does not directly apply to this thesis. However, the dimensions contained in the table represent a good way of scoping the area under research as illustrated in table 2.1.

This thesis will present a traceability framework that works on any given SDLC regardless of methodology and process followed. It is integrated into the software development tool chain without required alternations of the targeted tools and creates tracelinks based on the events emitted. It combines the different approaches of event-based traceability, graph-based database, schemaless relationships and a single supertype.

Dimension	Concrete example
What	Events that stem from evolving artifacts
Who	The tools that evolve artifacts of a SDLC
Where	In a graph database
How	By making a generic plug-in
Why	To answer the questions found in the company interviews
When	On-line; when the events are running

Table 2.1: Scoping the area under research through the traceability dimensions.

2.3 Industry interviews

In this section we will introduce some case companies from the industry that have interest in improving their traceability. The interview conducted will lead to a list of capabilities that our tool should have in order to show its validity towards the enterprise environment targeted.

2.3.1 Interview setup

The interview is being conducted in relation to [42]. We conduct expert interviews with the purpose of thorough exploration to define real-world problems. The interview has been conducted at a conference in Trondheim on April 18[19]. Prior to this meeting, a number of informal meetings have taken place where the problem of traceability and configuration management have been discussed in broader terms. Both companies have an interest in the framework and knows about the proposed solution presented in this thesis before the interview. The interview happened during a discussion session about traceability. It is, therefore, more of a guided observation than a formal interview. The interview itself is not publicly available due to sensitive information being disclosed. Meanwhile, the statements presented in this section have been verified for approval by the participants (see Appendix Mail correspondance for more information).

2.3.2 Case companies

Employees from two case companies have been interviewed for this thesis. Both companies are large-scale global companies with distributed development.

Case A is a large automotive company. They have over 15,000 employees worldwide, and software development in several countries. As an automotive company, they need to comply with the ISO standard 26262 titled "Road vehicles – Functional safety"[15]. It is a risk-based standard that identifies critical failure points and prescribes the implementation of countermeasures or backup systems. Traceability is a key feature to have when documenting measures to ensure a sufficient and acceptable level of safety is being achieved.

Case B is a major Danish company with 10,000+ employees globally, where more than 100 people in different locations around the world are working with software engineering. Software development is done both in-house as well as by outsourcing.

2.3.3 Findings

During the interviews, there were statements that came into discussion multiple times. The companies describe these statements as key features or problems that they think can be solved through traceability. The statements are described as capabilities the framework must possess.

Finding 1 Ability to visualize the pipeline from a given event.
Example:
As a developer, I want to know who is using this component in other libraries etc.
As a developer, I want to know where my commit is in the pipeline. Has it been deployed, or is it halted somewhere?

Finding 2 Ability to measure the lead time between two events.
Example:
Measure the time from when an issue is created to first commit is pushed to the repository.
Measure the complete pipeline time from commit to executable creation/product release.

Finding 3 Case A: making compliance with ISO 26262.
Example:
Compare two builds from identical change sets. When testing embedded software, the build used in the simulator and the build used in the

actual device are differently compiled, and are, therefore, two different artifacts.

Finding 4 Management: Has this issue been resolved in this release?
Example:
Have all commits connected to this issue been incorporated in this release?
What tests were run to secure that it was resolved?

Finding 5 We want to have an event mechanism that can drive the continuous delivery pipeline and that is tool agnostic.
Example:
Every tool should be able to trigger a build in the pipeline.
We want to be able to change vital components in the pipeline without losing the pipeline itself.

Findings 1-4 are about visualizing the events recorded or its data in one way or another. In that way, the tools will only have to produce data to the framework to fulfill these requirements.
Finding 5 is about triggering events upon other events. This requires that the tools also become consumer of events and some kind of triggering mechanism based on certain criteria.

2.4 Summary

In this chapter we have made good progress analyzing the details of the problem domain and specifying the delimitations of the thesis. We have conducted an interview with two case companies with large-scale distributed software development. They come from different sectors, making the findings more generic and applicable to a broader range of industries. The findings described in this chapter will serve as the foundation when designing the framework in the following chapter.

Chapter 3

Framework design

This chapter describes the decisions and thoughts that went into constructing the framework presented in this thesis.
The industry interviews lead to five findings that show how the companies could benefit from such a framework. Four of the listed findings are about recording and visualizing the events happening in the pipeline. They can be implemented without changing the internals of the given tools in the pipeline. The last finding is about having an event-triggering mechanism built in that can activate and partially control the pipeline. In order to make this feature, the framework must know much more about the process, and some kind of pipeline DSL must be implemented. The pipeline must have some criteria that activate the tools accordingly. The goal of this thesis is to make a lightweight, non-obstructive framework that can aid in horizontal traceability. We will therefore only in a very reduced manner take finding 5 into consideration when designing the framework.

We want to construct a framework that tracks the events in the pipeline and then links down to the underlying artifacts from each tool. In that way, the overall managing of artifacts and their versions are kept in the respective tools while still maintaining a visualization of the horizontal traceability of the given pipeline.

3.1 Overall architecture

As described in the Artifacts and events subsection, the framework will consist of multiple data senders, one from each instance of a given tool, and a central database where all the data is stored. It will store the

events that the different tools emit. It will not save tool-generated artifacts related to the events, but instead link back to them.

3.2 Event-driven traceability

Each phase in the SDLC maintains its own set of artifacts. They are rarely shared with other phases. Instead they evolve during events from one phase to the next. In that way a requirement gets broken down into tasks etc., as the figure below illustrates:

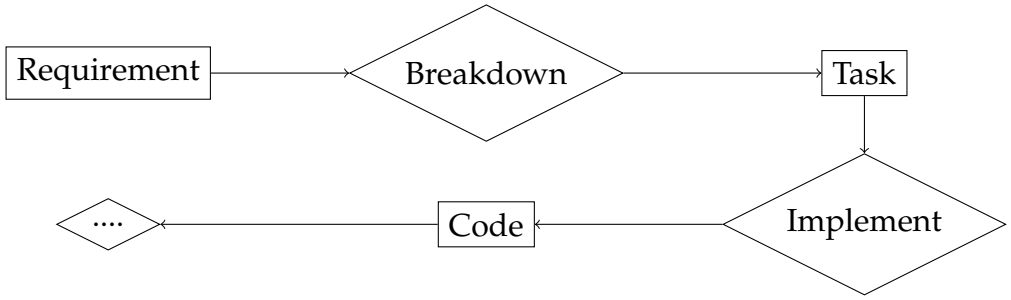


Figure 3.1: Artifacts and event in the SDLC. Rectangles illustrates artifacts, and diamonds represent evolving events.

Every time an artifact evolves from one phase to another, it does so by an event as shown in figure 3.1. In order to make horizontal traceability, we need to link events from one phase to the next. That way we end up with a chain of connected events that all have relations to the underlying artifacts. We are building the framework in the belief that any given tool A knows who activated it e.g. tool B or a human, enabling us to construct a network of traces by joining all the tools' individual events with its parent events.

3.3 Domain requirements

In this section, we are outlining requirements stemming from an implicit knowledge that is not taken directly from the interviews or studies of the companies. They are derived from the environment that the framework must operate in as well as from the nature of the data they must support. They do not necessarily mark a concrete implementational issue, but are equally important in order to secure the quality of the framework. The

section is divided into subsections that each represents a trait or restriction that the final design needs to accommodate for when operating in large enterprises.

3.3.1 Scalability

The framework needs to operate in large enterprise environments with lots of different software projects running simultaneously. Each of the projects have thousands of artifacts, and for that reason at least an equal amount of events. The tool needs to accommodate for distributed software development where multiple teams contribute to the same repositories. This requires a system that can handle peak loads without dropping events and that has a high degree of availability. We, therefore, need some kind of load balancer and messaging queue in order to compensate for peak loads and also downtime in the database.
There is a large variety of message brokers and protocols in the market; AMQP, JMS, MSMQ to name a few. We want to have a protocol that was language agnostic, leaving room for implementing clients in any language dictated by the enterprise. Of the three listed protocols, only AMQP delivers that; JMS is Java only, and MSMQ is built on Microsoft technologies with only a 10-year-old abandoned library to the JVM languages.

RabbitMQ[21] is an enterprise grade open source message broker, talking through the AMQP message protocol as the standard. It features a lot of the patterns described in [40] like clustering capabilities as well as persisting and guaranteed delivery of messages. It also has features like message filtering (called routing) and publish-subscribe channels (called fan-out) making it possible to build in finding 5 (2.3.3) at a later point if need be. This makes us able to make a stateless framework because queuing, for instance, is handled by the message broker, making our framework more scalable overall.

We choose RabbitMQ because of its ability to scale, its language agnostic protocol and the fact that it has the features needed from a message broker in our framework.

3.3.2 Geographical distribution

Working with geographically distributed teams, time zones become an issue. If two events, A and B, are executed at the order $A...B$ with 10

seconds interval, and $B \rightarrow A$, then the tracetracelinklink will be rightfully made. But if the time zones of A and B differ by one hour, the time difference would be recorded an hour wrong on either side of the time zone difference. We, therefore, need to record timestamps in a format that either is noting the time zones or that has a universal time schema.

Unix time, also known as Epoch time, displays the time in number of seconds from January 1st 1970 00:00:00 UTC time. It ignores local time zones by stating that all time originates from UTC. The time format is an integer number, making math calculations of the duration between two timestamps trivial.

ISO 8601 [16] is the ISO standard for displaying both the time and date. It is formatted in the following way: $[YYYY] - [MM] - [DD]\mathrm{T}[hh] : [mm] : [ss]W$ where the T is a delimiter between date and time, and W indicates the time zone. If the time zone is not indicated, the local time zone is inferred. If used by the framework, the time zone must be obligatory as the notion of "local" time zone is nonexistent. The benefit here is that it is human-readable as well as easily comparable and sortable with a trivial string comparison. The downside is that calculating a duration between two timestamps requires a lot more parsing.

As stated in the Findings section of the analysis, we are using the timestamp to tell the duration between two events in a given chain. We only care about the duration between the events regardless of their different time zones. Consequently, we choose **Unix** time as the given time format when dealing with events.

3.3.3 Namespace

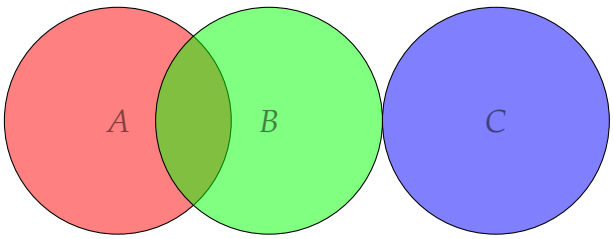


Figure 3.2: Namespace overlaps between the tools A and B but not C

When working with a heterogenous set of monolithic tools, we cannot guarantee the uniqueness of namespaces set by each tool resulting

in what [50] calls an ambiguity problem. In our test setup described in the later chapter Framework validation, we are working with three specific tools; Jira, Gitlab, and Jenkins.

Jira uses a namespace for identifying issues that looks like this: $[XXXX - Y]$ where X represents the first 4 letters in the project name, and Y the consecutive number of issues for that project.

Jenkins has a namespace that has the following convention: $[X\#Y]$ where X represents the build job name, and Y represents the consecutive build number from that build job.

The two examples presented here do not have a direct overlap of namespaces and serve only as an illustration of the different namespaces. But with an unknown set of tools it is not guaranteed that there are no coalitions within the selected toolstack. In a large enterprise, multiple instances of a given tool can be deployed, making it possible to have ambiguity problems in the namespace of a single tool.

We regard namespace coalitions as a task best solved by the enterprise itself, as they already handle it on a tool level and most likely have some corporate guidelines for namespacing. We have made it possible to prefix the id in the data format through the DSL, making the users able to solve any occurrence of the ambiguity problem regarding different tools.

3.3.4 Timing

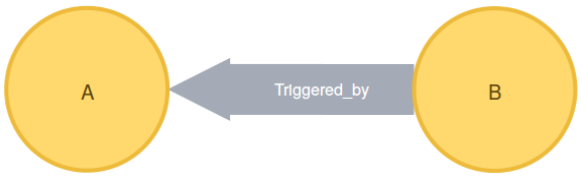


Figure 3.3: Nodes with a relationship from B to A

In order to make tracelink $B \rightarrow A$ for the events emitted first A then B, the nodes need to be inserted in the chronological order. With our RabbitMQ message bus, we make sure that we uphold a first in, first out (FIFO) queue into the database. But we still need to tackle two other scenarios with timing:

1. Tool *A* is triggering *B* before reporting to the framework
2. Tool *A* is sitting on a heavily delayed line, making the messages from *B* reach first

In both of the scenarios, we are dealing with events getting out of order. If the events are inserted out of order, the ability to create tracelinks is broken. This is because the framework is stateless meaning that it does not temporarily hold events in memory to see if other nodes are coming in to resolve the broken link.

The solution here is to make a shadow node of *A* as a placeholder for the event that is missing as seen on figure 3.4. This has several benefits over a stateful solution where traces will be retried several times until given up. The first one is that we maintain the traceability when

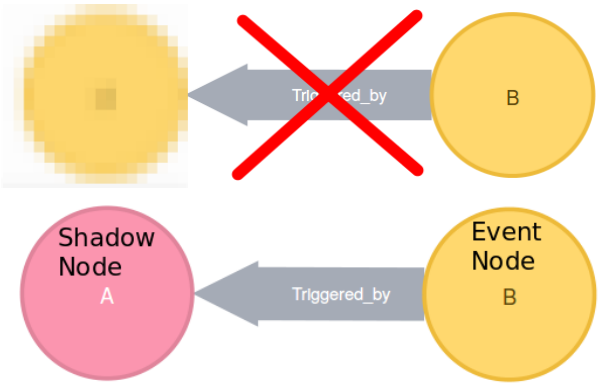


Figure 3.4: Node *B* with a relationship to shadow node *A*

node *A* gets persisted; it will just merge its data into the shadow node, transforming it into the real one. Another benefit is that we do not need to have a separate queue for unresolved relationships that can run full or terminate at some point. The database also marks the shadow nodes in order for the framework to identify them and report if the level of shadow nodes rises.

These scenarios may seem like thought experiments only, but they stem from the work conducted in the Framework validation chapter. In the concrete setup, an integration between Gitlab and Jira was made in such a way that if the commit message to a certain repository included the string "FIXED {task#}", Gitlab would call Jira to close the task. The order of events would be as follows:

1. Gitlab receives the push containing the commit message
2. Gitlab calls Jira to close the task with the given id
3. Jira calls the framework with reference to the commit event.
4. Gitlab calls the framework to persist the commit and push events.

In this scenario the framework does not know of the commit before *after* the task has been closed based on the given commit, resulting in an out-of-order insertion into the database.

3.4 Database model

Over the past decades the number of different database management systems has grown, with a lot of new systems comprised under the NoSQL umbrella[45]. With the increase of volume and diversity of data and usage, the decision on the database model is a key factor for scalability and performance when constructing new software.

Relational databases are great for structured data where the model is defined up front. But it puts a strict schema on the data, making it unsuited for our heterogeneous toolstack and its data models.

Document, key-value and wide-column data stores all give the flexibility of a schemaless data model, but they either do not support relationships or do not operate well when traversing n-degree relations as well as the graph database.

Graph databases put the concept of relationships between data first. Based on graph theory, they have the notions of nodes and edges to form a graph. They are addressing the performance hit normal databases suffer from when traversing through multiple relationships e.g. several table jumps.

We use a **graph database** to store the data. This has several advantages over the alternative database systems. When querying our database, we typically want to find the nodes that are on the same graph as a given node A. This is the key point for graph databases' existence.

We chose Neo4J database as the software implementation as it is one of the most used graph databases[6] in the world. It is capable of making directed binary relations between two nodes while still allowing the relation to be traversed in both directions.
According to [32] it provided the second best performance on large datasets only surpassed by DEX (now Sparksee). The reason for choosing Neo4J over Sparksee is its open source license (GPL version 3[13]) as well as the large community around the product.

3.4.1 Cypher query language

Cypher query language is a declarative graph query language designed by the Neo4J team to query the graph database. It is the graph database equivalent to Structured Query Language (SQL)[9]. Instead of tables, rows and columns the language has nodes, relations and properties of the two. A common query is built up on the three keywords MATCH, WHERE, RETURN.

MATCH defines the graph pattern that consists of nodes and relationships we want to retrieve. One query can have several match statements to describe the pattern in further detail.

WHERE defines limitations to one of the nodes or relationships in the pattern. A lot of the limitations can also be described in the MATCH clause, as seen below.

RETURN defines what part of the graph pattern you would like to retrieve from the database, just like the select part of an SQL statement.

In the listing below is an example of a CQL statement used later in the thesis.

```
1 MATCH (n:Event)-[r:Triggered_by*]-(m:Event)
2 WHERE n.id={event id}
3 RETURN n,r,m
```

Listing 3.1: Cypher query for getting all nodes of type Event in the graph connected to a given node n

Translated into plain English, we want to get all the nodes of type Event that are connected via a relation of type Triggered_by in any degree (the asterisk in the end) and any relation direction from a given node n.

3.5 Data format requirements

Unless deliberately made so, each tool in an SDLC has its own data model and artifact types. But even if we have an arbitrary amount of different tools, they all need to confine into some general requirements in order for their data to be chainable and useful. This balance puts a lot of restrictions into what we can expect to get from each tool. If too many required fields are put in the data format, it ends up excluding tools, with the possible outcome of breaking the tracelinks. If too few required fields are put in, the data will loose the ability to display information that is needed by the companies. We try to give some assumptions on what to require from each tool.

Id As the main goal is to link events together, one must be able to tell them apart. The only way to do that is to have a unique id on each and every one of them.

Timestamp In order for the framework to expose how long it takes a given chain to complete, one must know the start and end time of the chain. But since it never knows when it ends (one event could occur weeks after another in the chain), it needs to track each event with a timestamp. Either that timestamp is emitted through the tool, or our plugin must append this at the point of execution.

Previous id As stated in the Thesis delimitation, the tool needs to know what activated it. Who were its predecessors? If a build B is started by a commit A, then B must know of A in order to make the tracelink $A \leftarrow B$. One event can have one or several parent events. That can, for example, be a release that has several commits on it since the last release.

Type Knowing the type of event would also help. Is it a code commit, a test execution, or a release? As stated in Thesis delimitation, these types cannot be a fixed set of values since we do not know all the possible tools a given company wants to incoorporate in their SDLC.

URL Since we are tracing the events, but want to get hold of the underlying artifacts, we need to have a link back to the corresponding artifact. This is done by specifying a URL to the data origination. If the artifact

is not avaliable through a web address, then the link must be to the corresponding server where the artifact is located.

Healthy event If a build has failed, or the packaging of the component is executed with a warning, the pipeline is usually stopped. However, a halted pipeline can be due to many different scenarios, and for that reason it could be a benefit knowing whether or not the event chain is still healthy or not. In this way, a developer can see the status of the commit, aiding in Finding 1 from the industry interviews.

Data The last part is all the tool specific data. We do not give any limitations or format to this because we have no control over the tools used. The tool specific data is collected so that companies can make custom queries about this kind of data if they choose so.

We have tried to make a data format that is information rich enough that we can accommodate the industry deduced findings while still being applicable to the vast majority of the tools used in an SDLC.

3.5.1 Message format

After describing the data that needs to be included in the framework, we also need to define a message format. Our message broker RabbitMQ's default wire-level protocol is AMQP, which is agnostic about the data format our messages are in. Consequently, we could use any format that can be used to contain data; XML, CSV, YAML, JSON etc. We chose JSON as the format for sending the event messages because JSON has a less verbose syntax than XML. It is also the javascript data transfer format, so if a viewer to the data in the database was to be developed it would not be necessary to make any translations. In order to validate whether a given message is against the format, we created a JSONSchema, which validates against the Data format requirements.

```
1 {
2   "type": "object",
3 "oneOf" : [{
4   "properties": {
5     "id": {
6       "type": "string"
7     },
```

```
    "prev": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "url": {
      "type": "string"
    },
    "type": {
      "type": "string"
    },
    "timestamp": {
      "type": "string",
      "format":"date-time"
    },
    "healthy": {
      "type": "boolean"
    },
    "data": {
      "type": ["object","string","array"]
    }}

  }],
  "required": [
    "id",
    "prev",
    "type",
    "timestamp",
    "healthy",
    "data"
  ]
}
```

In that way, we can have the schema validation, which is usually one of the strengths of XML, together with the web-centric format of JSON.

3.6 Data parsing

Since the tool landscape is so diverse, our framework needs to accommodate for an arbitrary set of tools their data formats. As an example

of this diversity, Wikipedia lists 37 version control systems, where 28 are marked as actively developed[3]. It is not feasible to make a tool up front that has a converter to every target tool imaginable. We need to define a way to make it possible and easy to parse any given format of data into our chosen format to make accurate tracelinks.

For extracting the relevant data one could use different path-traversing libraries like JSONPath for JSON[12] or XPath for XML reading[17]. But sometimes we need to make alternations on the data like:

1. Concatenations of two values
2. Loop through some values and make Event nodes for each of them
3. Make evaluations (ifs) etc.
4. Take values from multiple paths into the same array

These functions are neither comprised in JSONPath nor XPath, and it is therefore necessary to extend the capabilities of the libraries:

There are two ways to make our framework this extensible: Have plugins with general purpose language or create a domain specific language.

3.6.1 Framework plugins with GPL

One solution is to make an architecture that enables pluggable parsers. The basic idea is that the parser needs to confine to an interface like the one described below:

```
1 package dk.itu.tracy.parser;
2
3 import java.util.List;
4
5 import dk.itu.tracy.entity.Entity;
6
7 public interface Parser {
8         public List<Entity> parse(String inputString);
9 }
```

Listing 3.2: Parsing Interface in Java GLP

The *inputString* is the string coming from the target tool. A specific implementation of the interface must then produce one or more entity objects and return them to the framework. This was the first way of trying to solve the parsing problem, and listing 3.3 shows the first way to parse the JSON emitted from Jenkins.

```
1  public List<Entity> parse(String jsonString) {
2     Entity ent = new Entity();
3     List<Entity> out = new ArrayList<>();
4     JsonObject json = new
          JsonParser().parse(jsonString).getAsJsonObject();
5     long date = json.get("timestamp").getAsLong();
6     ent.setTimestamp(new Date(date));
7     ent.setId(json.get("id").getAsString());
8     try {
9        ent.setUrl(new URI(json.get("url").getAsString()));
10    } catch (URISyntaxException e) {
11       e.printStackTrace();
12    }
13    if(json.get("building").getAsBoolean())
14       ent.setType("jenkins-build-started");
15    else
16       ent.setType("jenkins-build-done");
17    ent.setData(json);
18    JsonArray ja=json.get("changeSet").getAsJsonObject()
19       .get("items").getAsJsonArray();
20    String[]prev = new String[ja.size()];
21    for (int i = 0; i < prev.length; i++) {
22       prev[i]=ja.get(i).getAsJsonObject()
23          .get("id").getAsString();
24    }
25    ent.setPrev(prev);
26    out.add(ent);
27    return out;
28 }
29 }
```

Listing 3.3: Parsing Jenkins code in Java GLP

The downside of this method is the required knowledge about programming, specifically Java, and the framework used for parsing. Another headache is that it does not support scripting, meaning that it

needs to be built and linked to the framework for every code change on compile time.

3.6.2 DSL

Another solution is to build a Domain Specific Language to solve the task. DSLs are languages tailored to a specific application domain[34]. This has the benefit of being more concise to the specific purpose. DSLs can either be internal -embedded in a GPL language- or external -having its own parser.

Internal DSLs have the benefit of making the GPL available to the coder when needed while still having the domain specific syntax. One downside of the internal DSLs is that they piggy bag on the mother GPL and therefore still conforms to the syntax limitations of that given language. Another consequence of making an internal DSL is that error reporting is done through the GPL, often resulting in complex stack-traces etc.

External DSLs are not bound by any parent language syntax, and error messages are also custom created by the DSL author. The downside is that everything needs to be implemented by oneself.

We chose to make an internal DSL in the Groovy language[14]. Since the rest of the framework is coded in Java, it made sense to take a JVM based language. [35] lists several JVM languages suited for writing internal DSLs. Groovy was chosen because of its large resemblance in syntax to Java as well as its relative ease of developing a DSL in. The language homepage has a lot of documentation on how to write a DSL, as well as it has the book cited above. Groovy also has built-in support for scripting enabling us to change the DSL at runtime instead of compile time, making adaption of changes in formats and new tools faster.

3.7 DSL

When making a DSL, there are several things one must take into consideration. Is the DSL going to be graphical/visual or text based? How

are the grammar and concrete syntax of the DSL? According to [47] programmers prefer text based DSLs over visual DSLs. As Fowler[34] argues, experience shows that a DSL should not read like natural language as it often leads to a complicated understanding of the semantics with the added syntactic sugar.

According to [28] there are some design principles when designing a DSL:

Identify the purpose of the language: Our DSL is used to describe the parsing of data from an arbitrary format to our format defined above. It is going to be used by people with software development skills where tools like XPath and String manipulations are familiar concepts.

Keep the language simple: We modeled our DSL around the Entity Java class as it is mirroring the data format described above. It is also the message format we send over the wire. When designing the internal DSL, we only made common scenario methods available in the DSL. If more advanced things are needed, the whole Groovy language can be used inside the DSL.

Test the language with dynamic instances: We have designed the language by testing it up against three different kinds of tools and multiple different instances of data emitted from each tool. That way we uncovered features that the language needed to posses in order to fulfill its purpose.

Use the problem domain as inspiration: We modeled the vocabulary towards the vocabulary of the problem domain; parsing. As it is a technical DSL, there are many technical terms in it. We tried to hide the underlying GPL data structures when wanting to evaluate to a given format. So instead of the user writing *java.lang.String* she could just type *String*. In that sense only defining the datatype as an abstract concept, but not knowing the internal implementation class we try to make the language more alligned with the problem domain instead of the underlying GPL language.

An example of the concrete syntax of our DSL looks like the following. For comparison, this is also targeting Jenkins as its source as the parser file in Data parsing section:

```
1  parser 'JSON'
2  entity {
3      id = eval('$.fullDisplayName')
4      prev = eval('$.changeSet.items[*].id','java.util.List')
5      url = eval('$.url')
6      type = (eval('$.building')=='false')?'Jenkins building
           stopped':'Jenkins building started'
7      data = all
8      healthy = (eval('$.result')=='FAILURE')?false:true
9      timestamp = new Date(eval('$.timestamp', 'java.lang.Long'))
10 }
```

Listing 3.4: DSL code for parsing Jenkins files

The first line defines the parser used. This can be JSON, XML or the like. The parser needs to match the input format given from the target tool. Thereafter we build the instance of class *Entity* by extracting the relevant parts of the targeted tools data into the properties of the *Entity* class.

Groovy has optional parentheses, so if a method contains at least one argument we can leave out the parentheses. But there must be no ambiguity before the omissions can happen, or the results can be unexpected. Consequently, we recommend the use of parenthesesfor the sake of clarity.

In this example, we reduced the amount of code needed to be written by 66% compared to the parser example in Data parsing. Besides the Entity keywords, a collection of other keywords has been designed for the DSL:

parser indicates what parser needs to be used. The only implemented data format at the time of writing is JSON, but the architecture is made so that all data formats can be supported.

all inserts all the target tool data into the property. Used for the *data* property of the data format.

eval parses the given string into the data received from the target tool. Uses JSONPath for JSON and will eventually use XPath for XML etc. This method can take one or two arguments:

eval(String key) evaluates the key and parses the result to a String representation.

eval(String key, String returntype) evaluates the key and parses the result to a representation of the given return type. To hide the underlying data structures of Java, the user only needs to write *list*, *string*, *long* or *boolean* to get the equivalent classes in Java as return types.

array is used to combine one or more parsings into one array. Removes null references as well.

parsedate uses the Java *SimpleDateFormat* class to parse a given string into a Date object.

With this small set of keywords in our concrete syntax, we target common scenarios our DSL must encounter. Furthermore, when more advanced features need to be tackled it is easy to make GPL methods available.

Interface We still need to be able to plug in several parsers that can be utilized by our DSL. In that way, new data formats can be added to support the DSL as long as they confine to the four methods below:

```
1 package dk.itu.tracy.dsl;
2 public interface Parser {
3     String eval(String key);
4     public <T> T eval(String key,String clazz);
5     void setup(String input);
6     String toJson(String input);
7 }
```

As long as a given dataformat can be parsed through the use of queries (i.e. XPath for XML, YPath for YAML files), it can be implemented in the DSL.

By making this DSL we separated the file format (XML, YAML, JSON) from the data format of the specific tools data structure. The code that handles the file format can then be reused every time that particular format is present. The only thing that needs to be described per tool is how to parse the different data structures that reside.

3.8 Overall framework design

During this chapter, we have analyzed and presented parts of the chosen design for the framework. We have looked at the problem do-

main, the environment that the framework must operate under, and designed a framework on the basis of the Findings. This section will try to give a broad overview of the architectural design of the framework, seen from a deployment perspective. For full sourcecode, see appendix Code. The framework consists of three components written in Java or

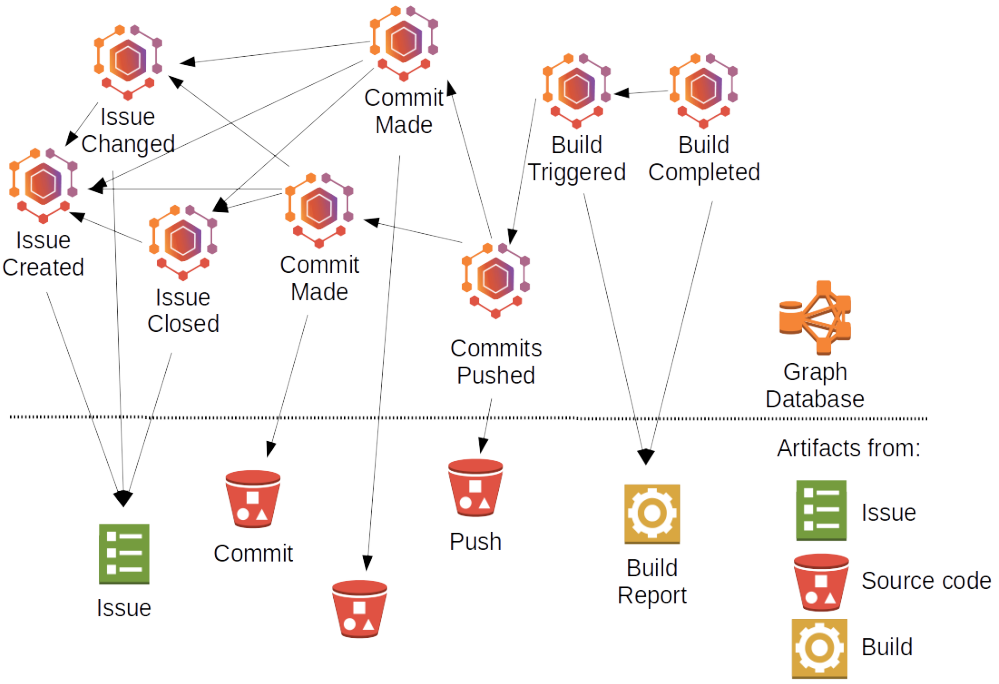


Figure 3.5: Abstract illustration of the correspondence between the events captured by the framework and the individual tools artifacts. Icons are taken from AWS Simple Icons set [2].

Groovy using Maven as the dependency manager; Tracy, REST2AMQP, and AMQP2NEO.

Tracy is the main component of the framework. It consists of the following packages:

CLI is a small java main class that takes three arguments; properties for RabbitMQ connection, data from the target tool and the DSL for parsing the data file parsed.

DSL is the package containing the Groovy DSL as well as the surrounding code that parse data from Java to Groovy and back. Is also includes the interface for implementing new parsers.

Entity only contains the data transfer object (DTO) that is used to internally represent the JSON data format.

Facade contains the code that communicates with Neo4J and RabbitMQ respectively as well as a parser to and from JSON and DTO.

REST2AMQP is a REST enabled wrapper around Tracy. It utilizes Javas JAX-RS technology with the reference implementation Jersey.

AMQP2NEO is the bridge between the AMQP messages and the Neo4J graph database. It utilizes the classes and methods of Tracy to receive and persist the messages.

The point of the architecture design is to separate the logic of the framework itself from its endpoints. In that way, the capabilities of the framework can easily be integrated into other areas where it is needed.

One could argue that the CLI needs to be in a separate module. The reason we included it in the Tracy component was based on the fact that it was just one Java class with a main method. In that way, we do not have to maintain a whole component for one class.

Chapter 4

Framework validation

Based on the outcome from the industry interviews, described in the Findings subsection, we will validate the designed frameworks ability to fulfill the capabilities found.
Unfortunately neither of the two companies interviewed had the possibility to set up the framework in their respective environments, but company A has been consulted in the making of the test cases in order to ensure that they mimic their environment to best effort. We are therefore confident that the validations in this chapter can be transferred to real world environments. The first section will describe our test setup that emulates an SDLC pipeline. We outline the concrete tools and their qualifications. The following section will explain the test cases constructed in order to generate events from the tools chosen. The third section describes the results generated and the evaluations whether or not the tracelinks based on the data collected in the graph database have been made.

4.1 Test setup

For our test setup we have chosen to include three systems; a task management system, a version control system and a build system. These tools are core components in all SDLCs, and it is deemed very likely that the framework's ability to generate tracelinks can be extended to all other tools in a given setup. This section will describe the specific tools chosen for the setup as well as the DSL generated to parse their data.

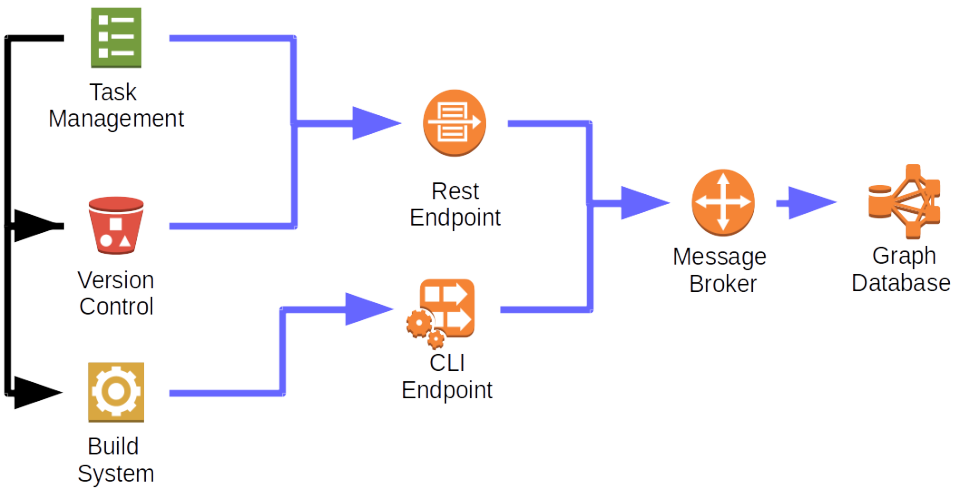


Figure 4.1: Illustration of the test setup. The black arrows illustrate the flow of inter-tool messages. The blue arrows illustrate the framework messages and endpoints.

Task management The tool chosen for the setup is Jira[18] from Atlassian. It provides issue tracking and project management. One of the main reasons for choosing this is that both interviewed companies are either using it or have an interest in evaluating its capabilities. It has a very simple set of artifacts from the start and can be fully customized in terms of workflow and properties on the given artifacts.

Jira has webhooks for emitting an event whenever there is a change in the system, be it workflow related or user management etc. We have configured Jira to only send events concerning the tasks. An example of one such data file is displayed in the appendix Example of tool emitted data.

In order to pass the several hundred lines long file into our entity structure, we have implemented a parser in our DSL:

```
1 parser 'JSON'
2 entity {
3     id = eval('$.issue.id')
```

```
4     prev = array(retrievePrev(),eval('$.issue.id'))
5     url = eval('$.issue.self')
6     type = eval('$.webhookEvent')
7     healthy=true
8     data = all
9     timestamp = new Date(eval('$.timestamp', 'java.lang.Long'))
10 }
11
12 def retrievePrev(){
13 String issuetrack =
      eval('$.issue.fields.comment.comments[0].body')
14 if (issuetrack!=null)
15 return issuetrack.substring(issuetrack.lastIndexOf(']')-39,
      issuetrack.lastIndexOf(']'))
16 return null
17 }
```

Listing 4.1: DSL code for parsing Jira files

Unlike the Jenkins DSL parser described in listing 3.4 we had to make a GPL based method in order to get the SHA-1 id from the Gitlab commit since it was not available in a property of its own. This example illustrates well the benefits of having a mother GPL language to engage with when special cases need to be coded.

Version control system (VCS) The tool chosen as VCS is Git, an open-source distributed VCS that rapidly is becoming the defacto standard system to use. It is used in one of the most complex and actively developed codebases in the world, the Linux kernel. We use a repository management system on top of Git called GitLab. It is similar to e.g. Github and gives a web based interface as well as security management. Gitlab also has webhooks to send messages whenever a push is sent to a given repository.

The DSL implemented for this data is a bit different than the two others. It needs to generate multiple entity classes for one data file. One for the overall push and one for each of the commits contained in that push. Lines 5-13 take care of making an entity for each commit, and lines 16-24 make the push entity, with the reference back to all the commits.

```
1 parser 'JSON'
2
```

```
3  list = eval '$.commits[*].id','List'
4  list.eachWithIndex { val, idx ->
5     entity {
6         id = eval('$.commits['+idx+'].id')
7         prev = ""
8         url = eval('$.commits['+idx+'].url')
9         type = 'commit'
10        healthy=true
11        data = all
12        timestamp =
              parseDate(eval('$.commits['+idx+'].timestamp'),
              "yyyy-MM-dd'T'HH:mm:ssX")
13    }
14 }
15
16 entity {
17     id = eval('$.before')+eval('$.after')
18     prev = eval('$.commits[*].id','List')
19     url = eval('$.repository.homepage')
20     type = eval('$.object_kind')
21     healthy=true
22     data = all
23     timestamp = new Date()
24 }
```

Listing 4.2: DSL code for parsing Gitlab files

Build system The tool chosen as build system is Jenkins. It is the leading open source automation server with over 100,000 installations worldwide, and over 1000 plugins. It is used for testing, building, packaging and deploying software in all kinds of languages. Both companies are using it as well.
The DSL created for parsing Jenkins files is shown in listing 3.4. In that, we made use of ternary operators that stem from the GPL.

This setup enables us to perform test cases in a simplified environment and evaluate the usefulness of the framework based on those cases.

4.2 Test cases

In order to evaluate the framework's ability to comply with the capabilities, some test cases need to be conducted. A collection of four test cases where constructed. They represent some common scenarios in a regular SDLC and will serve as the basis for the evaluation of the framework. Each test case is listed below with a description of the case as well as a reasoning for its existence.

1: Simple Make a task, start the task, deliver some code, start a build, close the task.
This is the simplest way to trigger all the tools involved in the setup and also resembles the very basic setup of small simple repositories.

2: Branching Make a task, start the task, make a branch, deliver some code, start a build, merge the branch, close the task.
This scenario covers the branch by feature/activity strategy described in [53, 8].

3: Branch/Build/Merge/Build (BBMB) Make a task, start the task, make a branch, deliver some code, start a build, merge the branch, start a build, close the task.
The test case simulates a more advanced scenario where the build servers act as the safeguard to the master branch[20]. Only if the build succeeds, the commits from the branch are merged into master. This strategy ensures that the master branch is releasable at all times.

4: Build/Build Make a task, start the task, deliver some code, start a build, start another build, close the task.
This test case is primarily made to test the Finding 3 compliance. Here, two builds from the same repository and set of commits need to be verified as containing the same code. The two builds are made because of different environments in the simulation test facilities and in the real test vehicles. To simulate that situation we set up two Jenkins builds to run. When one has successfully run without failure, it starts the next build.

Even though its primary reason for this test case is the ISO26262 compliance, case company A, which does not operate under these

compliance rules, has similar scenarios in their pipeline e.g. when one codebase gets compiled to several platforms.

A short video have been produced to show you how the simple test-case is running in the test setup. The url for the video is the following: `https://www.youtube.com/watch?v=lrNjMbQXpjU`

4.3 Results

We have structured this section by the findings listed in the Findings subsection. For each finding, we will run the appropriate test cases and display the results generated. We will also evaluate whether the capability has been fulfilled or not. We will not produce any results for Finding 5, as stated before. This is due to a completely different capability of the framework that has been deemed out of scope for this thesis. Finding 3: *Case A: making compliance with ISO 26262* is a special case as well, making it only applicable to test case 4. It has therefore only that test case as the target.

For every test case run, the graph database will collect all the events translated into event nodes with the data defined in our format, and sub-nodes with all the tool specific data. To illustrate what the data look like and the amount of nodes generated, figure 4.2 shows a scenario where one commit is pushed to a given repository, following a Jenkins build which has a start and stop event.

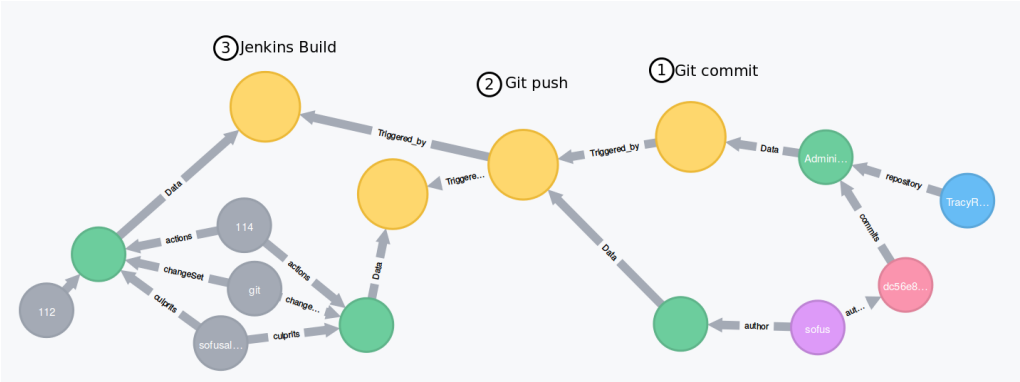


Figure 4.2: Illustration of event nodes (colored yellow), and all the tool specific data (any other color).

The illustrations are taken from Neo4J's cypher query browser. In future

illustrations, we are filtering all the sub-nodes to give a cleaner view of the graph.

4.3.1 Finding 1: Ability to visualize the pipeline from a given event.

For every test case in this subsection a graphical illustration of the nodes is being displayed. Listing 4.3 shows the query used to get this graph illustration. It starts with one event node in the graph and retrieves all the corresponding event nodes connected to it. Therefore, it does not matter which one of the event nodes we retrieve as our query is direction agnostic.

```
1 MATCH (n:Event)-[r:Triggered_by*]-(m:Event)
2 WHERE n.id={event id}
3 RETURN n,r,m
```

Listing 4.3: Cypher query for visualizing the SDLC pipeline

1: Simple gives the following result on the simple test case:

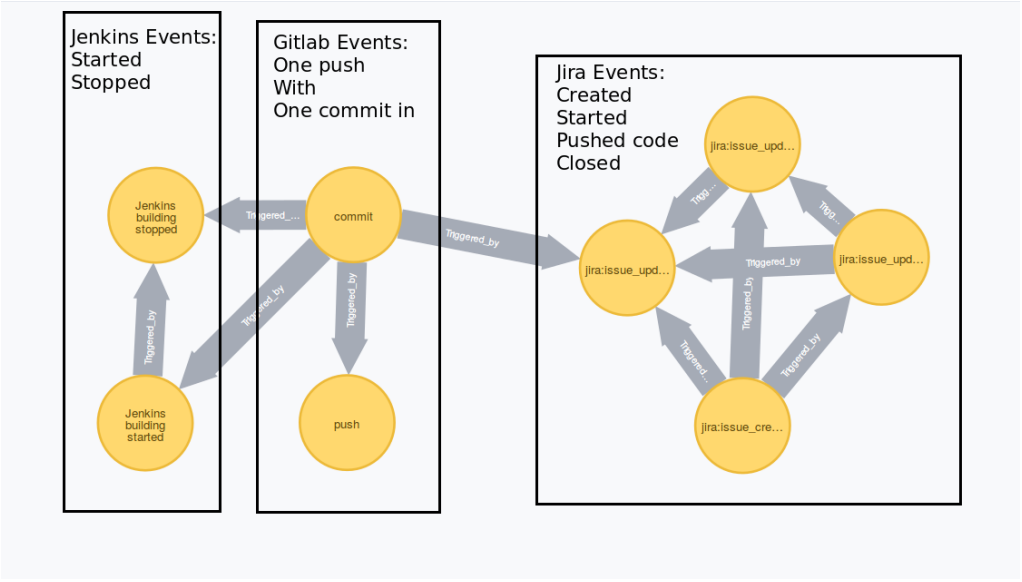


Figure 4.3: Illustration of the events collected in the "simple" test case.

In figure 4.3 we can see that all the events emitted in the test case have been traced, and the tracelinks have been created. The graph shows two nodes for Jenkins. This is because we put a call to our framework both

the pre and post condition of the build. Likewise there are four nodes representing Jira. They are there because there are four events happening to that artifact (issue): the creation of the issue, the phase change from created to start working, the reference to the pushed commit and the closing of the issue in the end.

Because all nodes have been connected correctly, we see this test case as successfully fulfilling the capability.

2: Branching When making this test case, we start up with a master branch. We then create another branch called development, and push changes to that. When done, we let Jenkins take the development branch, and build and test the code. If the tests are successful, we will merge the commits from the development branch into the master branch as illustrated in 4.4. In this example, only a single commit was made, but it makes no difference since the commits all will be built and merged.

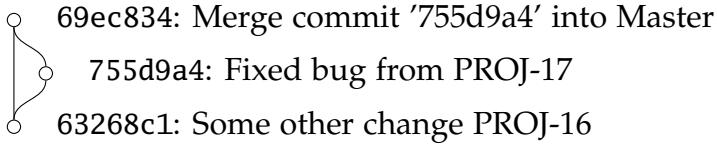


Figure 4.4: Illustration of the two branches *master* (left) and *development* (right). A change is made at development and merged back to master.

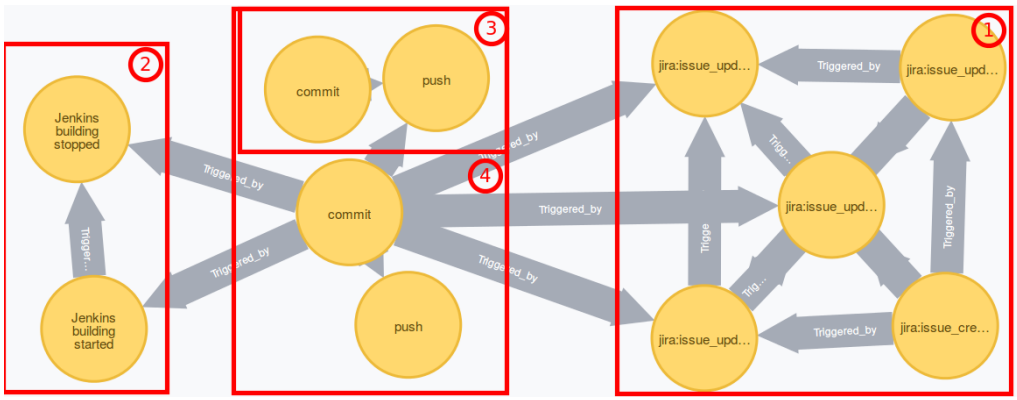


Figure 4.5: Illustration of the events collected in the "branching" test case. (1) are Jira events, (2) are Jenkins events, and (3) and (4) are the merge commit and commit with the actual change respectively.

In figure 4.5 we can see that all the events emitted in the test case have been traced and that the tracelinks have been collected. Therefore we see this test case as successfully fulfilling the capability.

3:Branch/Build/Merge The important thing from this test case to the one before is whether or not both builds will be traceable back to the same commits even when they build from different branches.

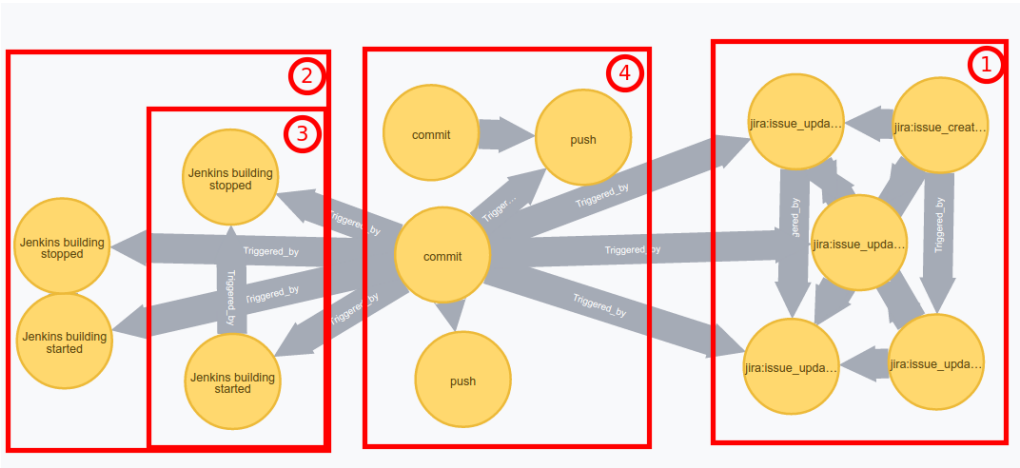


Figure 4.6: Illustration of the events collected in the "BBMB" test case. (1) are Jira events, (2) and (3) are Jenkins events, and (4) are the merge commit and commit with the actual change respectively.

As seen on figure 4.6 both build events have made tracelinks back to the commit. Therefore we see this test case as successfully fulfilling the capability.

4:Build/Build Seeing how two builds in the last test case were successfully traced, one could imagine that this test case is superfluous. But the two builds are from the same branch of code and for that reason interesting enough to research.

Figure 4.7 is almost identical to figure 4.6 except for the merge commit made in the latter. We deem this test a success as well.

4.3.2 Finding 2: Ability to measure lead time between two events

The result here is finding out how long a given pipeline has run. By continuously querying several instances of the pipeline we are able to see if

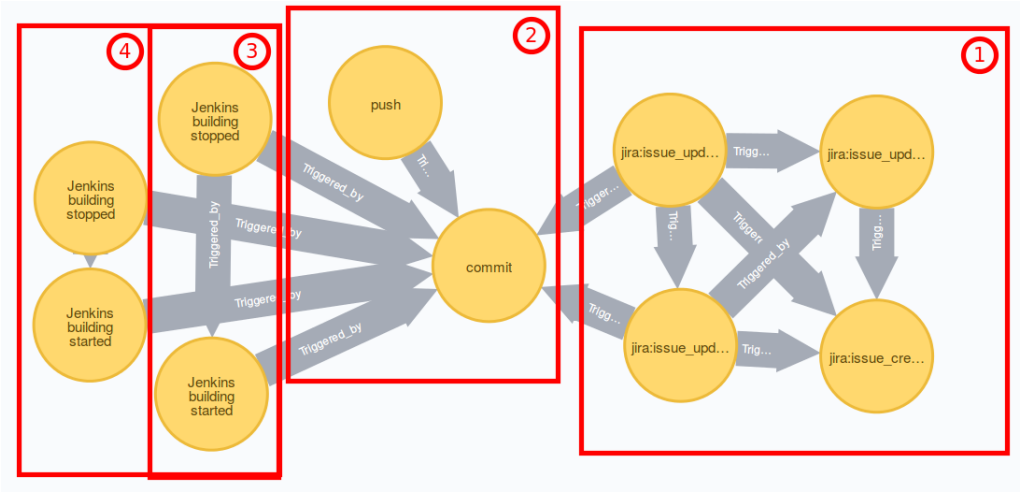


Figure 4.7: Illustration of the events collected in the "Build/build" test-case. (1) are Jria events, (2) are Git commit and push events, and (3) and (4) are the two Jenkins builds that refer to the same commit

there is an increase in the duration, enabling us to identify bottlenecks etc.

```
1 MATCH(n:Event)-[r:Triggered_by*]-(b:Event)
2 WHERE id(n)={event id}
3 RETURN (max(b.timestamp)-n.timestamp)/1000
```

Listing 4.4: Cypher query for finding out the duration of a given pipeline

The Cypher query shown in listing 4.4 takes the start node and traverse from that out to every Event node in its graph. The node that has the largest unix timestamp is then compared to the start node and the returned result is the number of seconds between the two events. As these results vary depending on the time it takes to perform the tests, there is no way to reproduce the test and get the same result.

By manually examining the timestamps on the nodes in the graphs created, the results in this test scenario have been verified. We deem this capability successfully tested.

4.3.3 Finding 3: Case A: making compliance with ISO 26262.

The query created to test this capability is somewhat the same as the query for Finding 1, with the exception that we are only interested in the build events created by Jenkins:

Test case	Result
1: Simple	196 seconds
2:Branching	116 seconds
3:Branch/Build/Merge	3363 seconds
4:Build/Build	320 seconds

Table 4.1: Lead time between issue creation and the last event of the pipeline

```
1 MATCH(n:Event)-[r:Triggered_by]-(b:Event)
2 WHERE n.id IN["{git SHA-1}"] and b.type STARTS WITH "Jenkins"
3 RETURN n,r,b
```

Listing 4.5: Cypher query for getting the builds connected to an array of commits.

4:Build/Build The query successfully returns the four events representing the two builds made in the pipeline as seen in figure 4.8. We therefore deem this capability successful as well.

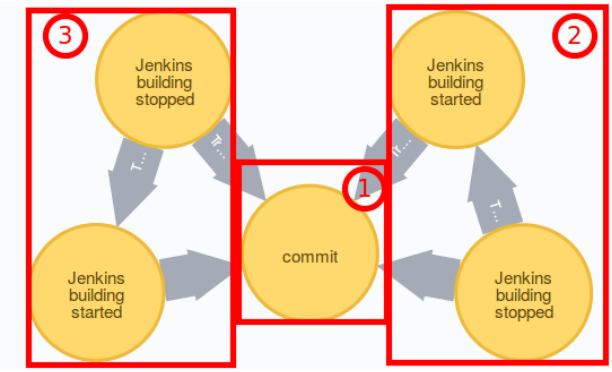


Figure 4.8: Illustration of the events collected in the "Build/build" test case. (1) is the Git commit, (2) and (3) are the two Jenkins builds that refer to the same commit.

4.3.4 Finding 4: Management: Has this issue been resolved in this release?

As we do not have any artifact management software in the test setup it is not possible to show the full set of tracelinks from issue to artifact. But since Finding 4 is basically a specialized version of Finding 1, we

treat it as fulfilled if Finding 1 is so. In order to retrieve only the set of nodes that is needed to fulfill this capability, the query displayed in listing 4.6 -where n is the Jira issue and b is the Jenkins build- needs to be examined.

```
1 MATCH(n:Event)-[r:Triggered_by*]-(b:Event)
2 WHERE n.id IN["{Jira Issue id}"]
3 AND b.id IN ["{Jenkins build id}"]
4 RETURN n,r,b
```

Listing 4.6: Cypher query for checking if a given issue is covered in a build.

4.4 Summary

In this chapter we have conducted a series of test cases to ensure that the constructed framework had the desired capabilities. For each test scenario the resulting graph has been shown, and for every finding a matching cypher query has been constructed to retrieve the desired data from the database.
The framework has during the test cases performed shown to possess the capabilities wanted. We have shown that the framework is able to make the tracelinks by parsing tool specific data emitted from the tools selected for the test.

Chapter 5

Limitations

In this chapter we will touch upon some of the limitations encountered either in the validation method or in the original thesis delimitation. The chapter is divided into sections that each represents an identified limitation. Assessments of the severity of the limitation, as well as possible mitigations, will be discussed through each of them.

5.1 Validation

It is not possible to test every given setup and function that a normal enterprise will come across. In order to tell with certainty if the framework has actually attained the desired capabilities, it needs to be deployed to production at a company.

Because we were unable to do that, we created a test scenario with a small sample of test cases. The small sample size of test cases represents normal operations in an SDLC and we would argue that more advanced setups are just alternations of these 4 test cases. Furthermore, we have collaborated with one of the companies to ensure that the tests conducted resemble that of a real environment. But as we have not tested the framework in production, we have not been able to tackle the unforeseen circumstances that are always present in practice.

5.2 Performance

The solution created in this thesis is designed towards enterprises, and it is therefore important to validate its abilitiy to function in large scale

projects. The tests conducted in this thesis does not account for performance and stress testing of the framework in any way. Likewise there has been no performance optimization of the framework. An example of such lack is that every Cypher query takes up a session to the database, and all queries are run in isolated transactions. Every part of the framework runs single threaded.
All these examples could be changed to improve efficiency *if* the framework hits a bottleneck.

The framework produced is seen as a prototype, a way of investigating *if* this approach was feasable or not. When designing it we have made sure that the technologies chosen have been industry proven.

5.3 Version control traceability

When dealing with Git as a VCS, there are some scenarios where the traceability inside the VCS itself will be broken, making our graph broken as well.

cherry-picking When cherry-picking a commit from one branch to another inside a repository, the id of that commit changes. This brakes traceability when e.g. one bugfix is being implemented in several branches (Current, version 1.0, version 0.9). This problem is somewhat mitigated by mentioning the task in the commit, making the task management system aware of the commit. But for our tool, it will seem like there are three different commits being made to solve the same bug.

rebase Likewise cherry-picking, rebasing illustrates the same problem. If you change the path on which the commit is located in the repository, the id changes. This has some consequences outside traceability as well, making the authors of git come with this word of causion on using rebase[10]:

> Do not rebase commits that exist outside your repository.

In the context of the quote, "your" refers back to the users local repository, meaning that you should not rebase commits that any other user depends on in their branch.

These examples are Git specific, but other versioning tools have similar limitations. The limitations concerning the VCS would also be

present in any other traceability tool, as the limitiations are created internally in the target tool.

5.4 Event and Artifact id namespace

Every event in our traceability model needs to have an id. That id is usually the related artifact's id that the event then inherits. The id is unique if the event and artifact are the same and therefore have a 1:1 relationship. An example is the Git commit, as the commit both represents the event and the artifact.
However, this is often not the case.

One event, several artifacts Some events concern several artifacts. A Git push is an event itself and does not have a particular artifact connected. Instead, the push represents a transfer of commits from one repository to the next. In that case, we need to make up an artificial id for the event as shown in the last entity made on listing 4.2 line 17. The artificial id makes it hard for any other event to refer back to that event. That is the only one of this kind discovered during the thesis writing. In this particular case the push itself is not responsible for alternations, giving it an informative role only.

We argue that that would be the logical case for any other event that does not have an artifact underneath, because if an event alternates an artifact, it will have knowledge of that particular artifact, giving it an id.

One artifact, several events A single artifact can also have several events related to it. For example, Jenkins has a build (artifact) that starts and stops (events). Jira has an issue (artifact) that is created and optionally updated (event). As a result, there will be several events in the graph with the same id, which can be seen in the Results section, e.g. at illustration 4.6 in selections 1, 2 and 3. This devaluates the ability to make tracelinks on an event level.

We solve this by making the reference from Jira (issue) towards Gitlab (commit). In that way, it is only the specific updates to an issue that actually reference a commit that gets the tracelink. But this is only possible if one of the sides of the tracelink has a 1:1 relationship to its artifact.

This limitation lacks a good mitigation strategy. In order to identify each event individually, we need to set some requirements on the toolstack that cannot be honored by several of the tools investigated. Jenkins only has an id to the build, and not to its start and stop. In the same way, the id is unique on an issue basis in Jira. Therefore, we cannot enforce an event level unique id schema without excluding some tools for which reason we need to make due with the current level of uniqueness in the data model.

The level of uniqueness is fine-grained enough for our framework to possess the capabilities wanted, as the results shows. Therefore, we do not deem this as a severe limitation.

5.5 Schemaless types

As stated in the Analysis we wanted to take a different approach to traceability than found in the literature by not having a fixed set of types, but only a single supertype with the flexibility of naming the type what suits the company. This has been impossible to evaluate on since we have not conducted trials of either of the approaches in real life environments. In the test setup used in this thesis, only a fixed set of types was used, but these can change from one setup to another, from one toolstack to the next. When looking at the proposed schemas in Literature review, it is clear to see that the schemas not only try to make horizontal traceability across the toolstack but also vertical traceability inside each tool section. Looking back at figure 2.1, the framework proposed here in this thesis only covers about half of the types presented, i.e. Tests, Implements, Refines, Documents, and maybe Realizes depending on the context. That makes a comparative study up against the schema-full and schema-less approach difficult to conduct. In defense of our approach, one could state that we considered the approach chosen suitable enough for the purpose specified.

5.6 Summary

In this chapter we discussed some of the limitations that we encountered either in our thesis delimitation or in the conducted framework validation. For every limitation listed, an explanation, assessment and optionally a mitigation strategy were presented. These limitations help

us in evaluating the results produced in the Framework validation chapter.

Chapter 6

Future research

A thesis is a compact project only spanning 4 months from initiation to finish, making the scope of the project focused in order to produce results in that time frame. For that reason, a lot of things were deemed out of scope although still important if this framework would see real world adoptation. This chapter describes some of the activities that could be relevant from an academic and/or professional perspective.
As in the Limitations chapter there is a section dedicated to each of the topics. Each section explains the possible branch of research and describes its usefulness.

6.1 Visual tool/front end

As stated in the Thesis delimitation section, having a visual front end that has great usability and ease of use is very important in a successful adoption of tools in any SDLC. As [44] puts it:

> Tools are central enablers, like it or not.

The more available a tool is, the more it will be used. An example of such visualization tool is figure 2 in [31] also displayed in this thesis as figure 6.1. The figure shows a screenshot of a tool where a given developer can see the event chain from a commit throughout the pipeline.

Furthermore, a study of usability could also aid in uncovering more use cases that the framework could be applicable in helping. In much the same way as [46] describes production of software in research, some

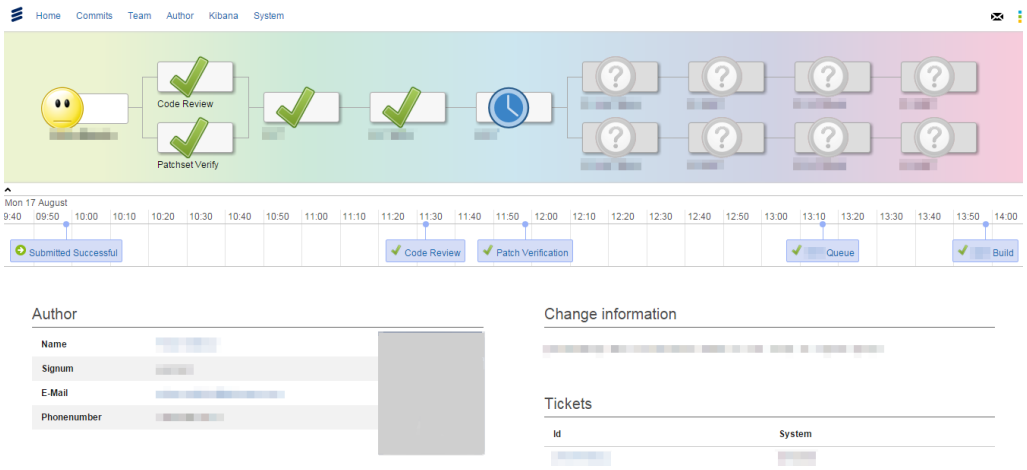


Figure 6.1: A developer-centric "Follow Your Commit" visualization of the Eiffel framework

things do not pop up as possibilities, problems or alternations before a prototype of the software is used.

6.2 Log files aka *expostfacto* events

With enterprise companies, more often than not, the software components developed are not newly started but have had years of development behind them. That means that the framework constructed here would be inserted in an already running SDLC with lots of artifacts created.
By utilizing historical data, the framework could benefit the traceability needs in a given SDLC from the instant it was implemented.

One way to suck up historical data is to look into the log files of the given applications. Log files are usually line based, with poor formatting compared to the JSON event data formats that have been worked on. The focus of an application log is more on the system's performance and health than on what the users of the system are doing. A topic of research could therefore be to analyze and make parsers that take log files instead of event calls and make the tracelinks in the graph database.

Another way could be to make extractors from the database of each application. This, however, requires a deep knowledge of the internal structure of the data in the given database. Database schemas are usually also considered internal to the application, with changes occurring without warning.

6.3 Named relations

As stated in the Limitations chapter, we did not evaluate whether or not working without a fixed set of relation types was beneficial. Right now the type of relation that is established between two nodes in Neo4J is hardcoded as *triggered_by*, as also shown in figure 3.3. This makes it impossible to establish the relationship types illustrated in figure 2.1. In order to make this type controlled by the user, it must be inserted into the data format. It is a small change in the DSL engine, and the primary reason this is not implemented is that the capabilities required from the interviews did not require this. But in order to gain a more detailed traceability like the ones described in [52, 51], the proposed change is needed.

The benefit of gaining named relationship types is that more generic queries to the database can be made.
Imagine that we want to display all builds to a given issue:

```
1 MATCH(n:Event)-[r:Triggered_by*]-(b:Event)
2 WHERE n.id IN["{Jira Issue id}"]
3 AND b.type STARTS WITH "Jenkins"
4 RETURN n,r,b
```

Listing 6.1: Cypher query for finding the builds related to a given Jira issue

Listing 6.1 illustrates that with the current implementation, we need to know what the type name of the given build events is. Here they start with "Jenkins", but if the company is using Gerrit, Travis or any other build tool, this query would not work.

```
1 MATCH(n:Event)-[r:Triggered_by*]-(b:Event)-[re:Build]-(e:Event)
2 WHERE n.id IN["{Jira Issue id}"]
3 RETURN n,r,b,re,e
```

Listing 6.2: Cypher query for finding the builds related to a given Jira issue.

In listing 6.2 we have deleted the part of the Cypher query where Jenkins was mentioned and instead inserted a *build* relation in the path pattern. Neo4J supports multiple labels to each relation and node, allowing both a general *triggered_by* label as well as a specific one like *build*.

6.4 Alternatives

While doing research for the thesis, we got in contact with an employee at Ericsson who worked with traceability inside the company. Ericsson has their own self build tracing and orchestrating system called Eiffel [11, 31], also mentioned in the Literature review section. Their system combines the traceability features researched in this thesis together with an orchestration mechanism that Finding 5 is requesting. It uses a fixed set of messaging formats passed through a central message broker and stored in a NoSQL document database. The protocol on which their framework relies is being open sourced and further developed on Github[11]. The underlying architecture remains at this time proprietary and there are no known publicly available implementations of the protocol yet. [31] displays good results in the usage of the framework internally.

Continuous delivery orchestration. If Finding 5 was to be pursued, it could be beneficial to incorporate the Eiffel protocol as it has been tried and tested in a production environment at a large enterprise. As the implementation around the protocol is not open sourced, one needs to implement the architecture. The framework presented in this thesis could serve as a foundation for the new architecture, taking advantage of the DSL language and graph database while extending the clients to also be consumers and therefore triggering mechanisms. This would remove the flexible type system as the protocol has a fixed set of types associated with it, but the framework would gain time-proven efficiency from using a protocol that has been in production for a long time.

Chapter 7

Conclusion

The goal of this thesis project was to tackle the horizontal traceability problem that enterprise companies have while they are developing software at scale. Large software vendors are claiming that they achieve both vertical and horizontal traceability as long as you use their software everywhere in the tool stack. This makes the selection of best-of-breed tools impossible and enforces vendor lock-in.

When having a heterogeneous software tool stack, integration rarely happens to more than the neighboring tools in the pipeline. With integration restricted to the neighboring tools only, horizontal traceability is not present.

This thesis tries to tackle the problem of horizontal traceability in a heterogeneous tool stack. In contrast to most efforts and literature published, this thesis tried another approach where the subject for traceability is the events, not the artifacts themselves. By combining this event based approach to traceability, defining a DSL for parsing data from any possible tool, and leveraging the strengths of a graph database into a prototype framework, we have shown that this approach is capable of making the horizontal trace lines that the industry wants. The framework leverages the fact that when a tool is activated in the pipeline, it knows who the activating "parent" is and is thus able to make the wanted horizontal traceability.

Through case company interviews we defined capabilities that the framework must possess. One of the capabilities found in the interviews were excluded because of the large implication to the framework. If this capability of a pipeline triggering mechanism was to be implemented,

a recommendation to include the protocol open sourced from Ericsson has been given.

The framework has been tested whether it fulfills the remaining capabilities through several test cases. The framework passed all test cases constructed, and validated all the capabilities chosen. The artificial test scenarios were, albeit validated by one of the case companies, a limitation to the study. In order to gain more confidence in the framework constructed, its performance and ability to function in real-world scenarios, industry tests need to be made as well. In order to get the best evaluations and secure a high level of adoption of the framework in a real world scenario, a visual front end that is easy to use must be made.

There is a strong need for efficient traceability in the software development industry. This thesis presents a prototype of a generic framework. It is capable of integrating into any heterogeneous tool stack the company has in their SDLC, giving much wanted horizontal traceability.

7.1 Acknowledgement

A lot of people helped during the process of working with this thesis. They have made small and large contributions from technical discussions, proofreading, opening doors to contacts in the case companies etc.
I want to thank the following people:
My supervisor Thomas Hildebrandt.
My friends Laura Hauch and Jens Egholm.
All the people at Praqma, especially Lars Kruse, Andrey Devyatkin and Thierry Lacour.
The JOSRA group.
Employees at the two case companies.
Daniel Ståhl from Ericsson.
Without these people, the thesis would not have been.

Appendix A

Mail correspondance

Sofus Albertsen <sofusalbertsen@gmail.com> 20. apr. (for 9 dage siden)

t

He

Jeg håber i kom godt hjem til DK.

Jeg har lyttet til optagelserne af diskussionen i Tronhjem og udledt et par generelle udtalelser omkring sporbarhed, som jeg håber du kan nikke genkendende til.
Hvis der er nogle af udtalelserne der ikke passer til jer, eller er upræcise, så skriv endelig tilbage.
Det vil betyde meget for mit arbejde med specialet hvis jeg må bruge som case virksomhed. Det vil betyde at jeg skriver at har disse behov. Det vil derfor ikke kræve noget andet fra jer, en nogle informationer om jeres softwareudviklings teams (hvor mange er i, hvor mange lande er der teams i, etc).

Jeg har identificeret 5 punkter som blev udtalt som behov oppe i Tronhjem.
Pkt. 4 var fra der bragte det på banen, og jeg tænker ikke er ISO26262 har noget med jer at gøre. Så hvis i føler der er det samme behov for juridisk compliance, så må du meget gerne skrive hvilken myndighed eller lign i skal være compliant overfor.

As a Developer:
1: Ability to visualise the pipeline from a given event.
Example: Who is using this commit / my component
Where is my commit in the pipeline

2: Ability to measure lead time between two events.
Example: From Issue creation to first commit.
From commit to executable creation/product release.

3: Want to have an event mechanism that can drive the continous delivery pipeline and that is tool agnostic.
Example: Every tool should be able to trigger a build in the pipeline.

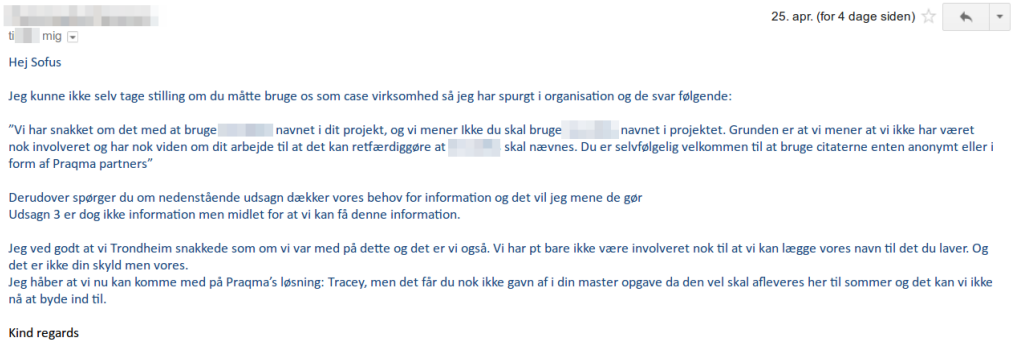
4: Legal issues: making compliance with ISO 26262
Compare two builds from identical changesets (ISO_26262)

Management:
5: Has this issue been resolved in this release?
Is all commits connected to this issue been incoorporated in this release.

Du må meget gerne skrive tilbage om hvilke af disse du mener er relevante for .
Og så håber jeg meget at jeg må skrive jer på som case virksomhed.

Med venlig hilsen

Figure A.1: Case A: Response



25. apr. (for 4 dage siden)

til mig

Hej Sofus

Jeg kunne ikke selv tage stilling om du måtte bruge os som case virksomhed så jeg har spurgt i organisation og de svar følgende:

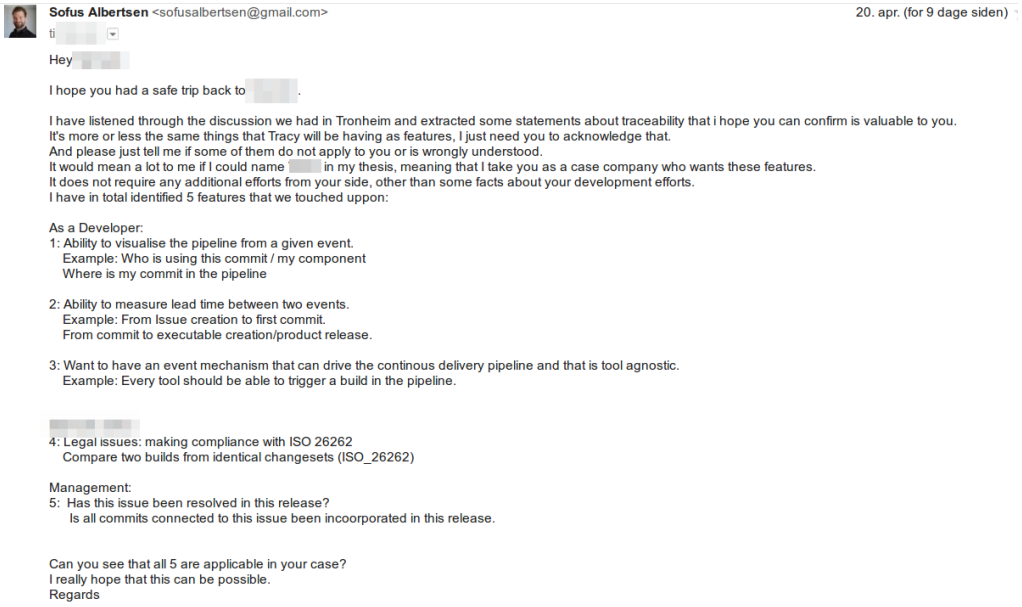
"Vi har snakket om det med at bruge navnet i dit projekt, og vi mener Ikke du skal bruge navnet i projektet. Grunden er at vi mener at vi ikke har været nok involveret og har nok viden om dit arbejde til at det kan retfærdiggøre at skal nævnes. Du er selvfølgelig velkommen til at bruge citaterne enten anonymt eller i form af Praqma partners"

Derudover spørger du om nedenstående udsagn dækker vores behov for information og det vil jeg mene de gør
Udsagn 3 er dog ikke information men midlet for at vi kan få denne information.

Jeg ved godt at vi Trondheim snakkede som om vi var med på dette og det er vi også. Vi har pt bare ikke være involveret nok til at vi kan lægge vores navn til det du laver. Og det er ikke din skyld men vores.
Jeg håber at vi nu kan komme med på Praqma's løsning: Tracey, men det får du nok ikke gavn af i din master opgave da den vel skal afleveres her til sommer og det kan vi ikke nå at byde ind til.

Kind regards

Figure A.2: Case A: Reply



Sofus Albertsen <sofusalbertsen@gmail.com> 20. apr. (for 9 dage siden)

til

Hey

I hope you had a safe trip back to .

I have listened through the discussion we had in Tronheim and extracted some statements about traceability that i hope you can confirm is valuable to you.
It's more or less the same things that Tracy will be having as features, I just need you to acknowledge that.
And please just tell me if some of them do not apply to you or is wrongly understood.
It would mean a lot to me if I could name in my thesis, meaning that I take you as a case company who wants these features.
It does not require any additional efforts from your side, other than some facts about your development efforts.
I have in total identified 5 features that we touched uppon:

As a Developer:
1: Ability to visualise the pipeline from a given event.
Example: Who is using this commit / my component
Where is my commit in the pipeline

2: Ability to measure lead time between two events.
Example: From Issue creation to first commit.
From commit to executable creation/product release.

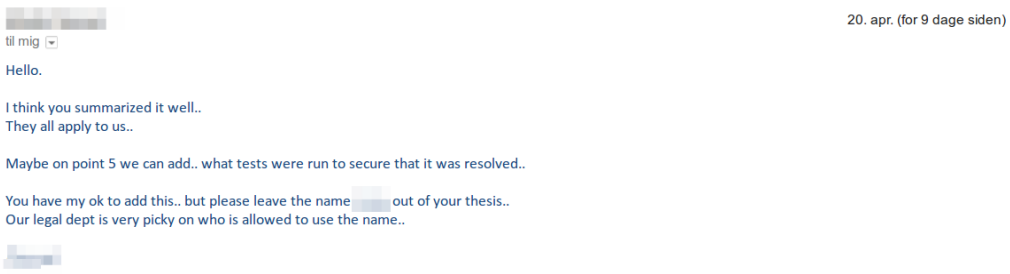
3: Want to have an event mechanism that can drive the continous delivery pipeline and that is tool agnostic.
Example: Every tool should be able to trigger a build in the pipeline.

4: Legal issues: making compliance with ISO 26262
Compare two builds from identical changesets (ISO_26262)

Management:
5: Has this issue been resolved in this release?
Is all commits connected to this issue been incoorporated in this release.

Can you see that all 5 are applicable in your case?
I really hope that this can be possible.
Regards

Figure A.3: Case B: Response



20. apr. (for 9 dage siden)

til mig

Hello.

I think you summarized it well..
They all apply to us..

Maybe on point 5 we can add.. what tests were run to secure that it was resolved..

You have my ok to add this.. but please leave the name out of your thesis..
Our legal dept is very picky on who is allowed to use the name..

Figure A.4: Case B: Reply

Appendix B

Example of tool emitted data

You can find many more examples of both tool emitted data, and their corresponding Entity formats in the Tracy code project under src/-main/resources.

B.1 Jira data

```
{
  "timestamp": 1460367244452,
  "webhookEvent": "jira:issue_updated",
  "user": {
    "self":
       "http://172.17.0.1:8040/rest/api/2/user?username=gitlab",
    "name": "gitlab",
    "key": "gitlab",
    "emailAddress": "gitlab@noname.com",
    "avatarUrls": {
      "48x48":
         "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=mm&s=48",
      "24x24":
         "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=mm&s=24",
      "16x16":
         "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=mm&s=16",
      "32x32":
         "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=mm&s=32"
    },
    "displayName": "gitlab",
```

```
    "active": true,
    "timeZone": "Etc/UTC"
  },
  "issue": {
    "id": "10006",
    "self": "http://172.17.0.1:8040/rest/api/2/issue/10006",
    "key": "PROJ-7",
    "fields": {
      "issuetype": {
        "self":
            "http://172.17.0.1:8040/rest/api/2/issuetype/10001",
        "id": "10001",
        "description": "A task that needs to be done.",
        "iconUrl":
            "http://172.17.0.1:8040/secure/viewavatar?size=xsmall&avatarId=1031
        "name": "Task",
        "subtask": false,
        "avatarId": 10318
      },
      "components": [],
      "timespent": null,
      "timeoriginalestimate": null,
      "description": null,
      "project": {
        "self":
            "http://172.17.0.1:8040/rest/api/2/project/10000",
        "id": "10000",
        "key": "PROJ",
        "name": "ProjectName",
        "avatarUrls": {
          "48x48":
              "http://172.17.0.1:8040/secure/projectavatar?avatarId=10011",
          "24x24":
              "http://172.17.0.1:8040/secure/projectavatar?size=small&avatarId=
          "16x16":
              "http://172.17.0.1:8040/secure/projectavatar?size=xsmall&avatarId
          "32x32":
              "http://172.17.0.1:8040/secure/projectavatar?size=medium&avatarId
        }
      },
      "fixVersions": [],
      "aggregatetimespent": null,
```

```
51        "resolution": null,
52        "timetracking": {},
53        "attachment": [],
54        "aggregatetimeestimate": null,
55        "resolutiondate": null,
56        "workratio": -1,
57        "summary": "Task to be done",
58        "lastViewed": null,
59        "watches": {
60          "self":
              "http://172.17.0.1:8040/rest/api/2/issue/PROJ-7/watchers",
61          "watchCount": 2,
62          "isWatching": true
63        },
64        "creator": {
65          "self":
              "http://172.17.0.1:8040/rest/api/2/user?username=sofusalbertsen",
66          "name": "sofusalbertsen",
67          "key": "sofusalbertsen",
68          "emailAddress": "sofusalbertsen@gmail.com",
69          "avatarUrls": {
70            "48x48":
                "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?d=mm&s=4
71            "24x24":
                "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?d=mm&s=2
72            "16x16":
                "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?d=mm&s=1
73            "32x32":
                "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?d=mm&s=3
74          },
75          "displayName": "sofusalbertsen@gmail.com",
76          "active": true,
77          "timeZone": "Europe/Copenhagen"
78        },
79        "subtasks": [],
80        "created": "2016-04-11T09:28:08.864+0000",
81        "reporter": {
82          "self":
              "http://172.17.0.1:8040/rest/api/2/user?username=sofusalbertsen",
83          "name": "sofusalbertsen",
84          "key": "sofusalbertsen",
85          "emailAddress": "sofusalbertsen@gmail.com",
```

```
        "avatarUrls": {
          "48x48":
              "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
          "24x24":
              "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
          "16x16":
              "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
          "32x32":
              "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
        },
        "displayName": "sofusalbertsen@gmail.com",
        "active": true,
        "timeZone": "Europe/Copenhagen"
      },
      "aggregateprogress": {
        "progress": 0,
        "total": 0
      },
      "priority": {
        "self": "http://172.17.0.1:8040/rest/api/2/priority/3",
        "iconUrl":
            "http://172.17.0.1:8040/images/icons/priorities/medium.svg",
        "name": "Medium",
        "id": "3"
      },
      "labels": [],
      "environment": null,
      "timeestimate": null,
      "aggregatetimeoriginalestimate": null,
      "versions": [],
      "duedate": null,
      "progress": {
        "progress": 0,
        "total": 0
      },
      "comment": {
        "startAt": 0,
        "maxResults": 1,
        "total": 1,
        "comments": [
          {
```

```
122             "self":
                   "http://172.17.0.1:8040/rest/api/2/issue/10006/comment/10020",
123             "id": "10020",
124             "author": {
125               "self":
                     "http://172.17.0.1:8040/rest/api/2/user?username=gitlab",
126               "name": "gitlab",
127               "key": "gitlab",
128               "emailAddress": "gitlab@noname.com",
129               "avatarUrls": {
130                 "48x48":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
131                 "24x24":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
132                 "16x16":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
133                 "32x32":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
134               },
135               "displayName": "gitlab",
136               "active": true,
137               "timeZone": "Etc/UTC"
138             },
139             "body": "Issue solved with
                   [a95a9f2e88a0d421a5635e6f3bdcaf50fa80fb10|http://gitlab.example.com/roo
140             "updateAuthor": {
141               "self":
                     "http://172.17.0.1:8040/rest/api/2/user?username=gitlab",
142               "name": "gitlab",
143               "key": "gitlab",
144               "emailAddress": "gitlab@noname.com",
145               "avatarUrls": {
146                 "48x48":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
147                 "24x24":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
148                 "16x16":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
149                 "32x32":
                       "http://www.gravatar.com/avatar/7f9d20a30446ae4e1d1148ec3d652ad4?d=
150               },
151               "displayName": "gitlab",
```

```
            "active": true,
            "timeZone": "Etc/UTC"
          },
          "created": "2016-04-11T09:29:19.496+0000",
          "updated": "2016-04-11T09:29:19.496+0000"
        }
      ]
    },
    "issuelinks": [],
    "votes": {
      "self":
         "http://172.17.0.1:8040/rest/api/2/issue/PROJ-7/votes",
      "votes": 0,
      "hasVoted": false
    },
    "worklog": {
      "startAt": 0,
      "maxResults": 20,
      "total": 0,
      "worklogs": []
    },
    "assignee": {
      "self":
         "http://172.17.0.1:8040/rest/api/2/user?username=sofusalbertsen",
      "name": "sofusalbertsen",
      "key": "sofusalbertsen",
      "emailAddress": "sofusalbertsen@gmail.com",
      "avatarUrls": {
        "48x48":
           "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
        "24x24":
           "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
        "16x16":
           "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
        "32x32":
           "http://www.gravatar.com/avatar/98e92798e4582a868b00a48857856a67?
      },
      "displayName": "sofusalbertsen@gmail.com",
      "active": true,
      "timeZone": "Europe/Copenhagen"
    },
    "updated": "2016-04-11T09:34:04.450+0000",
```

```
188       "status": {
189         "self":
              "http://172.17.0.1:8040/rest/api/2/status/10001",
190         "description": "",
191         "iconUrl":
              "http://172.17.0.1:8040/images/icons/status_generic.gif",
192         "name": "Done",
193         "id": "10001",
194         "statusCategory": {
195           "self":
                "http://172.17.0.1:8040/rest/api/2/statuscategory/3",
196           "id": 3,
197           "key": "done",
198           "colorName": "green",
199           "name": "Done"
200         }
201       }
202     }
203   },
204   "changelog": {
205     "id": "10006",
206     "items": [
207       {
208         "field": "status",
209         "fieldtype": "jira",
210         "from": "3",
211         "fromString": "In Progress",
212         "to": "10001",
213         "toString": "Done"
214       }
215     ]
216   }
217 }
```

Listing B.1: Jira data file

B.2 Gitlab data

```
1 {
2   "object_kind": "push",
3   "before": "a95a9f2e88a0d421a5635e6f3bdcaf50fa80fb10",
4   "after": "590670a5ede2aef3569d84df5f53f41bbb5e441b",
```

```
  "ref": "refs/heads/master",
  "checkout_sha": "590670a5ede2aef3569d84df5f53f41bbb5e441b",
  "message": null,
  "user_id": 2,
  "user_name": "Jenkins",
  "user_email": "jenkins@jenkins.com",
  "project_id": 1,
  "repository": {
    "name": "TracyRepo",
    "url": "git@gitlab.example.com:root/john.git",
    "description": "",
    "homepage": "http://gitlab.example.com/root/john",
    "git_http_url": "http://gitlab.example.com/root/john.git",
    "git_ssh_url": "git@gitlab.example.com:root/john.git",
    "visibility_level": 20
  },
  "commits": [
    {
      "id": "590670a5ede2aef3569d84df5f53f41bbb5e441b",
      "message": "Fixes PROJ-7",
      "timestamp": "2016-04-11T09:34:02+00:00",
      "url":
          "http://gitlab.example.com/root/john/commit/590670a5ede2aef3569d84df5
      "author": {
        "name": "Jenkins",
        "email": "jenkins@jenkins.com"
      },
      "added": [],
      "modified": [
        "README.md"
      ],
      "removed": []
    }
  ],
  "total_commits_count": 1
}
```

Listing B.2: Gitlab data file

B.3 Jenkins data

```
{
```

```
  "actions": [
    {
      "causes": [
        {
          "shortDescription": "Started by user anonymous",
          "userId": null,
          "userName": "anonymous"
        }
      ]
    },
    {
      "buildsByBranchName": {
        "refs/remotes/origin/master": {
          "buildNumber": 201,
          "buildResult": null,
          "marked": {
            "SHA1": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
            "branch": [
              {
                "SHA1":
                   "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
                "name": "refs/remotes/origin/master"
              }
            ]
          },
          "revision": {
            "SHA1": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
            "branch": [
              {
                "SHA1":
                   "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
                "name": "refs/remotes/origin/master"
              }
            ]
          }
        }
      },
      "lastBuiltRevision": {
        "SHA1": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
        "branch": [
          {
            "SHA1": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
```

```
            "name": "refs/remotes/origin/master"
          }
        ]
      },
      "remoteUrls": [
        "http://172.17.0.1/root/john.git"
      ],
      "scmName": ""
    },
    {},
    {}
  ],
  "artifacts": [],
  "building": true,
  "description": null,
  "displayName": "#201",
  "duration": 0,
  "estimatedDuration": 7956,
  "executor": {},
  "fullDisplayName": "Test-job #201",
  "id": "201",
  "keepLog": false,
  "number": 201,
  "queueId": 120,
  "result": null,
  "timestamp": 1458479882954,
  "url": "http://172.17.0.1:8080/job/Test-job/201/",
  "builtOn": "",
  "changeSet": {
    "items": [
      {
        "affectedPaths": [
          "README.md"
        ],
        "commitId": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
        "timestamp": 1458479857000,
        "author": {
          "absoluteUrl":
              "http://172.17.0.1:8080/user/sofusalbertsen",
          "fullName": "sofusalbertsen"
        },
        "comment": "test",
```

```
        "date": "2016-03-20 14:17:37 +0100",
        "id": "fc81df1fe70f7a3272008278d02cfa8aa4e95b90",
        "msg": "test",
        "paths": [
          {
            "editType": "edit",
            "file": "README.md"
          }
        ]
      }
    ],
    "kind": "git"
  },
  "culprits": [
    {
      "absoluteUrl":
          "http://172.17.0.1:8080/user/sofusalbertsen",
      "fullName": "sofusalbertsen"
    }
  ],
  "mavenArtifacts": null,
  "mavenVersionUsed": "3.3.9"
}
```


Listing B.3: Jenkins data file

Appendix C

Code

All the code produced for this thesis is handed in in the seperate zip package on LearnIt, as it is impossible to read code in pdf. If the code for some reason is not avaliable or you are not supervisor or censor, then look at the following three sites: It is divided into three Eclipse projects using Maven:

Tracy Framework This is the lib that is comprised in the REST implementation and in the AMQP2NEO converter. Has a Command line interface built in.

AMQP2NEO converter to get the Json from AMQP into Neo4J.

REST2AMQP Endpoint for REST-services like Gitlab and Jira

Bibliography

[1] Application lifecycle management - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Application_lifecycle_management. (Accessed on 26-02-2016).

[2] Aws simple icons. https://aws.amazon.com/architecture/icons/. (Accessed on 28-04-2016).

[3] Comparison of version control software - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Comparison_of_version_control_software#General_information. (Accessed on 03-04-2016).

[4] Continuous delivery. http://www.martinfowler.com/bliki/ContinuousDelivery.html. (Accessed on 20-04-2016).

[5] Continuous integration. http://www.martinfowler.com/articles/continuousIntegration.html. (Accessed on 19-04-2016).

[6] Db-engines ranking - popularity ranking of graph dbms. http://db-engines.com/en/ranking/graph+dbms. (Accessed on 25-04-2016).

[7] Extreme programming rules. http://www.extremeprogramming.org/rules.html. (Accessed on 20-04-2016).

[8] Feature branching your way to greatness | the agile coach. https://www.atlassian.com/agile/branching. (Accessed on 11-05-2016).

[9] For relational database developers: A sql to cypher guide. http://neo4j.com/developer/guide-sql-to-cypher/. (Accessed on 17-05-2016).

[10] Git - rebasing. https://git-scm.com/book/en/v2/Git-Branching-Rebasing. (Accessed on 05/26/2016).

[11] Github - ericsson/eiffel: The eiffel framework vocabulary, descriptions, guides and schemas along with links to relevant implementation repositories. https://github.com/Ericsson/eiffel. (Accessed on 20-05-2016).

[12] Github - jayway/jsonpath: Java jsonpath implementation. https://github.com/jayway/JsonPath. (Accessed on 04-04-2016).

[13] The gnu general public license v3.0 - gnu project - free software foundation. http://www.gnu.org/licenses/gpl-3.0.en.html. (Accessed on 25-04-2016).

[14] The groovy programming language. http://www.groovy-lang.org/. (Accessed on 25-04-2016).

[15] Iso 26262-1:2011(en), road vehicles - functional safety - part 1: Vocabulary. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en.

[16] Iso 8601:2004 - data elements and interchange formats – information interchange – representation of dates and times. http://www.iso.org/iso/catalogue_detail?csnumber=40874. (Accessed on 26-04-2016).

[17] javax.xml.xpath (java platform se 7). https://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/package-summary.html. (Accessed on 04-04-2016).

[18] Jira software - issue & project tracking for software teams | atlassian. https://www.atlassian.com/software/jira. (Accessed on 28-04-2016).

[19] Josra. http://www.josra.org/gatherings/4th-gathering.html. (Accessed on 02-05-2016).

[20] Josra. http://www.josra.org/blog/An-automated-git-branching-strategy.html. (Accessed on 12-05-2016).

[21] Rabbitmq - messaging that just works. https://www.rabbitmq.com/. (Accessed on 21-04-2016).

[22] Solvency ii (including "omnibus ii") - european commission. `http://ec.europa.eu/finance/insurance/solvency/solvency2/index_en.htm`. (Accessed on 25-02-2016).

[23] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[24] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

[25] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 95–104. ACM, 2010.

[26] Hazeline U Asuncion, Frédéric François, and Richard N Taylor. An end-to-end industrial software traceability tool. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 115–124. ACM, 2007.

[27] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[28] Andrzej Wasowski. Thorsten Berger. Introduction to model-driven software engineering with domain-specific languages. Unpublished manuscript., 2016.

[29] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, (6):27–35, 2007.

[30] Jane Cleland-Huang, Carl K Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *Software Engineering, IEEE Transactions on*, 29(9):796–810, 2003.

[31] Jan Bosch Daniel Ståhl, Kristofer Hallén. Continuous integration and delivery traceability in industry: Needs and practices. Unpublished paper, accepted to SEAA 16., 2016.

[32] David Dominguez-Sal, P Urbón-Bayes, Aleix Giménez-Vanó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, and Josep-Lluis Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Web-Age Information Management*, pages 37–48. Springer, 2010.

[33] U.S. Food and Drug Administration. General principles of software validation; final guidance for industry and fda staff. `http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm`, 1 2012. (Accessed on 25-02-2016).

[34] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[35] Debasish Ghosh. *DSLs in Action*. Manning Publications, 1 edition, 12 2010.

[36] Orlena Gotel and Anthony Finkelstein. Contribution structures (requirements artifacts). In *Second IEEE International Symposium on Requirements Engineering, March 27 - 29, 1995, York, England*, pages 100–107, 1995.

[37] Orlena CZ Gotel and Anthony CW Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE, 1994.

[38] Jane Huffman Hayes and Alex Dekhtyar. Humans in the traceability loop: can't live with'em, can't live without'em. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 20–23. ACM, 2005.

[39] James D Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 29(6):481–494, 2003.

[40] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[41] Lars Klimpke and Tobias Hildenbrand. Towards end-to-end traceability: Insights and implications from five case studies. *Software Engineering Advances, International Conference on*, 0:465–470, 2009.

[42] Steinar Kvale and Svend Brinkmann. Interview–introduktion til et håndværk, 2. *Hans Reitzler Forlag: København*, 2009.

[43] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Softw. Pract. Exper.*, 26(10):1161–1180, October 1996.

[44] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Motivation matters in the traceability trenches. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 143–148. IEEE, 2009.

[45] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[46] Jay F Nunamaker Jr, Minder Chen, and Titus DM Purdin. Systems development in information systems research. *Journal of management information systems*, 7(3):89–106, 1990.

[47] Richard F. Paige, Jonathan S. Ostroff, and Phillip J Brooke. Principles for modeling language design. *Information and Software Technology*, 42(10):665–675, 2000.

[48] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, Jan 2001.

[49] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.

[50] Patrick Rempel, Patrick Mader, and Tobias Kuschke. An empirical study on project-specific traceability strategies. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 195–204. IEEE, 2013.

[51] Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and System Modeling*, 9(4):473–492, 2010.

[52] George Spanoudakis and Andrea Zisman. Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, 3:395–428, 2005.

[53] Brian A White. *Software configuration management strategies and Rational ClearCase: a practical introduction*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[54] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Softw. Syst. Model.*, 9(4):529–565, September 2010.