Computer Science

TECHNISCHE UNIVERSITÄT
ILMENAU

Masterthesis

Analysis of Software-Engineering-Processes

von

Clemens Teichmann

- *Matriculation number.: 49816* -

- *IN-Master* -

German Aerospace Center

Location: Berlin

Simulation and Software Technology
Department: Distributed Systems and Component Software

Prof. Dr.-Ing. Detlef Streitferdt (JP)

State of Affirmation

I hereby assure that this thesis was exclusivly made by myself and that I have used no other sources and aids other than those cited

Berlin, October 30, 2013

Contents

List of Figures

Listings

List of Abbreviations

AUP	**A**gile **U**nified **P**rocess
CI	**C**ontinuous **I**ntegration
DBMS	**D**ata**b**ase **M**anagement **S**ystem
DLR	German Aerospace Center
IDE	**I**ntegrated **D**evelopment **E**nvironment
OPM	**O**pen **P**rovenance **M**odel
PrIMe	**Pr**ovenance **I**ncorporating **Me**thodology
RUP	**R**ational **U**nified **P**rocess
SC	Institute Simulation and Software Technology
SPEM	**S**oftware & Systems **P**rocess **E**ngineering **M**etamodel
SVN	**S**ub**v**ersio**n**
VCS	**V**ersion **C**ontrol **S**ystem
VSS	Department Distributed Systems and Component Software

1 Introduction

This chapter begins with a short description of the working environment and continues with the motivation for this thesis and its distinctions from other fields of research. Finally, the chosen approach is described and the structure of the thesis laid out.

1.1 Working Environment

This thesis was written at the German Aerospace Center in the department for Distributed Systems and Component Software of the Institute of Simulation and Software Technology.

1.1.1 German Aerospace Center

The German Aerospace Center (DLR) is the research center of the federal republic of Germany for aeronautics, space, energy, transport and security. In this extensive fields it has many national and international cooperative ventures and beside its own research, it is responsible for the German space program as Germany's space agency. It currently occupies approximately 7400 employees at 16 locations in Germany and three offices abroad: Cologne (headquarters), Augsburg, Berlin, Bonn, Braunschweig, Bremen, Goettingen, Hamburg, Juelich, Lampoldshausen, Neustrelitz, Oberpfaffenhofen, Stade, Stuttgart, Trauen, Weilheim, Brussels, Paris, Tokyo and Washington D.C.

The mission of the DLR is the exploration of the earth and the solar system, the development of environment-friendly technologies for energy supply, mobility, communication and security. To achieve that, it follows different guidelines like "One DLR", excellence in science and professionalism, attention to precision and reliability and many more. [DLR]

1.1.2 Institute Simulation and Software Technology

The Institute Simulation and Software Technology (SC) has the mission of research and development in software engineering technologies. Fields of research include component-based software for distributed systems, software technologies for embedded systems and software quality assurance. It takes part in demanding software development projects at the DLR as well as with external partners and develops and incorporates state-of-the-art software technologies. Bureaus are located at Cologne, Braunschweig and Berlin and it is divided into the departments "Distributed Systems and Component Software" and "Software for Space Systems and Interactive Visualization". Those departments work on unified multi-domain simulation libraries, grid computing, modeling and simulation, software test and many more. [SC]

1.1.3 Department Distributed Systems and Component Software

The mission of the Department Distributed Systems and Component Software (VSS) is to provide software engineering expertise at the DLR as well as development and research in software technologies. Its fields are separated into software engineering including methods, tools, test and architecture, high performance computing like parallel numerical algorithms and distributed software systems including software integration technology, integration frameworks and data or knowledge management. It also takes a big part in the consulting and development for the entire DLR. [VSS]

1.2 Motivation

The complexity of software and its development had rapidly increased in the last years from sequential to iterative software development processes. From a simple Waterfall model industry and research advanced to more complex processes like the **R**ational **U**nified **P**rocess (RUP) [Kru04] which include more tasks and developers to generate high quality software. That increases the complexity not only of the process but also of the part of it a single developer has to understand and documentation becomes even more important. In addition to the increased team sizes the number of particular roles within a team also increased. Instead of software developers there are process managers, designers, developers and dedicated testers, e.g.

Besides the process structure, software engineering methods and tools are employed to automate the development steps. A version control system holds a complete history of the source code, allowing the development team to distribute implementation tasks while working on the same code basis. An issue tracking system helps to link requirements and implementation as well as a bug report system later in maintenance. A continuous integration framework runs software tests before building the software regularly and a wiki serves as documentation platform, for the code and the process itself. This increased number of tools leads to the distribution of process information on the different tools, making the control of the process even harder.

High software quality demands a well defined process as well as an assurance that the process is followed correctly by all its participants. For this purpose, a provenance system has been developed in a previous master thesis [Wen10]. This system creates cross links between information of all different tools used in the development process and visualizes the origin of data. With this provenance data a centralized, complete view is available which should help to design a comprehensible and reproducible software development process. Since research to provenance is pretty new, there are still many unanswered questions like how provenance data reflects the development process or process-specific questions.

The subject of this master thesis is the analysis of the existing provenance model. A theoretical view on the software development process as a provenance graph is explained as well as the deviation from the old provenance model used in [Wen10]. Since the automatic insertion of data relies on all tools capabilities, a closer look on the information retrieval from different tools is given. Furthermore, an analysis of the amount of provenance data generated is given and some flaws or missing object as well as possible extension points in the existing model exposed. To compare performance of different query languages, existing and new provenance questions are newly implemented in Cypher. At last, a short overview on privacy of data regarding provenance is presented.

1.3 Distinction

The research of data provenance touches other fields in software engineering research, such as repository mining, (requirements) traceability and rationale management. While those technologies have a similar purpose, their scope or approach differ from the ones of data provenance. To make a distinction between those fields, this chapter gives a short description of these three software engineering disciplines and explains the differences.

1.3.1 Repository Mining

Repository Mining is the empirical and systematic investigation of software repositories as described by Kagdi et al [KCM07]. The purpose is to identify uncovered information, relationships or trends in the observed repository, such as finding reused or cloned code or connect events that happen often in a timely manner. Investigated artifacts are the differences in the repository like addition, deletion or modification of code and meta-data containing user IDs, timestamps and comments. Its main scope are version control systems like **S**ub**v**ersio**n** (SVN) or Git, but it also includes the connection to bugtrackers.

While repository mining gives an in-depth analysis of the source code and its meta-data, the scope is limited on the repository and gives a very code-centric view. Aside from the relationships to a bugtracker, it just focuses on one part of the software development and not on the process as a whole. Mining a repository also includes the process of getting data by code analysis or metrics while provenance information are all available and a user just needs to query for the correct data.

1.3.2 Traceability

Traceability in the software development process is generally the link between two or more artifacts, often between requirements and parts of the source code or tests. In [Lev10] requirements traceability is described as "the ability to describe and trace the life of a requirement, in both a forward and backward direction". So traceability looks at a requirement from its creation to implementation. With this information questions such as "what source code files contribute to what requirement" or "how good every requirement is covered by tests" can be answered. It will also give information about when and why requirements changed.

Similar to repository mining, requirement traceability has a very small scope compared to provenance information since it sets its focus mainly on the requirements and their links to other artifacts and not the complete process. Additionally, its technical implementation is very difficult since many approaches use a traceability matrix which must be maintained manually and ca not be automated very well.

1.3.3 Rationale Management

Another research field similar to provenance is rationale management. The purpose of rationale management is "capturing, representing, and maintaining records about why developers have made the decisions they have." [DP06] It should give a complete model of design decisions throughout the whole development and capture the discourse and knowledge. To determine this process of decision making, four components are saved: the problem, the available options for the problem, additional criteria like non-functional requirements and the debate that lead to the decision. The decision making process is recorded with this information and understandable for every participant in the development, helping to understand why software is implemented the way it is and to simplify changes.

Rationale management focuses mainly on the question why developers choose a particular design while provenance concentrates on what happens in the process and when, making links between artifacts. So again, the scope of this approach is smaller and mostly includes requirements and design. In the future it will be a task in provenance to record design decisions but there is currently no concept to do so automatically which makes implementation of rationale management difficult.

1.3.4 Conclusion

There are many other research fields that have similar purposes as provenance, but not one considers the whole development process. They either focus on requirements (traceability), design decisions (rationale management) or source code (repository mining), but none consider an integrated view of the entire process. Besides, those technologies do not record their information in a structured manner like provenance which makes automation of the analysis difficult and therefore is a very important feature of provenance. Some parts of the development process like documentation and building the software are completely left out by those approaches.

1.4 Approach

This master thesis analyzes the current provenance questions and infrastructure based on the work of [Wen10] at the DLR. Existing questions regarding provenance information are recapped and new queries are developed to verify the software development process. The new provenance model of the software development process is presented but because actual provenance data does not exist to this date, all analysis is based on theoretical assumptions and an example process. Afterwards, the model is examined in regards of data volume and a discussion about flaws in the system is given with possible solutions. Finally, all questions are implemented in the graph query language Cypher.

1.5 Structure

Following this introduction, this master thesis is structured in seven additional chapters:

Chapter 2 The second chapter explains the background of this thesis. First it describes the development of software, a brief history and which process is incorporated at the department SC. Then it provides a definition for data provenance, its concept and the provenance model PROV-DM. Finally, it gives an overview of the provenance infrastructure using the **D**ata**b**ase **M**anagement **S**ystem (DBMS) Neo4J, its query language Cypher and the recording interface PyProv.

Chapter 3 Every project starts with the identification of its requirements. For this thesis, the requirements are a number of questions regarding the software development process and the provenance information which can be retrieved from all incorporated tools. This chapter recaps the questions from previous work, adds additional questions for the process and presents the data which is provided by the tools.

Chapter 4 Because the previous work on a provenance model for the development process was outdated and a new modeling standard was introduced, the provenance model was renewed using the PROV data model. With the knowledge of the two aforementioned chapters, this one presents the updated model for the software development process which is the source for the analysis later in this thesis.

Chapter 5 The fifth chapter combines the knowledge about the software development process at SC with the provenance model of it. Using a process with example data, one iteration of the software life-cycle is presented as a provenance graph. It displays how a graph builds up over time and how the data is connected and stored while the iteration is running. Additionally, an introduction into the **S**oftware & Systems **P**rocess **E**ngineering **M**etamodel (SPEM) is given which can be used to model the process formally.

Chapter 6 After the model and the process graph are analyzed, this chapters breaks down the provenance model further. First, a compilation of generated nodes and relationships in the graph database is presented. Afterwards, a discussion about flaws in the current model and infrastructure is performed. It exposes problems in the recording of provenance information and gives convenient solutions.

Chapter 7 The implementation of the questions to verify the development process is given in this chapter. All queries are written in Cypher and divided into content and process queries. Query results can not be presented in this thesis, due to the fact that real life provenance data was not available at the time of this writing.

Chapter 8 Finally, the complete thesis is reviewed in the last chapter. The work is summarized and an outlook on existing problems and future development is provided.

2 Background

This chapter describes the background of this master thesis and everything the reader should know in advance. It covers the general and specific software development process, the concept of provenance and the PROV data model, the incorporated graph database Neo4J and the provenance recording interface PyProv. The questions about software development and the analysis of the process are all based upon the knowledge presented in this chapter.

2.1 Software Development Process

This section provides a general description of software and the development process as well as a brief history given in [Raj11]. Following that, the development process used at SC will be explained and a typical iteration with different tools is illustrated.

2.1.1 General

All software today share some collective properties. These properties include complexity, invisibility, changeability, discontinuity and conformity. Complexity of software describes its size, e.g. number of components, classes or modules etc. and how clear its structure and function is for the user and developer. Because one can not see, feel or touch software, it is invisible and therefore hard for untrained people to imagine how it works. Changeability is one of the most important properties of software today. Because software is basically source code and this code can be easily changed with a text editor, changes can happen very fast and immediate. However, it is not that easy to make changes in bigger projects and ensure the correct function of software. This goes hand in hand with discontinuity which means that small changes can have a huge impact on its functionality. The last property is conformity, meaning that software always has some kind of domain for which it is developed and works for example with specific naming conventions.

The development of software started in the 1950s and experienced three main paradigms shifts. In the beginning of software development, there were no software engineers. Programmers often come from different disciplines and wrote little applications. Most development in that time was ad-hoc, i.e. no processes was followed and tool usage was low. As soon as programs became more complex but not well documented, the knowledge about the source code resided just by the programmer and was not available to the company or third persons. Besides, the requirements by the customers grew, so there was a need for some kind of development schema.

To cope with these new requirements, the first process called “Waterfall model” was created. It is a planned approach for software design and implementation and includes four to seven phases, depending on different authors and granularity. The original seven phases [Roy70] are:

- Requirements specification
- Design
- Construction
- Integration
- Testing and Debugging
- Installation
- Maintenance

The idea was to follow each step one at a time and to document all results from every phase. If the team finished one phase, they would go to the next, but it was not taken into account that one might have to return to a previous phase. So once the requirements were collected and the architecture designed, the model would not allow to make a change on the result of these steps. This entails that these processes were not flexible enough to suite modern demands since requirements change more often during development.

To take volatility of requirements and changeability of software into account and handle those problems, new processes arouse marking a new paradigm shift. These new models were called incremental or iterative and focused on a closer engagement of the customer into the development process and short development phases called iterations. Today we have different life span/cycle models consisting of phases like an initial development, evolution (software changes and new versions), servicing, phaseout and closedown.

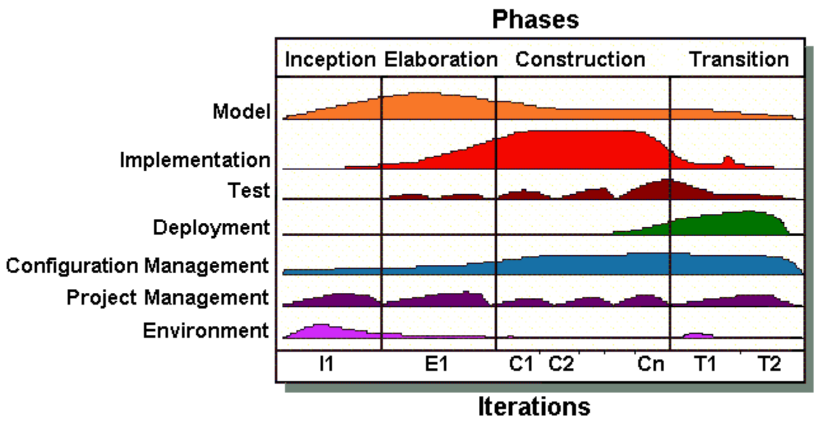

Figure 1: Phases of AUP [Amb12]

2.1.2 Software Development at SC

The default software development process at the department VSS is the **A**gile **U**nified **P**rocess (AUP) (see figure 1). It is based on RUP but simplifies it at some aspects to make it more agile. The key principle of AUP is "serial in the large, iterative in the small". It serves as a basic process and needs to be tailored for every project. While developing in a process using AUP, the focus is on the following values

- Evolvability,
- Correctness in requirements, unit and integration tests,
- Production efficiency through automation and
- Reflection. [Sch13c]

It also defines five different roles, consisting of Customer, Project Lead, Developer and Software Responsible. Deviant from the definition of the AUP, there are six phases [Sch13c] to go through, illustrated in figure 2:

Inception
: set the project vision, understand the requirements and tailor the process while keeping the customer heavily involved

Elaboration
: plan initial iterations, stabilize the project vision and defines processes and tools

Construction
: continuous planning and "just in time" modeling, Test-Driven Development and CI as well as continuous feedback to and from the customer

Transition
: deliver the software into production, do a final user acceptance test and perform a retro-perspective on the project so far

Maintenance
: similar to the **Construction** phase, plan releases, maintain the user group and some marketing activities

Retirement
: a controlled retirement of the software and a final retro-perspective

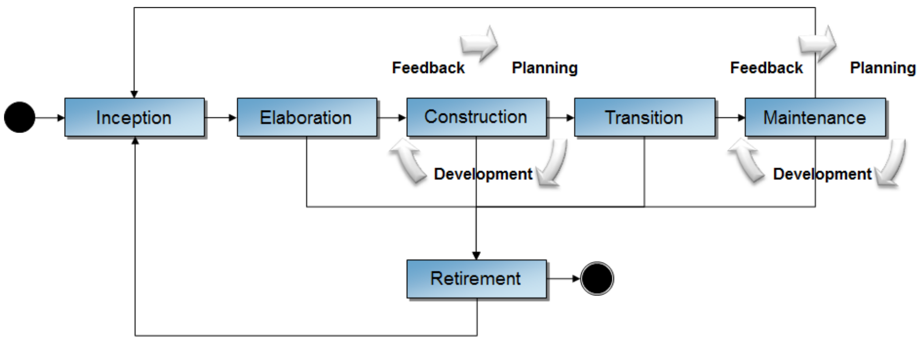

Figure 2: Software Life-Cycle at SC [Sch13a]

Additionally, the phases Construction and Maintenance include iterations which are basically cycles of planning, development and feedback. The Construction phase divides furthermore into a kick-off meeting, the realization and stabilization parts of the code and a review of that iteration. Around four to six weeks length is stated as a best practice time for an iteration, but with the remark "the shorter, the better". The disciplines executed by each participant of such a process include modeling, implementation, testing, deployment, configuration management, project management and environment.

Beside those management specifications, there is also a list of tools given which have proven themselves in software development. For version control, the widely known SVN is used. But in addition, the software deployment tool RepoGuard is employed. RepoGuard takes advantages of SVNs hook scripts to execute automated checks on commit time, e.g. if the code follows the specified coding standards or if the commit message includes an issue ID. As an issue tracker, Mantis is recommended and Jenkins is used as a continuous integration tool. The wiki engine MoinMoin serves as a tool to document the software and the process. On developer side, there is an **I**ntegrated **D**evelopment **E**nvironment (IDE), typically Eclipse, and a web browser accessing ViewVC to view the source code. [Sch13b]

How a typical iteration uses those tools can be seen in figure 3. At the beginning, the release and iteration planning are performed using Mantis. All issues for the iteration are generated and assigned to the release. After that, the issues are assigned to the developers and the design phase starts. Before the next step, every developer should update his repository to the latest revision. Then a cycle of coding starts and unit testing, running the build process locally first and integration testing by committing the code to SVN using the assigned Mantis ID. When a developer finishes this cycle, i.e. by having implemented and tested his code for his issues, the issue will be resolved. Working items for this step are a Mantis update and the documentation for those issues and a passed Jenkins build. At the end of each iteration the release process is performed and the issue is closed. [Sch13a]

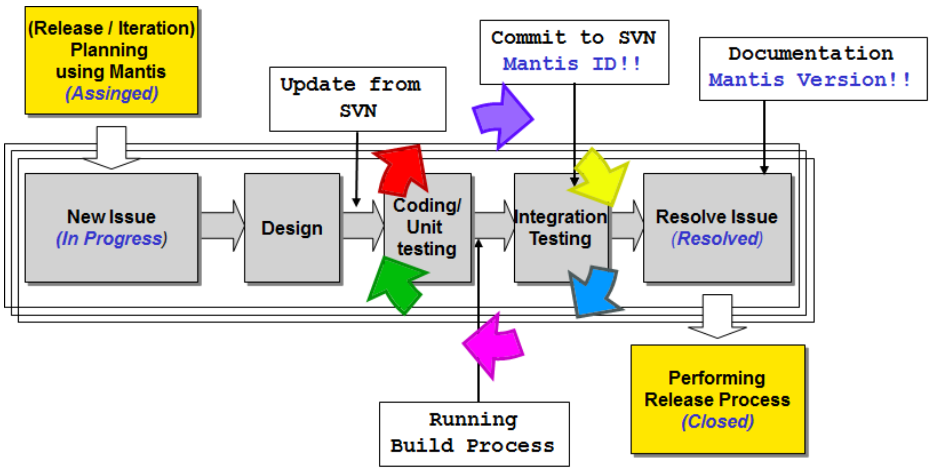
(Release / Iteration) Planning using Mantis (Assinged)
Update from SVN
Commit to SVN Mantis ID!!
Documentation Mantis Version!!
New Issue (In Progress)
Design
Coding/ Unit testing
Integration Testing
Resolve Issue (Resolved)
Running Build Process
Performing Release Process (Closed)

Figure 3: Iteration plan [Sch13a]

2.2 Provenance

In this chapter, a definition and brief history of provenance is given. Afterwards, the purpose and approach of data provenance is explained and the PROV data model is illustrated with examples.

2.2.1 Definition

The word *Provenance* comes from the French "provenir" and means "to come from". Provenance is a well defined term in Arts describing the origin of a painting from its creation to all different locations and collectors. In computer science, data provenance, sometimes also called data lineage, represents the origin of data, from its generation to all its usage and deviation [MGM+08]. Groth et. al. give the following definition for provenance.

Definition 1. *The provenance of a piece of data is the process that led to that piece of data. [GJM+06]*

The research on provenance began with the EU Provenance project in 2004. This was a three year project with many participants from European universities and research facilities to give a definition of data provenance and to establish a common context for a provenance system. Its results are the **O**pen **P**rovenance **M**odel (OPM) [MCF+11], an architectural design and the **Pr**ovenance **I**ncorporating **Me**thodology (PrIMe) [MMG+06]. With OPM a first model for recording provenance in a graph model was established and later refined. The architectural design and PrIMe are still the standard for their fields with only minor adjustments. Since 2011, almost the same participants from the European project take part in the W3C Provenance Working Group named PROV [PRO]. The goals of this project are the revision of the old model and generating example provenance systems for different applications. So far, this project introduced a new model for provenance data called PROV-DM.

2.2.2 Concept

As stated before, the main goal of provenance is to record the complete process of the generation, usage and deviation of data. PROV serves as a specification to record, store and query provenance data as can be seen in figure 4. Its purpose is to illustrate how data was collected, determine ownership and rights over an object, give judgment whether or not to trust data, verifying a process and steps and reproducing results. Provenance data is meta data from different systems connected in one model. [Sch13d] This model can be viewed from three different perspectives: an agent-centered, object-centered or process-centered view. [BDG+12] The agent-centered view focuses mainly on all participants of a process, how they are involved at certain activities or entities and what role they have. An object-centered view looks at the entities of a process, how they were generated and used as well as the deviation from other entities. The scope of a process-centered view is the performed activities, how they interact with each other and pass different entities. As this thesis focuses on the software development process, it represents a process-centered view on the provenance data. However, different views can be applied on the same set of provenance data in regard to what is most interesting for the viewer.

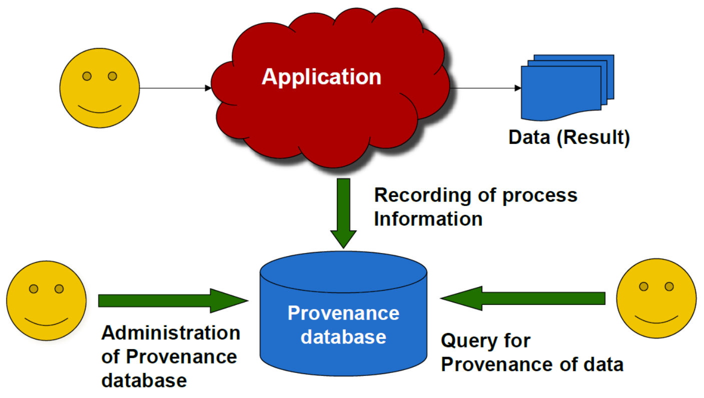
Application
Data (Result)
Recording of process Information
Administration of Provenance database
Provenance database
Query for Provenance of data

Figure 4: Concept of Provenance [Sch13d]

To reach its goals, a provenance system has to analyze the process, its subprocesses, all involved actors and the handled data. With the information from different tools a directed graph is build with nodes for activities, actors and data and edges for relationship between those. This graph is always directed from the present to the past, showing what happened in the process.

Provenance has many different use cases in research and industry. For example, it can be used in an organ transplantation management system to create transparency over the donor or organ assignment. Entities would be the donated organ, agents the donor, the receptor and all the hospital staff involved in the process and activities would be the acceptance of a receptor or the surgery. As for this thesis, it can also be used to monitor development processes or reproduce simulation data. Generally speaking, provenance can be applied to either cost intensive or security heavy processes (e.g. satellite programs) or to assure responsibilities.

2.2.3 PROV Data Model

This section gives a general overview of the provenance data model to establish a common knowledge basis for the software development model in section 4. Provenance data is saved in a property graph model with nodes and edges. Its core structures are illustrated in figure 5.

Nodes are artifacts like entities, activities and agents. An entity symbolizes a piece of data like an image, a website article or a set of simulation results, anything that can be generated or used in the process. An activity is a process which uses or generates data and is somehow attributed to an actor. This may be writing a news article or altering a photo

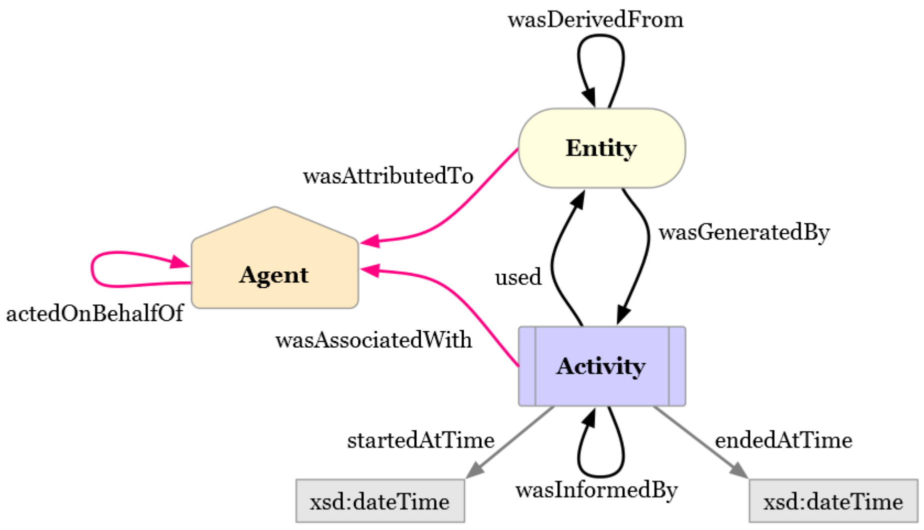

Figure 5: PROV core structures [BDG+12]

done by a person or a step in an automated manufacturing process started by a controlling computer. Agents are objects that interact with entities through activities and are therefore associated with those entities. An agent can be a living person, a computer program or any object that can start or end activities. Also illustrated in figure 5 are the relationships between provenance objects to themselves. As mentioned before, an entity may be derived from another entity in the process. This happens when an activity uses a set of entities and generates new entities, then those generated entities are automatically derived from all involved precedent entities. If information is passed from one activity to another using an entity, one activity informed the other. Activities can also be delegated between different agents, meaning that one agent acted on behalf of another, e.g. a developer acted on behalf of his chef who acts on behalf of the firm. They are all in some way responsible for the activity but there is no concrete information what responsibilities and delegation were given. To complete this core structure, some additional provenance objects are integrated, like collections of entities or a so called bundle, which is the provenance of provenance data. [BBC+12]

As stated in the beginning of this section, a provenance graph is a property graph. This means that all nodes and edges can have additional parameters added to them, symbolizing properties of the object or the link. A property of a node may be the name of a user (agent) or a storage location of an item (entity), while properties on edges determine in what role a entity might be used or which responsibilities where passed from on agent to another. This helps to define more specific information about the process. Probably the most important use of properties are time declarations. A time stamp refines the chronological events of the process by adding concrete time information to nodes and edges. With this property, start and end time of activities or generation time of entities can be specified.

Some capabilities of a provenance system are especially interesting for this master thesis:

- **Representing diverse entities involved**: Any work item of a process can be modeled into the provenance graph, expanding the expressiveness of the model. Furthermore different users can obtain different information from the graph by diverse views. A project manager might consider a process-centered view on the model by looking on chronological dependencies of all performed steps while a developer has an object-centered view on his source code and how it was affected by different issues, other code or other developers.
- **Stating partial or incomplete provenance**: A provenance graph might be incomplete, not by missing artifacts of the process but by omitting information. When it comes to interaction between institutes and exchanging provenance information, one party might only hand out partial provenance information, just enough to satisfy the other parties' needs. The expressiveness of the model might suffer from this, but the model is still correct.
- **Describing the commonalities in the derivation of two entities**: Especially when it comes to code and issues, the derivation of those two are very important to draw conclusion on why the code changed and to what requirement it is related. In the provenance graph it is either possible to make an implicit declaration with the usage and generation of entities by an activity or an explicit declaration using a derivation relationship and give further information to that process.
- **Relating versions of objects over time**: This is a subset of the aforementioned capability. While the model can illustrate the derivation between two entities, it can also specify how one entity changed in the process. Contrary to the derivation, the relation here is between the same artifact although it would be displayed as two entities in the model. So the version change of an updated wiki page or a fixed code file can be tracked and shows the development of one artifact over time.
- **Supporting queries over provenance of different granularity**: Like the different views on a provenance model, there will also be a need for different granularities of queries. One user might want to see a detailed view of a source code file with all modifying activities and working agents, so he needs more in-depth information. Another user justs needs overall data like accumulation of data (for example how much commits user X performed in a given time frame) or that there is documentation for every issue, not interested in the actual content of the documentation. [BDG+12]

2.3 Graph Database

A graph database is a database which stores its information in a graph structure. Those DBMS' are part of the so called NoSQL concept, including different information storage and retrieval system beside the relational approach. NoSQL databases also include key-value, column and document databases which all have different purposes and goals, from handling big data to not following a fixed data model. In distinction to traditional relational DBMS, NoSQL databases often come with no pre-defined scheme.

The purpose of graph databases is to store highly connected data, so basically objects with many relationships between them. In many real life structures like social networks, the relationships between objects are as important as the objects themselves and therefore contain a lot of information, which only reveals if multiple objects are connected. Graph databases also allow dynamic working and an evolvable data model because most graph DBMS do not require a pre-defined scheme, increasing complexity and changeability of graphs. The current realizations of graphs in databases are a triplestore using the Resource Description Framework, which stores its data in a subject-predicate-object format, or a real graph database like Neo4J.

2.3.1 Neo4J

Neo4J [Neo12] is todays most popular graph database. [DBE] Its initial release was in 2007 by Neo Technology. This cross-platform DBMS is written in Java and implements a graph structure with nodes, relationships and properties on the aforementioned, leading to a so called property graph (see figure 6). Neo4J allows its user to store their data in a directed graph and set typed relations between objects. This relation type is like a special property, it describes the relationship between two objects further like a “knows” or “has” connection. With version 2.0 nodes have labels, which mark their object type to different granularity. This allows the developer to group nodes to sets with the same label. The node type does not have to be a property or relationship anymore but can easily identified via its labels. Neo4J also supports full ACID transactions and is scalable up to several billion nodes and relationships as well as billions of properties and up to 32 thousand different relationship types.

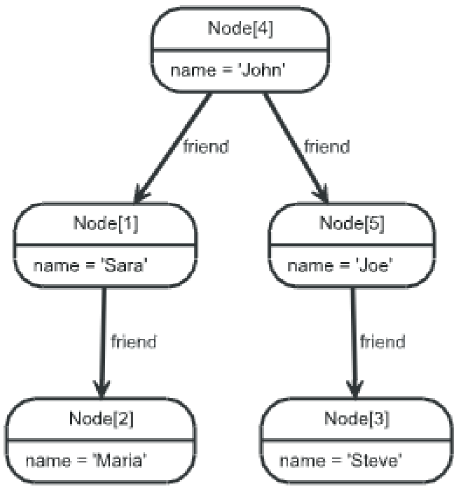

Figure 6: Example Graph [Neo13]

2.3.2 Cypher

Cypher is the query language of Neo4J, developed by the same team. [Neo13] It is a declarative graph query language that allows the user to express what information he wants. Therefore, it is an alternative to imperative languages like Gremlin, which focuses on how to retrieve data. Cypher allows the user to build traversals and paths in a graph but can also query and update without traversals using indexes on node and relationship properties. The language itself is inspired by SQL and uses similar keywords. The code example 1 shows a query on the graph in figure 6. In the first line, the auto indexing function of Neo4J is used to retrieve all nodes with the property name and its value “John”. Afterwards, all outgoing “friend”-relationships of those nodes are matched twice, resulting in a friend of a friend relationship. So this query would return node four (John), two (Maria) and three (Steve).

```
START john=node:node_auto_index(name = 'John')
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

Listing 1: Cypher example

2.4 PyProv

PyProv is the working title of a provenance recording interface. It is an open source program and currently in prototype state. [Tei13] PyProv is a redevelopment of prOOst (**pro**venance with **O**PM **st**ore) [Ney11] which is also an interface to save provenance information implemented in Java. Since prOOst has not been maintained after its development has finished, the program is out-dated. The provenance model is based upon the OPM and the libraries, it depends on, have changed making it difficult to use. So instead of rewriting the whole program to use new versions of the libraries, a new interface was written in Python, implementing the new data model PROV and using up-to-date software libraries.

PyProv serves as an interface between the tools in the software development process and a database to record provenance information. It implements a REST API to retrieve provenance data from the tools, which use their internal event system to send those information, and connects to a Neo4J database as back-end for storage. In addition, PyProv provides an interface to transmit queries and their results between a user and the provenance store. Whether recording or querying, all data must come from a POST request. To receive provenance information, the content of the request is formatted in JSON and the content-type should be set to `application/json`.

So the user is not meant to communicate with the interface to store provenance information, this task is up to the employed tools. The user works in the IDE and commits new source code, which triggers the event system of the **V**ersion **C**ontrol **S**ystem (VCS) and the bugtracker. In addition, all users can manually access the bugtracker, the CI system and the wiki via their web interfaces. All those tools send their provenance information to PyProv via a

HTTP request, which passes the data to the graph database. The Neo4J database also comes with a web interface but this should only be used for maintenance work.

The next paragraph explains some in-depth details about the design of PyProv and the libraries used. As mentioned, PyProv is written in Python and provides a web front end and a graph database as back end. For the back end connection PyProv uses the library py2neo. py2neo allows the developer to access Neo4J via a RESTful web interface and work with the graph database inside the code. Code 2 gives a little example how to establish a connection to the root direction of a local database and create two nodes and one relationship between the two of them. py2neo also allows the developer to generate a node or relationship object and add properties before they are written to the database. To enhance performance, one may subsume various write jobs to a so called batch and send data bundled in one transaction instead of many transaction for every job.

```python
from py2neo import neo4j

graph_db = neo4j.GraphDatabaseService("http://localhost:7474/db/data/")
a, b, ab = graph_db.create(node(name="Alice"), node(name="Bob"), rel(0, "
    KNOWS", 1))
```

Listing 2: Connection to local graph database using py2neo

The web front end was programmed by means of the microframework Flask. Flask is based on Werkzeug and Jinja 2. Its goals is to be small and simple and not making the developer any prerequisites or constraints. Therefore it has very little core functionalities to set up a web server fast as can be seen in the code example 3 and is expandable through extensions.

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

Listing 3: A minimal Flask application

Using Flask, the REST API of PyProv is set up and allows all tools to send their provenance data. In the following, every API implemented to use PyProv at the department SC is described with their corresponding input data.

The build API enables Jenkins or other CI tools to add their build information to the database. Every build is identified by the used revision, the result of the build and the maven version executing the build.

After all tests ran, the coverage API takes the coverage result from the CI tool. For the coverage information, the revision number and the percentage of covered code is needed.

http://host:port/prov/build		
revision	integer	the revision number the build based on.
result	integer	the build result. Zero indicates success, all other failure.
maven	string	the version of the executing Maven.

Table 1: Build REST API

http://host:port/prov/coverage		
revision	integer	the revision number the coverage report based on.
coverage	float	the percentage of covered code.

Table 2: Coverage REST API

To add documentation to the database, the documentation API is used. It takes the user name and the changed page as a required input and optionally the change message and the corresponding issue.

http://host:port/prov/documentation		
user	string	the user name in the format “ForenameSurname”.
page	string	the name of the changed page.
message	string (optional)	an optional message for the edit by the user.
issue	integer (optional)	the issue belonging to the change.

Table 3: Documentation REST API

The provenance information of a software release can be added via the release API. It takes the identifier of the release, e.g. the version number, the revision the release is based on and the resulting file. If multiple files are generated, the function must be called for every file.

http://host:port/prov/release		
id	integer	the identifier of the release (version number).
revision	integer	the revision number the release is based on.
file	string	the resulting release file.

Table 4: Release REST API

The issue API allows to write issue changes into the database. This process is specified by a user and an identifier for the issue. Furthermore, three optional parameters can be sent. The first one is the new status of the issue, the second one the assigned user for that issue and at last a release number the issue is associated with. Even though the three parameters are optional, at least one has to be given to write an issue change.

http://host:port/prov/issues		
user	string	the user name in the format “ForenameSurname”.
id	integer	the identifier of the changed issue.
status (optional)	string	the new status of the issue.
assignee (optional)	integer	the new assignee of the issue.
release (optional)	string	the corresponding release number.

Table 5: Issue REST API

The test API allows the CI system to send information about the software tests to the database. A test is specified by the revision it was running on and the number of successful and failed tests.

http://host:port/prov/test		
revision	integer	the revision number corresponding to the tests.
success	integer	the number of successful tests.
failure	integer	the number of failed tests.

Table 6: Test REST API

The VCS API consists of two functions, one for successful and one for failed commits. The parameters for a successful commit are the committing user, the new revision number, the corresponding message and issue identifier.

http://host:port/prov/vcs/revision		
user	string	the user name in the default account name format.
number	integer	the new revision number
message	string	the commit message.
issue	integer	the corresponding issue identifier.

Table 7: New revision REST API

Instead of the revision number and issue identifier, the failed commit function takes just the output of the VCS which specifies the problem.

http://host:port/prov/vcs/failure		
user	string	the user name in the default account name format.
message	string	the commit message.
output	string	the reason for the failure, provided by the VCS.

Table 8: Commit failure REST API

3 Requirements

The design and model of the software development process as a provenance graph requires two essential parts: the questions being asked and the data from the tools incorporated into the process. On the one hand, provenance questions determine the complexity and granularity of the model by indicating what information has to be available in the model. On the other hand, the tools have different kinds of information retrieval mechanism which define what data can be part of the provenance graph. This chapter lists all questions regarding the development process and describes in which way data can be retrieved by different tools used in the process.

3.1 Provenance Questions

The main task of provenance is to answer questions about the process and all its corresponding data. The recording of provenance data without having such questions would be useless. Also, questions about the process determine the requirements for the provenance model. The questions to be answered and the data needed mark the starting point for modeling a provenance graph. Therefore, this section gives a number of questions, divided into content and process questions. Content questions target specific data of the graph such as a user name or aggregated information such as the number of commits on one issue. Process questions deal with the sequential order of activities and the compliance with the applied software development process. While the provenance questions about the content are taken over from [Wen10], the process questions were not considered in the previous work.

3.1.1 Content Questions

As mentioned before, the following questions aim for specific data within the software development process. Those questions are used to give information about different aspects of the process such as error detection, statistical analysis or developer rating.

- **Error Detection:** A build or a test might fail during the process and by analyzing the preceding commits with its generated revision and the associated developer, the source of the failure can be located.
- **Statistical Analysis:** These questions aim mostly for aggregated data and can give conclusion about the complexity of an issue or the quality of the software documentation.
- **Developer Rating:** This is probably the least popular purpose of provenance and there are concerns about protection of privacy, but developer data might be helpful to improve the process by identifying a developer's skills like unit testing or writing documentation and therefore assign them to specific tasks.

In the following, a total number of twenty questions are given. Those questions are just an excerpt of the many possible questions on provenance data and shall give an overview about its feasibility. The complexity of the questions varies, some just need information from a single tool while others combine data from multiple tools.

1. **Who is responsible for implementing issue X?** This question only aims for an issue and its assigned developer. To answer it, the issue must be found via its identifier and the value of its assignee-property should be returned.
2. **What is the current build status of the project?** The answer to this question is again just a simple property lookup to obtain the status of the automatic build in the CI system. The last build must be determined via its time stamp and the result of this build should be returned.
3. **What is the current overall code coverage?** Like the question before, this one deals with the coverage in the CI, The code coverage indicate code quality be assuring the code is well covered with tests. The last executed coverage process has to be found and the percentage of covered code returned.
4. **From which revision was release X built?** Since software releases are performed multiple times throughout the development process, it is necessary to identify the revision on which the release is based to provide bug fixes later on. It should just return the number of the revision associated with the release.
5. **How many unsuccessful commits did user X do?** The VCS or a hook script might reject a commit for various reasons. The occurrence of multiple failures may indicate a problem with the configuration of the VCS. To answer it, all failed commits of a user have to be retrieved.
6. **How did the number of unit tests change in the last month?** Unit tests are a good indicator for software quality, more tests with a good coverage result in better software. This question aims for the last test and the one from one month ago and compares the total number of tests.
7. **Which developer is most active in contributing documentation?** Since documentation is essential for understanding the software, more contributed documentation may result in rewards for the corresponding developer. To answer this question, the total amount of documentation from every developer is calculated.
8. **How many releases have been produced this year?** This question aims for the productivity in the process. Stagnation or delay in the development may cause a deviation in the release schedule. Therefore, it should simply sum up all releases in the last year.
9. **How many issues were implemented by developer X for release Y?** The answer to this question may indicate two things. On the one hand, issues implemented by one user is an indication of productivity of the user. On the other hand it shows the complexity of an issue. The scope of the question is from the specified release to all related, closed issues which are assigned to the specified user.
10. **How many commits did developer X contribute to release Y?** This question is again an indicator for the productivity of a developer. However, it may just show a different level of efficiency between a developer with few commits and a developer with many commits. From the specified release, the number of all commits are summed up and returned.

11. **How many developers contributed to issue X?** Similar to question nine, an issue with many contributing developers is probably very complex. This can happen if the issue was too large to be handled from one or two users and should therefore be split in multiple issues. To answer this question, all versions of the specified issue are found and all contributing users who did issue changes, commits or documentation on the issue are summed up.
12. **Which developers contributed to release X?** This question is for statistical analysis and shows, which developers were working on a release, whether it was a commit, documentation or changing an issue.
13. **How many commits were needed for issue X?** Like question eleven, this one looks at a specified issue and rates its complexity. But this question counts the number of commits on the issue, regardless which user committed code.
14. **How much time has been spent implementing issue X?** A long implementation time for an issue may be another indicator for a very complex issue, but it may also be a low prioritized issue. To answer it, the time stamp of the last version or the version with the status "resolved" respectively and the first version of the issue have to be compared.
15. **What documentation belongs to issue X?** This question helps to find documentation on a given issue, which might be distributed over multiple systems. It just takes the issue and finds all belonging documentation.
16. **Which features are part of release X?** During the release planning, all functionalities for the upcoming release are defined, but some functions might not be implemented in time and ca not be part of the release. This question is useful to compare the actually implemented features with the planned ones. To do so, all issues assigned to a specified release are returned.
17. **Who is responsible for reducing the code coverage?** Code coverage can reduce if a developer writes new code but does not generate tests for this code. This question aims to find the user responsible for reducing the code coverage. If the coverage percentage has dropped between two coverage reports, those are selected and the developer who did the commit, on which the latter coverage report is based on, is returned.
18. **Which requirement causes the most build failures?** Another indicator for a complex issue or a badly designed requirement can be the number of failed builds. To answer this question, all issues and their related commits are selected and the one with the highest number of failed builds returned.
19. **Which changeset resulted in more failing unit tests?** An increasing number of failed unit tests can happen if two users were changing similar code. This may indicate some kind of dependency between the issues they were working on. The question's goals is to find commits, which changeset resulted in an increasing number of failed unit tests.

20. **Which version of maven caused the build failure of revision X?** A change in the maven version may cause a build to fail due to configuration issues. To detect this kind of change, the version of maven associated to a failing build for the specified revision is returned.

3.1.2 Process Questions

While the aforementioned questions ask about the content of the provenance graph, the following questions aim for the software development process. They aim for different aspects such as process validation and process optimization.

- **Process Validation:** The main goals of a development process are distribution of workload and the assurance of software quality. But a process can only reach these goals, it all developers follow the process as specified. Therefore, the process has to be validated using the provenance graph and queries.
- **Process Optimization:** Every process can be improved and optimized for its working environment and its involved developers. A good indicator for this is a deviation from the process due to implementation problems for example.

The first half of the following ten questions check the order of tasks performed in the process, while the last five questions cover some work products.

21. **Was the issue status set to ”in progress” before any commit was performed?** Developers should only commit code on issues which status' they updated previously to “in progress” to indicate, what they are currently working on these issues. This way the project manager or team leader, what issues are soon to be completed and plan next assignments or iterations.
22. **Were all tasks of the CI executed before a build?** If there is a problem with the CI system and for example tests are not executed before a build, the query for this question would recognize such misbehavior.
23. **Was the list of done completed before an issue was set to ”resolved”?** Typically, the list of done is used to specify work products which have to be done to a defined point in time. It includes a successful build with no failed tests and documentation. Only if all these tasks are completed an issue status can be changed to “resolved”.
24. **Was the issue status ”resolved” before the release process?** This questions checks that a release only includes resolved issues. If an issue was planned for a release but did not finish in time, it may be a too complex feature.
25. **Was the issue ”closed” after the release process?** After the software is released, all included issues should be closed to indicate the completion of those issues.
26. **Did anyone commit code on the issue except the assignee?** To check, whether the assignee was the only committing developer for an issue or not can show different problems. On the one hand, the issue might be too big and complex for one developer

to implement, therefore one or more developer were needed additionally. On the other hand, it might indicate, that the developer skills were not sufficient to solve the issue and this knowledge can be applied for future iteration planning.

27. **Did anyone write documentation for the issue except the assignee?** This question basically has the same purpose as the aforementioned but in regards of the documentation for an issue.
28. **When did the coverage percentage dropped below X?** In contrary to question 17, this one examines if the code coverage drops below a specified value. Depending on the project settings, a low coverage percentage may display a warning to the user.
29. **When did the number of failed unit test exceed X?** Similar to the previous question, a above specified number of failed unit tests indicates a problem in the software.
30. **Was there a build not performed on the last revision?** While it is not mandatory all the time, e.g. for a release, a build should usually use the last committed revision.

3.2 Provenance Data from Tools

To record provenance data from different tools, it is necessary that all tools implement a system to detect actions performed or trigger some kind of event. Furthermore it is important to now, what meta-data can be recorded from each tools. This section gives an overview of how to get provenance relevant information from the tools used in the development process.

3.2.1 SVN/RepoGuard

The event system of SVN is based on a server-side hook system. [CSFP02] Every repository has a folder called “hooks” with templates for five different events:

start-commit runs before the actual transaction. This can be used to check privileges of the committing user.

pre-commit runs after the transaction is completed but before the actual commit is written. Checks on the code or commit message are usually performed with this script to verify the commit before it is saved. A failure in this script causes the whole commit to fail.

post-commit runs after the commit is written and a new revision created. So a failing post-commit hook would not stop the commit and is therefore typically used to perform actions with the commit like sending emails or start a build job.

pre-revprop-change runs before a revision property, like `svn:log`, is modified.

post-revprop-change the counterpart to the aforementioned hook, but this hook will only run if a pre-revprop-change script is available.

From those different events, the pre- and post-commit are the most interesting for provenance recording. A hook script can be any kind of executable file, it must just have the name of the desired event. So a simple command or bash script can be used as well as a python file. Its important to check privileges to execute the file though. For pre- and post-commit hooks, SVN passes two parameters, the repository path and the transaction id. With those to parameters, the commit can be analyzed using svnlook and its options. These options give you different kind of information about the commit:

- **author** returns user who executed the commit
- **cat** lists the content of a file, identified by an extra argument.
- **changed** returns the paths of all changed files and folders. The modification is further specified via letters, “A” for added, “U” for updated and “D” for deleted.
- **date** returns a time stamp in the format YYYY-MM-DD.
- **diff** lists the differences of a specified or all changed files and properties.
- **dirs-changed** returns the directories that were themselves changed.
- **history** gives information about the history of a path in the repository or the root directory if no path is given.
- **info** returns the author, time stamp, log message size and the log message.
- **lock** if a lock exists on a path in the repository, it will be returned with this option.
- **log** shows the log message.
- **propget** print the value of a property on a path in the tree.
- **proplist** lists the names and values of file and directory properties on all paths.
- **tree** illustrates the complete repository in a tree format.
- **uuid** returns the repository’s universally unique identifier.
- **youngest** displays the last revision number.

For the current model, the options “author”, “log” and “youngest” would be sufficient, but with further development on the model and new questions, more fine-grained data like the actual difference between the last and newest revision might be interesting.

Instead of implementing this hook system from scratch, the open-source tool RepoGuard is used. RepoGuard was developed at the DLR and is written in Python. It is “an advanced validation framework with built-in integrations for several common version control systems” [Sch]. Using the hook system of SVN, RepoGuard builds an environment where all information from the transaction respectively the commit is available. It comes with some built-in function to increase automation of the development process by linking different tools, e.g. SVN and Mantis. It is DLR-widely used to check commit messages (length and issue id), code standards or file encoding. To do so, it implements a system of checks and handlers. The first tests the commit or its meta-data for defined rules or specifications, the latter performs actions on incoming commits like sending a mail to all other developers for information. For integration into the provenance system, RepoGuard must simply implement a new simple handler to send all given SVN data to the provenance store.

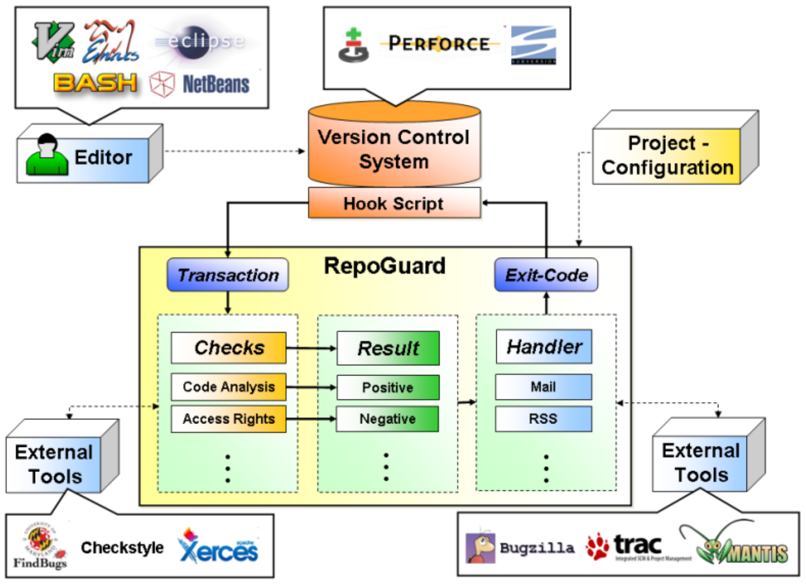

Figure 7: Architecture of RepoGuard [Sch]

3.2.2 Mantis

Mantis' event system is called custom functions, which are basically functions written in php. [Man] Each function is called upon a specified event and functionality has to be implemented as needed by the user. The file `custom_function_api.php` holds all custom functions, important for provenance are

- `custom_function_default_issue_update_notify`,
- `custom_function_default_issue_create_notify` and
- `custom_function_default_issue_delete_notify`.

One of those three functions are called every time an issue is created, updated or deleted. There are similar functions to be called before any of those events occur, identified by a "validate" instead of "notify". The first two get the issue identifier as a parameter, the last gets the data from the deleted issue. Thus any necessary provenance information is given or can be extracted with the identifier.

3.2.3 Jenkins

To react on events, Jenkins integrates a plugin system. [Jen] Just as Jenkins itself, all plugins are written in Java and there are many available on the web. So instead of implementing a completely new plugin, the so called "notification"-plugin can be used. This allows to send job status via HTTP, TCP or UDP and formats all notifications as JSON data like this:

```
{
  "name":"JobName",
  "url":"JobUrl",
  "build":{
    "number":1,
    "phase":"STARTED",
    "status":"FAILED",
      "url":"job/project/5",
      "full_url":"http://ci.jenkins.org/job/project/5"
      "parameters":{"branch":"master"}
   }
}
```

Listing 4: Notification Plugin JSON Example

The actual job data is far more complex and includes all necessary provenance information, so the problem reduces to extracting all data from the notification.

3.2.4 MoinMoin

As with all other tools, the MoinMoin wiki engine also comes with an event system. [Moi] In the file MoinMoin/events/`__init__.py` are many different events defined and are later called upon their incidence. Events can be for instance a page change, a page rename, a page deletion or a file attachment. Thus the current provenance model can be easily expanded with more events than just the page change. Every method has at least the page name as a parameter, other information differs. All events can call handlers to pass their information, so a new handler to send the provenance can be easily implemented.

```
class PageChangedEvent(PageEvent):

    name = u"PageChangedEvent"
    description = _(u"""Page has been modified""")
    req_superuser = False

    def __init__(self, request, page, comment):
        PageEvent.__init__(self, request)
        self.page = page
        self.comment = comment
```

Listing 5: PageChangedEvent from MoinMoin

4 Concept: Provenance SD Model

This chapter is about the implementation of the software development process into a provenance model. It gives an overview about the domain of software development as a specification of the PROV data model and illustrates all its components. This model is a remake of the work of [Wen10] who used the OPM to model the process and which was the subject of analysis for this thesis.

As mentioned in section 2.2.2, this thesis follows a process-centered view on the provenance data. For this reason, the model is described using the activities as main objects with their input and output data as well as their associated agents. The subprocesses of the software development are Issue Tracking, Coding, Testing, Building, Documentation and Release.

Issue Tracking is an activity performed throughout the whole development iteration. From distribution of work in the beginning to the release in the end, issues are created and updated. The main tasks of issue tracking are the assignment of work products to the developers, submitting bugs and plan the functionalities of the next release.

Coding is one of the main tasks in the process. After the developers designed their assigned issues, there will be a multitude of commits for the implemented code since it is very unlikely that a developer finishes his work in one work step. A commit is always related to an issue and will create a new revision and a changeset in the VCS.

Testing is part of the CI. Every developer should test his code on their local machine and integrate it into the complete system, invoking another test. Since recording local tests is not part of the model yet, only tests on the CI are recorded. Part of each test is a coverage report, giving the percentage of covered code by the test.

Integration of the software happens after all tests and the build ran and creates the actual product. So this is also part of the CI and results in a number of output files.

Documentation is an activity and work product at the same time. Code comments, user or developer guides, meeting logs or architecture design documents can all be part of the documentation and are stored in the central documentation system.

Release is the last activity in the iteration. A final build is performed and the result is packed to a distributable software package. Also the next iteration is prepared by closing old and creating new branches and updating the issue tracker.

Every one of these activities has a specific number of input and output data as well as an associated user or tool. The following figures illustrate all subprocesses mentioned above and their relationships to other object in the PROV scheme. The Eclipse plug-in Neoclipse has been used to create these graphical representations of the graph. Note that the color of nodes depends on its incoming relationships and since they are automatically assigned upon connection to the database and can not be changed manually, these figures may have a different color scheme than those in later sections.

All nodes have two properties in common, a 'type' property and a 'prov_type' property. These two specify the node type in terms of the software development process and the provenance model respectively. So with the value of 'prov_type' nodes are distinguished into agents, activities and entities while the value of 'type' categorizes nodes into a user, issue or commit node. All other properties are given in the figures with their key and the corresponding data type. The letter in the brackets marks a mandatory (M) or optional (O) property for this node. All relationships follow the conventions of the PROV data model. In the current model, there are no properties on relationships, so all are illustrated as simple arrows with their corresponding type.

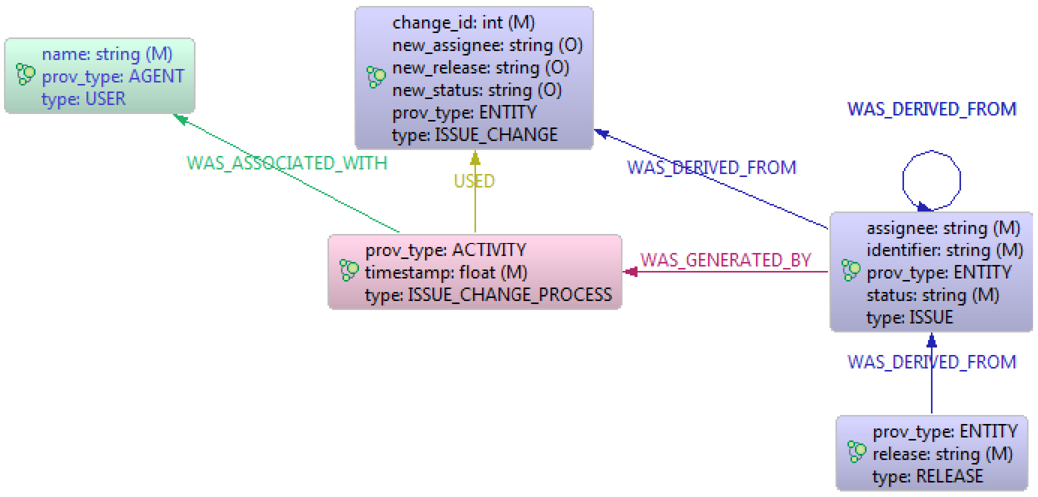

Figure 8: PROV scheme for the issue change process

The issue or bug tracking is modeled as an issue change process as can be seen in figure 8. Like all activities, its only property is a mandatory time stamp in the Unix time format. The input for an issue change process is the issue change, which holds all relevant data and is used by the process. It holds one mandatory property, the identifier of the changed issue and three optional properties: the new assignee, the status update and the new release. While these three are optional, at least one of them must be present, since otherwise there would be no change on the issue. The issue change process is also associated with a user who modified the issue and is specified via an unique name like his account name. As a result of the process, a new issue node is generated with three mandatory properties: the identifier, the status and the assignee. Since the new issue node is the result of the change, a generation-relationship to the process node and a derivation-relationship to the issue change node are created. Once a node has been written it must not be altered in the PROV specification, therefore an issue change always generates a new issue node with the same identifier. With a derivation-relationship between two issue nodes, the issue is presented in its various states throughout the process. At last, if the property 'new release' is set on the issue change, a release node will be generated and linked to the issue.

The design and implementation of the software, summarized as coding, is modeled in figure 9 as a commit activity. This subprocess is again performed by a user and used a changeset. The changeset has the mandatory property 'change_set' which holds the commit message given by the user. Since a commit message always requires the issue identifier to be successful, there is also a relationship between a successful commit and the issue to which the commit belongs to. As a result of the commit, either a new revision or a commit failure is generated. If the commit was successful, the data is written to the VCS, marking a new revision in the repository. This is modeled with a revision node and its property 'revision' with the revision number. If a commit fails for some reason, a commit failure node is recorded. This node holds the information as to why the commit failed, given by the hook script (RepoGuard) or the VCS. If the commit fails, the commit node is not linked to the issue node.

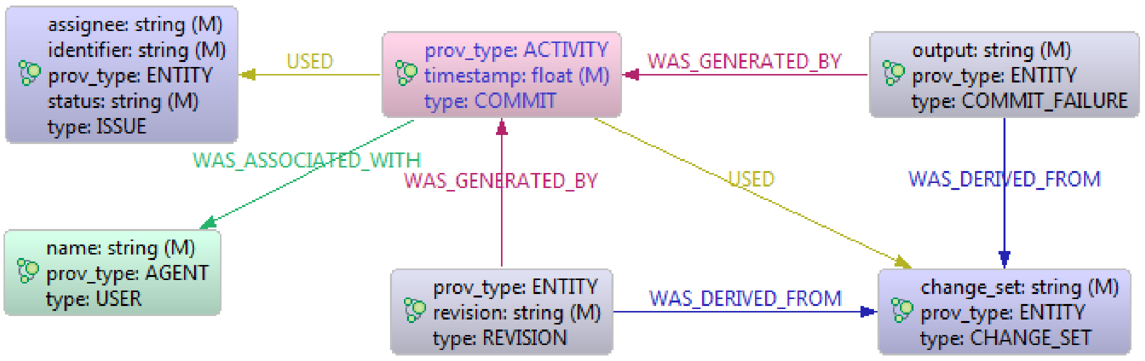

Figure 9: PROV scheme for the commit process

The next two subprocesses, testing and integration, are both tasks within the CI and illustrated in figure 10. Testing is further divided into the actual unit test and the corresponding coverage report process. All three tasks are linked to an revision on which their work is based on. This marks the revision number which was tested and built by the CI and it is also the linking element of those three processes. A unit test process generates a test result which contains the number of successful and failed tests for this run. Following the unit test, the coverage process generates the code coverage report for the test. The only property of the coverage result is the percentage of covered code, not specifying the metric for calculation. The third process, the build, creates a build result which holds the exit code of the build process. Successful builds exit with a zero, all other numbers specify different kind of failures. The build is also linked to an agent 'Maven', which indicates the maven version used to build the project.

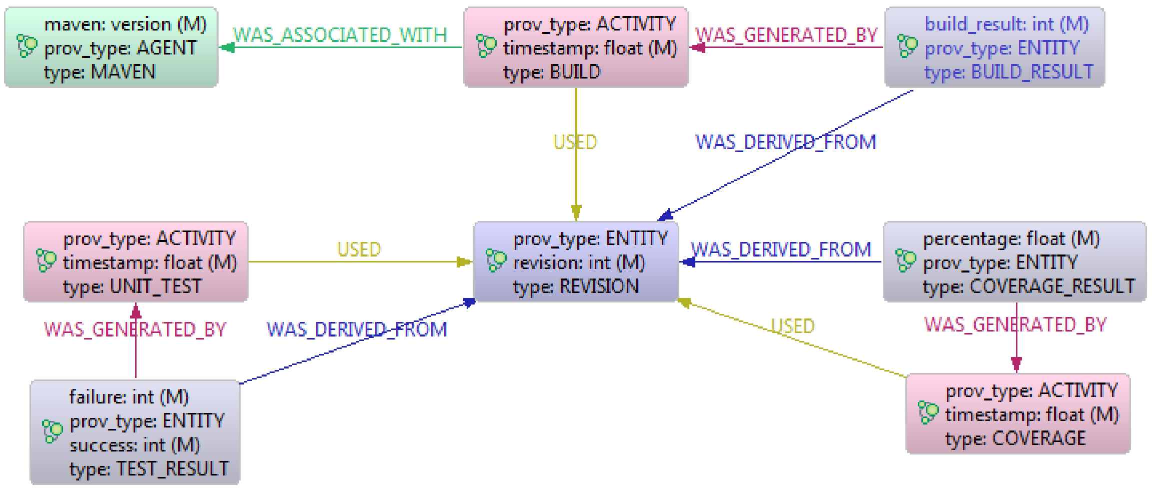

Figure 10: PROV scheme for the continuous integration including test, coverage and build process

In figure 11 the documentation change process is depict. This process is associated with a user who modifies a documentation page. As a result of this task, a new 'doc_change' entity node is generated with the mandatory property 'document' and the optional 'comment'. The document property marks the name of the modified page, e.g. a wiki page, while the comment is an additional information given by the user which summarizes the change. If the comment includes a corresponding issue identifier, the documentation change process can be linked to this issue node, indicating its use for the process.

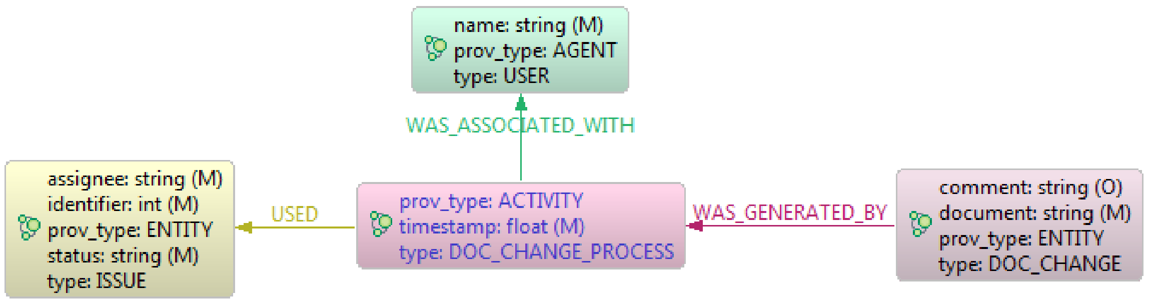

Figure 11: PROV scheme for the documentation change process

The last activity in the iteration, the release process, is modeled as shown in figure 12. Like the subprocesses of the CI system, the release process works with a specified revision, most likely the newest. Therefore, it is linked to the revision node as part of its input. The release produces a various number of files, each file as a separate node with its name as a mandatory property. With the version number of the release every file is also linked to the corresponding release node generated in the issue tracking process (see figure 8).

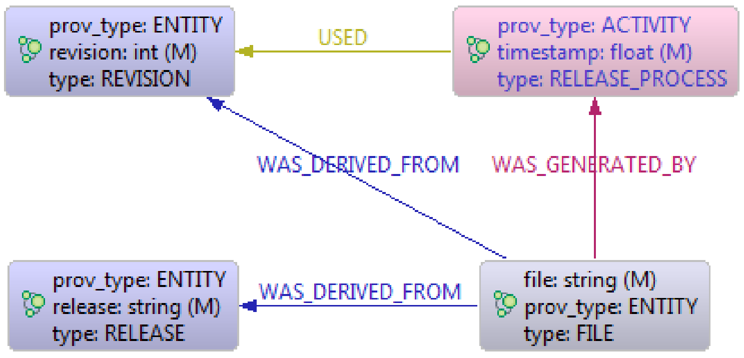

Figure 12: PROV scheme for the release process

5 Software Development Process

In section 2.1.2 an overview for the software development process was given. This section describes the employed AUP in two more in-depth ways, as a graph and as a formal model. The graph model is the representation of the process as a provenance graph. It shows what subprocesses generate what kind of data in an iteration and how they are connected in the provenance store. The formal model is the definition of the software development meta-model. It serves as a verification of the graph model, so the provenance data can assure that the process was followed as specified by the formal model. There will only be a theoretical view on how to formalize such process using the SPEM, because there is no currently existing formal model and the whole process is just textual documented.

5.1 Graph Model

The main purpose of the provenance for the software development process is to record all steps and subprocesses for two tasks, first to answer questions regarding the software product and its development and second to assure software quality by verifying the process. Both tasks require queries on the database, the latter also requires to understand how the process is recorded to compare it with a formal model.

In the following all steps of an iteration are illustrated. Since no real provenance data exists, an example process following the software development process as described in section 2.1.2 is outlined. This example process has two users with one issue to work on and displays how the graph is built up while the development proceeds. Although all activities occur very fast after another, the chronological order matches with the specified process.

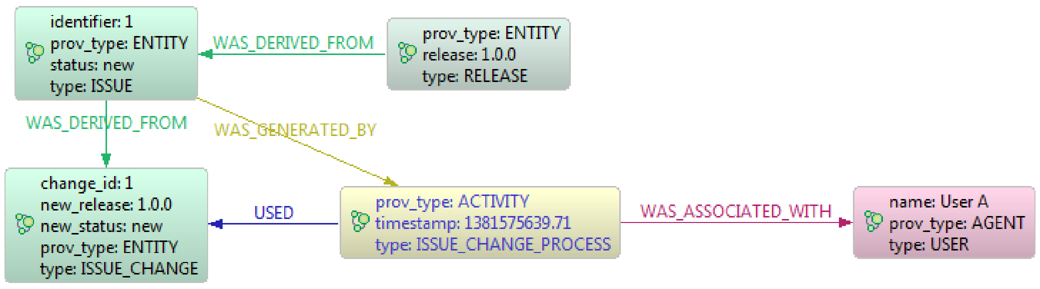

Figure 13: A new issues is created

An iteration starts with the creation of new issues using the web interface of Mantis. These can be new functionalities for the software or minor to major bug fixes. Issues worked on in one iteration should contribute to the next release, which is planned beforehand. In figure 13 'User A' starts the issue change process. This process generates a new issue with the status 'new' using the issue change. The issue change displays the current change of the issue holding all modified data while the issue may inherit some of its properties from its foregoing issue. Since the issue contributes to the release '1.0.0', a derivation-relationship is created between those two.

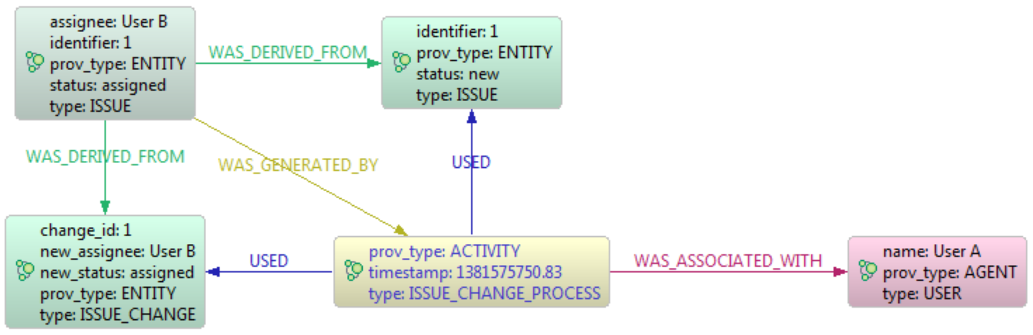

Figure 14: The issue assigned to User B

In the next step, all issues are assigned to developers who should work on them. The assignment can be conducted by every user with the corresponding privileges on the issue tracker, in this example 'User A' who assigns the issue to a 'User B' as can be seen in figure 14. Since an already existing issue is changed, the issue change process now uses both, the issue change and the old issue and sets the relationships accordingly.

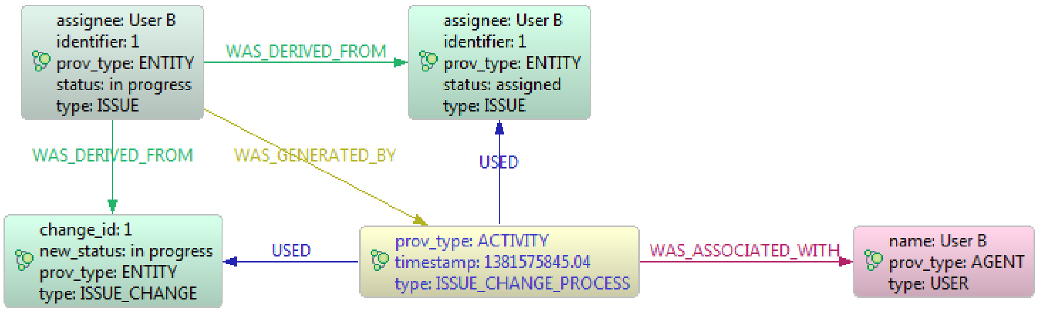

Figure 15: Starting to work on the issue

The moment the issue is assigned, the developer can start to design and implement the issue. Because one developer may have multiple issues assigned to him, he has to mark the issue he is currently working on. This step is usually done by the IDE as soon as you select the issue in the Mylyn plug-in, but can also be changed using the web interface. In figure 15 the assignee 'User B' set the status of the issue to 'in progress'. This figure also shows that the issue change holds just the change information and all unmodified properties inherit their values from the previous version.

While designing and implementing software components, the developers commit their progress to the VCS. Since the provenance recording system is in the hook scripts of the repository, a developer may use the IDE, a command line or any other tool to commit his data. Figure 16 shows the commit process started by 'User B'. With the issue identifier in the

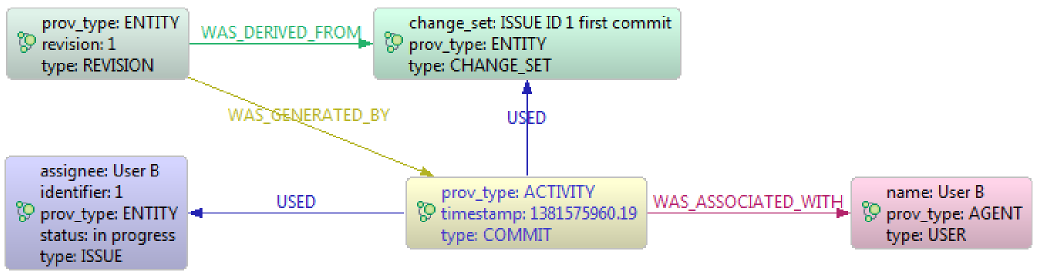

Figure 16: The first commit on the issue

commit message, the newest version of the corresponding issue is automatically connected to the commit process with an usage-relationship. For this example it is assumed that developer 'User B' did three commits for his issue, but since all have similar graph representation, only the first is shown.

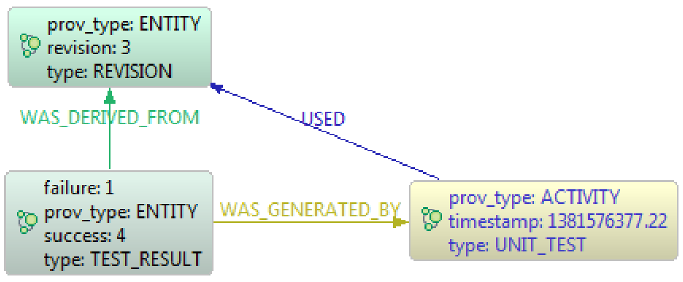

Figure 17: Testing the last revision

Before an issue can be resolved, it has to be built and documented using two systems, the CI system and a wiki. The build process usually triggers the test of the software with its coverage report. Although these are three different activities, they occur most time together on the CI system. The following three figures illustrate the sequential order of those build steps. Since local tests and builds are not recorded, they occur either on a commit of a new revision, on a scheduled way such as nightly builds or manually executed by a user, depending on the settings.

First all tests are bundled in a test process with a result as shown in figure 17. The process and the result are both connected with the last revision which is the current state of the software. Following a test, the coverage report is automatically created, again making the connection to the last revision. After both, the test and the corresponding coverage report have finished, the build process starts. This process uses the last revision to build the code, so they are also related. Maven as a build management tool is the agent for this activity.

The last work product needed for this iteration is the documentation for the implemented issue. 'User B' writes a new wiki page, accessing the MoinMoin wiki via its web interface. Only the modified page is mandatory, but since the comment on the change includes the

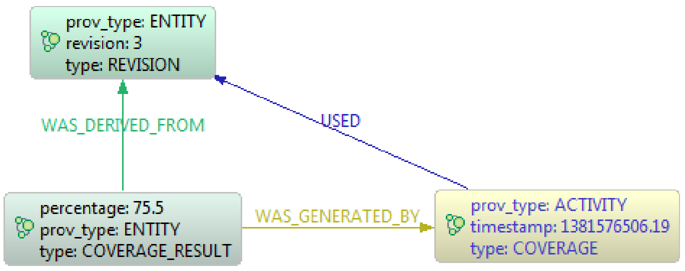

Figure 18: The coverage report for the test

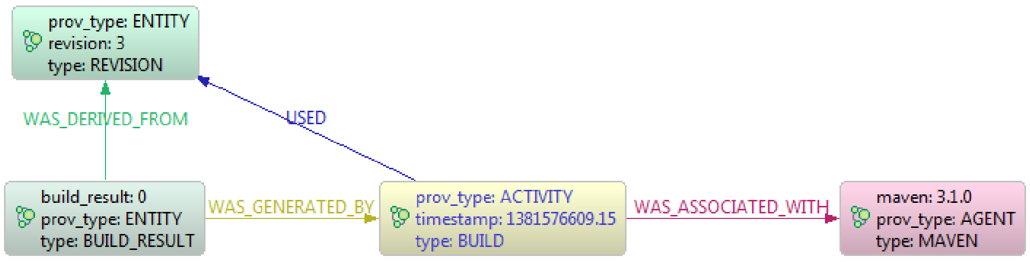

Figure 19: Building the last revision

corresponding issue identifier, the documentation change process can also be related to the issue node. Now that all work products - tests, coverage report, successful build and documentation - for the issue are completed, the issue can be resolved. This indicates that all tasks for the issue are finished and it is ready for the release. Similar to the issue changes before, figure 21 shows the status update for the issue by 'User B'.

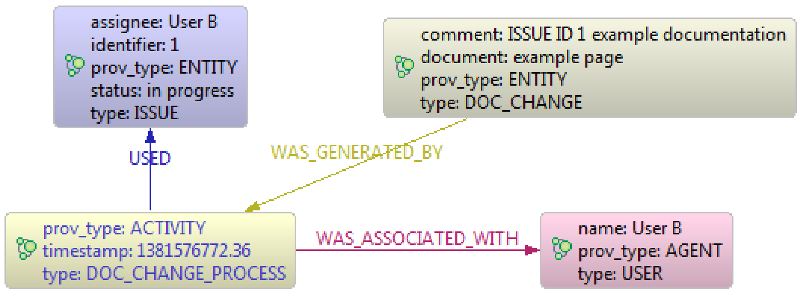

Figure 20: Documentation for the issue

With a resolved issue and assuming that this iteration had only this issue, the release process can now be started. This usually happens with a release script to pack all data and documentation into a software bundle. Again, the release process and the release file are connected with the last revision which the release is based on. As stated in section 2.4, every file is recorded individually to the release process by calling the release API multiple times. That is why there is only one release file in figure 22 while the complete example has three files.

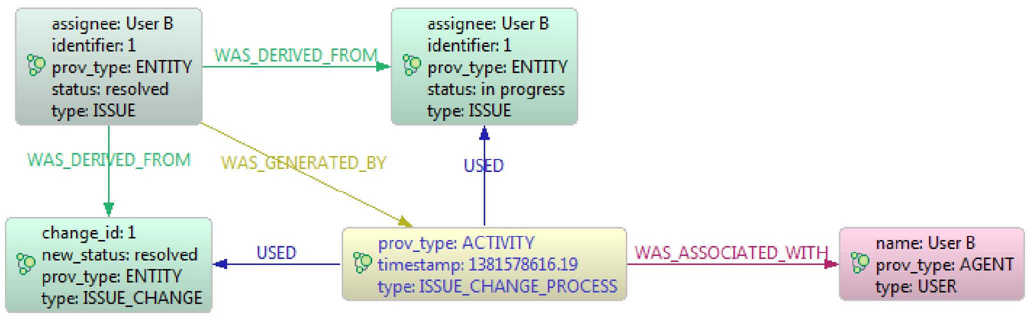

Figure 21: Set the issue to resolved

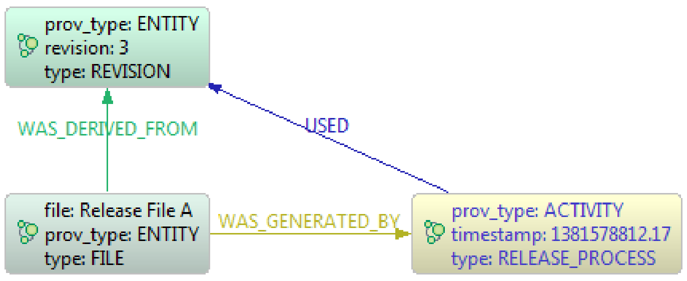

Figure 22: Perform the release process

The last task of an iteration is to either close resolved issues or re-assign them for the next iteration because of a bug or fault in its functionality. In figure 23 the issue is closed by 'User A', the same one who created and assigned the issue in the first place.

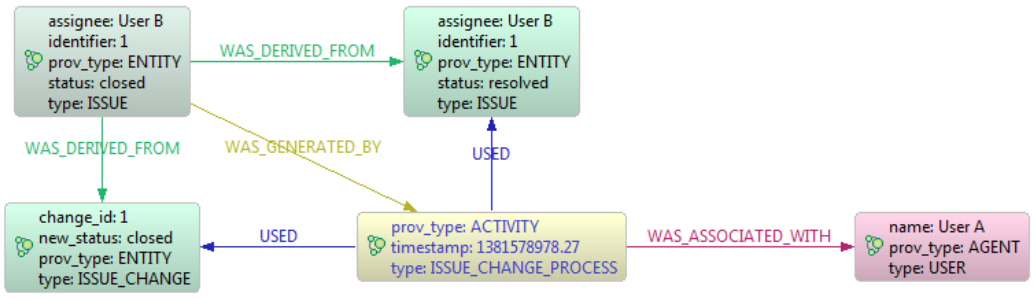

Figure 23: Closing the issue

5.2 Formal Model

As mentioned in the introduction of this section, the software development process at the department SC is only defined on a textual basis. The software project manual [Sch13c] documents phases, iterations and tasks, but does not specify the process with a formal model. This textual documentation is not applicable to compare it with the provenance graph to verify the process. To define a formal model and compare it with the graph model automatically, a computer-readable definition is necessary. This kind of definition can be generated using the SPEM. In the following, a brief overview about the purpose and capabilities of this meta-model is given as well as an outlook how to specify the current process with it.

The Software & Systems Process Engineering Meta-Model Specification is “used to define software and system development processes and their components.” [OMG08] It gives its user all necessary concepts to model and document development methods with minimal elements to include a wide range of processes. The initial version was developed in 2002, the current version 2.0 was released 2008. SPEM uses UML2 for process modeling and divides its elements in seven packages which build on top of each other:

- **Core** includes the basis for all other classes. Every element which is needed in every other package is defined here.
- **Process Structure** accommodates simple and flexible activities and methods. Elements defined in this package can be pictured as tasks in a process with refer to a performing actor as well as input and output work products.
- **Process Behavior** extends the process structure with behavioral models. The process structure is a static breakdown structure and this packages allows the user to add behavior using links to external models like the Activity diagram in UML2.
- **Managed Content** provides concepts for managing the textual content and human-consumable documentation. This includes documentation in natural language or guidance for best development practices.
- **Method Content** contains methods and techniques for software development which are independent of any specific process or project such as a task, role or work product.

It defines reusable methods which can be implemented into a development process.

- **Process with Methods** combines the process structure with the method content. Elements of the aforementioned packages are adjusted and placed into the context of a life-cycle model. It describes for example, what activities and work products are incorporated for each phase or milestone of the process.
- **Method Plugin** introduces "concepts for 'designing' and managing maintainable, large scale, reusable, and configurable libraries or repositories of method content and processes." [OMG08]

Figure 24 shows the conceptual usage framework of SPEM and how the packages work together. As mentioned, the method content represents all process-independent knowledge about software development like content on agile development or test guidance. The processes for performing projects such as embedded system development are defined alongside. They include all tasks and activities for specified development processes. Every project uses parts of both packages and merges them into a new process framework which is customized for its needs and the resulting project template defines procedure in the context of a specific project.

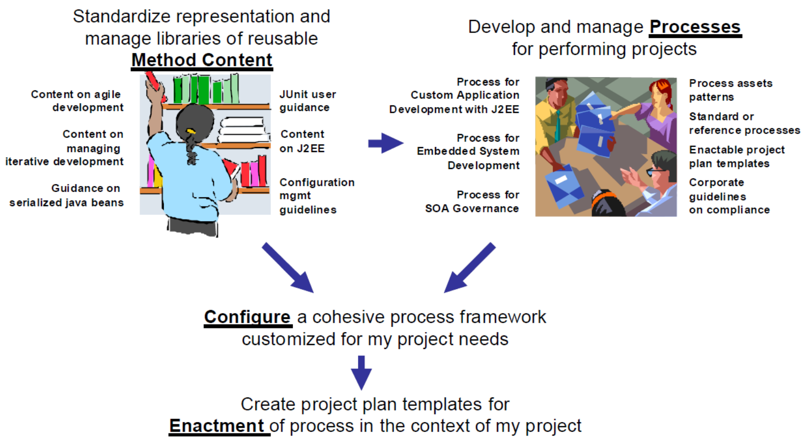
Standardize representation and manage libraries of reusable Method Content
Content on agile development
Content on managing iterative development
Guidance on serialized java beans
JUnit user guidance
Content on J2EE
Configuration mgmt guidelines
Develop and manage Processes for performing projects
Process for Custom Application Development with J2EE
Process for Embedded System Development
Process for SOA Governance
Process assets patterns
Standard or reference processes
Enactable project plan templates
Corporate guidelines on compliance
Configure a cohesive process framework customized for my project needs
Create project plan templates for Enactment of process in the context of my project

Figure 24: Conceptual usage framework [OMG08]

After this theoretical overview of SPEM, next follows an example how an activity is defined using the model. In figure 25 a "Use Case Analysis" activity is depicted with its corresponding actors and work products. Different symbols are used to specify roles, tasks and work products and directed links with stereotypes illustrate their relationship to each other. The presented activity uses a mandatory use case and an optional analysis model to analyze the use cases. The activity is performed by a designer with additional help from a system analyst and results in an analysis model and an optional use case realization. As can be seen in this example, the specification of a process looks similar to the provenance graph of a process. Both use a graph model with directed relationships and distinguish their elements into work products (entities), activities and roles (agents). To model a process with SPEM, the eclipse process framework composer can be used. [Hau07] This tools incorporates several example methods and processes and allows its user to define and model new development processes. Using the EPF Composer, a formal model for the process can be specified beforehand and later verify the process during and after development with the graph model.

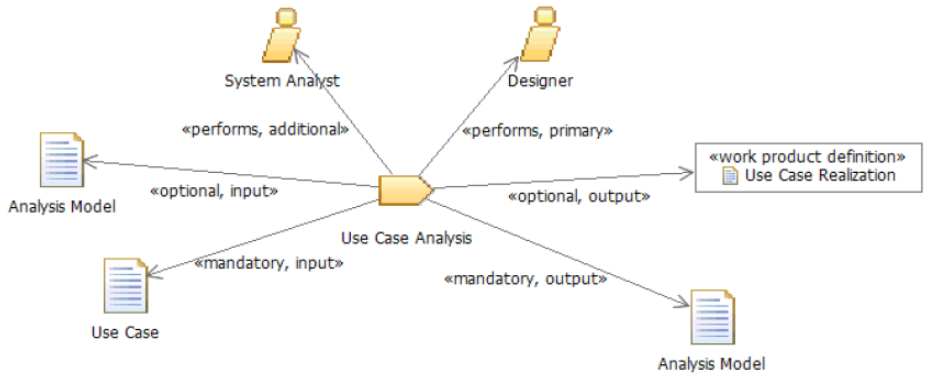

Figure 25: Example activity in SPEM [OMG08]

6 Analysis of Provenance Data

With the description of the provenance model and the representation as a graph, the provenance data is analyzed in this chapter. The first section examines how many nodes and relationships are created for each step in the software development and how much data is generated by the process on a daily and weekly basis. The following section discusses some flaws in the model and the infrastructure and presents possible solutions and extension points for them.

6.1 Data Volume

Different tasks in the development process are executed with a different frequency. While code commits may happen multiple times a day, tests, coverage reports and builds are typically scheduled once per day, e.g. as a nightly build process, but can also be kicked off by a user. New issues are generated at the beginning of an iteration and changed (status change or commit on the issue) multiple times throughout the iteration. Documentation should be added at the end of one cycle, this may also take multiple updates. The release process is also performed at the end of an iteration, this will likely only happen once.

The nodes for a user, maven and a release will only be created on their first appearance and later just queried to set relationships. For the analysis of data volume we assume that the nodes user and maven are already stored in the database because they do not change often in the software development.

A commit is likely to be the most executed task in the process. Regarding the issue complexity and the developer's coding experience, it will happen at least once daily. Since both successful and failed commits are recorded the number of data increases. The following data is recorded for one commit:

- the commit process with a time stamp
- the change set including the commit message
- a relationship between the process and the user as well as the relationship between the process and the change set
- for a successful commit:
  - the revision with its number
  - the relationship between the process and the issue and between the revision and the change set as well as the commit process
- for a failed commit:
  - the failure with the corresponding output message
  - the relationship from the failure to the change set as well as the commit process

This makes a total count of three nodes and four to five relationships per commit, depending on successful or failed commit.

Next are the three processes started by the continuous integration framework, test, coverage and build. Depending on the settings, these processes are performed on a daily basis like a nightly build or every time a user commits new code to the linked repository. A test generates the following data:

- the test process with a time stamp
- the test result with the number of successful and failed tests
- the relationship between the result and the test process as well as the revision
- the relationship between the process and the revision.

With every test there is a coverage report on how many lines of code are covered by all tests. It produces likewise provenance data:

- the coverage process with a time stamp
- the coverage result with the relatively covered lines of code
- the relationship between the coverage result and the coverage process as well as the revision
- the relationship between the coverage process and the revision.

After the test and the coverage are finished, the actual build process begins and creates the following data:

- the build process with a time stamp
- the build result with an exit code
- the relationship between the build process and the maven version as well as the revision
- the relationship between the build result and the build process as well as the revision.

So with every successful run of the continuous integration framework, six nodes and ten relationships will be stored in the provenance database.

Issue management takes place in the whole iteration, so combined with commits it will produce the most provenance data. Whether a new issue is created or changed affects only the number of relationships:

- the issue change process with a time stamp
- the issue change containing the ID, and three optional parameters, depending on what changed: the status, the assignee and the release
- the issue with its identifier, the status and the assignee
- the relationship between the change process and the changing user as well as the change
- the relationship from the issue to the change and change process
- if the issue change contained a release id, a relationship between release and issue is set
- if an old issue is updated, there will also be two additional relationships, one between the new and old issue and one from the change process to the old issue.

If a new issue is created at the start of an iteration, three nodes and four relationships will be saved. Updated issues generate the same amount of nodes but two to three additional relationships depending if a release is added to the issue.

Beside those regular software development steps there are some infrequent processes. The creation of a part of the documentation occurs after an issue is resolved, so the following data is generated at the end of an iteration:

- the documentation change process with a time stamp
- the documentation change with the edited page and the alteration
- the relationship between the documentation change process and the editing user
- the relationship between the documentation change and the documentation change process
- if a corresponding issue is specified which is best practice but not necessary a relationship between the documentation change process and that issue is generated.

For one documentation two nodes and two to three relationships will be saved, depending on a denoted issue.

At last, new software releases will only be performed at the end of an iteration, so it will produce little provenance information in contrast to the other development steps:

- the release process with a time stamp
- the release archive with a string of all corresponding files
- the relationship between the release process and the revision used for this process
- three relationships from the archive, to the revision, the release and the process.

This makes two nodes and four relationships for a new release after a cycle is finished.

Following this overview of the amount of provenance data produced by each process, a sample project shall illustrate the data generated on one day as well as at one iteration. Instead of using the example iteration described in section 5, a new process is presented to give an example closer to existing team sizes and number of work products. It is assumed that six developers work on the software and the time span for one iteration is two weeks. The six developers commit about five to six times a day, as the analysis of a existing project at VSS suggests. Also, the continuous integration framework is configured to start a test with a coverage report and build the software. The software is built automatically on a nightly basis. This sums up to 51 to 60 nodes and 80 to 94 relationships written to the database per day. Issues are created and updated in the whole iteration, making at least five changes, adding further fifteen nodes and 29 relationships for each issue. For this example, one developer works on one issue per iteration. After coding and integration, every developer will expand the documentation, generating twelve nodes and eighteen relationships. At the end of each iteration, a release is produced which writes two nodes and four relationships to the database. So for one iteration, there are 155 to 164 nodes and 276 to 290 relationships. With an iteration length of two weeks, the development process generates a little more than 4000 nodes and almost 7500 relationships in one year. This does not come near Neo4J's maximum capacity which is about 34 billions nodes and relationships [Neo13]. Also writing and reading time should not be a problem since these process steps do not happen all at once but with a certain time gap between them. Even if all developers commit their code at the same time, PyProv is able to buffer the load and summarize it to one writing job.

6.2 Model Discussion

This section discusses the current model and recording process and explains some flaws, possible improvements and how to expand the model to retrieve more interesting data from the process to analyze it further. As mentioned in section 2.4, the recording interface prOOst used outdated dependencies and more important the OPM which was replaced with the new provenance model PROV. With new capabilities of modeling provenance data, the model for the software development process has to be revised. Besides some parts of the model or recording interface are imprecisely defined. The questions whether or not a changeset should contain the commit message or all changes from the commit, for example. In the following, these flaws are exposed and if possible, a convenient solution is given. This discussion is divided in three parts: model, recording and other flaws. While many problems are covered in this section, not all possible flaws can be discussed and some formal errors might be unrecognized.

6.2.1 Model Flaws

The problems discussed here refer to the provenance model as described in section 4 and originate from the improved provenance model or imprecise specifications.

As mentioned in the introduction to this section, there are inconsistent specification which information is part of the changeset of a commit. [Wen10] describes a changeset in the software development model as "all changes that shall be performed in the repository, including the commit message." Conflicting with this definition, the current recording interface only stores the commit message in the database. A detailed commit message may cover all changes of a commit, but in some cases, this is too vague. For example, question 19 from section 3.1 asks for the changeset of a specific commit. To analyze the problem with this changeset in the query, it is more convenient to have to detailed change information at hand. However, as soon as the corresponding revision number is retrieved, a user can also switch to the VCS to examine the changeset. While provenance system record many information about the process, it should not be an overhead of data mapping all data from all tools. To cope with this problem, the different capabilities of SVN to return the changeset can be used. Currently, the "log" command returns only the commit message. A more detailed view would be given by the "changed" command returning a list of changed files. The most in-depth information is returned by the "diff" command, listing all changes in the files. For future implementations, the "changed" command is probably most convenient. Together with the committing developer, the querying user gets a short list of files which may cause the problem and can report a bug to the developer. At the same time, it does not increase the amount of data much, because it is still a string combining the commit message and a list of files.

Another problem in regards of the representation of commits in the provenance model is the graph structure of repositories. VCS such as SVN or GIT use multiple lines of development: the trunk as the main development path, various branches to implement features separately and tags to save completed versions of the software. This structure is nowhere represented in the current model. All commits are recorded without information which part of the repository is affected by it. Since it has no effect on the incremental revision count whether a commit is performed on a branch or the trunk, the graph structure is flattened to a single line of development. The representation of this structure could happen with various model designs. One solution would be to save the repository path as a property of the revision node. Another, more significant approach would be to model all repository paths as nodes in the provenance graph. This increases the complexity, but in the same time enhances expressiveness of the model. Additional nodes can include a path property and be in relation to the commit activity as in figure 26. This is probably the most convenient solution, because it requires little model changes and can be used to represent actions on the repository itself in the future. For example, if development in a branch is finished and it is merged back into the trunk, a commit combined with a derivation relationship between the two repository nodes would identify the merge more clearly than with a single commit.

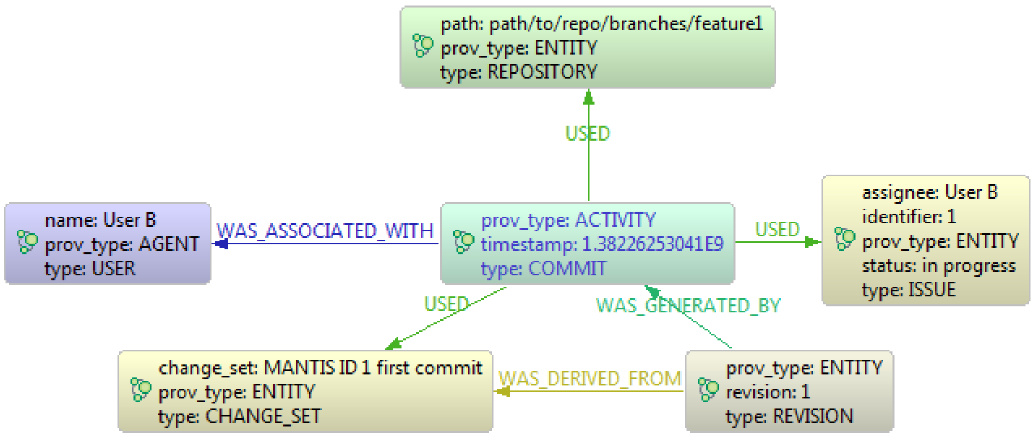

Figure 26: Repository node in the commit process

The next enhancement uses the new relationships of the PROV model. A new relationship called "WasAttributedTo" can be set from an entity to an agent and describes "the ascribing of an entity to an agent" [BBC+12]. This relationship can be used to link an issue with its assigned user (see figure 27) simplifying queries such as for question 1 and 9. It would also be possible to replace the "assignee" property with this relationship.

In the current model, the assignee for an issue is a mandatory property, but an optional one for an issue change as seen in figure 8. While the latter is plausible because the assignee does not change often, the former might be a problem. The identifier and status of an issue are clearly required properties and have to have a value throughout the whole process, but a user is not assigned immediately upon creation. During the iteration and release planning, issues are generated first to identify the relevant features and after that assigned to a user with suitable skills. To fix this problem short-term, it would be sufficient to set an empty

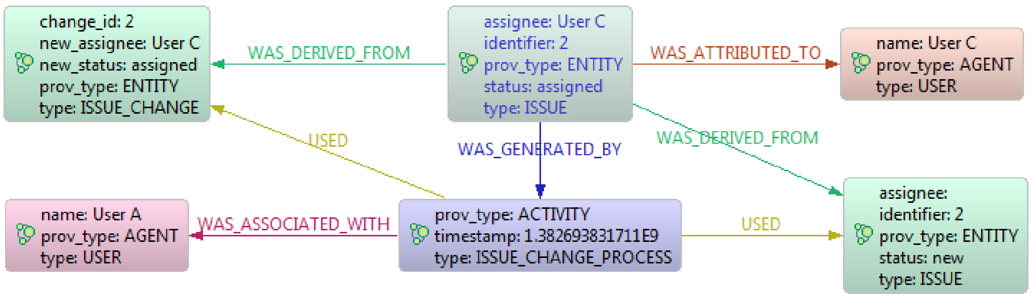

Figure 27: WasAttributedTo-relationship between issue and user

string as an assignee upon issue creation. For future development however, the assignee for an issue should be an optional node property.

The next paragraph describes problems concerning activities in the CI system. Currently, the test and coverage activities are only linked to their results and the revision they used. This can create problems as soon as those activities run on the same revision twice. If a developer writes new test cases and runs them on a revision which was already tested with fewer cases, the provenance graph would include two tests and two coverage reports referring to the same revision. At this point, it is not distinguishable which coverage report belongs to which test. The same applies to the build node, which is also not connected with those two nodes. To improve the model, a relationship between these three nodes would be helpful. The PROV-DM specifies communication between activities as “the exchange of some unspecified entity by two activities, one activity using some entity generated by the other” [BBC+12]. So this relationship describes how activities can exchange data and trigger each other. With this “WasInformedBy”-relationship, the connection between the nodes of the CI system would be established and their dependency to each other displayed. Figure 28 shows the direction of these links. If a user commits new code, the CI system tests the code, so the test activity was informed by the commit. As soon as the test finishes, the coverage report is started, followed by the build process. The communication is not between the coverage node and the build node, because a coverage can be an optional part of the CI, but tests and builds are always part of it. With this modeling approach, it is more obvious, which coverage report belongs to which test and more importantly, the casual connection between commit, test and build are described in the model. This way, an additional agent, who starts every one of these tasks, is not necessary.

Another missing agent problems occurs at the release process. The release is not associated with a user yet and a communication-relationship to any activity is not appropriate, because this process is started by a user. The release process is either performed with a release script or as part of the CI which can start an extra release-build, generates all files and packs them together. If the release is performed on Jenkins, the triggering user can be recorded; a release script would need additional information about the user.

The last model flaw concerns the Maven node for the build process. In the process described

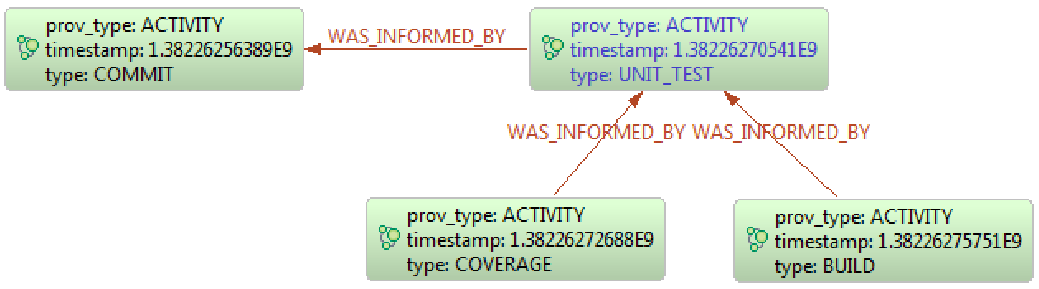

Figure 28: WasInformedBy-relationship in the CI

in chapter 5, Maven takes no active role and can therefore not be associated as a user to the build node. This might change if local builds are recorded, but until now, builds are triggered by the CI system and the aforementioned communication with other activities. A possible solution would be to add the information about Maven to the build node. This way, question 20 could still be answered and the model is more accurate. In addition, by including a Maven node, only one possible build management tool is represented in the model, so other popular tools like Ant or CMake can not be described. If this agent continues to be part of the model, it should be a more abstract representation of build tools to cover multiple programs.

6.2.2 Recording Flaws

While model flaws are problems of the design of the provenance mode, recording flaws with regards to the implementation of this systems. This applies to the recording interface as well as the event system of the incorporated tools to retrieve provenance information.

To include SVN into the recording, one should bear in mind, that one hook script is insufficient and both, the pre- and the post-commit hook should be used. The pre-commit hook usually checks for some specifications, for example the length of a commit message, a specified issue identifier or coding standards. While this covers most sources of possible commit failures, it does not check, if a commit was actually written to the repository. As mentioned in section 3.2.1, pre-commit hooks are executed when the transaction finishes but the new revision is yet to be saved. So if the commit fails to write the revision, this kind of hook would not recognize it. To be on the safe side, a post-commit hook should clarify that a commit was successful and the revision was saved properly. A convenient solution would be to check for failures in regards to the specifications in the pre-commit hook and record commits with a post-commit hook, either successful or failed.

In the current recording interface, successful commits are related to a given issue while failed ones are not. It might be interesting to see not only how many commits a user needed to implement an issue, but how many failed ones, too. The model would allow such a connection, so as long as the commit does not fail because of a missing issue identifier, it would be easy to record the corresponding issue too. The best solution for this would be to add the issue as an optional parameter for PyProvs REST API as can be seen in table 8.

During development, a user may work on an issue and discover, that his implementation also fixes a reported bug for example. It is now possible to include both identifiers in the commit message because RepoGuard does not restrict the number of issues for one commit. This is not considered in the recording interface but not uncommon in practice. A simple solution would be to accept an array of issues in the REST API and set multiple usage relationships to each of these issues.

The next flaw applies to the CI system and the three activities performed by it. Independent from the execution schedule of tests, coverage reports and builds, these three tasks are most likely performed one after another. The event system of Jenkins identifies this as one single task and the notification plugin sends the information of all those three activities at once after the build terminates. The current recording interface got a different function for each of these activities and is not capable of saving them all at once. There are two ways to cope with this problem. On the one hand, the Jenkins plugin could gather all data and make three distinct calls to the REST API. To implement this, either the Notification plugin mentioned in section 3.2.3 has to be customized or a new plugin has to be written. On the other hand, the recording interface could provide an additional API function to save all three activities at once. The most convenient solution is probably an adjustment of the Jenkins plugin, because all tools need to be customized and this would not alter the recording interface which will stay more abstract this way.

The following adjustment reduces the number of nodes and simplifies query 8. Currently, multiple release files are recorded by calling the release function multiple times. For every call, a new file and release process node is stored in the database. While the relationship to the release node clarifies the relation between files and release, multiple activity nodes inflate the database and are technically wrong in terms of the process because only one release occurs. A better solution would be to call the REST API once for the release and commit all release files at once using an array in the JSON data.

6.2.3 Other Flaws

This section deals with flaws that can not be assigned to a model or recording problem, because they concern the incorporated technology or information integrity of the provenance system.

With further development, Neo4J implements some new features. One of the most important are the new *labels* as part of version 2.0. A label can be attached to any node in the database and marks the type of the node. This new functionality allows the user to classify nodes into distinct sets of nodes. As the current model uses the properties “type” and “prov_type” to specify the nodes, labels would provide the same capability while reducing complexity and enhancing performance. Labels do not appear as a property and can be queried like an index, which would clear up nodes and indexes at the same time. Also, Neo Technology probably improves speed with this technique in the future by making sets of nodes manageable for the DBMS.

Another problem of the existing model is information integrity and trust. Provenance data is very sensible data because it includes personal information about the developers as well as the internal development process of the company. Therefore, the model and infrastructure have to ensure that the recorded data is stored unaltered and consistent with the incorporated tools. While pseudonymization of personal data and security mechanism for infrastructure are also open problems, it is more important to assure that first, the recording interface receives and stores the data consistent and that second, the user of this system accesses the correct data. A starting point for the first part of this problem would be the integration of the development environment, most important the IDE of the developer. With the actions of the IDE integrated into the recording process some information can be recorded redundant, for example the changeset of a commit in the IDE and on the VCS system or the issue change on both systems. This can be used to compare both data sets and ensures, that a developer was actually the initiator of e.g. a commit, and that the data was not modified while transmitting. The second part of the problem needs some kind of identification mechanism for the database. Currently, the database contains no information, what project or repositories are observed. As mentioned earlier, a node for the VCS repository would be a way to identify the recorded process, but this can only be a starting point. The connection between provenance data and the actual data must be established in any way so that it can be assured, that a user looks at the correct data. The cryptographic authentication of history in GIT is a good example for this problem. GIT saves every revision with a secure hash which could be included in the provenance graph. With the hash information, a user can identify the repository to which the revision belongs and therefore be sure, that the provenance graph represents the repository.

7 Implementation: Queries

With the software development process as a provenance graph in the Neo4J database, the questions from section 3.1 can be answered using queries on the database. The queries are again divided into two parts, one querying for content within the model and the other querying information about the process itself. [Wen10] implemented the same content questions as part of his thesis using Gremlin as a query language. This thesis uses Cypher instead to give a different example for querying a graph database and more important, to compare both languages in terms of performance. Even though Cypher is currently slower than Gremlin, as layed out in [HP13], it might enhance performance in the next years since Neo Technologies will optimize it for their DBMS.

These code examples do not claim to be the only or best queries for the corresponding question, but they give an overview of the typical usage. Since no real datasets exist at the time of this thesis, the queries are tested on the sample data of section 5.1 and results can not be illustrated. Some characteristics of Cypher are explained in the following.

- To find the starting node of the query, an index search is performed most of the time. The index `Nodes` holds all nodes with their properties including the node type. This allows the user to search for a specific attribute or to start with every node of a given type. The values of an index search have to be given as a string but are converted internally to the correct type.
- Deviating from the cypher example 1 in section 2.3.2, every node in the `MATCH` clause is bracketed.
- Neo4J uses an internal identifier, automatically set for each node upon creation. While this identifier is incremented every time, the DBMS might reuse an identifier after the corresponding node was deleted. Therefore, ordering nodes by their identifier to retrieve the last written one can result in unexpected behavior. Using the time stamp of the activities is one better way to distinguish the chronological order of nodes.
- All keywords of cypher are capitalized to give a better visual distinction, they can also be uncapitalized. Relationship types on the other hand are stored capitalized and therefore must be queried the same way.

7.1 Content Queries

The following queries answer questions about specific values of nodes like a user name or the amount of documentation. They traverse the graph using one or more start nodes and retrieve a value of a property. To start from a node with a specific value, Cypher uses parameters indicated by curly brackets, which allows the language to reuse queries and enhance performance.

1. Who is responsible for implementing issue X? The start node for this query (see listing 6) is the issue with the identifier X. After the index search (line 1), all nodes which used the issue and their corresponding users are collected (line 2). Then, all commit nodes are selected (line 3) and the name of the corresponding user is returned uniquely (line 4).

```
1 START issue=node:Nodes(identifier={id})
2 MATCH (issue)<-[:USED]-(commit)-[:WAS_ASSOCIATED_WITH]->(user)
3 WHERE commit.type = "COMMIT"
4 RETURN DISTINCT user.name
```

Listing 6: Cypher query for question 1

An alternative query to this question is given in listing 7. Instead of finding all users who did a commit on the issue, the assignee-property (line 2) of the issue can simply be returned.

```
1 START issue=node:Nodes(identifier={id})
2 WHERE issue.status="assigned"
3 RETURN issue.assignee
```

Listing 7: Alternative query for question 1

2. What is the current build status of the project? Listing 8 starts with selecting all build result nodes (line 1) and finds their generating build process (line 2). In the last three lines, the build result of the last build is returned, using the time stamp of the build to order the results.

```
1 START builds=node:Nodes(type='BUILD_RESULT')
2 MATCH (builds)-[:WAS_GENERATED_BY]->(build)
3 RETURN builds.build_result
4 ORDER BY build.timestamp DESC
5 LIMIT 1
```

Listing 8: Cypher query for question 2

3. What is the current overall code coverage? As in the previous query, this one returns the last code coverage as can be seen in listing 9. All coverage results are selected (line 1), the outgoing generation-relationship to the process is matched (line 2) and the percentage of the last coverage is returned (line 3-5).

```
1 START coverages=node:Nodes(type="COVERAGE_RESULT")
2 MATCH (coverages)-[:WAS_GENERATED_BY]->(coverage_process)
3 RETURN coverages.percentage
4 ORDER BY coverage_process.timestamp DESC
5 LIMIT 1
```

Listing 9: Cypher query for question 3

4. From which revision was release X built? The query in listing 10 starts with the specified release version (line 1). Then, it searches for a path from the release to the revision (line 2). Note that the connection node in the middle is not specified with an identifier but just with empty brackets. This node, a release file, holds no useful information for the query and can be left out. Because every file is linked with the revision, the query just returns the revision number once (line 3).

```
1 START release=node:Nodes(release={release})
2 MATCH (release)<-[:WAS_DERIVED_FROM]-()-[:WAS_DERIVED_FROM]->(revision)
3 RETURN DISTINCT revision.revision
```

Listing 10: Cypher query for question 4

5. How many unsuccessful commits did user X do? Listing 11 selects the specified user by his name first (line 1) and then matches the path to all nodes generated by him (line 2). The commit failures are extracted (line 3) and the total amount of failures returned (line 4).

```
1 START user=node:Nodes(name={user})
2 MATCH (user)<-[:WAS_ASSOCIATED_WITH]-()<-[:WAS_GENERATED_BY]-(failure)
3 WHERE failure.type = 'COMMIT_FAILURE'
4 RETURN count(failure)
```

Listing 11: Cypher query for question 5

6. How did the number of unit tests change in the last month? This query is more challenging since it includes time measurement. First all tests are selected (line 1) in listing 12. Then all tests from one month ago up until now are collected using the `TIMESTAMP`-function (line 2). This function returns the current time as an unix time stamp and is part of Cyper 2.0, in previous versions, the time stamp must be given as a parameter. Line three and four just sort all tests by their time stamp. The next step is to retrieve the first and last test from this collection (line 5), match their corresponding results (line 6 and 8) and return the difference between successful and failed tests (line 10). Since the nodes are not necessarily returned in order of their node identifier, ordering of the tests is necessary. The `ORDER BY` clause can not be placed after the statement in line five, because the variable 'test' is not available anymore at that point.

```
1 START test=node:Nodes(type="UNIT_TEST")
2 WHERE test.timestamp <= TIMESTAMP() AND test.timestamp >= TIMESTAMP()
      -60*60*24*30
3 WITH test
4 ORDER BY test.timestamp
5 WITH HEAD(COLLECT(test)) as test1, LAST(COLLECT(test)) as test2
6 MATCH (test1)<-[:WAS_GENERATED_BY]-(result1)
7 WITH result1, test2
8 MATCH (test2)<-[:WAS_GENERATED_BY]-(result2)
9 RETURN result1.success - result2.success, result1.failure - result2.failure
```

Listing 12: Cypher query for question 6

7. Which developer is most active in contributing documentation? Starting with all documentation nodes (line 1), every user who has written a piece of documentation is matched (line 2). The total amount of documentation by user is then calculated, returning just the highest count (line 3-5) in listing 13.

```
START docs=node:Nodes(type="DOC_CHANGE_PROCESS")
MATCH (docs)-[:WAS_ASSOCIATED_WITH]->(user)
RETURN count(user) as number_of_documentation, user.name
ORDER BY number_of_documentation DESC
LIMIT 1
```

Listing 13: Cypher query for question 7

8. How many releases have been produced this year? Similar to question six, the query in listing 14 works with time stamps. It returns the amount of releases between now and one year ago using the `TIMESTAMP`-function.

```
START release=node:Nodes(type="RELEASE_PROCESS")
WHERE release.timestamp <= TIMESTAMP() AND release.timestamp >= TIMESTAMP()
     -60*60*24*365
RETURN count(release)
```

Listing 14: Cypher query for question 8

9. How many issues were implemented by developer X for release Y? Listing 15 shows the query for this question. It selects the given release first (line 1) and then finds all issues it was derived from (line 2). In line three, it filters issues with the status resolved and the specified user as an assignee. If the status of an issue is set to resolved, it means the issue is implemented and all necessary work products like documentation and a passed build are given. Other issues might be planned for this release, but did not finish in time. Last, the amount of issues is returned (line 4).

```
1 START release=node:Nodes(release={release})
2 MATCH (release)-[:WAS_DERIVED_FROM]->()<-[:WAS_DERIVED_FROM*]-(issue)
3 WHERE issue.status="resolved" AND issue.assignee={developer}
4 RETURN count(issue)
```

Listing 15: Cypher query for question 9

10. How many commits did developer X contribute to release Y? Similar to the previous query, this one counts all commits by a specified user for a given release. Again, it starts at the release node (line 1) and matches the way to the user via all commits on all issues regarding the release (line 2). In the current recording interface, a release is only linked with the issue where it was specified, most likely the issue with status "new". But an assignment of a user or a commit on the issue will occur on later versions of the issue, so the query must retrieve all issues derived from the one with the relationship to the release. This is indicated by the asterisk after the second derivation-relationship. Since other activities can work with an issue, e.g. documentation, the type is specified and the developer is filtered (line 3). Finally, the amount of commits by this user is returned (line 4).

```
1 START release=node:Nodes(release={release})
2 MATCH (release)-[:WAS_DERIVED_FROM]->()<-[:WAS_DERIVED_FROM*]-(issue)<-[:
     USED]-(commit)-[:WAS_ASSOCIATED_WITH]->(user)
3 WHERE user.name={user} AND commit.type="COMMIT"
4 return count(commit)
```

Listing 16: Cypher query for question 10

11. How many developers contributed to issue X? Contributing to an issue can be done as a change on the issue itself or as a commit or documentation activity. Given the start node via the issue identifier (line 1), two relationships from the issue are matched (line 2). Since the two relationships have a different direction, the arrow is left out in that part of the query as can be seen in listing 17. In the end, the amount of distinct users is returned (line 3).

```
1 START issue=node:Nodes(identifier={id})
2 MATCH (issue)-[:WAS_GENERATED_BY|USED]-()-[:WAS_ASSOCIATED_WITH]->(user)
3 RETURN count(DISTINCT user)
```

Listing 17: Cypher query for question 11

12. Which developers contributed to release X? Like the previous query, contributing to a release takes place by changing an issue, committing code or writing documentation. Selecting the release node first (line 1), all issues and the activities using these issues have to be found. The match clause in listing 18 first finds all issues for the release and then all derived issues as well as the issue change processes on them using multiple relationship types and a variable length (line 2). Next, the subprocesses and their corresponding users are matched and returned uniquely (line 3).

```
1 START release=node:Nodes(release={version})
2 MATCH (release)-[:WAS_DERIVED_FROM]->()-[:WAS_DERIVED_FROM|WAS_GENERATED_BY
     *]-()<-[:USED]-()-[:WAS_ASSOCIATED_WITH]->(user)
3 RETURN DISTINCT user.name
```

Listing 18: Cypher query for question 12

13. How many commits were needed for issue X? The query starts with the specified issue (line 1 in listing 19) and matches to all nodes using this issue (line 2). The commit nodes are filtered (line 3) and the amount of commits returned (line 4).

```
1 START issue=node:Nodes(identifier={id})
2 MATCH (issue)<-[:USED]-(commit)
3 WHERE commit.type="COMMIT"
4 RETURN count(commit)
```

Listing 19: Cypher query for question 13

14. How much time has been spent implementing issue X? Listing 20 begins with selecting all versions of the specified issue (line 1), matches the path to their generating process (line 2) and extracts the ones with status “in progress” or “resolved” (line 3). To distinguish both nodes, they are put in a collection and separated into two nodes (line 3). Finally, the time between both processes is returned in seconds (line 4).

```
1 START issue=node:Nodes(identifier={id})
2 MATCH (issue)-[:WAS_GENERATED_BY]->(process)
3 WHERE issue.status = "in progress" OR issue.status="resolved"
4 WITH HEAD(COLLECT(process)) as process1, LAST(COLLECT(process)) as process2
5 RETURN process2.timestamp - process1.timestamp
```

Listing 20: Cypher query for question 14

15. What documentation belongs to issue X? The query in listing 21 works similar to query thirteen, but instead of the amount of commits, it returns every documentation item on a specific issue.

```
1 START issue=node:Nodes(identifier={id)
2 MATCH (issue)<-[:USED]-()<-[:WAS_GENERATED_BY]-(doc)
3 WHERE doc.type="DOC_CHANGE"
4 RETURN doc.document
```

Listing 21: Cypher query for question 15

16. Which features are part of release X? Starting from the specified release (line 1), the query in listing 22 uses two match clauses, one to match the path from the release to all versions of the related issues and their change process and the other to match the corresponding release process (line 2). Line three first checks, if the issue status is set to “resolved” and if the release process happened after the change to this status. This ensures, that the issue really was part of the release and not just supposed to be. In the last line, the distinct identifiers are returned.

```
1 START release=node:Nodes(release={version})
2 MATCH (release)-[:WAS_DERIVED_FROM]->(new_issue)<-[:WAS_DERIVED_FROM*]-(
      issue)-[:WAS_GENERATED_BY]->(change_process), (release)<-[:
      WAS_DERIVED_FROM]-()-[:WAS_GENERATED_BY]-(release_process)
3 WHERE issue.status = "resolved" AND release_process.timestamp >
      change_process.timestamp
4 RETURN DISTINCT issue.identifier
```

Listing 22: Cypher query for question 16

17. Who is responsible for reducing the code coverage? Contrary to the previous query, the one in listing 23 uses three different start nodes, one revision node and two coverage result nodes (line 1). Three matches are performed in line 2, two from the coverage results to the revision they are derived from and one from the revisions to the committing users. Both coverage matches hold the same paths with the same nodes on them, so a decreased percentage between two coverage reports can now be found (line 3). The two other conditions in the where clause filter every revision which was performed between the reduction of code coverage, because every commit between these two might be responsible. Finally, every associated user is returned once.

```
1 START revision=node:Nodes(type="REVISION"), coverage1=node:Nodes(type="
      COVERAGE_RESULT"), coverage2=node:Nodes(type="COVERAGE_RESULT")
2 MATCH(coverage1)-[:WAS_DERIVED_FROM]->(rev1), (coverage2)-[:WAS_DERIVED_FROM
      ]->(rev2), (revision)-[:WAS_GENERATED_BY]->()-[:WAS_ASSOCIATED_WITH]->(
      user)
3 WHERE coverage1.percentage > coverage2.percentage AND revision.revision>rev1
      .revision AND revision.revision<=rev2.revision
4 RETURN DISTINCT user.name
```

Listing 23: Cypher query for question 17

18. Which requirement causes the most build failures? Listing 24 starts by selecting all build result nodes and then matches the path to the issue. The where clause extracts every result unequal zero and the issue nodes (line 2). Then it returns the issue identifier and the highest amount of build failures for that issue (line 3-5).

```
1 START result=node:Nodes(type="BUILD_RESULT")
2 MATCH (result)-[:WAS_DERIVED_FROM]->(revision)-[:WAS_GENERATED_BY]->(commit)
      -[:USED]->(issue)
3 WHERE result.build_result <> 0 AND issue.type="ISSUE"
4 RETURN issue.identifier, count(result) as failed_builds
5 ORDER BY failed_builds DESC
6 LIMIT 1
```

Listing 24: Cypher query for question 18

19. Which changeset resulted in more failing unit tests? The query in listing 25 begins with three nodes, two test results and one revision (line 1), and works similar to query seventeen. Every revision for every test is matched twice and the changesets of every revision is found (line 2). Then it searches for an increase in failing tests and finds the revisions between this increase (line 3) similar to listing 23. Afterwards, all changesets and revision numbers are returned uniquely.

```
1 START test1=node:Nodes(type="TEST_RESULT"), test2=node:Nodes(type="
      TEST_RESULT"), revision=node:Nodes(type="REVISION")
2 MATCH (test1)-[:WAS_DERIVED_FROM]->(rev1), (test2)-[:WAS_DERIVED_FROM]->(
      rev2), (revision)-[:WAS_DERIVED_FROM]->(change_set)
3 WHERE test1.failure < test2.failure and revision.revision > rev1.revision
      and revision.revision <= rev2.revision
4 RETURN DISTINCT change_set.change_set, revision.revision
```

Listing 25: Cypher query for question 19

20. Which version of maven caused the build failure of revision X? The last query selects the specified revision (line 1) and matches the paths to the builds and their results (line 2) as can be seen in listing 26. Then a check is performed to filter the builds, because the coverage process might use the revision node too, and extract the failing builds (line 3). With those builds (line 4), the path to the maven node is matched (line 5) and returned (line 6).

```
1 START revision=node:Nodes(revision={number})
2 MATCH (revision)<-[:USED]-(build)<-[:WAS_GENERATED_BY]-(result)
3 WHERE has(result.build_result) AND result.build_result <> 0
4 WITH build
5 MATCH (build)-[:WAS_ASSOCIATED_WITH]->(maven)
6 RETURN DISTINCT maven.maven
```

Listing 26: Cypher query for question 20

7.2 Process Queries

The queries presented in this section answer questions about the order of activities in the process or some other specifications. They compare timestamps of activities, specified values or count entities to determine the existence of all required work products.

21. Was the issue status set to "in progress" before any commit was performed? The query in listing 27 starts with all issue nodes (line 1), matches the path to the commits for each issue (line 2) and checks if the issue had any other status than "in progress" (line 3). Every commit on an issue with a different status is a violation against the process and the committing user is returned (line 4).

```
1 START issue=node:Nodes(type="ISSUE")
2 MATCH (issue)<-[:USED]-(commit)-[:WAS_ASSOCIATED_WITH]->(user)
3 WHERE issue.status<>"in progress" AND commit.type="COMMIT"
4 RETURN user.name
```

Listing 27: Cypher query for question 21

22. Were all tasks of the CI executed before a build? Before a build starts, all other activities in the CI should be finished. Listing 28 checks this condition by selecting all build nodes (line 1) and matches the path to the revision and every activity node that worked with the same revision (line 2). The tests and coverage reports are filtered (line 3) and counted (line 4). If the number of tasks is below two (line 5), either a unit test or a coverage report was not performed indicating an error and the corresponding build will be returned (line 6).

```
1 START build=node:Nodes(type="BUILD")
2 MATCH (build)-[:USED]->(rev1)<-[:USED]-(n)ode
3 WHERE (node.type="UNIT_TEST" or node.type="COVERAGE")
4 WITH build, count(node) as ci_tasks
5 WHERE ci_tasks<2
6 RETURN build
```

Listing 28: Cypher query for question 22

23. Was the list of done completed before an issue was set to ”resolved”? The list of done includes a unit test, a successful build and documentation for an issue and it has to be completed before an issue is set to resolved. To check this, the query in listing 29 finds every item on the list separately and counts the occurrence of each. Starting with all resolved issues (line 1), the query first matches the path to the documentation for that issue (line 2 and 3) and counts the documentation change processes (line 4). Then it proceeds in the same manner for successful builds (line 5 to 7) and for the tests without a failing one (line 8 to 10). If any item does not exist, i.e. the count for it is zero (line 11), the issue is returned (line 12) because is should not be set to resolved.

```
1 START resolved=node:Nodes(status="resolved")
2 MATCH (resolved)-[:WAS_DERIVED_FROM]->(issue)<-[:USED]-(doc_change)
3 WHERE issue.type="ISSUE" AND doc_change.type="DOC_CHANGE_PROCESS"
4 WITH resolved, count(doc_change) as doc_count
5 MATCH(resolved)-[:WAS_DERIVED_FROM]->(issue)<-[:USED]-()<-[:WAS_GENERATED_BY
      ]-()<-[:USED]-()<-[:WAS_GENERATED_BY]-(build)
6 WHERE issue.type="ISSUE" AND build.type="BUILD_RESULT"  AND build.
      build_result=0
7 WITH resolved, doc_count, count(build) as build_count
8 MATCH(resolved)-[:WAS_DERIVED_FROM]->(issue)<-[:USED]-()<-[:WAS_GENERATED_BY
      ]-()<-[:USED]-()<-[:WAS_GENERATED_BY]-(test)
9 WHERE issue.type="ISSUE" AND test.type="TEST_RESULT"  AND test.failure<>0
10 WITH resolved, doc_count, build_count, count(test) as test_count
11 WHERE doc_count < 1 OR build_count < 1 OR test_count < 1
12 RETURN resolved.identifier
```

Listing 29: Cypher query for question 23

24. Was the issue status ”resolved” before the release process? This query checks, if all issues planned for a release were resolved before the release was performed. It selects all releases first (line 1 in listing 30) and finds the corresponding process (line 2). With those two nodes, all related issues with their generating process are found (line 4) and a check is performed, if their issue change to resolved happened after the release (line 5). If this condition is true, an issue was changed too late and therefore violates the process definition. The release and the corresponding issue are returned in the last line.

```
1 START release=node:Nodes(type="RELEASE")
2 MATCH (release)<-[:WAS_DERIVED_FROM]-()-[:WAS_GENERATED_BY]->(
      release_process)
3 WITH release, release_process
4 MATCH (release)-[:WAS_DERIVED_FROM]->()<-[:WAS_DERIVED_FROM*]-(issue)-[:
      WAS_GENERATED_BY]->(change_process)
5 WHERE issue.status="resolved" AND release_process.timestamp < change_process
      .timestamp
6 RETURN release, release_process, issue.identifier
```

Listing 30: Cypher query for question 24

25. Was the issue ”closed” after the release process? Like the previous one, the query in listing 31 checks if the issue was closed after the release was performed. It works almost in the same way, except the where clause in line 5 checks for the status “closed” and the if the release process happened after the change process.

```
1 START release=node:Nodes(type="RELEASE")
2 MATCH (release)<-[:WAS_DERIVED_FROM]-()-[:WAS_GENERATED_BY]->(
      release_process)
3 WITH release, release_process
4 MATCH (release)-[:WAS_DERIVED_FROM]->()<-[:WAS_DERIVED_FROM*]-(issue)-[:
      WAS_GENERATED_BY]->(change_process)
5 WHERE issue.status="closed" AND release_process.timestamp > change_process.
      timestamp
6 RETURN release, release_process, issue.identifier
```

Listing 31: Cypher query for question 25

26. Did anyone commit code on the issue except the assignee? Listing 32 starts with all issue nodes (line 1) and matches the path to the commits on the issue and their attributed users (line 2). Then it checks, if a user committed code on an issue without being the assignee of it (line 3) and returns their names (line 4).

```
1 START issue=node:Nodes(type="ISSUE")
2 MATCH (issue)<-[:USED]-(commit)-[:WAS_ASSOCIATED_WITH]-(user)
3 WHERE issue.assignee <> user.name AND commit.type="COMMIT"
4 RETURN user.name
```

Listing 32: Cypher query for question 26

27. Did anyone write documentation except the assignee? Similar to the aforementioned query, this one searches for documentation written by a user who was not the assignee for the issue belonging to the documentation. The only difference in listing 33 is the different node type in the where clause.

```
1 START issue=node:Nodes(type="ISSUE")
2 MATCH (issue)<-[:USED]-(doc)-[:WAS_ASSOCIATED_WITH]-(user)
3 WHERE issue.assignee <> user.name AND doc.type="DOC_CHANGE_PROCESS"
4 RETURN user.name
```

Listing 33: Cypher query for question 27

28. When did the coverage percentage dropped below X? While the CI system warns if the percentage of covered code drops below a specified value, it only saves this information for a short period of time. The query seen in listing 34 retrieves every revision, where the code coverage dropped below a specified percentage. It selects all coverage results first, matches the path to the coverage activity and the corresponding revision (line 2) and performs a check with the given parameter (line 3). If the value is too low, it returns the coverage process and the revision number (line 4).

```
START result=node:Nodes(type="COVERAGE_RESULT")
MATCH (result)-[:WAS_GENERATED_BY]->(coverage)-[:USED]->(rev)
WHERE result.percentage < {X}
RETURN coverage, rev.revision
```

Listing 34: Cypher query for question 28

29. When did the number of failed unit test exceed X? This query works the same way the previous one does, except it works with test results and checks, if the number of failed tests exceed a given value as can be seen in listing 35.

```
START result=node:Nodes(type="TEST_RESULT")
MATCH (result)-[:WAS_GENERATED_BY]->(process)-[:USED]->(rev)
WHERE result.failure > {X}
RETURN process, rev.revision
```

Listing 35: Cypher query for question 29

30. Was there a build not performed on the last revision? The last process query starts with all build and commit nodes (line 1 in listing 36) and matches two paths, one from the build to its used revision and one from the commit to its generated revision (line 2). Then it performs checks on three conditions in line 3: first it compares if the build was performed after the commit, second it checks of the revision generated by the commit has a revision property, i.e. is not a commit failure and third it compares the revision numbers. If the build was performed after the commit but the generated revision number is higher than the one used by the build, it means that the build was not performed on the newest revision. If this case occurs, the build and the two revision numbers are returned in the last line.

```
1 START build=node:Nodes(type="BUILD"), commit=node:Nodes(type="COMMIT")
2 MATCH (build)-[:USED]->(rev1), commit<-[:WAS_GENERATED_BY]-(rev2)
3 WHERE build.timestamp > commit.timestamp AND has(rev2.revision) AND rev2.
      revision > rev1.revision
4 RETURN build, rev1.revision, rev2.revision
```

Listing 36: Cypher query for question 30

8 Conclusion

The final chapter summarizes the outcome of this master thesis and gives an outlook on existing problems and future developments. After an introduction in the working environment, the software development process and provenance in general, the requirements for this thesis were presented. These included recapped and new questions about the process and an updated list of retrievable data from the incorporated tools. Next, an extensive explanation of the redesigned software development model using the PROV data model was given. An example iteration of the process as a provenance graph was examined in the following chapter and SPEM was presented as a meta-model to describe the development process with a formal model. Chapter 6 gave an in-depth analysis of the process with first a list of provenance data produced by the process and last a discussion about current flaws and problems in the model and infrastructure. The second to last chapter re-implemented the stated provenance questions in the graph query language Cypher to give a comparison to the queries previously implemented in Gremlin.

A contribution to the research field of provenance toke place in three different points in this thesis:

- The SPEM is introduced as a formal way to model the software development process. In the future, this standard can be employed as a model to verify the provenance graph automatically.
- With the re-implementation of queries in Cypher, a comparison between the graph query languages Cypher and Gremlin can be performed. While both implementation can be optimized, these queries can be executed on real data and give an overview of the difference in performance of both languages.
- The identified flaws do not only concern the model presented in this thesis, but also the provenance model itself. They can be used as a starting point for future development and improvements of the system.

The conditions of this approach were:

- The modeled process is a specialization of the software development process incorporated at the department SC. Not all possible questions can be answered, but is presents the capabilities of data provenance.
- The recording infrastructure is constrained to a specific set of tools used in the process. However, with enhancements in PyProv, the provenance data can be stored in a more abstract way and more software development tools can be integrated.

The following paragraphs list open issues and give an outlook about future developments.

Model Optimization The provenance model for the software development process presented in this thesis is generated for a specific process used at the department SC. With the listed flaws, the model should be optimized in the future to cope with additional requirements such as enhanced expressiveness, information integrity and security. In addition, the model could be revised and adjusted to be more abstract and thus be applicable to a wider range of development processes.

Automatic Verification Part of the motivation for this thesis was quality assurance and therefore the verification of the process. While it had been shown, how such a process is modeled as a graph and how process information is retrieved with queries, an automatic verification system would be eligible. Warnings about deviations from the process could identify problems with the incorporation of the process and help to perform counteractions as close to the deviation as possible. For this automatic verification, the process has to be modeled formally using for example SPEM, so that a the provenance graph can be benchmarked.

Model Extension It has been shown that the PROV data model can represent a specific set of tools. In the future, more tools need to be introduced to answer more questions about the process and to increase the expressiveness of the model. The extensions range from alternative tools presented in this thesis to novel tools such as the IDE Eclipse. Those additional model elements entail new problems and challenges.

Process Optimization Another purpose of provenance is the optimization of software development processes. With all recorded data, it still needs to be evaluated, how the provenance data supports an optimization. Questions such as "which developer writes the most documentation" are ambiguous and can be interpreted in different ways. A conclusion has to be drawn how the recorded data can be evaluated and interpreted.

Protection of Data Privacy At last, provenance data includes a lot of private data about the developers, which need an access restriction for all unauthorized persons. Laws of the country have to be observed, thus provenance data may not include any personal data in the graph or only in a pseudonymized manner. This also concerns the analysis and evaluation of the recorded information. Provenance data may only be used to validate the development process and not to rate involved developers.

References

[Amb12] Scott W. Ambler. The agile unified process (aup). http://www.ambysoft.com/unifiedprocess/agileUP.html, 2005-2012. Last accessed on October 30, 2013.

[BBC$^+$12] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. Prov-dm: The prov data model. Technical report, W3C, 2012.

[BDG$^+$12] Khalid Belhajjame, Helena Deus, Daniel Garijo, Graham Klyne, Paolo Missier, Stian Soiland-Reyes, and Stephen Zednik. Prov model primer. Technical report, W3C, 2012.

[CSFP02] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2002.

[DBE] Db-engines ranking of graph dbms. http://db-engines.com/en/ranking/graph+dbms. Last accessed on October 30, 2013.

[DLR] Dlr at a glance. http://www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10443/637_read-251/#gallery/8570. Last accessed on October 30, 2013.

[DP06] Allen H. Dutoit and Barbara Paech. Rationale management in software engineering. Handbook of Software Engineering and Knowledge Engineering, 2006.

[GJM$^+$06] Paul Groth, Shen Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An architecture for provenance systems, February 2006.

[Hau07] Peter Haumer. Eclipse process framework composer. Technical report, Eclipse Foundation, 2007. http://www.eclipse.org/epf/general/EPFComposerOverviewPart1.pdf.

[HP13] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM.

[Jen] Jenkins wiki. https://wiki.jenkins-ci.org/display/JENKINS/Home. Last accessed on October 30, 2013.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, J. Softw. Maint. Evol.: Res. Pract.:77–131, 2007.

[Kru04] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[Lev10] Darren Levy. Why is requirements traceability so important? http://www.gatherspace.com/static/requirements_traceability.html, May 2010. Last accessed on October 30, 2013.

[Man] MantisBT Team. *Mantis Manual*. http://www.mantisbt.org/docs/master-1.2.x/en/developers.pdf.

[MCF+11] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011.

[MGM+08] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The provenance of electronic data. *Commun. ACM*, 51(4):52–58, April 2008.

[MMG+06] S. Munroe, S. Miles, P. Groth, S. Jiang, V. Tan, L. Moreau, J. Ibbotson, and J. Vazquez-Salceda. Prime: A methodology for developing provenance-aware applications. Technical report, University of Southampton, 2006.

[Moi] *MoinMoin Documentation*. http://moinmo.in/Documentation.

[Neo12] Neo Technology. Neo4j - the world's leading graph database, 2012. http://neo4j.org/.

[Neo13] Neo Technology. *The Neo4J Manual*. http://docs.neo4j.org, 2.0.0 edition, 2013.

[Ney11] Miriam Ney. proost - wisdom through provenance. http://sourceforge.net/p/proost/wiki/Home/, 2011. Last accessed on October 30, 2013.

[OMG08] OMG. Software & Systems Process Engineering Metamodel Specification (SPEM). Technical report, "OMG", April 2008. http://www.omg.org/spec/SPEM/2.0.

[PRO] Provenance working group. http://www.w3.org/2011/prov/wiki/Main_Page. Last accessed on October 30, 2013.

[Raj11] V. Rajlich. *Software Engineering: The Current Practice*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2011.

[Roy70] Winston W Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, number 8 in 26. Los Angeles, 1970.

[SC] Simulation and software technology. http://www.dlr.de/sc/desktopdefault.aspx/tabid-1185/1634_read-3062/. Last accessed on October 30, 2013.

[Sch] Tobias Schlauch. Repoguard. http://repoguard.tigris.org/. Last accessed on October 30, 2013.

[Sch13a] Tobias Schlauch. Introduction to the software life-cycle at sc. Technical report, Simulation and Software Technology, 2013.

[Sch13b] Tobias Schlauch. Software development tools. Technical report, Simulation and Software Technology, 2013.

[Sch13c] Tobias Schlauch. Software project manual. Technical report, Simulation and

Software Technology, 2013.

[Sch13d] Andreas Schreiber. Increasing software quality using the provenance of software development processes. In *ESA Software Product Assurance Workshop 2013*, 2013.

[Tei13] Clemens Teichmann. Pyprov. `https://github.com/onyame/pyprov`, 2013. Last accessed on October 30, 2013.

[VSS] Distributed systems and component software. `http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1199/1657_read-3066/`. Last accessed on October 30, 2013.

[Wen10] Heinrich Wendel. Using provenance to trace software development processes. Master's thesis, University of Bonn, 2010.