Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN EN BIO-INGENIEURSWETENSCHAPPEN
Departement Computerwetenschappen
Software Languages Lab

Source Code Archeology using Logic Program Queries across Version Repositories

Proefschrift ingediend met het oog op het behalen van de titel Master in de Ingenieurswetenschappen: Computerwetenschappen, door

Reinout Stevens

Juni 2011

Promotor: prof. dr. Wolfgang De Meuter
Begeleiders: dr. Andy Kellens, dr. Coen De Roover

Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
Department of Computer Science
Software Languages Lab

Source Code Archeology using Logic Program Queries across Version Repositories

Dissertation submitted in partial fulfillment of the requirements for the degree of Master in Engineering: Computer Science, by

Reinout Stevens

June 2011

Promotor: prof. dr. Wolfgang De Meuter
Advisors: dr. Andy Kellens, dr. Coen De Roover

©Vrije Universiteit Brussel, all rights reserved.

Abstract

Software engineers need answers to a wide series of questions concerning the source code while developing or maintaining software projects. A question might be finding out all the usages of a method throughout the software system. In the past, tools based on logic meta-programming (LMP) have proven to be successful in answering such questions related to the source code of a system.

Some questions can only be answered by consulting the history of the software project. For example, the question: "Who modified this class the last?", can only be answered by consulting the history. This history is stored inside a version repository, which are an industry best practice.

The research concerning the use of LMP to query multiple versions is limited. In this area the available tools only answer specific questions or are tailored towards a specific goal. To our knowledge, no tools that allow developers to express custom queries across different versions exist.

In this dissertation, we investigate different configurations in the design space of LMP for querying the history of a software project. We consider three different design dimensions for such a tool based on LMP: the program representation, the specification language and the detection mechanism. For example, we experiment with different logic specification languages. The research focuses on the comparison of these different configurations. We bundle these different configurations in the umbrella tool suite called ABSINTHE.

As a validation, we compare the different configurations of our tool by expressing queries that answer questions commonly asked by software developers concerning the history of a software project. These queries illustrate the strengths and weaknesses of each configuration.

Samenvatting

Software-ontwikkelaars hebben nood aan antwoorden op een wijd spectrum van vragen omtrent de broncode gedurende de ontwikkeling en het onderhoud van softwareprojecten. Een mogelijke vraag is het nagaan waar een methode overal gebruikt wordt in het softwaresysteem. In het verleden zijn tools gebaseerd op logisch meta-programeren (LMP) succesvol gebleken om zulke, aan broncode gerelateerde, vragen te beantwoorden.

Sommige vragen kunnen enkel beantwoord worden gebruik makende van de geschiedenis van het software project. Bijvoorbeeld, de vraag: "Wie heeft deze klasse het laatst aangepast?", kan enkel beantwoord worden met behulp van de geschiedenis. Deze geschiedenis is vervat in een version repository. Deze zijn onmisbaar in de ontwikkeling van hedendaagse software projecten.

Het onderzoek van het gebruik van LMP om te redeneren over meerdere versies is zeer beperkt. Hier bestaan slechts tools voor die enkel vooraf gedefinieërde vragen toelaten, of gericht zijn op slechts één taak. Naar ons weten bestaan er geen zo een tools die gebruik maken van de historiek van een softwareproject om vragen te beantwoorden.

In deze thesis onderzoeken we verschillende configuraties in de designspace van LMP voor de geschiedenis van een software project te bevragen. We beschouwen drie verschillende designdimensies voor een op LMP gebaseerde tool, namelijk de programmarepresentatie, de specificatietaal en het detectiemechanisme. We experimenteren bijvoorbeeld met verschillende logische specificatietalen. Het onderzoek focust zich op het vergelijken van deze verscheidene configuraties. We groeperen deze verschillende configuraties in een overkoepelende toolsuite genaamd ABSINTHE.

De validatie van onze tool gebeurt door in de verschillende configuration bevragingen uit te drukken welke vaak gestelde vragen van software-ontwikkelaars over de historiek van een softwareproject beantwoorden. Deze bevragingen duiden de sterktes en zwakten van elke configuratie.

Acknowledgments

Bestest [sic] reader, thou art browsing through the pinnacle of my academic career. Although this says a lot about my work during the past five years, I am still glad to have written this pizzazzy document. I would not have been able to do so without the help of some people, to whom this page is dedicated.

First of all I would like to express my gratitude to Andy and Coen who guided me throughout the past year.

I would like to thank Wolfgang for the promotion of my dissertation, as well as the help during my studies.

I thank the many people who distracted me while working. These are, in no particular order but mostly alphabetical: Bart, Brecht, Eline, Ian, Joeri, Kristof, Laure, Mattias, Reinout, Robbe, Roeland, Sander, Stijn, Tim and Tim.

Last I want to thank the people at the software languages lab, and the coffee machine. The latter fed my daily need for caffeine, chocolate milk, chocolate milk "deluxe" and water that is supposed to taste like soup. I do not dare to imagine what the outcome of this dissertation would be in case their was actual soup.

In case you are not mentioned here, it was completely deliberate and premeditated.

Contents

1 Introduction **13**
- 1.1 Version Repositories . . . 14
- 1.2 Tool Support for Querying Version Repositories . . . 14
- 1.3 Evaluation . . . 16
- 1.4 Overview . . . 16

2 Querying Version Repositories **17**
- 2.1 Questions Developers Ask about Version Repositories . . . 17
- 2.2 Version Repositories . . . 18
- 2.3 Related Work . . . 19
  - 2.3.1 Querying a Single Version . . . 19
  - 2.3.2 Querying Multiple Versions . . . 22
  - 2.3.3 Related Technologies . . . 25
- 2.4 Conclusion . . . 26

3 Overview of the Tool Suite **27**
- 3.1 Program representation . . . 27
- 3.2 Specification Language . . . 28
  - 3.2.1 Kripke Structures . . . 29
  - 3.2.2 Linear Temporal Logic . . . 30
  - 3.2.3 Computation Tree Logic . . . 31
  - 3.2.4 Quantified Regular Path Expressions . . . 32
- 3.3 Detection Mechanism . . . 32
  - 3.3.1 SLD Resolution . . . 33
  - 3.3.2 SLG Resolution . . . 34
- 3.4 Conclusion . . . 35

4 Representing Program Versions **37**
- 4.1 Design Space . . . 37
- 4.2 Meta-model . . . 38
- 4.3 Implementing an Efficient Model . . . 38
- 4.4 The Model in Practice . . . 40
- 4.5 Declarative Interface to the Model . . . 40
- 4.6 Conclusion . . . 40

5 Specification Language **43**
5.1 Soul . . . 44
5.2 Linear Temporal Logic . . . 46
5.2.1 Specification . . . 47
5.2.2 Example Queries . . . 47
5.3 Computation Tree Logic . . . 49
5.3.1 Specification . . . 50
5.3.2 Example Queries . . . 51
5.3.3 Differences between CTL and LTL . . . 54
5.4 Quantified Regular Path Expressions . . . 55
5.4.1 Specification . . . 57
5.4.2 Example Queries . . . 59
5.5 Conclusion . . . 61

6 Detection Mechanism **63**
6.1 SLD Resolution . . . 63
6.2 SLG Resolution . . . 67
6.2.1 SLG Resolution for Positive Programs . . . 67
6.2.2 Negation and Recursion through Negation . . . 67
6.2.3 SLG Resolution using SLD Resolution . . . 69
6.3 Linear Temporal Logic using SLD resolution . . . 72
6.4 Computation Tree Logic using SLG resolution . . . 73
6.5 Quantified Regular Path Expressions using SLG resolution . . 74
6.6 Conclusion . . . 77

7 Evaluation **79**
7.1 Evaluation Strategy . . . 80
7.2 SOUL as Specification Language . . . 80
7.2.1 Example Queries . . . 80
7.2.2 Evaluation . . . 83
7.3 Linear Temporal Logic as Specification Language . . . 84
7.3.1 Example Queries . . . 84
7.3.2 Evaluation . . . 86
7.4 Computation Tree Logic as Specification Language . . . 87
7.4.1 Example Queries . . . 87
7.4.2 Evaluation . . . 89
7.5 Quantified Regular Path Expressions as Specification Language 90
7.5.1 Example Queries . . . 90
7.5.2 Evaluation . . . 93
7.6 Discussion . . . 93
7.6.1 Properties in a Single Version . . . 93
7.6.2 Global Properties . . . 93
7.6.3 Sequence of Properties . . . 94
7.6.4 Overlap between LTL and CTL . . . 95

7.6.5 Complexity 95
7.6.6 Conclusion 96

8 Conclusion and future work **97**
8.1 Summary of the Dissertation 97
8.2 Contributions 99
8.3 Future Work 100
8.3.1 Technical Improvements to the ABSINTHE Prototype . 100
8.3.2 Further Research 100

1 Introduction

Developers need answers to a wide series of questions while developing or maintaining software projects. Some of these questions can easily be answered as they focus on one entity or have little ambiguity [18]. One of these questions is “Where is this method called in the system”. Often, these questions can be answered by using a proper integrated development environment (IDE). The IDE can add extra functionality to the interface, facilitating easy browsing throughout the software project.

There are questions that cannot be answered by examining the code. Consider a developer who tracked a bug down to a method. This method is not written in a straightforward manner, seems quite complicated and has hardly any documentation. Insider information from the developer who contributed most to the method is required to patch the bug. Yet determining who contributed most to a specific code fragment is not easily done with the standard available tools.

Fritz et al. identified a number of questions developers commonly need answers to in their daily work [18]. These include:

- Who has the knowledge to do the code review?
- Who owns this piece of code? Who modified it most?
- What caused this build to break?
- How did the code evolve?

A lot of these questions concern the history of the code. As this information is not easily accessible for the developer, finding the correct answer can be a tedious task.

1.1 Version Repositories

In a company many developers work simultaneously on the same project, each using a different workstation. Some developers are resolving bugs, others are implementing features for the new version while some developers are implementing an experimental algorithm. Managing these different changes to the project cannot be done efficiently by hand.

A version repository manages the different versions of a software project. Developers can commit their changes to the repository, which will create a new version. Every other developer can pull this new version to his workstation. If another developer had already committed other changes the versioning software will merge both versions. This might result in some conflicts if both versions contain the same file with different modifications. The versioning software will try to resolve these conflicts. The common approach is to query the user for the correct merge of all changes.

Developers can also create a new branch of the software, in which they can test experimental features. The rest of the team can continue with the main version, unaffected by those features. In a later stage they can decide to merge the branch with the main version, making the features available for everyone. In case the experiments were not successful they can decide not to merge that branch.

There are two sorts of versioning systems: distributed version control and centralized version control. The distributed model is peer-to-peer like, with each developer having a complete repository on his workstation. Patches can be sent to other repositories. This means that there is no "main repository", only a set of working copies. Examples of the distributed model are Mercurial [2] and Git [1]. The centralized version control stores the repository on the server, and people can get a specific version. Changes are always sent to the server. Examples are Subversion [4] and Monticello [3]. The latter is specific to the Smalltalk programming language.

1.2 Tool Support for Querying Version Repositories

The aforementioned developer questions often concern the evolution of source code. A version repository contains the history of a software project. Querying such a repository is needed to answers non-trivial questions developers ask. Logic program query languages have already proven themselves for querying a single version [33, 22, 27, 11]. We believe that developer questions concerning the evolution of a software project are best answered using a logic program query language as well.

There is therefore a need for a tool that allows developers to reason over the history of a project. We propose to create such a tool called ABSINTHE.

We have the following desiderata:

- Developers should be able to get answers to their questions in a reasonable time. They need the answer to their questions the moment they have the question. When the computation takes several hours it could be easier to find the answer yourself.
- The tool should enable developers to specify queries themselves, rather than being restricted by a set of predefined questions.
- The tool must scale to large projects with hundred different versions.
- Writing down queries should be done in an concise language with clear semantics.

The academic community around "mining software repositories" also focuses on software repositories. Ad-hoc tools are used to detect correlations or trends in the evolution of a software project, but to our knowledge they are unable to answer the developer questions we target.

We consider three different design dimensions for a program query language: the specification language, the program representation and the detection mechanism.

Program Representation The program representation is a model of the history of the project. The history is stored inside a version repository, but cannot be queried by the developer. We require a model that properly scales with the number of versions, both in memory usage and access time.

Specification Language The specification language is the language in which the developers express queries. We require a concise language that allows expressing a wide series of questions.

Detection Mechanism The detection mechanism detects what parts of the model comply with the query. The detection mechanism must be fast and scale with the amount of available entities.

We explore which configurations in the design space of a logic program query language support querying the history of a software project. To this end, we will motivate, implement and evaluate different variants of SOUL. SOUL [33] is a logic programming language similar to Prolog, but with dedicated features for querying versions.

All of these variants share a common representation of the information inside a repository, stored in a model. This model contains every version of the software project. Each version provides us with the necessary information, like the classes, methods and variables. The model provides us with a graph of versions.

These variants differ in their specification language. We investigate what specification languages are suited for expressing queries for developer questions. We investigate normal declarative programming using SOUL, two temporal logics, linear temporal logic (LTL) and computation tree logic (CTL), and quantified regular path expressions.

In a temporal logic, formulas are evaluated against an implicit world. A formula might be true in one world, yet fail in another. The temporal logic provides operators to change this implicit world. We map our versions to these worlds, allowing temporal logic to be used to query the version model. Both LTL and CTL are good representatives of a temporal logic. Although many variations on both exist we are confident that these do not differ enough to change the outcome of our experiments.

Quantified regular path expressions [13] are similar to regular expressions. It reasons over sequences of versions instead of a sequence of characters in a string. The language allows expressing sequences of versions, where certain properties have to hold in each of those versions.

1.3 Evaluation

To evaluate our approach we assess which SOUL variants and their corresponding configuration for the model, specification language and detection mechanism are most suitable for our purposes. We use the questions identified by Fritz et al. [18] as example questions, and examine how we can express these questions with our tool. We also use the tool to answer some questions of our own. We evaluate each configuration whether it supports expressing these questions and evaluate the conciseness of the resulting queries.

1.4 Overview

The rest of this document is structured as followed:

- In chapter 2 we discuss the context and related work
- In chapter 3 we provide an overview of our proposed tool
- In chapter 4 we discuss how we can represent a version repository
- In chapter 5 we discuss the proposed specification languages
- In chapter 6 we discuss the possible detection mechanisms
- In chapter 7 we evaluate the different configurations of the proposed tool
- In chapter 8 we provide a conclusion and discuss future work

2 Querying Version Repositories

Software engineers need answers to a wide range of questions while developing current software projects. While some of these questions are easy to answer using the features of modern IDEs, particular questions require information regarding the history of the system. One example of such a question is "What developers modified the `Logger` class?".

To our knowledge there exists no tool that allows developers to formulate queries that answer these questions. In this chapter we will give an overview of questions developers ask about version repositories. We explain what a version repository exactly is. We will look at the state of the art in tool support for querying version repositories.

2.1 Questions Developers Ask about Version Repositories

Fritz et al. [18] conducted an empirical study, asking eleven professional developers what questions they have while working on a project, for which there is no current tool support. They identified the following categories for the questions: source code, change sets, teams, work items, comments, web/wiki, stack traces and test cases. The study resulted in 46 different questions.

The following questions concern the history and evolution of a software system:

1. Who to assign a code review to? / Who has the knowledge to do the code review?

2. What is the evolution of the code?
3. Who is working on the same classes as I am and for which work item?
4. Who has made changes to my classes?
5. Who is using that API [that I am about to change]?
6. Who created the API [that I am about to change]?
7. Who owns this piece of code? / Who modified it the latest?
8. Who owns this piece of code? / Who modified it most?
9. Who to talk to if you have to work with packages you have not worked with?
10. What classes have been changed?
11. What is the most popular class? [Which class has been changed most?]

Note that many variations of these questions are possible. To answer these questions we need at least all the classes, methods and fields available in each version. Other information like the source code and author (person who committed the version to the repository) should be available as well. This leads to the need for a rich model of the version repository of the software project.

2.2 Version Repositories

A version repository is used to keep track of all the different versions of a software project. Nowadays it is common practice to store a software project inside a version repository. This repository is used to store and undo changes between versions, implement experimental features and share changes between different developers. We will describe the common uses with a scenario.

Brian is given an assignment on the first day of his new employment. He has a examine and fix a simple bug to make him familiar with the project. First of all he needs to get the source code of the project. The whole project is stored in a version repository on the server of the company. Brian checks out the latest version of the project to his workstation. After some time he finds the cause of the bug and fixes it. He commits his changes to the repository, allowing his colleagues to get his changes. But in the mean time someone else also made some changes to the project and published them. The versioning software will try to merge both versions. This works fine for files Brian did not change, but there is one file that both he and someone else changed. The versioning software alerts Brian and presents him with

an interface showing the conflicting merge. Brian merges both changes by hand, and his version is finally stored on the server.

The scenario illustrates that developers can retrieve any version of the software project, add changes to the repository and merge different versions into one version. What we have not discussed is branching. Branching is used to experiment with the project, where the developer is not actually sure his approach will be successful. Instead of mixing the experimental version with the main one the developer branches it. Other developers can check out the main version, unaffected by the experimental changes. Their changes are still added to the normal version, while the developer continues to develop in his own branch. If the approach was successful he can merge the branch back with the main version. In case it was not he can simply revert to the main branch.

Most versioning software can be used to keep track of any sort of file. They do this by just comparing files line-by-line. The advantage is that developers can not only store source code, but also binaries, images etc. in the version repository. The downside is that the version repository has no knowledge of the structure of a file. It will for example solve merge conflict on a line-by-line basis. Examples are Subversion [4], Git [1] and Mercurial [2].

There is versioning software designed for a specific programming language. An example is Monticello [3] for the Smalltalk programming language. Monticello for example has knowledge of the classes and methods. Although this information is used for better merging of versions, Monticello does not offer tools to extract this information.

2.3 Related Work

In the following section we discuss the work related to our research. First, we discuss the available research for querying a single version of a software project. Second, we discuss the available research for querying multiple versions of a software project. Finally, we discuss related technologies.

2.3.1 Querying a Single Version

Logic meta-programming (LMP) is proven to be suited for querying a single version of a software program. It advocates using formulas in an executable logic to specify the characteristics of a pattern. This is done by reifying the program under investigation such that the logic variables can range over the elements of the program. Executing a proof procedure establishes whether certain program elements exhibit the characteristics specified by the formula.

LMP allows queries to be expressed in a declarative style. The advantage is that users can denote the characteristics source code has to exhibit, and the detection mechanism will identify the corresponding source code.

Backtracking allows multiple answers to be returned on demand of the user. The advantage is that not all the answers have to be computed at once. It also provides an easy way to browse returned answers.

There has been extensive research around LMP to query a single version of a software project. In what follows we provide an overview of some of the work done in this domain.

PQL Program Query Language (PQL) [27] is a query language that can be used to detect patterns statically as well as dynamically. A PQL query is a pattern that is matched on the execution trace of the program. A conservative static analysis provides all possible matches. These matches are further refined using dynamic analysis. The static analysis allows that only certain parts of the application have to be monitored. The dynamic analysis will then catch all violations when the program runs.

A user can define actions that are executed when a violation occurs. Such action may be logging or inserting a breakpoint. PQL is mostly used to find application errors and security flaws. Listing 2.1 depicts such a query that enforces that every stream is eventually closed. The example is adapted from [27].

```
1 query forceClose()
2 uses object InputStream in;
3 within _ . _ ();
4 matches {
5   in = new InputStream();
6   ~in.close();
7 }
8 executes in.close();
```

Listing 2.1: Ensuring that every stream is closed using PQL, adapted from [27]

SOUL The Smalltalk Open Unification Language [33], known as SOUL, is one of the earliest logic programming language. It has domain-specific features (e.g. template terms and domain-specific unification) for querying the source code of a program. Template terms allow users to write the characteristics of a pattern through a code excerpt. Without templates this needs to be done using logic queries, which tend to become convoluted. `if jtStatement( st){ ?x = (?type)?e;}` is an example template term. A template term resembles normal program code but logic variables allow the query to remain open.

One of the characteristics of SOUL is that it allows reasoning over the abstract syntax tree (AST). SOUL features symbiosis with Smalltalk. This

allows the usage of the object representing the AST as first-class citizens in queries. This differs from other program query languages that have to reify these objects, for example in a logic fact base.

The domain-specific unification allows specialized unification. Listing 2.2 depicts an example person class written in Java. The user now express a pattern that matches all the accessors for the *age* variable. Imagine for a moment that the name of the field and the operand of the return statement are reified as strings. The unification would unify the shadowed variable used on line 4 with the instance variable. The method on line 3 would not match, as `"age"` does not unify with `"this.age"`.

The domain-specific unification used in SOUL treats reified objects differently. This allows the unification of the two variables on lines 2–3, while the variable on line 4 does not unify. SOUL uses static analyses similar to PQL to gather this information.

```
1 class Person {
2   private Integer age;
3   public Integer getAge() { return this.age; }
4   public Integer notGettingAge(Integer age) { return age; }
5 }
```

Listing 2.2: A Person class with an accessor for the age in Java. The other method does not return the instance variable, as it is shadowed.

CodeQuest CodeQuest [22, 21] allows a developer to write logic queries over Java programs. It does this by creating a fact base of the software system. This fact base contains relations between the different entities, the reading and writing of fields and method invocations. It does not provide submethod information or ASTs. This fact base is quantified over using logic program queries.

Listing 2.3 depicts such a rule, adapted from [22]. This query finds all methods that implement an abstract method.

```
1 implementation_of_abstract(Abstract, Implements ) :-
2   modifier(Abstract, abstract),
3   overrides(Implements, Abstract ),
4   not(modifier(Implements, abstract)).
```

Listing 2.3: Finding a method that implements an abstract method using CodeQuest, adapted from [22]

JTL The Java Tools Language (JTL) [11] is a language to reason over Java code. JTL represents the program under investigation in a relational database. JTL features a terse, but readable specification language. JTL has predicates that allow users to specify queries that superficially resemble the corresponding source code. These queries are then evaluated using Datalog. JTL uses the bytecode representation of the Java program instead of the actual source code.

JQuery JQuery [14] is a plugin for Eclipse, designed for navigating through the source code using the results of program queries. It provides a view that can be customized with logic queries to find the parts of the code that are relevant for the user. These queries are expressed in a Prolog-like language.

Conclusion

The aforementioned languages indicate an interest at querying and reasoning over software projects. This is done either over the AST of the project, or by storing a representation of the project in a logical fact base.

The shortcoming in all these languages is that they are limited to one version of the software system. This makes queries about the evolution of the code impossible, as this information can only be gathered from the history of a software project.

2.3.2 Querying Multiple Versions

Our research focuses on using the history of a software project to answer user-specific questions. In the following subsection we discuss some of the prior work concerning the querying of multiple versions of a software project.

SCQL SCQL [23] is a query language to reason over the evolution of a version repository. To this end, it creates a graph of the repository. Each author, file and revision is a vertex in this graph. Each revision is assigned a timestamp and is connected with the corresponding files and author for that revision. SCQL provides a temporal specification language that allows a user to express relationships as “previous”, “after”, “always”, “never” etc. A user could check whether a file is always modified by the same author. SCQL does not create a model for the source code in each version. This means that the possible queries are coarse-grained.

Table 2.1 depicts a list of primitives of the specification language of SCQL, adapted from [23]. Here we can see that this approach only provides coarse-grained access to the repository. This is also illustrated by listing 2.4, adapted from [23]. This query looks for an author that only modifies files that were already modified by another author.

```
E(changer, Author) {
  E(owner, Author) {
    changer!=owner &&
    A(revision, changer.revisions) {
      A(file, revision.file) {
        Ebefore( r2, file.revisions, revision) {
          isAuthorOf(owner, r2)
        }
      }
    }
  }
}
```

Listing 2.4: Is there an author “changer” who only modifies files that author “owner” has already modified using SCQL? Example is adapted from [23]

Predicate	**Description**
isaMR(ϕ)	is ϕ a Modification Request (MR)?
isaRevision(ϕ)	is ϕ a Revision?
isaFile(ϕ)	is ϕ a File?
isaAuthor(ϕ)	is ϕ an Author?
numberToStr(i)	Represent i as a string.
length(ϕ)	Length of the string ϕ.
substrϕ, k, l)	Return a substring of ϕ of length l at k.
eq(ϕ, θ)	are ϕ and θ equivalent strings?
match(ϕ, θ)	is θ is a substring of ϕ?
isEdge(ϕ, θ)	is there an edge from ϕ to θ?
count(S)	counts the elements in a subset.
isAuthorOf(ψ, ϕ)	is ψ an author of ϕ?
isFileOf(τ, ϕ)	is τ a File of ϕ?
isMROf(ϕ, ϕ)	is ϕ a MR of ϕ?
isRevisionOf (θ, ϕ)	is θ a revision of ϕ?
revBefore(θ, θ_2)	is there a revision path from θ to θ_2?
revAfter(θ, θ_2)	is there a revision path from θ_2 to θ?

Table 2.1: The primitives of the SCQL model, adapted from [23]

```
frequentlyChangedMethod(?method,?history,?threshold) if
  isHistory (?history),
  currentSystemVersionInHistory(?version,?history),
  methodInSystemVersion(?method,?version),
  countall(
    ?vmethod,
    and(anyVersionOfEntity(?vmethod,?method),
        previousVersionOfEntity(?prevmethod,?vmethod),
        methodWasChanged(?vmethod,?prevmethod)) ,
    ?number),
   [?number >= ?threshold]
```

Listing 2.5: Finding frequently changed methods using Time Warp, adapted from [31]

Time Warp Time Warp [31] extends the SOUL language to reason over FAMIX [30] and Hismo [20] models. FAMIX is a language independent meta-model to represent object-oriented source code. Hismo is a history-aware model for object-oriented languages. Time Warp provides predicates that extract information out of the FAMIX model. It provides temporal predicates to reason over temporal relations between Hismo entities.

Listing 2.5 depicts a query that looks for methods that are frequently changed expressed in Time Warp. This example is adapted from [31]. Time Warp uses SOUL as a specification language, with ad-hoc support for temporal queries. This results that each query is evaluated in an explicit version, as can be seen on lines 4–5. Our approach differs in this aspect as we are researching what specification language is suited to express such queries.

Conclusion

There is limited research in querying the history of a software project. SCQL does not consult the source code of each revision to create a model. The smallest entity available in the model is a file. We suspect this level of granularity is too low, limiting the questions that can be answered with this tool.

Time Warp features an idea similar to ours, yet is only a proof-of-concept. The specification language requires an explicit manipulation of the reference version. This is the version where predicates are evaluated in.

Our research focuses on expressive means to describe characteristics of source code elements that are temporally dispersed. We want a tool that can reason over hundreds of different versions of a software system. To our knowledge no such tool exists.

2.3.3 Related Technologies

Our tool suite needs to extract information from a version repository to create a model. The model then provides a graph-like structure that is being used in the queries. In this subsection we describe other technologies that either query graphs or that gather information from version repositories.

Mining Software Repositories

The "mining software repositories" community focuses on spotting trends and correlations in the evolution of a software system. There tools are often limited to just this task, which is reflected by the information those tools extract from the software repository to verify the claim. While lots of approaches exist, we only discuss some representative examples:

- Hindle et al. [24] use the commit messages of each revision to track what each developer is working on. They present a way to visualize the topics developers work on, allowing a team leader to see what their team members are working on. It tracks which topics come and go over time, while other topics could recur.
- Giger et al. [19] track the semantic evolution of a software repository, in combination with a bug tracker, for bug prediction. Instead of using line-by-line comparison they use fine-grained source code changes that contain semantic information about the changes.

The mining software repositories community often uses software repositories as the main source of information. Their approaches differ from ours in that they only focus on one specific task. Their tools are not suited for developers to reason over repositories, or to extract information that is of interest to the developer.

Graph Querying

Model Checking Model checkers enable developers to verify properties of finite state automata. This can be used for software verification if the software can be modelled as a graph of different states. An example can be verifying elevator software, where each elevator always is in one specific state. LTL, CTL and regular path expressions are often used as the specification language for model checking [10, 7].

Dwyer et al. [16] provide a list of common patterns expressed in each specification language. This shows us that each specification language has its own expressive power, although there is a large overlap. This comparison does not readily translate to our setting. For instance, model checkers only determine whether a property holds for a certain state, while we need concrete answers to our questions. More specifically model checkers can use

negation in their queries, while we are reluctant to do so as this does not provide variable bindings.

In general, we are interested in answering other kinds of questions than model checkers. Model checkers learn us that LTL, CTL and regular path expressions are commonly used to query graphs. We will use these specification languages to query a graph of versions of a software system.

Regular Path Expressions Liu et al. [26] present an efficient implementation of extended regular path queries. They state that regular path expressions are not as powerful as other languages, but compensate this by being more convenient and are expressive enough to capture common and important properties.

Drape et al. [15] use regular path expressions to verify if certain compiler optimizations are legal. A compiler can for example replace a variable with its initial value if the variable never changes between the point of initialization and usage. We can express this with a path expression over the flow graph of the program.

2.4 Conclusion

Version repositories contain information that is needed to answer some of the questions software engineers ask while maintaining and developing software projects.There has been plenty of research regarding reasoning and querying over one version of a software project. However, these tools can only be used to answer questions that concern a single version of the program under investigation. To our knowledge there are no tools available that allow developers to query the history of a software project in a concise language. There is therefore a need for our tool.

3 Absinthe: Overview of the Tool Suite

Developers need answers to a wide series of questions while developing current software projects. While some of these questions are easy to answer using the features of modern IDEs, particular questions require information regarding the history of the system. This information resides inside the version repository. We propose a tool called ABSINTHE that allows developers to query these version repositories.

We consider three different dimensions for a tool for querying version repositories. The first dimension is the program representation. This is a model for the version repositories, that contains as much relevant information as possible. The second dimension is the specification language. This is the language in which we express our queries. The last dimension is the detection mechanism. The detection mechanism determines what answers are valid for the query.

In what follows we discuss these three dimensions in detail. For each dimension we list the requirements for our tool. We discuss what configurations we consider for each dimension. Each dimension will be explained in detail in the following chapters.

3.1 Program representation

Most version repositories do not contain the needed semantic information about the contents they store, making it hard to directly launch queries on their data. A solution is to build a model of the version repository that we can query. One of the advantages of this approach is that we have a generic model that can be used independently of the underlying versioning system.

There are several requirements for this model. We want our tool to be able to reason over hundreds of different versions. The model has to contain all the information of these versions. We require a memory usage that is better than creating a complete representation of each version. The reason for this is that the changes between two versions are usually limited to a relatively small set of entities. This means that the rest of the system is unchanged, and information can be reused from the previous version. We also require a fast lookup. A query of a developer will be evaluated in a certain version, and extracting the information of that version from the model should not delay the computation. The model should be generic enough so that it can cope with the differences between the different versioning software available. Yet it should contain enough information so that developers can use it to answer their questions.

One way to implement the program representation is creating a model for each version. As each version contains all the information the access time is very fast. But as already stated this approach will have a large memory footprint, and we do not consider it. Another approach is to only store the changes between each version. This will have the smallest memory usage possible. The access time will be slow, as the model will have to simulate all the changes between the first version and the one it is trying to access.

A third option is to reuse those parts of the model that have not changed between versions, and updating the ones that have. We suspect this will provide a decent trade-off between performance and memory usage. We do not consider the other two options as viable solutions, and do not consider them in our experiments.

We make sure the model can cope with different versioning software. We will use the model to represent repositories stored in SVN, Monticello and the Store from VisualWorks and Pharo (both are Smalltalk IDEs). The latter is used as versioning software for SOUL and contains 180 different versions. We think this is a large enough repository to test our tool.

3.2 Specification Language

The specification language is the language in which the developer expresses queries. The query specifies what parts of the program representation the developer is actually interested in.

Our tool focuses on reasoning over multiple versions. The specification language should provide an easy way to express queries that span multiple versions. A formula might hold in one version, yet fail in another. Evaluating a formula in a specific version should be integrated in the specification language, and is not the responsibility of the developer. The semantics of the specification language should be simple and clear. ABSINTHE is to be used by developers to express queries in a concise way to answer questions

they have while developing software projects.

In this section we focus on the management of the version in which a formula is evaluated. It is also possible to evaluate complex queries in just one version. SOUL has rich features that allow a developer to reason over a specific version. We will not focus on these features.

Managing in what version a formula is evaluated can be done in several ways. An ad-hoc manner is that developers create rules that take a version as an argument, and are responsible for managing the version in which rules are evaluated. Although this approach suffices for some repository queries, we opted for one that results in more descriptive and expressive specifications. There are several logic formalisms in which the "current world" is implicit. Implicit means that the developer can use predicates without specifying a version. They are automatically evaluated against this implicit world. The developer can use operators which transition between different worlds.

These logics are called temporal logics. Plenty of different temporal logics have been proposed. We suspect that each logic will have some queries that are easily expressed in it, while others are harder. To test this we have opted for three different formalisms, namely linear temporal logic (LTL), computation tree logic (CTL) and quantified regular path expressions (QRPE). The latter is not a temporal logic, but it also provides this implicit world.

In the following sections we explain each formalism we have implemented. In chapter 5 we have an in-depth comparison of each formalism. In what follows we provide an overview of each formalism we consider.

We begin by providing some information on Kripke Structures, a graph-like structure that is used in both LTL and CTL.

3.2.1 Kripke Structures

A Kripke structure is defined as a 4-tuple:

$$M = (S, I, R, L)$$

S A finite set of states

I A set of initial states $I \subseteq S$

R A transition relation $R \subseteq S \times S$ where $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$

L A labeling function

Informally we have a graph, where each node is at least connected with one node. In each node some properties hold, designated by the labeling function. This graph also has some initial states. Figure 3.1 depicts the following Kripke structure:

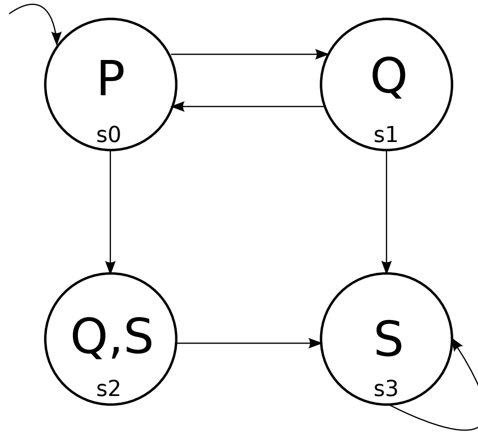


Figure 3.1: A Kripke Structure

$$
\begin{aligned}
S &= \{s_0, s_1, s_2, s_3\} \\
I &= \{s_0\} \\
R &= \{\{s_0, s_1\}, \{s_1, s_0\}, \{s_0, s_2\}, \{s_1, s_3\}, \{s_2, s_3\}\} \\
L &= \{\{s_0, \{P\}\}, \{s_1, \{Q\}\}, \{s_2, \{Q, S\}\}, \{s_3, \{S\}\}\}
\end{aligned}
$$

Kripke structures are used in linear temporal logic and computation tree logic. Kripke structures can be easily mapped to our version model, with the exception that we do not require a transition in each state. In a Kripke structure this means that every path has an infinite length, as there is no end state. In our model we have states without transitions. If we want to strictly comply with the definition we could add a transition from each end state to itself, but this would result in counterintuitive solutions for our repository queries.

3.2.2 Linear Temporal Logic

Linear temporal logic is a temporal logic in which time is regarded as a sequence of states. LTL reasons about infinite paths throughout the Kripke structure. A path π is a sequence of states $s_0, s_1, s_2, \ldots$ where for each i holds that $\{s_i, s_{i+1}\}$ is $\in R$ of the Kripke structure. In our program representation the paths we will reason about are finite, as there are no cycles and a finite number of versions. We say that for a given state s_i a LTL formula holds if there exists a path starting in s_i for which the formula holds. The detection mechanism will try the formula for every path starting in s_i to detect where it holds.

Note that we could also say that the formula has to hold for every path starting in that state. Yet these two are duals of each other: if a formula

```
1 ?first isOldestVersion,
2 ?path isPathStartingFrom: ?first where:
3   globally(?class isClass)
```

Listing 3.1: LTL Querying: What class has always existed on one path?

has to hold on every path then it means the same as ¬formula cannot hold on some path and vice versa.

To illustrate LTL we use it to answer the question: “What class has always existed on one path?”. Listing 3.1 depicts the query for this question. We first find the oldest version of the software history. We are only interested in a class that has existed on one path. The *isPathStartingFrom:where:* predicate will look at every path starting at the provided version, and returns the solutions one by one. The *globally* predicate says the formula has to hold on every version on the path. Our formula here is `?class isClass`. In case the variable *?class* is unbound it will be bound to a class in the current version. In case *?class* already is bound the predicate will verify that *?class* exists in the current version. `?class isClass` has an optional version argument. When this argument is not provided by the user it will be assigned to the current version. We use this technique for most of the predicates. The complete workings will become clear after explaining the detection mechanism in 3.3.

3.2.3 Computation Tree Logic

Linear temporal logic allows checking whether a formula holds for one path. Computation tree logic verifies if a formula holds for one state. The combinators are similar to LTL, but have an obligatory quantifier. This quantifier either specifies that the combinator has to hold for one of its successors, or for every successor of that state. This adds the power to reason over branches, where LTL has no such expressivity.

We use CTL to answer the same question as we answered with LTL: “What class has always existed on one path?”. Code 3.2 shows the answer to that question. We first find the start version, or the root of the version history. We then try for each class if that class exists in the root and each successor of the root version. This query is almost similar to the query we had in LTL. There is an overlap in expressivity in CTL and LTL, but there are queries that can be expressed in one but not in the other. We will discuss these in 5.3.3.

```
1 ?first isOldestVersion,
2 ?class isClass,
3 eGlobally(?class isClass) succeedsIn: ?first
```

Listing 3.2: CTL Querying: What class has always existed on one path?

3.2.4 Quantified Regular Path Expressions

LTL and CTL are both temporal logics, where LTL is used to verify a formula on a path, and CTL to verify if a formula holds in a specific state. The reasoning engine will look for a state or path where the formula holds. Expressing sequences of versions that each have to satisfy a certain property is much harder. To tackle these kinds of questions we have implemented quantified regular path expressions [13]. Quantified regular path expressions are similar to regular expressions [6]. The regular expression engine detects whether a string matches a certain pattern by consuming characters. The reasoning engine will verify if a pattern matches a sequence of versions.

Similar to CTL we have a quantifier for our pattern: this quantifier can be an existential one (denoted by e) or a universal one (denoted by a). The existential quantifier denotes that the pattern has to hold for one path between the start and end world. The universal one denotes that the pattern has to hold on all paths.

We will answer a similar question to CTL and LTL: "What classes always exist on a path between the first and the last world?". Note that in LTL and CTL the end world was not specified. Listing 3.3 depicts the query for the question. We begin with taking the root of the project and the latest version. We now launch an existential query, denoted by the e. We require that the path exists out of many versions, for which *?class* exists. As we want this to be the same class for every version we denote that this variable needs be stored across different versions. This is done by the list `<?class>`. In case we did not add that variable it would become local to one version, and may be assigned different values.

3.3 Detection Mechanism

The detection mechanism identifies which part of the model exhibits the characteristics specified by the query. SOUL is a logic programming language. In a logic programming language programmers write down rules. A rule consists out of a *conclusion* (also known as *head*) and zero or more *conditions*. A condition has a *functor*, which act as a sort of name. The functor may be followed by some variables. In case it is not the condition is known as an *atom*. Listing 3.4 depicts a rule with two conditions. The declarative

```
1 ?first isOldestVersion,
2 ?last isMostRecentVersion,
3 "the manyOf operator is equivalent
4 to the * operator in regular expressions"
5 e(manyOf(?class isClass)) matches:
6   graph(versionTrans, ?first, ?last, <?class>)
```

Listing 3.3: Path Queries: What classes always exist on a path between the first and last world

```
1 functorName(?var1, ?var2) if
2   conditionA,
3   conditionB(?var1, ?var2)
```

Listing 3.4: A SOUL rule

meaning of a rule is that the head of a rule is a logical consequence of its conditions.

The detection mechanism will be used to implement the specification languages. We require a mechanism that allows these specification languages to be easily implemented. We consider two different detection mechanisms. SOUL features selective definite linear clause resolution (SLD resolution). It is the proof procedure of most declarative languages like Prolog. We will also look at SLG resolution, which can also be used for declarative languages, but has different workings.

3.3.1 SLD Resolution

Selective definite linear clause resolution is a proof procedure. A proof procedure is a way to determine whether a given query holds for a certain model. To refute a goal SLD resolution will unify the head of the goal with all the available rules. Unifying means that both functors have to be equal, and that each variable can be unified. Two atoms unify if they are equal. In case there is a rule that unifies SLD resolution will replace the goal with the conditions of that rule. SLD resolution will then proof these conditions from the left to the right in the same way.

It is possible that there is more than one rule that unifies. In this case SLD resolution will add a *choice point*. The first rule will be tried first. In case it fails, or the user asks for more solutions, SLD resolution will *backtrack* to the last choice point and try a different rule. This may lead to more or different solutions.

The order in which the conditions are specified is not important for the declarative meaning, but does change the procedural interpretation. The conditions are proofed from left to right. In case SLD resolution encounters an infinite recursion in one of the conditions it will never return an answer. It may be possible that one of the conditions after the infinite loop fails, meaning if the programmer placed it before the infinite loop SLD resolution could have proofed the rule failed. The procedural semantics from SLD resolution differ from the declarative ones, and it is the responsibility of the programmer to know the difference.

3.3.2 SLG Resolution

There are several disadvantages to SLD resolution [28]. The first is that it might not terminate due to infinite recursion. The second is that it may repeatedly evaluate the same literal in a rule body, decreasing performance.

A solution to these problems is using "linear resolution with selection function for general logic programs", commonly known as SLG resolution. SLG resolution will store previously computed answers and will detect the recomputation of rules we are already proving.

The proof strategy for SLG resolution is similar to SLD resolution. It differs that when it tries to proof a rule it will first see if it is already proving the same rule. In case it would start proving it again it would result in an infinite computation. The strategy is to suspend that proof and backtrack to the last choice point. By selecting other rules it may discover other answers for the delayed rule. These answers can then be used to continue the delayed computation. This means that the order of the conditions of a rule is no longer important for the termination of the computation. It can still affect the efficiency. This results in a more declarative approach than with SLD resolution.

Answers of rules will be stored, preventing the need to reproof these rules. Instead the answers will just be returned.

We have several reasons to use SLG resolution. First of all several of the specification languages and their accompanying detection mechanism we will be using can easily be implemented using SLG resolution. A second motivation is that we suspect that we can reuse plenty of computations across versions. The number of changes between two versions is often limited to several classes, unaffecting the rest of the system. SLG resolution provides opportunities to reuse the computation in one version in several other versions, as long as the query uses entities that remained unchanged between these versions.

3.4 Conclusion

Developers have a lot of questions concerning the evolution of the project. The information about the history of a project is stored in a version repository. Repositories do not have any semantics about the contents they are storing. This means that the developer has to browse through the different versions himself if he needs to extract information. As this is a very inefficient approach, we propose a tool that allows developers to query these repositories.

First we create a representation of the version repository. This model contains all the different versions of the project, and adds semantics. It contains the different classes, methods, fields etc. for each version. The model abstracts away from the differences between the different versioning software that is currently available.

We then create a specification language in which we can express our queries. We explore integrating three formalisms for querying graphs in a logic program query language. Linear temporal logic an computation tree logic are two formalisms that are a decent representative of temporal logics. We also implemented quantified regular path expressions, in which developers can write down paths and the characteristics the versions on this path need to exhibit.

Finally we use SLG resolution to detect the correct answers for our queries. SLG resolution has several advantages over SLD resolution. Both CTL and quantified regular path expressions can be easily implemented using SLG resolution. We also suspect that SLG resolution can be used to reuse answers across different versions. SLG resolution is implemented as an extension of the SOUL programming language, which already features SLD resolution and facilities for reasoning about code.

4 ABSINTHE: Representing Program Versions

Software engineers need to answer a wide series of questions concerning the history of a software project. It is common practice to use versioning software for software projects. This means that the evolution of a software project resides in a version repository. Developers cannot query this version repository to retrieve the needed information to answer their questions concerning the history of the project.

Our tool will create a representation of such a version repository. This approach results in a uniform way to retrieve the information, independent of the versioning software used for the software project. In what follows we discuss several options to implement such a representation.

4.1 Design Space

The model needs a way to represent all the information available for each version in the repository. One way to do this is to create a model for each version. Hismo [20] is a history-aware meta-model for object-oriented languages that uses this approach. The advantage is that all the information is quickly available for a given version. The drawback is that this requires a lot of memory.

The amount of changed entities between two versions is often limited. The model could create a complete representation for the first version, and only store the changes for all successive versions. Praxis [8] uses this approach. This approach has the lowest memory usage, as only the minimum of needed information is stored. The drawback is that accessing information

in a version goes slower. The model needs to simulate all the changes from the first version up to the queried version to get the requested entities.

A third alternative is a hybrid model, that tries to find a balance between both extremes. By storing some extra information the model tries to compensate the slow access times, but still using less memory than a complete representation for each version. Our model uses this approach, which is based on Orion [25].

We re-use a model that already was available at the software languages lab where we did our research.

4.2 Meta-model

The model is the only way a user can retrieve information from the software project. This means that entities that cannot be stored in the model are inaccessible for the user, even when the entity is available in the version repository. To ensure that the model captures the subset of information found in the available versioning software, it provides support for different languages and versioning software. There is support for Subversion [4], Monticello [3] and the Store, a version repository used in VisualWorks [5]. At the moment there is support for both Java and Smalltalk, but this can easily be extended to other programming languages.

The model has several abstract entities. By subclassing these entities the model provides an easily extensible infrastructure to store all the information. This is needed as not all programming languages have the same constructs.

The model stores all the classes, packages and interfaces available. For each class the model also stores the available methods and fields. The model stores the relations between classes. Finally it stores what fields are referenced by each method.

SOUL provides facilities to reason over source code. Storing the source code directly in the model requires too much memory. The model provides a way to retrieve the source code by re-reading it from the repository and providing it to the user.

4.3 Implementing an Efficient Model

In this section we give a brief overview how the model achieves the trade-off between memory and speed. The actual design is not part of the scope of our research, and we do not discuss it here in detail.

Each entity is given a unique identifier. That identifier is used as a handle whenever an entity needs to refer to another entity. This identifier is unchanged over versions, even when the entity did change.

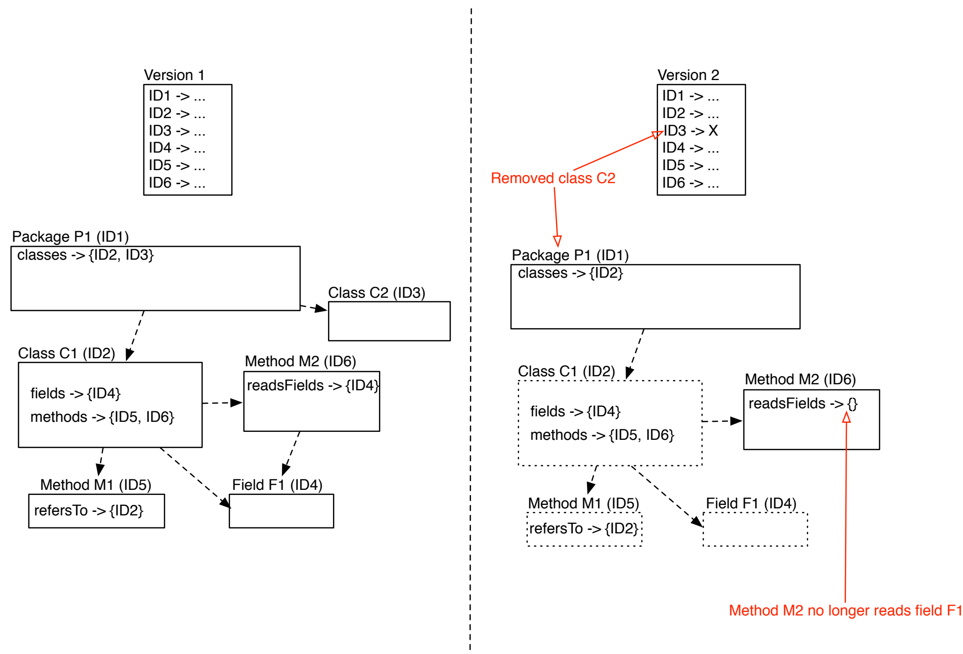


Figure 4.1: An overview how the model evolves between two different versions. A removed method is designated with a cross. Unchanged entities are designed with a dotted line.

Each version keeps track of all the identifiers and the corresponding entities. This way the model can find the correct entity for a given version. In case an entity did not change between two versions the new version just stores a reference to the old entity. A version still stores an identifier of a removed entity, but does not associates anything with it. We also provide an equality that is based on this identifier. This ensures that the an entity remains equal to itself across different versions.

All the methods of an entity are parametrized with a version argument. Instead of always specifying this version the model employs a different strategy. Versions do not return an entity, but rather a wrapper around an entity. This wrapper allows the user to query the entity without specifying the version. The wrapper delegates the query to the entity, but with the version as an argument. This is the same version that was used to retrieve the wrapper.

Figure 4.1 depicts how our model changes when a new version is loaded. Entities that are removed still have their identifier inside the table, but with no matching entity. Unchanged entities, denoted with dotted lines, are reused from the previous version.

4.4 The Model in Practice

We test our model by importing both SOUL and IntensiVE. SOUL is the programming language we use and extend to create our tool suite. Importing SOUL actually allows us to reason over the evolution of our own tool suite. IntensiVE is a tool that allows software architects and developers to monitor the internal quality of software development projects.

Table 4.1 depicts information about both projects, and the memory usage after importing the project.

	IntensiVE	**Soul**
versions	175	137
classes	42.730	26.447
changed classes	5301	3380
methods	414.745	302.103
changed methods	6483	4144
LOC	1.903.215	1.727.777
avg #classes	244	193
avg #methods	2369	2205
avg LOC	10875	12611
memory use	79,9 Mb	30 Mb

Table 4.1: Results of using our tool for both IntensiVE and Soul

4.5 Declarative Interface to the Model

The model is written in Smalltalk. SOUL features symbiosis with Smalltalk, allowing Smalltalk expressions to be used as conditions and for unification with other terms. We provide a declarative interface that queries the model. This interface will use the symbiosis with Smalltalk to extract the information, and allows a declarative programming style throughout the rest of the program.

Table 4.2 depicts a small overview of the interface. There are many other predicates available, similar in naming convention and usage.

4.6 Conclusion

We create a representation of a version repository. This model provides a uniform way to retrieve information from a repository, independently of the version software used for the project. This model needs to find a balance between access speed and memory usage. It does this by reusing unaffected entities from previous versions. We provide a declarative interface to this model so we can easily use it in the rest of the tool.

Predicate	Description
Versions	
?v isVersion	Entity is a version
?v isOldestVersion	Entity is the oldest version in the model
?v isMostRecentVersion	Entity is the latest version in the model
?v isVersionAtDate: ?d	Find the version at a particular date
?v isVersionBetweenDates: ?start and: ?end	Find all versions during a particular time interval
?d isTimestampOfVersion: ?v	Retrieve the time stamp of a version
?a isAuthorOfVersion: ?v	Retrieve the author of a version
Basic entities	
?c isClass : ?version	Entity is a class in a particular version
?c isClassInPackage: ?p : ?version	Class belongs to package in a particular version
?c isClassWithName: ?n : ?version	Class in version has name
?i isInterface : ?version	Entity is an interface in a particular version
?i isInterfaceInPackage: ?p : ?version	Interface belongs to package in a particular version
?i isInterfaceWithName: ?n : ?version	Interface in version has name
?m isMethod : ?version	Entity is a method
?m isClassMethod : ?version	Entity is a class (static) method
?m isMethodInClass: ?c : ?version	Method belongs to class
?m isMethodWithName: ?n inClass: ?c : ?version	Method with particular name in class
?p isPackage : ?version	Entity is a package in a particular version
?p isPackageWithName: ?n : ?version	Package with name
?v isInstanceVariableWithName: ?n inClass: ?c : ?version	Entity is instvar (field) with name in class
?v isClassVariableWithName: ?n inClass: ?c : ?version	Entity is classvar (static field) with name in class
?e isParseTreeOf: ?x : ?version	Retrieve the original AST-node from the repository for an entity
Entity relations	
?c classInHierarchyOf: ?super : ?version	Class belongs to hierarchy of superclass in a particular version
?c isSubclassOf: ?super : ?version	Class is a direct subclass of superclass
?c isSuperclassOf: ?sub : ?version	Class is a direct superclass of a subclass
?c classImplementsInterface: ?i : ?version	Class implements a particular interface
?i interfaceIsImplementedBy: ?c : ?version	Interface is implemented by a particular class in a version
?i isSubinterfaceOf: ?super : ?version	Interface is a subinterface of a particular interface
?m methodReferencesClass: ?c : ?version	Method refers to a particular class
?m methodSendsMessage: ?msg : ?version	Method sends a particular message
?m methodReads: ?v : ?version	Method reads a variable in a particular version
?m methodWrites:?v : ?version	Method writes to a variable in a particular version
?e wasChanged : ?version	Entity was altered in a particular version

Table 4.2: Excerpt from our library of logic predicates.

5
Absinthe: Specification Language

In the previous chapter we discussed how we created a model of a version repository. The model provides a way to query this version repository. The specification language is the language in which users of our tool express their queries. The specification language must make it possible to easily query both specific versions, as well as queries that span multiple versions.

In what follows we discuss the possible configurations for the specification language. The first specification language we consider is SOUL, a declarative programming language. We also discuss two temporal logics, namely linear temporal logic and computation tree logic. Finally, we discuss quantified regular path expressions.

We use the same running examples for each specification language. We explain the examples here.

E1 In example 1 we are interested in classes that have always existed in the software project.

E2 In example 2 we are interested in what version a class got introduced. We say a class is introduced in a version if the previous version did not contain that class. This works for every version except the first one, as it has no predecessor.

E3 In example 3 we are interested in an author who committed a new version 10 minutes after a previous committed version.

E4 In example 4 we are interested in two authors who committed at least two times after the other author in a row. This might indicate that those two authors are working as a team.

```
1 ?version isVersion,
2 forall(?version isVersion, ?class isClass : ?version)
```

Listing 5.1: What class has always existed using SOUL

5.1 Soul

Our tool is written in SOUL. This is a declarative programming language with a syntax similar to Smalltalk. SOUL uses SLD resolution as a proof strategy. We strongly suggest readers that are unfamiliar with SLD resolution to read chapter 6 first. All the variables are prefixed with a question mark.

It features symbiosis with Smalltalk. This means that Smalltalk terms can be used as a condition or for unification with other terms. It also allows us to easily query the model which is written in Smalltalk.

This is done by adding predicates that extract entities from a specific version of the model. We give a short overview of some of these predicates.

> `?class isClass : ?version` will unify the *?class* variable with all the available classes in *?version*.
>
> `?method isMethod : ?version` will unify the *?method* variable with all the available methods in *?version*. Similar predicates exists to further specify the methods. For example `?method isMethodInClass: ?class : ?version` retrieves all the methods from a specific class in a specific version.
>
> `?author isAuthorOf : ?version` will unify the *?author* variable with the author of *?version*.

Developers can also use SOUL as a specification language for our tool. Declarative programming languages are very well suited for reasoning about models. The user can use aforementioned predicates to specify what parts of the model he is interested in. Listing 5.1 depicts the query for E1. Ensuring that a class exists in every version is done using the *forall* predicate. Due to the working of *forall* this query does not return any bindings for *?class*. Listing 5.2 depicts a query which does. This is done by binding *?class* before using the *forall* predicate.

Specifying in what version a predicate has to hold is an annoyance. Most of the time a user first looks for the version, and uses that version in the following predicates. To solve this issue we implemented the predicates using annotated terms. An annotated term has an annotation, which is an optional argument. The predicates reasoning over versions all have an annotation,

```
1 ?version isVersion,
2 ?class isClass : ?version,
3 forall(?version isVersion, ?class isClass : ?version)
```

Listing 5.2: What class has always existed using SOUL, also returning variable bindings

```
1 ?version isVersion,
2 ?class isClass,
3 forall(?version isVersion, ?class isClass)
```

Listing 5.3: What class has always existed using SOUL with an implicit version

which is the version the predicate is evaluated in. In case the user does not provide a version the reference version will be used. The reference version is set whenever one of the predicates that look for a version, like *isVersion* or *nextVersionOf*, are used. This means that the predicate will be evaluated in the last version used in the query. The user can still specify the version by providing the annotation. Listing 5.3 depicts the query for E1, rewritten with the implicit version argument.

Listing 5.4 depicts the query for E2. Note that we first have to find a version where the class is available. This ensures that the *?class* variable is bound. If *?class* is unbound *not(?class isClass)* would mean that we are looking for a version where there are no classes at all.

Listing 5.5 depicts the query for E3. We begin by looking for the timestamp and author of a version. We then set the implicit version to one of the immediate successors. We require that the next version has the same author by reusing the variable *?author*. Finally we require that both timestamps are within 10 minutes of each other. SLD resolution will use backtracking to look for versions that exhibit these properties.

```
1 ?version isVersion,
2 ?class isClass,
3 ?prev isPreviousVersionOf: ?version,
4 not(?class isClass)
```

Listing 5.4: In what version a class got introduced using SOUL

```
?limit equals: [ 10 minutes ],
?version isVersion,
?author isAuthorOfVersion,
?timeA isTimestampOfVersion,
"set reference version to next version"
?next isNextVersionOf: ?version,
?author isAuthorOfVersion,
?timeB isTimestampOfVersion,
?timeA isWithin: ?limit ofTime: ?timeB
```

Listing 5.5: Which author committed a version 10 minutes after a previous commit using SOUL

```
?first isVersion,
?authorA isAuthorOfVersion,
?second isNextVersionOf: ?first,
?authorB isAuthorOfVersion,
not(?authorA equals: ?authorB),
?third isNextVersionOf: ?second,
?authorA isAuthorOfVersion,
?fourth isNextVersionOf: ?third,
?authorB isAuthorOfVersion
```

Listing 5.6: Which two authors committed two times in a row after each other using SOUL

Listing 5.6 depicts the query for E4. In this example the repetition is done manually. A better way would be to create a specific predicate that does the recursion, allowing an arbitrary number of repetitions.

5.2 Linear Temporal Logic

Linear temporal logic is a temporal logic that reasons over paths in a Kripke structure (see 3.2.1). We will use it to reason over paths in the representation of the version history. The representation is similar to a Kripke structure, except that not each version needs to have a successor. This results in the absence of cycles and infinite paths in our representation.

A path is a sequence of states $s_0, s_1, \ldots, s_n$ where s_i is connected with s_{i+1}. We say an LTL formula holds in a version if it holds for at least one path starting in that version. Another option is saying it has to hold for all the paths starting in that version. It is possible to express the latter approach using the first, by saying an expression holds on every path in case there exists no path where the expression does not hold. We can express

the first in terms of the latter in a similar way.

5.2.1 Specification

In LTL, there are five different temporal operators. An LTL formula is created by nesting these operators. We briefly explain and illustrate each operator. In our illustrations we will use the following conventions: versions coloured in blue are versions where ϕ holds, nodes coloured in green are nodes where ψ holds. ϕ and ψ are both expressions. These could be simple expressions, but also complex new LTL expressions. We use the following notation: π_i denotes the i'th states of the path. $\pi_i \models \phi$ means that ϕ holds in π_i. For every example we start at π_i. We will use these conventions in the rest of this document.

next(ϕ) The *next* combinator means that $\pi_{i+1} \models \phi$. Informally this means that ϕ has to hold in the next version. Figure 5.1a illustrates this.

globally(ϕ) The *globally* combinator means that $\forall j \geq i, \pi_j \models \phi$. Informally this means that ϕ has to hold for every subsequent state, including the current state, along the path. Figure 5.1b illustrates this.

finally(ϕ) The *finally* combinator means that $\exists j \geq k, \pi_j \models \phi$. Informally this means that ϕ eventually has to hold somewhere along the path, possibly in the current state. Figure 5.1c illustrates this.

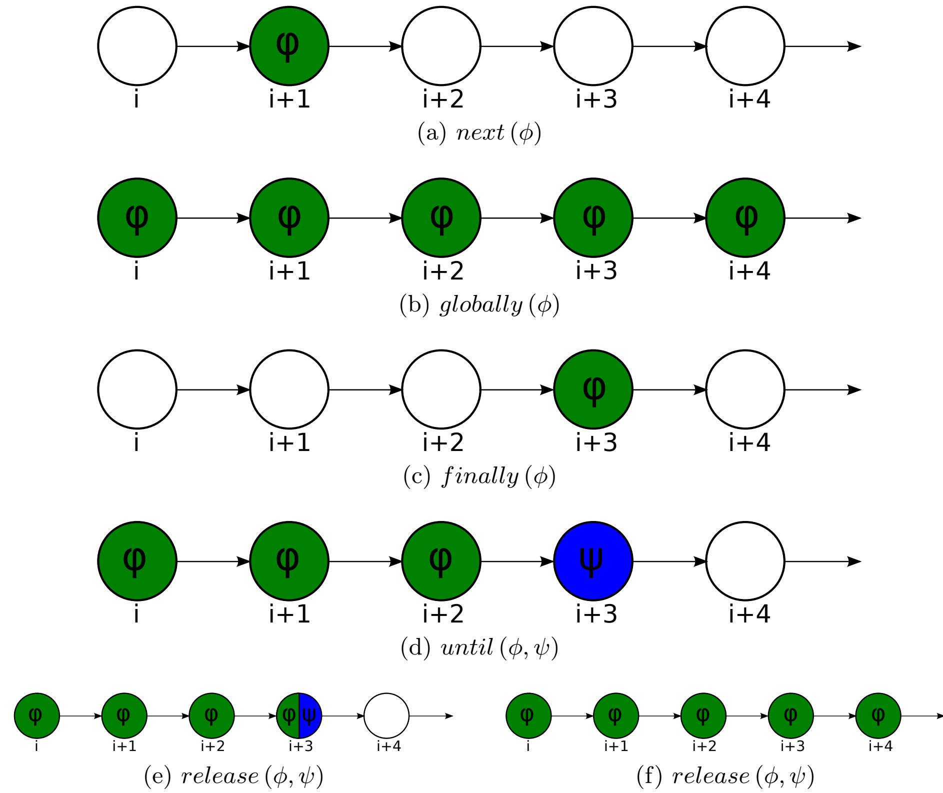
until(ϕ, ψ) The *until* combinator means $\exists k \geq i, \pi_k \models \psi, \forall j < k, \phi_j \models \phi$. Informally this means that starting in the current world ϕ has to hold until a world is reached where ψ holds. ψ has to hold eventually on the path. Figure 5.1d illustrates this.

release(ϕ, ψ) The *release* combinator can be rewritten as a combination of *globally* and *until*: *release* (ϕ, ψ) is the same *until* $(\phi, \phi \vee \psi) \wedge$ *globally* (ϕ). The main difference with until is that in case ψ never holds at the path that ϕ has to hold for the entire path. Figure 5.1e and 5.1f illustrate both cases.

5.2.2 Example Queries

We revisit our running examples and answer them using LTL. LTL is implemented as a meta-interpreter in SOUL. We added predicates to verify if a formula holds for one path or all paths of a certain version. The *holdsOnAllPathsStartingFrom:* predicate will check if the formula holds on every path starting in a version.

Listing 5.7 depicts us the query for E1. We first look for the oldest version and require that the class exists in every path starting in that version. This is the equivalent of checking if a class has always existed.



(a) *next* (ϕ)

(b) *globally* (ϕ)

(c) *finally* (ϕ)

(d) *until* (ϕ, ψ)

(e) *release* (ϕ, ψ)

(f) *release* (ϕ, ψ)

```
?version isOldestVersion,
globally(?class isClass) holdsOnAllPathsStartingFrom: ?version
```

Listing 5.7: What class has always existed on every branch using LTL

```
1 ?version isVersion,
2 ?path isPathStartingFrom: ?version where:
3   (and(next(?class isClass),
4       not(?class isClass))
```

Listing 5.8: What class is introduced in the next version using LTL

```
1 ?version isOldestVersion,
2 ?limit equals: [ 10 minutes ],
3 ?path isPathStartingFrom: ?version where:
4   finally(and(?author isAuthorOfVersion,
5              ?timeA isTimestampOfVersion,
6              next(and(?author isAuthorOfVersion,
7                       ?timeB isTimestampOfVersion,
8                       ?timeA isWithin: ?limit ofTime: ?timeB))))
```

Listing 5.9: Which author published a next version within 10 minutes of a previous commit using LTL

Listing 5.8 depicts us the query for E2. Just like in SOUL we have to ensure that the *?class* variable is bound before using it in a negation. In case it is not bound we are asking for a version that does not contain any classes. To circumvent this we first look for a class that exists in the next version.

Listing 5.9 depicts us the query for example 3. The query is similar to the one written in SOUL, except that we can make use of the *next* combinator, which increases the readability of the query.

Listing 5.10 depicts the query for E4. LTL has no support for repetition, and the only way to write down the query is by nesting the *next* combinator. This works for a limited number of repetitions. We can, just as with SOUL, implement a predicate that verifies the property. It is debatable if such a predicate would still be considered to be LTL, as the properties are normally checked in only one version.

5.3 Computation Tree Logic

Similar to LTL, computation tree logic also reasons over a Kripke structure. The difference with LTL is that it reasons over *states* instead of paths throughout the structure. In LTL we are limited to one path throughout the computation. CTL looks at all paths of the current state of the computation.

```
1 ?version isVersion,
2 ?path isPathStartingFrom: ?version where:
3  and(?authorA isAuthorOfVersion,
4     next(and(?authorB isAuthorOfVersion,
5       not(?authorA equals: ?authorB),
6       next(and(?authorA isAuthorOfVersion,
7     next(?authorB isAuthorOfVersion))))))
```

Listing 5.10: Which two authors committed two times in a row after each other using LTL

5.3.1 Specification

The operators are similar to those of LTL. The most important difference is that every operator has an obligatory quantifier. This quantifier can be either an existential or a universal one. The former denotes the formula has to hold on one path of the current state, while the latter indicates the formula has to hold on every path.

We discuss all combinations of operators and quantifiers. We use the same syntax as in LTL. We also employ the same colours for the illustrations: a blue world indicates ϕ holds in this world, while a green world indicates ψ holds in this world. All the illustrations can be found in figure 5.1.

eNext(ϕ) The *eNext* combinator means that $\pi_{i+1} \models \phi$, for a path starting in π_i. Informally this means ϕ has to hold in one of the immediate successors of π_i.

aNext(ϕ) The *aNext* combinator means that $\pi_i \models \phi$ for all paths starting in π_i. Informally this means that ϕ has to hold in each of the immediate successors of π_i.

eGlobally(ϕ) The *eGlobally* combinator means that $\forall j \geq i, \pi_j \models \phi$, for a path starting in π_i. Informally this means that ϕ has to hold in every state on one path starting in π_i.

aGlobally(ϕ) The *aGlobally* combinator means that $\forall j \geq i, \pi_j \models \phi$, for all paths starting in π_i. Informally this means that ϕ has to hold in every state on each path starting in π_i.

eFinally(ϕ) The *eFinally* combinator means that $\exists j \geq i, \pi_j \models \phi$, for a path starting in π_i. Informally this means that ϕ eventually has to hold in one of the successors of π_i.

aFinally(ϕ) The *aFinally* combinator means that $\exists j \geq i, \pi_j \models \phi$, for all paths starting in π_i. Informally this means that ϕ eventually has to hold in each path starting in π_i.

```
1 ?version isOldestVersion,
2 ?class isClass,
3 aGlobally(?class isClass) succeedsIn: ?version
```

Listing 5.11: What class has always existed using CTL?

```
1 ?version isVersion,
2 and(eNext(?class isClass),
3    not(?class isClass)) succeedsIn: ?version
```

Listing 5.12: What version introduced a class using CTL?

eUntil(ϕ, ψ) The *eUntil* combinator means that $\exists k \geq i, \pi_k \models \psi, \forall j < k, \pi_j \models \phi$, for a path starting in π_i. Informally this means that on a path starting in π_i ϕ has to hold until a world is reached where ψ holds. ψ has to hold eventually on that path.

aUntil(ϕ, ψ) The *aUntil* combinator means that $\exists k \geq i, \pi_j \models \phi$, for each path starting in π_i. Informally this means that on every path starting in π_i, ϕ has to hold until a world is reached where ψ holds.

5.3.2 Example Queries

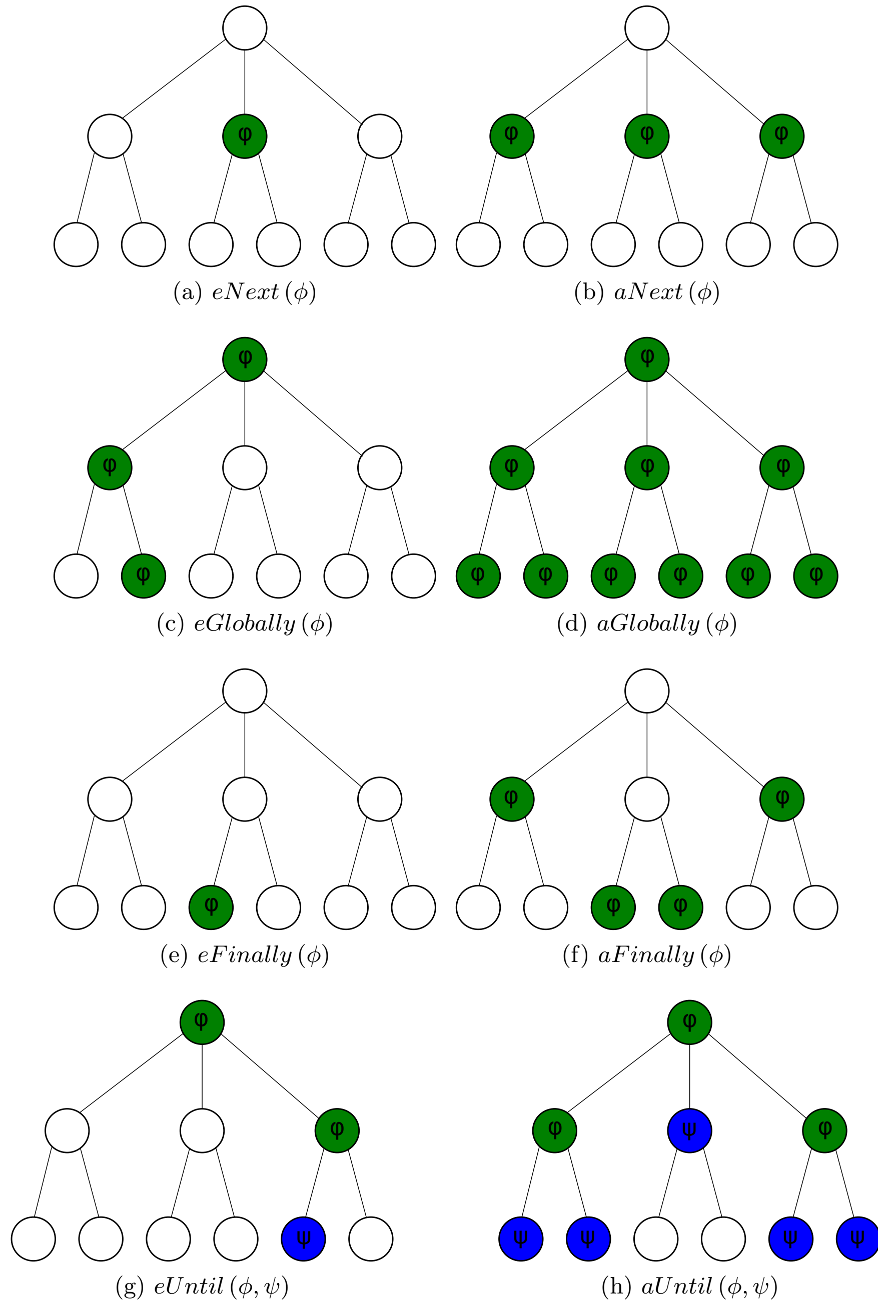
CTL is implemented as a meta-interpreter using SOUL. We added one predicate *succeedsIn:* which can be used to solve CTL queries. It takes a CTL formula and a start version as input. It succeeds if the formula holds for that version. In what follows we will write down queries for our running examples using CTL.

Listing 5.11 depicts us the query for E1. We use the *aGlobally* operator to ensure that the class exists in every version.

Listing 5.12 depicts the query for E2. As in LTL and SOUL we have to ensure that the *?class* variable is bound before using it in a negation. This is done by first looking for a class in the next version. Afterwards we can use the bindings in the negation.

Listing 5.13 depicts the query for E3. We look for a successor of the root of the software project that is committed by *?author* at *?timeA*. We require that the next version is published by the same author at *?timeB*, and that both timestamps are within 10 minutes.

Listing 5.14 depicts the query for E4. Just as with LTL we have difficulties expressing the repetition in a concise manner.



(a) $eNext\,(\phi)$
(b) $aNext\,(\phi)$
(c) $eGlobally\,(\phi)$
(d) $aGlobally\,(\phi)$
(e) $eFinally\,(\phi)$
(f) $aFinally\,(\phi)$
(g) $eUntil\,(\phi, \psi)$
(h) $aUntil\,(\phi, \psi)$

Figure 5.1: CTL operators with corresponding quantifier

```
1 ?version isOldestVersion,
2 ?limit equals: [ 10 minutes ],
3 eFinally(and(?author isAuthorOfVersion,
4       ?timeA isTimestampOfVersion,
5       eNext(and(?author isAuthorOfVersion,
6          ?timeB isTimestampOfVersion,
7          ?timeA isWithin: ?limit ofTime: ?timeB))))
8  succeedsIn: ?version
```

Listing 5.13: Which author published two version within 10 minutes using CTL?

```
1 ?version isVersion,
2 and(?authorA isAuthorOfVersion,
3    eNext(and(?authorB isAuthorOfVersion,
4       not(?authorA equals: ?authorB),
5       eNext(and(?authorA isAuthorOfVersion,
6     eNext(?authorB isAuthorOfVersion))))))
7    succeedsIn: ?version
```

Listing 5.14: Which two authors committed two times in a row after each other using CTL

5.3.3 Differences between CTL and LTL

Although there are plenty of queries that are expressible in CTL and LTL, both formalisms do not have the same expressive power. In this section we focus on those differences.

LTL is not equivalent to CTL

The first example is adapted from [7], chapter 2. Figure 5.2 shows two Kripke structures that are indistinguishable for LTL. LTL can only look at one path at the time. For the first structure it sees the paths $\{P, Q\}, \{P\}, \{\}$ and $\{P, Q\}, \{P\}, \{Q\}$. For the second structure it sees exact the same paths. LTL reasons over paths in the Kripke structure. As both structures have the same paths there exists no query in LTL which would succeed in one structure and fail in the other.

CTL reasons over worlds of the Kripke structure. It can look at more than one successor of a certain world. This means that we can write a query which would succeed in one structure and fail in the other. The query `aNext(and(eNext(Q),eNext(not(Q))))` is such a query. We say that for every next version of the start version there must exist a next version where Q holds and where Q does not hold. This is possible if there are two different successors. This is the case for the first structure, where the query succeeds. It fails in the second structure.

We can adapt the example to a more practical one. Imagine if you want to see if a branch introduced a class. This can be done by checking at the branching point whether there exists a successor for which the class exists, and a successor for which the class does not exist. This cannot be expressed in LTL.

CTL is not equivalent to LTL

Figure 5.3 depicts a Kripke structure that contains infinite paths. For this structure we can give a query in LTL that succeeds, and for which no alternative exists in CTL. The LTL query `finally(globally(P))` succeeds: for every path we take we always end up in a state where P will always hold. Either we have a path that leaves the first world. We end up in the last state, where P always holds. Or we have an infinite long path that always stays in the first world, yet here P also holds.

We cannot express this in CTL: trying the query `aFinally(aGlobally(P))` will fail. The query states that P will always hold at some point for whatever path we take. This fails due to the infinite long path that stays in the first state. There is always a successor where P does not hold. Figure 5.4 illustrates this. Note the left-most path, where there always is a successor where P does not hold. As the formula has to hold on all successors going one level deeper does not help, as the situation just repeats.

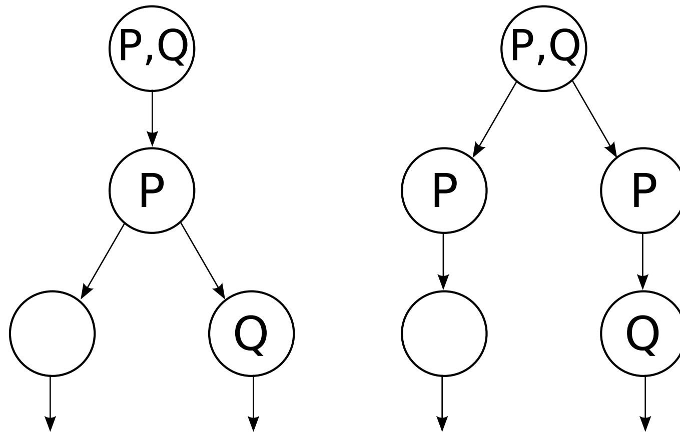


Figure 5.2: LTL cannot distinguish these Kripke structures. LTL reasons over sequences of worlds. The sequences here are exactly the same, meaning there is no LTL formula which succeeds in one structure but fails in the other. CTL reasons over worlds, and we can write down queries that succeed in one structure, yet fail in the other.

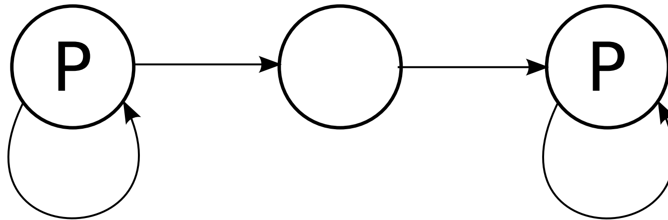


Figure 5.3: The LTL query *finally(globally(P))* succeeds for this Kripke structure. The CTL formula *aFinally(aGlobally(P))* fails here.

In our setting we do not have these infinite paths, and we suspect that LTL will actually just be a subset of CTL. This does not exclude LTL from our study, as queries might be easier to write and understand in LTL than in CTL.

5.4 Quantified Regular Path Expressions

Both LTL and CTL are temporal logics. In this section we will discuss a different sort of specification language, namely quantified regular path expressions. Quantified regular path expressions [13, 15] allow developers to express properties a path in the version graph has to exhibit. The reasoning engine will try to find a path that exhibits these characteristics.

Quantified regular path expressions are similar to regular expressions [6, 17]. Regular expressions provide means to describe a sequence of characters.

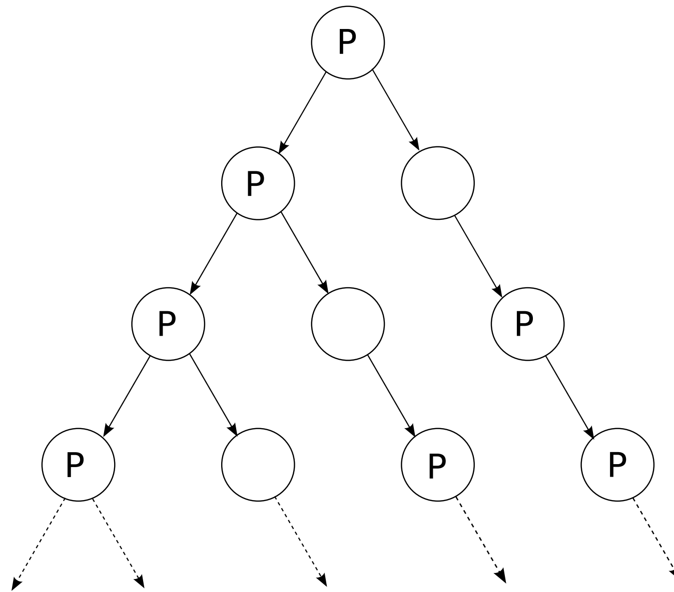


Figure 5.4: The reason why the CTL query *aFinally(aGlobally(P))* fails is due to the left-most infinite path, where there always is a successor where P does not hold

Quantified regular path expressions will describe a sequence of properties that have to hold for a sequence of states. A difference is that in regular expressions there is only one possible successor for each character, while a version could have multiple ones. We provide a quantifier, allowing the developer to specify if the regular path expression has to hold on all or just one path.

Regular expressions use special characters to denote the pattern. An example is the * operator, which denotes that the previous expression may succeed zero or more times. We will introduce similar operators using predicates. This allows for a syntax that is better integrated with the rest of the tool. In regular expressions one character normally denotes one character in the string. The reasoning engine evaluates one condition in a version, and evaluates the next condition in a successor of that version.

A quantified regular path expression consists of three parts. Listing 5.15 depicts us an example query. The quantifier indicates whether the expression has to hold on one or all paths. In the example the quantifier is indicated by *e*, which is the existential quantifier.

The setup is used to both customize and restrict the search space of the expression. In our example this is the last part, indicated by *graph*. It specifies what predicate has to be used to transition between two version. It specifies the start and end version of the query. Finally it specifies what bindings have to be passed between versions. This is indicated by <*?class*>.

These variables will also become available in the rest of the query.

The regular path expression is the sequence of expressions, prefixed with a quantifier. In the example we describe three versions, in which the first two versions both contain *?class isClass*, while the last one does not contain this class. The normal behaviour is that each expression describes one version. There are predicates that differ from this behaviour, which can be used to describe paths that have an unknown length.

In what follows we will describe these predicates. ϕ can be a sequence of expressions, while σ is limited to statements holding in one state.

```
1 ?first isVersion,
2 e(?class isClass, ?class isClass, not(?class isClass)) matches:
3  graph(versionTrans, ?first, ?last, <?class>)
```

Listing 5.15: An example quantified regular path expression

5.4.1 Specification

manyOf<$(\phi_1, \ldots, \phi_n)$ The *manyOf<* predicate consumes subsequent worlds as long as the expression ϕ succeeds. It will backtrack to shorter sequences if necessary. Note that *manyOf<* may consume zero states.

manyOf>$(\phi_1, \ldots, \phi_n)$ The *manyOf>* predicate is similar to the *manyOf<* predicate. Instead of trying the longest sequence first, it will start with zero worlds, and try more worlds when backtracking from later failure.

plus$(\phi_1, \ldots, \phi_n)$ The *plus* predicate is like the *manyOf<* predicate, except that the expression must succeed at least once.

not(σ) The *not* predicate is limited to one version, and succeeds if its argument fails in that version. No regular expression can be passed to *not*.

times$(i, \phi_1, \ldots, \phi_n)$ The *times* predicate takes a number i, and succeeds if the expression ϕ succeeds as many times as i.

We illustrate the difference between the more complex primitives. Figure 5.5 depicts an example path. We use the same conventions as with our illustrations of LTL. A green world designates that ϕ holds in this world, while a blue world designates that ψ holds.

We now compare the query *manyOf<*(ϕ),ψ with *manyOf>*(ϕ),ψ. Both are semantically equivalent, but differ operationally. Figure 5.6 shows us

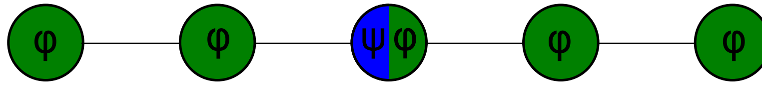


Figure 5.5: An example path that we will try to match with a quantified regular path query

how *manyOf* matches greedily, consuming as many worlds as possible. It is only if the rest of the query fails that it will backtrack, until ψ succeeds. The figure illustrates the different matches that occur to detect the path. The colours indicate what formulate is evaluated in each version.

In figure 5.7 we see how the *manyOf>* predicate first consumes zero worlds, consuming more and more until ψ finally succeeds. We see here that *manyOf>* needs fewer tries than *manyOf<*, and does not evaluates the formulas as much. It is the responsibility of the developer to know the difference between both.

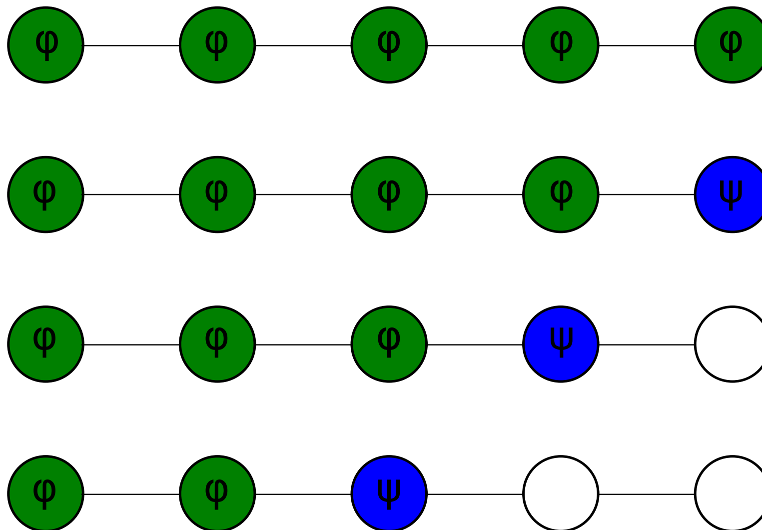


Figure 5.6: matching of *manyOf<*. The colours indicate what statement is evaluated in what version.

```
1 ?first isOldestVersion,
2 e(manyOf(?class isClass))
3  matches:
4   graph(versionTrans, ?first, ?last, <?class>),
5 ?last isEndVersion
```

Listing 5.16: Which class has always existed on a path using quantified regular path expressions

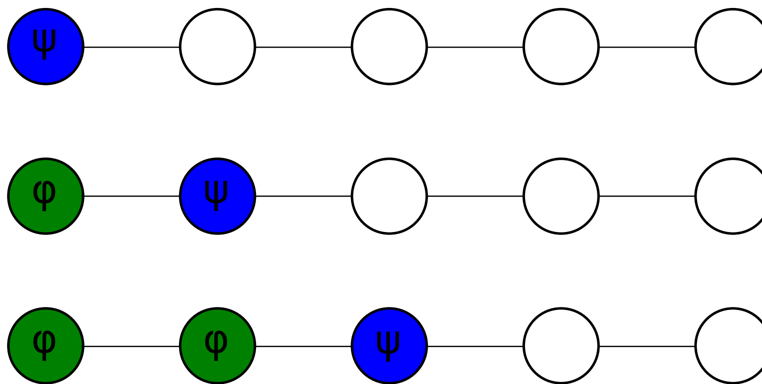


Figure 5.7: matching of *manyOf*>. The colors indicate what statement is evaluated in what version.

5.4.2 Example Queries

In what follows we provide the same example queries using quantified regular path expressions. Listing 5.16 depicts the query for a slight adaptation of E1. We are interested in a class that has always existed in a path. The reason we need to adapt our query is that path queries can only reason over the paths between two specific points. The universal quantifier does not help us here. Path queries require that the end state is always the same for every path. An unmerged branch in the system results in the query failing, as it is impossible to have the same end state. We could change the semantics of the universal quantifier, making it only consider versions where the end state is reachable. But then it would disregard those branches, and it would not verify all the versions. Path queries are only useful to reason over paths between two points, and not for every version.

Listing 5.17 depicts the query for E2. Just as in the other specification languages we have to ensure *?class* is bound before using it in a negation. A solution is to first look for a version where *?class* exists, and then ensure that the class does not exist in its predecessor. Notice that this query has a lot of setup before we actually use a path expression. We could have easily

```
1 ?version isVersion,
2 ?class isClass,
3 ?prev isPreviousVersionOf: ?version ,
4 e(not(?class isClass),
5   ?class isClass)
6 matches:
7   graph(versionTrans, ?prev, ?version, <?class>)
```

Listing 5.17: What version introduced a class using path expressions?

```
1 ?version isMostRecentVersion,
2 e(manyOf>([true]),
3   ?class isClass,
4   not(?class isClass))
5 matches:
6   graph(reverseVersionTrans, ?version, ?end, <?class>)
```

Listing 5.18: What version introduced a class using path expressions?

written this query without using path expressions at all.

A better way is depicted in listing 5.18. We use another transition predicate, which will go to a previous version instead of a next version of the current version. This means that we can first look for a class that exists in the current world, and check if it no longer exists in the “next” version.

Listing 5.19 depicts the query for E3. The *and* predicate is used to evaluate more than one expression in a version. We have to add *?limit* to the list of variables, if not it would not be bound. This is because the path queries are transformed before evaluating the query, and the *?limit* variable would not be available for the path query if it is not added to that list.

```
1 ?version isVersion,
2 ?limit equals: [ 10 minutes ],
3 e(and(?author isAuthorOfVersion,
4      ?timeA isTimestampOfVersion),
5   and(?author isAuthorOfVersion,
6      ?timeB isTimestampOfVersion,
7      ?timeA isWithin: ?limit ofTime: ?timeB))
8 matches:
9   graph(versionTrans,?version, ?end, <?author, ?timeA, ?timeB, ?limit>)
```

Listing 5.19: Which author published a next version within 10 minutes of a previous commit using path expressions?

```
1 ?version isVersion,
2 e(times(2,
3    ?authorA isAuthorOfVersion,
4    and(?authorB isAuthorOfVersion,
5       not(?authorA equals: ?authorB)))) matches:
6     graph(versionTrans, ?version, ?end, <?authorA, ?authorB>)
```

Listing 5.20: Which two authors committed at least two times in a row?

Listing 5.20 depicts the query for E4. Path queries are very well suited for repetition. Here we use the *times* predicate, which verifies that the statement holds at least two times.

5.5 Conclusion

The specification language is the language a developer will use to express queries. In this chapter we have discussed four different specification languages. SOUL is a logic programming language that features symbiosis with Smalltalk. Linear temporal logic and computation tree logic are two temporal logics with different predicates and expressive power. We also discussed quantified regular path expressions, a language where the developer can describe a path throughout the version history. For each specification language we have discussed the semantics and provided several examples.

6

Absinthe: Detection Mechanism

In this chapter we discuss the detection mechanism of our tool. The detection mechanism is an essential component that connects the program representation with the specification language. It identifies what parts of the model exhibits the characteristics specified by the query.

We consider two different proof strategies, namely selective definite linear clause resolution (SLD resolution) and linear resolution with selection function for general logic programs (SLG resolution). SOUL already features SLD resolution and can be used by our tool. We extend SOUL to incorporate SLG resolution.

In what follows we discuss both resolution strategies. We demonstrate how the specification languages can be expressed using one of the detection mechanisms.

6.1 SLD Resolution

SLD resolution is a proof strategy that is found in almost every declarative programming language. We explain the workings with an illustrative example. Listing 6.1 is a logic program with some simple family relationships. It contains two different rules. The *parent* predicate checks whether a person is the parent of another person. There are two different implementations. One implementation checks if that person is the mother, while the other implementation checks if that person is the father.

The other rule checks if one person is the grandfather of another person. This is done by checking if the person is the father of one of the parents of the other person. Finally there are two facts given, namely that John is the

```
1  ?X grandfather: ?Z if
2    ?X father: ?Y,
3    ?Y parent: ?Z.
4
5  ?X parent: ?Y if
6    ?X father: ?Y.
7
8  ?X parent: ?Y if
9    ?X mother: ?Y.
10
11 john father: mary.
12 mary mother: brian
```

Listing 6.1: A simple family relation

father of Mary, and Mary is the mother of Brian.

We are now interested in looking for the grandchild of John. We do this by launching the following query:

`john grandfather: ?grandchild.`

SLD resolution looks for all the rules where the head unifies with the given query. Two rules unify when they have the same functor, and all the arguments unify as well. Two atoms unify when they are equal. Two unbound variables always unify. Unifying a variable with an atom binds the variable to that atom. For `john grandfather: ?grandchild` only the first rule unifies. SLD resolution replaces the goal with the conditions of the unified rule. Put otherwise, proving the original query now becomes proving the conditions of the unified rule. This results in the following goals:

`john father: ?Y, ?Y parent: ?grandchild.`

SLD resolution proves the goals from left to right. The first goal is proven in a similar way. There is only one rule that unifies, namely `john father: mary`. This is a fact; a rule without conditions. A fact always succeeds, and SLD resolution simply removes the goal. It also binds *?Y* to mary, and continues proving the rest of the goals. The last goal now reads:

`mary parent: ?grandchild`

This time there are two rules that unify. In this case SLD resolution creates a choice point. In a choice point SLD resolution tries the first rule that unifies, but remembers there were other rules as well. In case the proof would fail for the first rule SLD resolution backtracks to the last choice point and tries the next rule.

Proving the first rule fails as the condition `mary father: ?Y` has no rule that unifies. SLD resolution backtracks to the last choice point, and tries the second rule. This time the body `mary mother: ?Y` has a rule that unifies. `mary mother: brian` is a fact, and succeeds.

SLD resolution proved our query, and binded *?grandchild* to brian. The

user can ask for more solutions. In this case SLD resolution will once more backtrack to the last choice point, notice that there are no more rules nor previous choice points and fail.

SLD resolution is commonly represented as a tree. Figure 6.1 depicts the refutation proof tree of our example query. It shows how each goal is replaced by the conditions of a unifying rule. Choice points are depicted by several branches in the tree.

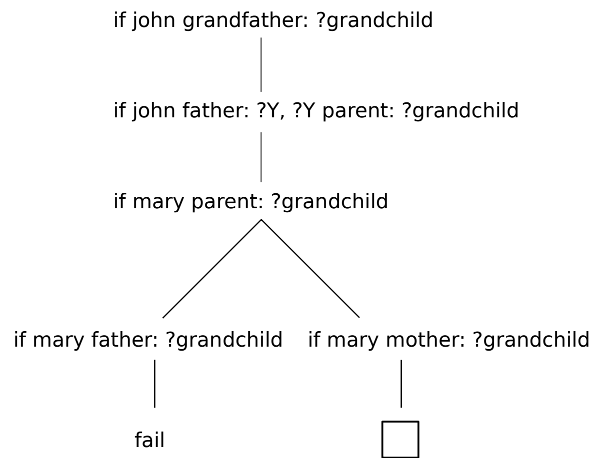


Figure 6.1: Refutation Proof Tree based on SLG Resolution

Negation

SLD resolution deals with negation by using negation as failure. This means that when SLD resolution tries to prove `not(Query)` it first proves `Query`. If this succeeds it knows there is a solution, meaning the negation has to fail. If it does not get a result this means that `Query` fails, or that the negation has to succeed. SLD resolution extended with negation as failure (SLDNF) is both used in SOUL and Prolog.

```
1 peter person.
2 jack person.
3 john person.
4
5 peter father: jack.
6
7 "the following query leads to floundering"
8 not(?X father: ?Y)
```

Listing 6.2: Floundering in SLG resolution

Special care is needed when using negation as failure with a query with

```
1  a edge: b.
2  b edge: c.
3  b edge: d.
4
5  ?X path: ?Z if
6    ?X path: ?Y,
7    ?Y edge: ?Z.
8
9  ?X path: ?Z if
10   ?X edge: ?Z
```

Listing 6.3: Infinite left-most recursion

unbound variables. Listing 6.2 depicts a program and query that has unbound variables.

We are interested in all the persons who are not a father. This is done with the following query: `not(?X father: ?Y)`. The expected answers are `jack` and `john`. SLDNF resolution tries to proof the negated goal by first proving the goal. The goal has one answer, namely `peter father: jack`. This means that the negated goal fails, and so does our query. A solution to this problem is to first bind first variable:

```
?X person, not(?X father: ?Y).
```

Drawbacks

SLD Resolution with negation as failure has some disadvantages [9]. First it may not terminate due to infinite recursion. The conditions of a rule are proved left to right, and goals are unified from the top to bottom. The order of these has no effect on the declarative meaning. It does change the procedural interpretation, and it is the responsibility of the programmer to get this correct. Infinite recursion happens when a rule recursively calls itself with the same variable bindings, up to variable renaming. Listing 6.3 depicts such a program. We launch the following query:

```
a path: ?X.
```

SLD resolution replaces the goal with the conditions of the first unified rule. These are `?X path: ?Y, ?Y edge: ?Z`. To proof the first condition SLD resolution will once again unify with the same rule, hereby replacing the condition with the conditions of that rule. This leads to a left-most infinite recursion. SLD resolution will not backtrack to the choice points, as the resolution never fails.

The declarative meaning of the program is correct, yet it is due to the procedural interpretation that it fails. Switching the statements on line 6 and 7 solves the problem.

Second, it may evaluate the same literal in a rule body multiple times.

When a predicate is already proven previously in the computation SLD resolution just proves it again. It only degrades performance, as it cannot fail.

6.2 SLG Resolution

SLG is a different proof strategy that tackles both problems of SLG resolution. It stores previously computed answers, and it will detect previously made calls. The latter will prevent calling a rule we are already calculating, which leads to infinite recursion. In this section we explain how SLG resolution works in detail, and we sketch a basic implementation. The examples we use can be found in [28]. We also refer to to [28] for the full implementation, which we do not discuss here.

SLG resolution does this by introducing two new concepts. First it introduced is a table, in which conditions are stored together with their answers. This table acts as a sort of caching mechanism. Second, it introduces waiting nodes. Waiting nodes are suspended computations that need additional data before the computation can continue. They are used to prevent infinite recursion by suspending the computation.

6.2.1 SLG Resolution for Positive Programs

A positive program is a program without negation. We once more use the program depicted in listing 6.3, but mark the path predicate as tabled. A tabled predicate will be resolved using SLG resolution instead of SLD resolution. We launch the same query:

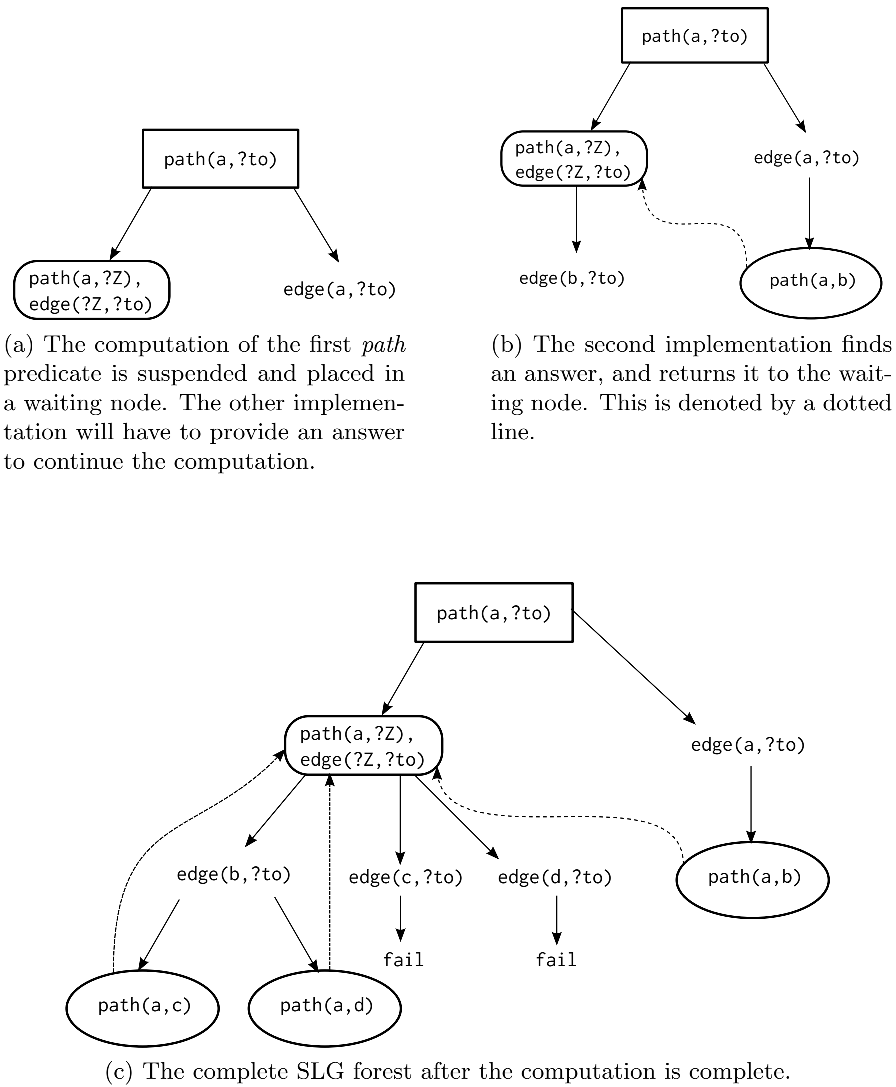
```
a path: ?X.
```

SLG resolution starts identical to SLG resolution, but instead of trying to resolve `a path: ?Z` it will detect that is the same call, up to variable renaming. The second call will be marked as a waiting node of the original call `a path : ?to`. A waiting node is suspended until SLG resolution finds an answer for the original call. This is possible if there are several implementations for the same rule. In our example the second `path` implementation will return `a path: b`, allowing SLG resolution to continue with the rest of the computation. It continues the waiting node with the answer, providing two more answers `a path: c` and `a path: d`. These answers are once again passed to the waiting node, but do not provide any more answers.

Figure 6.2 depicts us how the query is resolved using SLG resolution.

6.2.2 Negation and Recursion through Negation

A negated literal can only succeed if the positive counterpart is completely evaluated and has no answers. When we use recursion through negation some subgoals may depend on each other in a circular fashion. We can



(a) The computation of the first *path* predicate is suspended and placed in a waiting node. The other implementation will have to provide an answer to continue the computation.

(b) The second implementation finds an answer, and returns it to the waiting node. This is denoted by a dotted line.

(c) The complete SLG forest after the computation is complete.

Figure 6.2: Using SLG resolution to solve **a path:** ?to

```
1 a move: b.
2 b move: a.
3 b move: c.
4 c move: d.
5
6 tabled ?X wins if
7   ?X move: ?Y,
8   not(?Y wins)
```

Listing 6.4: Recursion through Negation

see in listing 6.4 that `a wins` depends on `not(b wins)` and `b wins` depends on `not(a wins)`. Using SLG we can still get answers defined under the well-founded semantics [32]. Both answers are undefined under the well-founded semantics, while `c wins` is true and `d wins` is false.

With SLD resolution the query `a wins` and `b wins` would lead to an infinite computation, as each calls the other. SLG resolution will delay the resolution of these conditions. As answers may depend on each other we attach conditions to answers. In the example `a wins` as an answer with `not (b wins)` attached as a condition. An answer is correct if all the attached conditions can be proven. The conditions are removed once proved later in the computation.

For answers where SLG resolution cannot prove the conditions it returns that those answers are undefined. For our example both `a wins` and `b wins` cannot be proven as they depend on each other. They are both undefined.

6.2.3 SLG Resolution using SLD Resolution

SLD resolution is already available in SOUL. There are two options to implement SLG resolution. Either we write a different reasoning engine that allows SLG resolution. Doing this in a clean manner, so that both engines can be used in the same manner, would require a lot of work.

SLG resolution can also be implemented using SLD resolution and using some extra data structures [12] [28]. This approach was developed to extend existing languages with SLG resolution. The only requirement is that one can somehow add extra facilities in the base language. SOUL has symbiosis with Smalltalk, which can also be used to extend the language. We opted for this approach over implementing a new reasoning engine.

We mark rules that have to use SLG resolution with the keyword `tabled`. These rules will be transformed when the program is compiled to an equivalent program using SLD resolution. In the rest of the program the developer can use these rules like he would use a normal rule. We sketch transformations for positive programs, but refer to [28] for the full implementation.

Using SLD Resolution to Simulate Positive Programs

We need a special data structure that manages the waiting nodes. Every call to a tabled predicate will be stored inside a table. This table keeps the answers for that call and a list of waiting nodes for the call. SLG resolution also uses this table to see if it is are already solving a query. Every call to a tabled predicate will be associated with an id, which uniquely identifies it. We use this identifier through the computation so we know what call we are computing.

A waiting node suspends a computation. This can be done by splitting the conditions of each rule up in new rules wherever a call to a tabled predicate happens. These are exactly the spots where the computation could be suspended. Providing an answer to a suspended node results in calling the newly generated rules with an answer. This is done by the *answer* primitive, which is added at the end of each rule. By adding it at the end of the rule we are sure that we have an answer for that rule. It both continues suspended nodes with the answer, but is also used to update the table where answers are stored for a predicate.

We need to make sure that wherever we have a call to a tabled rule in the conditions of a tabled rule that we consult the table. We might already be computing that rule, so we would have to add it as a waiting node instead. Every call to a tabled rule is replaced by the primitive *slgcall*. *slgcall* will see if a call is new, and in case it is not it will add a waiting node to the correct call in the table. *slgcall* also has a continuation as argument so it can continue the computation of a suspended node. It will also add an *answer* primitive to ensure we return new answers to all the waiting nodes.

If we want to consult the table to detect previously made calls we need a definition when two calls are identical. Two calls are identical if they have the same functor and number of arguments, and each argument is the same as the corresponding argument. This is very similar to the unification of two rules. The difference lies in unbound variables. At first it may seem enough to say two unbound variables are identical: the two rules `a path: ?X` and `a path: ?to` are both the same call, but have different variable names. Both these calls are identical. Saying unbound variables are identical might seem correct, but would fail for these two calls: `path(?X,?Y)` and `path(?X,?X)`. The first one calculates every path, while the latter detects cycles. The solution is to require that we have a bijection between both sets of unbound variables.

We want to make sure that we can use rules marked as tabled can be used throughout the program as you would use normal clauses. For this we will introduce a primitive `slg`. We replace the conditions of the marked rule by the expression `slg(?rule)`, where `?rule` is the conclusion of that rule.

Listing 6.5 and 6.6 depicts us how we can transform the example we

```
1 tabled ?X path: ?Y if
2   ?X path: ?Z,
3   ?Z edge: ?Y.
4
5 tabled ?X path: ?Y if
6   ?X edge: ?Y
```

Listing 6.5: The path program where the path rules are marked as tabled

```
1  ?X path: ?Y if
2    slg(?X path: ?Y)
3
4  slgpath(?Id,?X,?Y) if
5    slgcall(?Id, ?X path: ?Z, 'pathcont0').
6
7  pathcont0(?Id, path(?X,?Z)) if
8    ?Z edge: ?Y,
9    answer(?Id, ?X path: ?Y).
10
11 slgpath(?Id,?X,?Y) if
12   ?X edge: ?Y
13   answer(?Id, ?X path: ?Y)
```

Listing 6.6: The transformation of the path program depicted in listing 6.5

used in 6.3. The first rule is split in two parts, as there is a call to a tabled predicate so it might be suspended. The second part is added to the continuation. The second rule now contains an *answer* primitive at the end so suspended nodes can continue their computation.

These transformations only work for some positive programs. We need support to pass around variable bindings to the continuations. We also need more complex data structures and primitives to support negation and recursion through negation. Although we have implemented these, we will not explain these here. We refer to [28] for a complete implementation.

Symbiosis with Smalltalk

SOUL is a language that features symbiosis with Smalltalk. This means that Smalltalk blocks can be used together with SLG resolution. There is a conflict between blocks that contain side-effects and reusing previously computed answers.

We considered how to decide when two Smalltalk blocks are equal. This is needed to detect previously made calls with SLG resolution. We decided that two blocks are equal when their return values are equal. This means

```
1  finally(?term) : ?path if
2    ?path isPath,
3    equals(?path,<?first|?rest>),
4    ?first setAsReferenceVersion,
5    ?term
6
7  finally(?term) : ?path if
8    ?path isPath,
9    equals(?path,<?first|?rest>),
10   finally(?term) : ?rest
```

Listing 6.7: Implementation of finally using SLD resolution. The first rule checks if the term holds in the first version of the path, while the second rule recursively calls finally with the rest of the path. The predicate *setAsReferenceVersion* sets the implicit world where predicates are evaluated in.

that we have to evaluate a block before we can detect similar calls. This also means that a block will be evaluated, even if we can reuse previously computed answers. We would not get this behaviour if we would only compare the abstract syntax tree of each block. This approach would lead to unexpected behaviour. Two blocks might have the same semantics, yet have different abstract syntax trees. Or the abstract syntax tree could be identical for two blocks, but due to lexical scoping the result can be completely different.

We wanted to be sure that blocks are only evaluated once. The value of a block might be expensive to compute, or could contain side effects. Note that the latter is discouraged in SOUL, and there is no guarantee that the side effect will not happen twice. Recomputing a block several times is suboptimal, as one of the reasons to use SLG is preventing the computation of the same literal several times. We have chosen to just use the normal resolution strategy, where a block is evaluated upon unification. We only receive the return value of the block, and this value is used throughout the rest of the evaluation.

6.3 Linear Temporal Logic using SLD resolution

We have implemented LTL using SLD resolution. We have opted to implement all the predicates to reason over a single path. The implementation is straightforward and we will not discuss this in detail.

Listing 6.7 depicts the implementation of the *finally* predicate. The *finally* predicate succeeds if the term holds somewhere along the path. There are two possible rules: the first rule check whether the term succeeds in the

first version of the path. The second rule recursively calls itself with the rest of the path.

All the other predicates can be implemented in a similar way and are not discussed in this document.

6.4 Computation Tree Logic using SLG resolution

CTL can easily be implemented using SLG resolution. This is done by writing a few primitives that will use tabling. We can map all the other primitives to these primitives. We have the following equivalences:

$$eFinally\,(\phi) = eUntil\,(true, \phi) \tag{6.1}$$

$$aNext\,(\phi) = \neg eNext\,(\neg\phi) \tag{6.2}$$

$$aGlobally\,(\phi) = \neg eFinally\,(\neg\phi) \tag{6.3}$$

$$aFinally\,(\phi) = \neg eGlobally\,(\neg\phi) \tag{6.4}$$

$$aUntil\,(\phi, \psi) = eUntil\,(\neg\psi, \neg\,(\phi) \vee \neg\,(\psi)) \wedge eGlobally\,(\neg\psi) \tag{6.5}$$

With these equivalences we see that we only have two primitives that can span an indefinite number of versions. These primitives are `eUntil` and `eGlobally`. By implementing these as tabled we can make sure all the other primitives indirectly use tabling. We do not need tabling per se, as we are sure our version history does not contain any cycles. SLG resolution does prevent the recomputation of predicates.

We provide the implementation of some of these in listings 6.8, 6.9 and 6.10.

The implementation of *eFinally* makes use of the reduction to *eUntil*: *eUntil* succeeds if the first term succeeds until a version is encountered where the second term holds. The second term has to succeed eventually. The reduction works by providing a first term that always succeeds.

The implementation of *aNext* makes use of the reduction to *eNext*. It ensures there exists no next version where the term does not hold, which is the same as requiring that the term holds in all the next versions.

The implementation of *eUntil* is done by two rules. Either the second term succeeds, meaning *eUntil succeeds*. Or the first term has to succeed and we call *eUntil* recursively with a successor as reference world.

The disadvantage of reducing CTL to these primitives is that we have to use negation as failure. This prevents us from getting variable bindings.

```
1 eFinally(?term) succeedsIn: ?current if
2   eUntil([true],?term) succeedsIn: ?current
```

Listing 6.8: Reducing eFinally to eUntil

```
1 aNext(?term) succeedsIn: ?current if
2   ?current isVersion,
3   not(eNext(not(?term)) succeedsIn: ?current)
```

Listing 6.9: Reducing aNext to eNext

6.5 Quantified Regular Path Expressions using SLG resolution

In this section we discuss how path queries are resolved internally. Regular Path Queries are similar to regular expressions. Regular expressions can be translated to finite state machines. We have chosen for compiling to a nondeterministic automaton. As SOUL already provides a way to backtrack these kinds of automata are easy to write. In regular expressions there is only one possible successor, namely the next character. In our context each next version is a possible successor, making it possible that an expression matches only for one of those successors. SLD resolution already provides this backtracking.

Our translations are similar to the ones used in [13]. We illustrate them with a running example. We will translate the simplified query listed in 6.11, where we are looking for a class that got removed and is reintroduced later in the project. We have to keep track of two different states: the reference version and the state of the state automaton. Both can change independently of the other. Going to a different state does not mean we have to change the reference version and vice versa. The *manyOf* predicate creates one state where the goal is checked as long as it succeeds. But it does not have to succeed, meaning the reference version did not change, while the state automaton did.

We generate a goal for each expression in our path query. A goal will simply check if the expression holds in the reference world. We can add transitions between goals. We have several kinds of transitions, but the two most important are a normal transition and an epsilon transition. The first one will change the reference version and go to the next goal, while the latter will only change the goal. Every goal is called with a list of variables. In our example this is the list `<?name>`. These variables can be used in

```
tabled eUntil(?left,?right) succeedsIn: ?current if
  ?right succeedsIn: ?current

tabled eUntil(?left,?right) succeedsIn: ?current if
  ?left succeedsIn: ?current,
  ?next isNextVersionOf: ?current,
  eUntil(?left,?right) succeedsIn: ?next
```

Listing 6.10: Tabled implementation of eUntil

```
e(
  ?class isClassWithName: ?name,
  not(?class isClassWithName: ?name),
  manyOf([true]),
  ?class isClassWithName: ?name)
    matches:
      graph(?begin, ?end, <?name>, ?result)
```

Listing 6.11: A class that got removed and reintroduced later

the expression and the generated bindings will be passed to other goals too. Finally the `?result` variable will be bound to this list so we can use the binding in the rest of the program.

These transitions are captured by the `npattern` rules. Calling such a rule with a given reference version and state will return the new reference version and state. Depending on the sort of transition the goal will also be called. Given these primitives generating the automaton is not hard. Translating an expression will return two states, the begin and the end state. We will connect these states depending on the primitive.

The *manyOf* predicate will connect the end state of its expressions with its begin state. It will connect its begin state with the rest of the transformations with an epsilon transition, so it does not change the reference version. Figure 6.3 depicts how we translate the *manyOf* predicate of the example. With more complex examples the *manyOf* predicate could use more states, but will always have the same loop structure.

We described how the state machine can be constructed. We still need a predicate that manages both state machine and reference version, and call this predicate with the correct arguments.This is done by the primitive `nreach`. As we do the transformation at parse time we have to return a rule that will be called during the evaluation of the program. `somepath` is the returned function, and will be called when we evaluate the program. Listing 6.13 shows the generated code. This code is the same for every existential query. Figure 6.12 depicts the generated code from our example program.

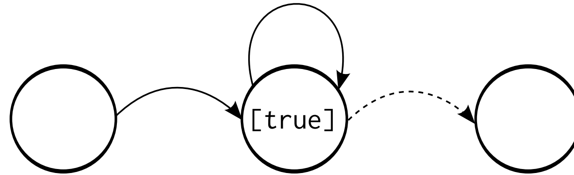


Figure 6.3: Transformation of manyOf. A dotted line implies an epsilon transition

```
1  goal(1,<?name>, ?currentWorld) if
2    ?class isClassWithName: ?name
3
4  goal(2,<?name>, ?currentWorld) if
5    not(?class isClassWithName: ?name)
6
7  goal(3,<?name>, ?currentWorld) if
8    [true]
9
10 goal(4,<?name>, ?currentWorld) if
11   ?class isClassWithName: ?name
12
13 goal(5,<?name>, ?currentWorld).
14
15 npattern(?phi, 1, 2, ?evalIn, ?nextEvalIn, ?fromWorld, ?toWorld) if
16   goal(1,?phi, ?evalIn),
17   versionTrans(?fromWorld, ?toWorld, ?nextEvalIn).
18
19 npattern(?phi, 2, 3, ?evalIn, ?nextEvalIn, ?fromWorld, ?toWorld) if
20   goal(1,?phi, ?evalIn),
21   versionTrans(?fromWorld, ?toWorld, ?nextEvalIn).
22
23 npattern(?phi, 3, 3, ?evalIn, ?nextEvalIn, ?fromWorld, ?toWorld) if
24   goal(3,?phi, ?evalIn),
25   versionTrans(?fromWorld, ?toWorld, ?nextEvalIn).
26
27 npattern(?phi, 3, 4, ?evalIn, ?evalIn, ?fromWorld, ?fromWorld).
28
29 npattern(?phi, 4, 5,?evalIn, ?evalIn, ?fromWorld, ?fromWorld) if
30   goal(4, ?phi, ?evalIn)
```

Listing 6.12: Transformation of example Path Query

```
1 tabled nreach(?phi, ?n1,?p1, ?n1, ?p1, ?current, ?next).
2 tabled nreach(?phi, ?n1,?p1, ?n2, ?p2, ?current, ?next) if
3   npattern(?phi, ?a1, ?a2, ?current, ?next, ?current, ?next)
4   nreach(?phi, ?na, ?pa, ?n2, ?p2, ?next, ?neext)
5
6 ?phi somepath: ?startWorld endWorld: ?endWorld if
7   nreach(?phi, ?startWorld, 1 , ?endWorld, 5, ?startWorld, ?next)
```

Listing 6.13: Starting the State Machine

The transformations are similar for universal queries, but instead of trying one successor we will check if the pattern matches for all successors.

SLG resolution prevents cycles in the computation. Our program representation does not contain any cycles, but we prefer a generic implementation that can also be used in other domains.

6.6 Conclusion

SLD resolution has several disadvantages [9]. It may not terminate due to infinite recursion and it may evaluate the same literal several time in a rule body. SLG resolution resolves these issues by reusing previously calculated answers and preventing infinite recursion. SLG resolution can be implemented by extending SLG resolution and some additional data structures. This approach allows us to extend the existing SLG resolution of SOUL without writing a new resolution engine. We have implemented our specification languages using one of the resolution strategies.

7

Evaluation

In the previous chapters we discussed the three design dimensions for our tool. The first dimension is the program representation. This is a model that contains the information found in a repository, and allows the developer to easily retrieve this information. The model needs to scale with the number of versions available. This means that the memory usage must be reasonable, while the access time to retrieve information from a specific version should still be fast. As the model will be constantly queried for information a slow access time would result in a slow performance overall.

The second dimension is the specification language. This is the language developers use to express their queries. The specification language needs support for temporal expressions, as the queries will span multiple versions. We discussed four different specification languages. SOUL is a declarative language we use to implement our tool. It can also be used to express queries. Computation tree logic and linear temporal logic are two temporal logics that can be used to express temporal queries. The last language is quantified regular path expressions, which allows developers to express the properties a path has to exhibit.

The last dimension is the detection mechanism. The detection mechanism is responsible for finding what parts of the model exhibits the characteristics specified by the specification language. We discussed both SLD and SLG resolution. Both resolution mechanisms allows us to easily implement all the considered specification languages.

7.1 Evaluation Strategy

Fritz et al. [18] conducted an empirical study to deduce questions developers commonly ask while developing software. We use the questions from this list that require information about the history of the software project. Some of these questions require a similar solution strategy, for example finding a version where a predicate holds. In this case we only answer one of those questions. We also added some questions taken from our own development expertise. These questions are denoted with an asterisk.

Afterwards we compare the different queries with the other languages.

Q1	What classes have been changed in the last version?
Q2	Who modified a class the last?
Q3	Who made changes to my classes?
Q4*	Which two classes are always modified together from a specific version?
Q5*	Which two authors committed at least two times in a row?
Q6*	What method got removed and is later introduced with the same name?
Q7*	What method got removed and is later introduced with a different name?
Q8*	What classes are introduced in the first version of a branch?
Q9*	Which class is replaced by another class?
Q10*	Is every method introduced with an associated unit test?
Q11	Which class has always existed?

7.2 SOUL as Specification Language

We implemented our tool in SOUL, as it is a declarative programming language that has extensive libraries to reason over code. We extended SOUL with predicates to retrieve information from our model in a declarative way. We implemented several specification languages on top of SOUL, but we can also use SOUL as a specification language. In this section we provide an overview of the queries expressed in SOUL.

7.2.1 Example Queries

Table 7.1 depicts the queries for the questions we want to answer.

Q1	1 ?latest **isMostRecentVersion**, 2 ?class **isClass** : ?latest, 3 ?class **wasChanged** : ?latest
Q2	1 ?version **isVersion**, 2 ?class **isClass** : ?version, 3 ?class **wasChanged**, 4 ?author **isAuthorOfVersion** : ?version, 5 not(?succ **isSuccessorOf**: ?version, 6 ?class **wasChanged** : ?succ)

We solve the query by finding a version where the class was changed. We then ensure that there exists no successor where that class is changed. This means that our version is the most recent version.

Q3

```
"ensuring this is a class we introduced"
?version isVersion,
?author isAuthorOfVersion : ?version,
?author equals: [ 'resteven' ],
?class isClass : ?version,
?previous isPreviousVersionOf: ?version,
not(?class isClass : ?previous),

"finding a version where the class got changed"
?changedVersion isSuccessorOf: ?version,
?class isClass : ?changedVersion,
?class wasChanged : ?changedVersion,
?changer isAuthorOfVersion : ?changedVersion,
not(?changer equals: ?author)
```

We first find the classes that we introduced. This is done by finding all versions we published (denoted by ensuring that the name of the author equals our username), and verifying that the class did not exist in a previous version. In the second part we look for versions that change our classes that are not published by us.

Q4

```
"find a version where both classes are changed"
?version isVersion,
?classA isClass : ?version,
?classB isClass : ?version,
not(?classB equals: ?classA),
?classA wasChanged : ?version,
?classB wasChanged : ?version,

"ensure that they are always changed together in the subsequent versions"
not(?other isVersion,
    not(?other equals: ?version),
    ?classA isClass : ?other,
    ?classB isClass : ?other,
    or(and(?classA wasChanged : ?other,
           not(?classB wasChanged : ?other)),
       and(?classB wasChanged : ?other,
           not(?classA wasChanged : ?other))))
```

We first find a version where both classes are changed together. We then use negation to ensure that there exists no successor where only one of the two classes are changed.

Q5

```
?versionA isVersion,
?authorA isAuthorOfVersion : ?versionA,
?versionB isNextVersionOf: ?versionA,
?authorB isAuthorOfVersion : ?versionB,
not(?authorA equals: ?authorB),
?versionC isNextVersionOf: ?versionB,
?authorA isAuthorOfVersion : ?versionC,
?versionD isNextVersionOf: ?versionC,
?authorB isAuthorOfVersion : ?versionD
```

Q6

```
?version isVersion,
?class isClass : ?version,
?method isMethodWithName: ?name inClass: ?class : ?version,
?removed isSuccessorOf: ?version,
not(?method isMethodWithName: ?name inClass: ?class : ?removed),
?reintroduced isSuccessorOf: ?removed,
?new isMethodWithName: ?name inClass: ?class : ?reintroduced
```

We only verify that a removed method is later replaced by a new method with the same name. We do not ensure that the semantics of this method are the same.

Q7

```
1  ?version isVersion,
2  ?method isMethodInClass: ?class : ?version,
3
4  "we bind the AST of the body of the method to ?body"
5  RBMethodNode(?,?,?body) isSmalltalkCodeOfMethod: ?method,
6  ?removedIn isSuccessorOf: ?version,
7  not(?method isMethod) : ?removedIn,
8  ?reintroducedIn isSuccessorOf: ?removedIn,
9  ?reintroducedMethod isMethodInClass: ?class : ?reintroducedIn,
10
11 "both AST's need to unify, meaning they are the same"
12 RBMethodNode(?,?,?body) isSmalltalkCodeOfMethod: ?reintroducedMethod
```

We make use of a SOUL predicate which provides the AST of a method. We only bind the body of this method to a variable. We later ensure that a method is introduced that has the same body. A method has the same body if both AST unify.

Q8

```
1  ?next isNextVersionOf: ?branchVersion,
2  ?class isClass : ?next,
3  not(?class isClass : ?branchVersion),
4  ?otherNext isNextVersionOf: ?branchVersion,
5  not(?class isClass : ?otherNext)
```

A branch point is a version that has more than one immediate successor. We can find classes that are introduced in a branch by looking what classes exist in one of those successors, but not in another one.

Q9

```
1  ?version isOldestVersion,
2  ?class isClass : ?version,
3  ?introduced isVersion,
4  ?refactorClass isClass : ?introduced,
5  not(?succ isSuccessorOf: ?introduced,
6     or(?class isClass : ?succ,
7        not(?refactorClass isClass : ?succ))),
8  not(?prev isPreviousVersionOf: ?introduced,
9     or(?refactorClass isClass : ?prev,
10        not(?class isClass : ?prev)))
```

We are looking for a class that got replaced by another class. As long as the refactoring class does not exist the original class has to exist, and once the refactoring class is introduced the original class must be removed. This is done by first finding a version where the refactoring class exists. We then ensure that there exists no successor where the refactoring class is removed or the original class exists. We also verify that there exists no predecessor where the refactoring class exists and the original class does not exist.

Q10

```
1  ?version isVersion,
2  ?method isMethod : ?version,
3  not(not(?unitTest isUnitTestFor: ?method : ?version))
```

We make use of a predicate *isUnitTestFor*. This predicate ensures that there exists a unit test for the given method[1].

[1]This information can be gathered from developer information, statical analysis etc. The implementation of this method lies outside the scope of this document.

Q11	1 ?first **isOldestVersion**, 2 "needed to get bindings" 3 ?class **isClass** : ?first, 4 ?succ **isSuccessorOf**: ?first, 5 not(not(?class **isClass** : ?succ))

Table 7.1: The queries for the example questions expressed in SOUL

7.2.2 Evaluation

SOUL has no knowledge of the temporal aspect of the queries. This means that it is the responsibility of the developer to manage the versions and in what version a predicate is evaluated. This results in two consequences.

First, every predicate is accompanied with a version. The pattern we notice is that a developer first specifies a version. He then uses this version in some conditions. The only query that differs from this pattern is Q8, where we specify two versions in the beginning. To remove this pattern we could use an implicit reference version in SOUL. The reference version would be the last version that is used in a query. The downside to this approach is that this lessens the declarative feeling of the language. In our opinion this outweighs the advantage and we do not opt for this option. Listing 7.12 depicts the query for Q1 rewritten using an implicit version.

```
1 ?latest isMostRecentVersion,
2 ?class isClass,
3 ?class wasChanged
```

Listing 7.12: Q1 rewritten using an implicit version in SOUL

Second, the temporal flow is hard to follow. One cause is that there is no proper indentation possible. This means that the user must disassemble each query to understand the meaning. This is not a problem for queries where the user is only interested in a version where a property holds. Q1 is an example that is concisely written and where we do not think any improvement is possible. Once the number of versions and the amount of relationships between versions increases the queries become hard to understand. Q9 is an example where both the flow and the meaning are hard to understand.

SOUL is ill-suited to express properties that have to constantly hold or hold between two versions. The only way to express this is to ensure that there is no such version where the property does not hold. Using negation means that no variable bindings will be created. Q4, Q10 and Q11 are examples where we use negation as failure to verify that a property holds continuously.

Expressing repetition cannot be done in a concise manner. Q5 is an example where we look for two occurrences of a pattern. A solution is to introduce a recursive predicate that checks for the repetition of a term.

7.3 Linear Temporal Logic as Specification Language

Linear temporal logic is a temporal logic that reasons over paths. A path is a sequence of subsequent versions. LTL evaluates predicates in the first version of the path. A developer can use combinators to traverse this path, hereby changing the version where predicates are evaluated in.

7.3.1 Example Queries

Table 7.2 depicts the queries for the questions we want to answer.

Q1

```
1 ?version isMostRecentVersion,
2 ?path isPathStartingFrom: ?version where:
3  and(?class isClass,
4      ?class wasChanged)
```

Q2

```
1 ?first isOldestVersion,
2 ?last isMostRecentVersion,
3 ?path isPathBetween: ?first and: ?last where:
4    finally(
5      and(
6        ?class isClass,
7        ?class wasChanged,
8        ?author isAuthorOfVersion,
9        next(
10         globally(
11           not(?class wasChanged)))))
```

Q3

```
1 ?version isOldestVersion,
2 ?path isPathStartingFrom: ?version where:
3  finally(
4    and(
5      next(
6        and(
7          ?class isClass,
8          ?author isAuthorOfVersion,
9          ?author equals: [ 'resteven' ],
10         finally(
11           and(
12             ?class wasChanged,
13             ?changer isAuthorOfVersion,
14             not(?changer equals: ?author))))),
15     not(?class isClass)))
```

First we look for a version we introduced. This is done by looking at next version and ensuring the class exists, while it does not exist in the current version. To prevent floundering we first look for a class that exists in the next version. We use another *finally* to ensure that that class is changed by another author.

<table>
<tr><td>Q4</td><td>

```
?version isOldestVersion,
?path isPathStartingFrom: ?version where:
 finally(
  and(
   ?classA isClass,
   ?classB isClass,
   ?classA wasChanged,
   ?classB wasChanged,
   next(
    not(
     finally(
      or(
       and(?classA wasChanged,
          not(?classB waschanged)),
       and(?classB wasChanged,
          not(?classA wasChanged)))))))
```

</td></tr>
<tr><td colspan="2">We first find a version where both classes are changed. We then ensure that both classes are always changed together by looking for a version where only one is changed, and negating that expression.</td></tr>
<tr><td>Q5</td><td>

```
?version isOldestVersion,
?path isPathStartingFrom: ?version where:
 finally(
  and(
   ?authorA isAuthorOfVersion,
   next(
    and(
     ?authorB isAuthorOfVersion,
     not(?authorA equals: ?authorB),
     next(
      and(
       ?authorA isAuthorOfVersion,
       next(
        ?authorB isAuthorOfVersion))))))
```

</td></tr>
<tr><td>Q6</td><td>

```
?version isOldestVersion,
?path isPathStartingFrom: ?version where:
 finally(
  and(
   ?method isMethodWithName: ?name inClass: ?class,
   next(
    and(
     not(?method isMethodWithName: ?name inClass: ?class),
     finally(
      ?reintroduced isMethodWithName: ?name inClass: ?class)))))
```

</td></tr>
<tr><td>Q7</td><td>

```
?version isOldestVersion,
?path isPathStartingFrom: ?version where:
 finally(
  and(?method isMethodInClass: ?class,
    RBMethodNode(?,?,?,?body) isSmalltalkCodeOfMethod: ?method,
    next(
     and(not(?method isMethodInClass: ?class)),
       finally(
        and(?reintroduced isMethodInClass: ?class,
          RBMethodNode(?,?,?,?body)
           isSmalltalkCodeOfMethod: ?reintroduced)))))
```

</td></tr>
</table>

<table>
<tr><td>Q8</td><td><pre>
1 ?version isOldestVersion,
2 ?path isPathStartingFrom: ?version where:
3 finally(
4   next(?class isClass)),
5
6 "we look for the branch point"
7 ?branchPoint isSecondToLast: ?path,
8
9 "second LTL query starting in branch point"
10 ?otherPath isPathStartingFrom: ?branchPoint where:
11 next(not(?class isClass))
</pre></td></tr>
<tr><td colspan="2">We can only write this query by writing two LTL expressions. The reason is that LTL can only reason over one path, and can not query multiple successors. We solve this by querying one of the successors in one LTL expression, and the other successor in a new LTL query.</td></tr>
<tr><td>Q9</td><td><pre>
1 ?version isOldestVersion,
2 ?path isPathStartingFrom: ?version where:
3 and(finally(?refactorClass isClass), "we need a binding"
4   until(
5   and(?class isClass,
6   not(?refactorClass isClass)),
7   and(?refactorClass isClass,
8   not(?class isClass)))
</pre></td></tr>
<tr><td>Q10</td><td><pre>
1 ?version isVersion,
2 globally(and(?method isMethod, ?unitTest isUnitTestFor: ?method))
3 holdsOnAllPathsStartingFrom: ?version
</pre></td></tr>
<tr><td>Q11</td><td><pre>
1 ?version isOldestVersion,
2 globally(?class isClass)
3 holdsOnAllPathsStartingFrom: ?version
</pre></td></tr>
</table>

Table 7.2: The queries for the example questions expressed in LTL

7.3.2 Evaluation

LTL is well-suited to express properties along a path. Both the use of combinators and proper indentation allows for concise queries that can be easily read and understood by the user. Examples are Q3 and Q6.

Queries become convoluted if we are only interested in one version where a property holds. The setup for the LTL expression requires more work than the actual expression, as can be seen in Q1.

The limitation to reason over one path also has its disadvantages. An example can be found in Q8, where we need two LTL queries. The reason is that we need to check a property on two different paths. Q2 is a more subtle example of this problem. We are looking for the person who modified a class the last. We implemented this by finding the version on a path between the first and last version where the class no longer changes. There could be a later version on a different path where the class is modified, yet LTL cannot detect it.

We can rewrite these queries using negation. An example is found in Q4, where we have to express a global property starting from a certain

version. LTL cannot verify this directly. We solve this by looking for a path where the property does not hold, and negating that expression. This is counterintuitive and does not provide variable bindings.

LTL provides no concise way to express repetitions along the path. Q5 is such an example. Generalizations of this query may not be expressible by LTL. An example question is to verify that every "even" version is published by one author, while every "odd" version is published by another.

Almost all the queries start with a *finally* combinator. The reason is that we are looking for a version where our expression has to hold, while our query start in the oldest version. We wrote down the queries this way as it is "stricter" to LTL. But SOUL is a declarative language, allowing us to rewrite the queries in a more concise way. The queries would be equivalent if we did not specify the start version and removed the *finally* combinator at the start of the query.

Another pattern we notice is an excessive use of *and*. The reason is that in the current implementation the combinators only have one argument. Verifying different properties in one version can thus only be done by using *and*. A solution is to allow combinators to have an arbitrary of comma-separated arguments that are evaluated in an implicit *and*.

7.4 Computation Tree Logic as Specification Language

Computation tree logic is a temporal logic, just as LTL. The difference is that CTL reasons over states instead of paths. It has the same combinators as LTL, but they are accompanied with a quantifier. This quantifier is either an existential or a universal one. Predicates are always evaluated in the current state, and the combinators can be used to change this state.

7.4.1 Example Queries

Table 7.3 depicts the queries for the questions we want to answer.

<table>
<tr><td>Q1</td><td><pre>1 ?last isMostRecentVersion,
2 and(?class isClass, ?class wasChanged) succeedsIn: ?last</pre></td></tr>
<tr><td>Q2</td><td><pre>1 ?first isOldestVersion,
2 eFinally(
3   and(
4   ?class isClass,
5   ?class wasChanged,
6   ?author isAuthorOfVersion,
7   aNext(
8   aGlobally(
9   not(?class wasChanged)))))
10 succeedsIn: ?first</pre></td></tr>
</table>

Q3

```
1 ?first isOldestVersion,
2 eFinally(
3   and(
4     eNext(
5       and(
6         ?class isClass,
7         ?author isAuthorOfVersion,
8         ?author equals: [ 'resteven' ])),
9     not(?class isClass),
10    eFinally(
11      and(
12        ?class wasChanged,
13        ?changer isAuthorOfVersion,
14        not(?changer equals: ?author)))))
15  succeedsIn: ?first
```

Q4

```
1 ?first isOldestVersion,
2 eFinally(
3   and(
4     ?classA isClass,
5     ?classB isClass,
6     ?classA wasChanged,
7     ?classB wasChanged,
8     not(?classA equals: ?classB),
9     aNext(
10      aGlobally(
11        or(and(?classA wasChanged, ?classB wasChanged),
12           and(not(?classA wasChanged),
13               not(?classB wasChanged)))))))
14  succeedsIn: ?first
```

We make use of the universal quantifier to ensure that both classes are either unchanged or changed together in every possible successor.

Q5

```
1 ?first isOldestVersion,
2 eFinally(
3   and(
4     ?authorA isAuthorOfVersion,
5     eNext(
6       and(
7         ?authorB isAuthorOfVersion,
8         not(?authorA equals: ?authorB),
9         eNext(
10          and(
11            ?authorA isAuthorOfVersion,
12            eNext(
13              ?authorB isAuthorOfVersion)))))))
14  succeedsIn: ?first
```

Q6

```
1 ?first isOldestVersion,
2 eFinally(
3   and(?method isMethodWithName: ?name inClass: ?class,
4     eNext(
5       and(
6         not(?method isMethodWithName: ?name inClass: ?class),
7         eFinally(
8           ?reintroduced isMethodWithName: ?name inClass: ?class)))))
9   succeedsIn: ?first
```

<table>
<tr><td>Q7</td><td><pre>
1 ?first isOldestVersion,
2 eFinally(
3   and(?method isMethod,
4   RBMethodNode(?,?,?,?body) isSmalltalkCodeOfMethod: ?method,
5   eNext(
6   and(not(?method isMethod)),
7   eFinally(and(?reintroduced isMethod,
8   RBMethodNode(?,?,?,?body)
9   isSmalltalkCodeOfMethod: ?reintroduced)))))
10 succeedsIn: ?first
</pre></td></tr>
<tr><td>Q8</td><td><pre>
1 ?first isOldestVersion,
2 eFinally(
3   and(eNext(?class isClass),
4   eNext(not(?class isClass))))
5   succeedsIn: ?first
</pre></td></tr>
<tr><td>Q9</td><td><pre>
1 ?first isOldestVersion,
2 and(eFinally(?refactorClass isClass),
3   aUntil(and(?class isClass, not(?refactorClass isClass)),
4   aGlobally(and(?refactorClass isClass, not(?class isClass)))))
5   succeedsIn: ?first
</pre></td></tr>
<tr><td colspan="2">We use aUntil to ensure that as long as the original class exists the refactoring class does not. Once the refactoring class is introduced it has to exist in all possible successors, while the original class does not.</td></tr>
<tr><td>Q10</td><td><pre>
1 ?first isOldestVersion,
2 aGlobally(
3   and(?method isMethod,
4   ?unitTest isUnitTestFor: ?method))
5   succeedsIn: ?first
</pre></td></tr>
<tr><td>Q11</td><td><pre>
1 ?first isOldestVersion,
2 aGlobally(?class isClass)
3   succeedsIn: ?first
</pre></td></tr>
</table>

Table 7.3: The queries for the example questions expressed in CTL

7.4.2 Evaluation

Most of the queries are very similar to the ones in LTL, namely the ones where we only use the existential quantifier are the same as in LTL. Examples are Q3, Q5 and Q6. We only discuss the queries that differ.

CTL shares some of the disadvantages of LTL. It has no concise way to express repetitions, as can be seen in Q5. But it allows us to reason over multiple paths at once. In LTL these were expressed using two different queries, while we can write them with one query using CTL. An example is Q8.

We use an implementation of CTL that is used as model checker. A model checker is not used to query the model, but to verify certain properties. This means that the implementation does not always return bindings for variables. In our setting these bindings are exactly what we are looking for. The reason we do not always get bindings is that we reduce the combinators to a minimum set. This reduction is done by using negation. This negation is exactly the reason we do not get bindings. Implementing

the correct semantics for these combinators instead of reducing them would solve this problem.

We also changed our original implementation so it no longer uses tabled resolution. We suspected that by storing previously computed answers we would get a speedup. The problem is that the transformational approach we use requires to compute all the answers for a tabled predicate. This differs from normal SLD resolution that returns the first found answer, and only continue the computation in case it backtracks. Computing all answers on a larger model takes too much time, making the runtime unacceptable. The tabled resolution for CTL is needed to prevent infinite computations due to cycles in the Kripke structure. In our setting there are no such cycles, and tabled resolution is not needed.

Not getting variable bindings and the unacceptable performance are not problems that are inherent in CTL but a result of the chosen implementation. These changes do not affect the queries, meaning we can still use the queries to evaluate the conciseness and expressivity of CTL.

7.5 Quantified Regular Path Expressions as Specification Language

Quantified regular path expressions allows us to describe a sequence of properties that have to hold on a path. It features regular expression like constructs that can be used to specify those properties.

7.5.1 Example Queries

Table 7.4 depicts the queries for the questions we want to answer.

<table>
<tr><td>Q1</td><td><pre>
1 ?recent isMostRecentVersion,
2 e(and(?class isClass, ?class wasChanged)) matches:
3   graph(versionTrans, ?recent, ?recent, <?class>)
</pre></td></tr>
<tr><td>Q2</td><td><pre>
1 ?first isOldestVersion,
2 e(manyOf>([true]),
3   and(?class isClass,
4   ?class wasChanged,
5   ?author isAuthorOfVersion),
6   manyOf<(not(?class wasChanged)))
7 matches:
8   graph(versionTrans, ?first, ?last, <?class, ?author>),
9 ?last isEndVersion
</pre></td></tr>
</table>

A path expression does not need to end in an "endversion". This is a version without any successors. In the example we are interested in the last person who modified the class. In case we do not require the path expression to stop in an endversion there might be a successor where the class is still modified by someone else. For this we introduced the predicate *isEndVersion*, which verifies that the version has no more successors. The `manyOf>([true])` statement skips an arbitrary number of versions. This is the equivalent of *finally* in LTL.

Q3

```
1  ?recent isMostRecentVersion,
2  "finding classes we introduced using reverseVersionTrans"
3  e(manyOf>([true]),
4    and(?class isClass,
5        ?author isAuthorOfVersion,
6        ?author equals: [ 'resteven' ]),
7    not(?class isClass))
8  matches:
9    graph(reverseVersionTrans, ?recent, ?start, <?class, ?author>),
10
11 "finding who made changes to this class"
12 e(manyOf<([true]),
13   and(?class wasChanged,
14       ?changer isAuthorOfVersion,
15       not(?changer equals: ?author)))
16 matches:
17   graph(versionTrans, ?start, ?changedVersion, <?class, ?author, ?changer>)
```

We make use of the predicate *reverseVersionTrans*. This predicate sets the reference version to a previous version instead of a successor. In the example we first look for a class we introduced. This is done by finding a version where the class does not exist, and ensuring it does exist in a successor. Due to floundering we first need bindings for the variable. The easiest solution is to use *reverseVersionTrans*. The query exists out of two path expressions, as we need the normal *versionTrans* for the second part of the query.

Q4

```
1  ?first isOldestVersion,
2  e(manyOf>([true]),
3    and(?classA isClass,
4        ?classB isClass,
5        ?classA wasChanged,
6        ?classB wasChanged,
7        not(?classA equals: ?classB)),
8    manyOf<(
9      not(
10       or(and(?classA wasChanged,
11             not(?classB wasChanged)),
12          and(?classB wasChanged,
13             not(?classA wasChanged))))))
14 matches:
15   graph(versionTrans, ?first, ?last, <?classA, ?classB>),
16 ?last isEndVersion
```

Q5

```
1  ?first isOldestVersion,
2  e(manyOf>([true]),
3    times(2,
4      ?authorA isAuthorOfVersion,
5      and(?authorB isAuthorOfVersion,
6          not(?authorA equals: ?authorB))))
7  matches:
8    graph(versionTrans, ?first, ?last, <?authorA, ?authorB>)
```

Q6

```
?first isOldestVersion,
e(?method isMethodWithName: ?name inClass: ?class,
  not(?method isMethodWithName: ?name inClass: ?class),
  manyOf<([true]),
  ?reintroduced isMethodWithName: ?name inClass: ?class)
matches:
  graph(versionTrans, ?first, ?last, <?name, ?class>)
```

Q7

```
?first isOldestVersion,
e(and(?method isMethodInClass: ?class,
     RBMethodNode(?,?,?,?body) isSmalltalkCodeOfMethod: ?method),
  not(?method isMethodInClass: ?class),
  manyOf>([true]),
  and(?reintroduced isMethodInClass: ?class,
     RBMethodNode(?,?,?,?body) isSmalltalkCodeOfMethod: ?method))
matches:
  graph(versionTrans, ?first, ?last, <?method, ?reintroduced, ?class>)
```

Q8

```
?first isVersion,
e(?method isMethod,
  not(?method isMethod))
matches:
  graph(reverseVersionTrans, ?first, ?branchPoint, <?method>),

e(not(?method isMethod),
  not(?method isMethod))
matches:
  graph(versionTrans, ?branchPoint, ?next, <?method>)
```

Path expressions can, just like LTL, only reason over one path. As we need to query two different successors of a version we have to use two path expressions in this query.

Q9

```
?first isOldestVersion,
?last isMostRecentVersion,

"only way to get a binding for ?refactorClass"
?refactorClass isClass : ?last,
e(manyOf(
    and(?class isClass,
       not(?refactorClass isClass))),
  manyOf(
    and(?refactorClass isClass,
       not(?class isClass))))
matches:
  graph(versionTrans, ?first, ?last, <?class, ?refactorClass>)
```

Q10

```
?version isOldestVersion,
not(
  e(manyOf([true],
    and(?method isMethod,
       not(?unitTest isUnitTestFor: ?method))))
  matches:
    graph(versionTrans, ?first, ?last, <?method>))
```

Q11

```
?first isOldestVersion,
"needed to bind the class"
?class isClass : ?first,
not(
  e(manyOf>([true]),
    not(?class isClass))
  matches:
    graph(versionTrans, ?first, ?last, <?class>))
```

Table 7.4: The queries for the example questions expressed in quantified regular path expressions

7.5.2 Evaluation

Quantified regular path expressions are limited to reasoning over one path throughout the version graph. This makes checking global properties hard. In Q11 we check for classes that have always existed, which can only be expressed with a double negation in path expressions. A query that has to reason over different paths must be split into several queries, just as with LTL.

Path expressions can only iterate once over a path. A query that starts and ends with a negation is hard to express as we encounter difficulties to get variable bindings. Q9 is such an example where we need to use SOUL to get bindings before writing our path expression.

Path expressions are very well suited to write repeated sequences of properties. Q5 is an example where path expressions is the only specification language where a concise query exists.

We notice the absence of the universal quantifier in our expressions. A reason is that the semantics of the quantifier are too strict, limiting the possible uses. The current semantics are that the expression has to hold on all the possible paths that start in the begin version. The problem is that these paths need to share the same end version, which is often not the case. A solution is to only consider paths between the begin and end version, disregarding the other ones.

7.6 Discussion

7.6.1 Properties in a Single Version

There are queries where we are only interested in finding a version where a property holds. An example is finding all the classes that have changed in a version. These kinds of queries are easily expressed in each specification language. The most complex part of the query is expressing the property, which is almost the same for every specification language. SOUL has an advantage that there is no setup required to specify this version. The disadvantage is that every predicate must be called with this version as an argument.

7.6.2 Global Properties

We are often looking for properties that continuously hold on all possible successors of a version. The only way to express those in SOUL is by using

double negation. The disadvantage is that those queries are counterintuitive and do not provide any variable bindings.

LTL can easily express if a property continuously hold on a path, yet has limited expressivity for properties that hold in all versions. LTL can verify that a property holds on all paths, but this only helps when the property has to globally hold starting in the start version. LTL cannot express that a property has to hold for all the successors of a version along the path. We can express this using negation just as in SOUL, resulting in the same drawbacks.

Path expressions can only verify properties that continuously hold on a path. Using double negation as in SOUL is not as straightforward with path expressions. This has to be done by finding a path where the property does not continuously hold, and negating that path expression. Examples can be found in Q10 and Q11. These expressions are not intuitive and provide no variable bindings.

CTL is the only specification language where we can easily express global properties by using the universal quantifier.

7.6.3 Sequence of Properties

We often have to describe subsequent versions for which some properties have to hold. In SOUL we can specify that a property has to hold on some subsequent versions by finding the first and last version of this sequence. We then use double negation to ensure the property holds on all versions between the beginning and the end of the path. We manage to express those properties, albeit in a complicated manner.

LTL and CTL are suited to express a sequence of versions where the same property has to hold in each version. This can be done by the *until* or *release* combinator. The *release* combinator is not provided in standard CTL, but can be easily implemented.

SOUL, LTL and CTL are not suited to express a sequence of versions where different properties have to hold. Q5 is example of such a query, where we have to chain the *next* combinator. We can express sequences over a limited amount of versions, but we are unable to do this when the number of versions is not known beforehand.

The only language where we can easily express a sequence of properties is path queries. The inherent support to evaluate each expression in a subsequent version is not available in other specification languages. We can use constructs as *manyOf* and *times* to evaluate a sequence of properties multiple times, even when the number of versions is not known beforehand.

7.6.4 Overlap between LTL and CTL

As previously mentioned there is a large overlap between LTL and CTL. We suspect that with the absence of infinite paths LTL is actually a subset of CTL. In case we use LTL to verify that an expression holds on one path an equivalent CTL expression can be constructed by using an existential quantifier. This means that an LTL formula is not easier to write than the equivalent CTL formula, and we see no reason to prefer LTL over CTL.

Some of the limitations are removed in CTL by using the universal quantifier. The limitations of LTL resulted in multiple complex LTL queries. Compare the query for Q8 in both specification languages, where CTL does not suffer from the same restrictions as LTL.

7.6.5 Complexity

SOUL is a declarative language. Declarative programming differs from object-oriented programming most programmers are familiar with. Developers that have programmed in similar declarative languages, like Prolog, should find SOUL to be easy and straightforward to use.

LTL and CTL are not commonly used by developers. Understanding the semantics and the subtle differences between the two specification languages may be difficult. Developers need to be able to quickly express a query, and first having to learn either LTL or CTL may be a daunting task for a novice developer.

Unlike LTL and CTL, regular expressions are widely spread and most developers are familiar with them. This and the descriptive approach of quantified regular path expressions results in a low adoption threshold for developers. Most queries can easily be expressed using path expressions, and we think this is an ideal language to quickly query a repository.

The specification language thus depends on the different stakeholders.

Absinthe for Team Leaders and Novice Developers

Developers can use Absinthe to answer the questions asked while developing and maintaining software projects. Team leaders can use Absinthe to verify software constraints and programming conventions. Both stakeholders have limited experience with declarative programming, and the steep learning curve for LTL and CTL can be too high to properly use the tool.

We propose quantified regular path expressions as an excellent specification language. Developers are acquainted with regular expressions, and the transition to quantified regular path expressions should be easy.

Absinthe for Tool Developers and Expert Developers

ABSINTHE can be used by tool developers, for example to integrate it in an IDE. Tool developers can implement queries for a predefined set of questions. Developers can then use these queries using their IDE without having to write the query themselves. These queries are written by expert developers that have knowledge of declarative and temporal programming.

We suggest a combination of path expressions and CTL as any query can easily be expressed by one of the two specification languages. LTL does not offer any advantage over CTL. Path expressions allow to concisely express queries that would be convoluted in CTL. CTL allows to express queries that reason over more than a single path that cannot be easily expressed using path expressions.

7.6.6 Conclusion

Each specification language can express most queries. The difference between each language is the conciseness for each query. These differ between the sort of query one wants to express. The complexity and learning curve for each specification language differs as well. SOUL and quantified regular path expressions are easily learned, while LTL and CTL have complex semantics.

Novice developers and team leaders can concisely express most queries using quantified regular path expressions. The learning curve is fairly low due to the similarities with regular expressions, which are commonly used by developers.

Tool developers and expert developers can use both CTL and path expressions. CTL can be used to form more complex queries, but has a higher complexity than path expressions. Tool builders could integrate these queries in an IDE so novice developers can use them as well.

8 Conclusion and future work

Software engineers need answers to a wide range of questions while developing current software projects. Fritz et al. [18] conducted an empirical study about what questions developers commonly ask. A series of these questions can only be answered by consulting the history of the software project. Such information can be found in version repositories, that have become industry best practice nowadays.

Program query languages have been used extensively [33, 14, 11, 27, 21] to reason over a single version of a software project. In contract, the research to query the history of a project is limited. We consider three different design dimensions for program query languages, the program representation, the specification language and the detection mechanism.

In this dissertation, we explored which configurations in the design space of a logic program query language support querying the history of a software project. We implemented each configuration in SOUL, a logic programming language similar to Prolog, but with dedicated features for querying a single version. These different configurations are bundled in the umbrella tool Absinthe.

8.1 Summary of the Dissertation

In this section we discuss the different logic meta-programming (LMP) configurations available in Absinthe. Each configuration is written as an extension of SOUL. They share a common representation of the information in a repository, stored in a model. They differ in specification language and detection mechanism.

We investigated how we can incorporate the specification language of each configuration within a logic programming language. We allow that these languages can be used in combination with other logic expressions. This also allows us to link different queries, preventing some of the limitations of the specification languages.

As an evaluation, we used each configuration to answer some of the questions that Fritz et al. [18] identified, in combinations with some questions from our own experience. We then used these queries to illustrate the strengths and weaknesses of each configuration.

We provide a summary of each considered configuration:

SOUL The first configuration we considered is using the base language SOUL as specification language. SOUL features SLD resolution as a detection mechanism, and allows the user to use "normal" declarative programming to specify his queries. In our evaluation, we observed that SOUL is suited to express queries requiring a property to hold in a single version. SOUL has no knowledge of the temporal aspect of these queries. This means that the user must explicitly denote what term is evaluate in which version. This makes queries spanning multiple versions very complex.

LTL The second configuration uses linear temporal logic as its specification language. LTL is a temporal logic that reasons over paths throughout a graph. Predicates are evaluated in an implicit version along this path. LTL features combinators that change this implicit version. We incorporated LTL expressions in SOUL through a meta-interpreter, which is resolved using SLD resolution. In our evaluation, we observed that LTL is suited to express properties that have to hold along a path. Expressing properties across different paths, such as global properties, cannot be easily done in LTL.

CTL The third configuration uses computation tree logic as its specification language. CTL is a temporal logic that reasons over nodes throughout a graph. It features similar combinators as LTL, but each combinator has a quantifier. We incorporated CTL expressions in SOUL through a meta-interpreter, which is resolved using SLG resolution. In our evaluation, we noticed that queries in CTL are very similar to the ones in LTL. They differ once we have to reason over different paths, which CTL can express concisely by using the aforementioned quantifiers. The drawback is that CTL has complexer semantics, and queries can become convoluted.

QRPE The final configuration uses quantified regular path expressions. Quantified regular path expressions features similar constructs as regular expressions. Quantified regular path expressions allows the user to

describe the properties a path has to exhibit. Quantified regular path expressions are implemented using a transformation from a path expression to a state machine. This state machine is resolved using SLD resolution. In our evaluation, we noticed that path expressions have a lower complexity than CTL, while they remain powerful enough to express most queries. Developers are familiar with regular expressions, lowering the learning curve for path expressions. Just as in LTL, quantified regular path expressions have troubles to express properties along different paths.

In general we consider quantified regular path expressions to be the easiest language for novice developers and team leaders. They have a lot in common with regular expressions. Most developers are familiar with regular expressions, resulting in a low learning curve for quantified regular path expressions.

Tool builders and expert developers may want to use CTL for some advanced queries. CTL has a higher complexity than the other specification languages, but is also more powerful.

8.2 Contributions

In this section we discuss the contributions of this dissertation. We make a distinction between our technical and the scientific contributions.

Technical Contributions

1. Several of the implemented configurations require the use of SLG resolution, while SOUL only featured SLD resolution. We extended SOUL with SLG resolution using a transformational approach [9, 28, 12]. Tabled rules can be transformed so they can be evaluated using SLD resolution. They make use of additional data structures that had to be implemented in the base language of SOUL, Smalltalk.

2. We incorporated three different temporal specification languages, LTL, CTL and quantified regular path expressions, into a logic programming language.

3. We provide an extensive list of queries that answer questions developers ask about the history of a software project, expressed in each considered configuration. This list is used to compare and evaluate the different configurations.

Scientific Contributions

1. We compared the aforementioned configurations to reason over the history and evolution of a software project.
2. As a result, we strongly suggest quantified regular path expressions as the specification language to reason over the history and evolution of a software projects.

8.3 Future Work

In this section we discuss directions of future work that address some of the limitations of our approach, or that build on the results obtained in this dissertation.

8.3.1 Technical Improvements to the Absinthe Prototype

SLG Implementation

The current SLG implementation transforms rules so they can be evaluated using SLD resolution. This approach requires that all answers for a query are computed before it can return the first answer. This differs from an independent SLG engine, which returns the first answer it encounters. We suspected the current implementation would work as the number of versions is limited, yet it does not scale due to the large amount of entities available in each version.

We opted for this approach as it allows us to reuse the existing SLD engine of SOUL. We will extend SOUL to include a new SLG engine that does not share the drawbacks of the current implementation.

Extending Path Expressions

There are several extensions and optimizations available for path extensions for quantified regular path expressions [29, 26]. An example of such an extension is the introduction of variable scoping. Variable scoping can be used to use a variable in just a subexpression. In the current implementation a variable is either global, or local to a single version. This limits the reuse of variables in repetitions, as these values need to be global and thus share the same value across repetitions.

8.3.2 Further Research

Empirical Study

So far the tool has been tested and used by several people at the software languages lab. We still lack input from developers in an industrial setting. An empirical study would show whether our tool can be easily used by developers to answer their questions. Such a study would illustrate the

usability of the tool, and what points need further refinement. This sheds light on what questions developers would actually answer using our tool, and what specification language they prefer.

Change-Resilient Unification

The amount of changed entities between two versions is limited. We suspect that we can reuse previously computed answers in one version for other versions as well. SLG resolution reuses previously computed answers for queries that were evaluated before. Whenever there is a call to a tabled predicate SLG resolution will first consult the tables to see if that query was evaluated before.

We think we can provide domain-specific equivalence for our model. In the current implementation two queries are considered to be equivalent if both the functor and the arguments (up to variable renaming) are equal. We can extend this to make unchanged entities across different versions equal. This would result that an answer computed in one version to be reused in a later version. The difficulty lies in finding dependencies between entities so our implementation remains correct.

Usability

The current syntax of path expressions is in a declarative style to integrate it with SOUL. Regular expressions use a different notation developers might be more familiar with. We will see if we can combine both the syntax of SOUL with a more regular expression-like syntax that remains intuitive and concise.

We want to provide a visual representation of the search and matching process of a path expression. A visual representation would indicate what parts of the expression were evaluated in what version. It also indicates which paths in the version graph matched an expression, and which failed and for what reason. The number of versions may be too high to fully represent the version graph. A solution could be to show only a representation of the general structure of the version graph. This could be done by only drawing the branch points of the version graph. The user could then focus on a specific part of the version graph.

New Applications within Software Engineering

We developed a tool that allows reasoning over multiple versions of a software project. This allows new applications within software engineering. We are interested in creating a tool that detects temporal bad smells. For example, a method that was not used in several versions may indicate that it is deprecated. Using this method in a later version may indicate faulty behaviour.

Similar checks can be integrated in the development of software. These checks would run when a developer wants to commit his changes, warning him of potential mistakes. These checks would consist out of temporal bad smells, but can also be specific for the stage of the software project. During the development of a beta-release one may require that only existing bugs are fixed. Such checks could be implemented by user-defined expressions.

The tool also allows extensions to be written for an IDE. An IDE could be extended with time aware browsing, allowing a user to browse and query the history of a software project.

Bibliography

[1] Git, the fast version control system. http://git-scm.com/, June 2011.

[2] Mercurial scm. http://mercurial.selenic.com/, June 2011.

[3] Monticello. http://wiresong.ca/monticello/, June 2011.

[4] Subversion. http://subversion.tigris.org/, June 2011.

[5] Visualworks. http://www.cincomsmalltalk.com/main/products/visualworks/, June 2011.

[6] A. V. Aho. *Algorithms for finding patterns in strings*, pages 255–300. MIT Press, Cambridge, MA, USA, 1990.

[7] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *System and Software Verification, Model-Checking Techniques and Tools*. Springer, 2001.

[8] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 511–520, New York, NY, USA, 2008. ACM.

[9] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43:20–74, January 1996.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[11] T. Cohen, J. Y. Gil, and I. Maman. Jtl: the java tools language. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 89–108, New York, NY, USA, 2006. ACM.

[12] P. C. de Guzmán, M. Carro, and M. V. Hermenegildo. A program transformation for continuation call-based tabled execution. *CoRR*, abs/0901.3906, 2009.

[13] O. de Moor, D. Lacey, and E. Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16:15–35, 2003.

[14] K. De Volder. Jquery: A generic code browser with a declarative configuration language. In *In Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006*, pages 88–102. Springer, 2006.

[15] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .net intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 133–144, New York, NY, USA, 2002. ACM.

[16] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[17] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[18] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[19] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 83–92, New York, NY, USA, 2011. ACM.

[20] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution: Research articles. *J. Softw. Maint. Evol.*, 18:207–236, May 2006.

[21] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *In ECOOP Proceedings*, pages 2–27. Springer, 2006.

[22] E. Hajiyev, M. Verbaere, O. de Moor, and K. De Volder. Codequest: querying source code with datalog. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 102–103, New York, NY, USA, 2005. ACM.

[23] A. Hindle and D. M. German. Scql: a formal model and a query language for source control repositories. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[24] A. Hindle, M. W. Godfrey, and R. C. Holt. What's hot and what's not: Windowed developer topic analysis. *Software Maintenance, IEEE International Conference on*, 0:339–348, 2009.

[25] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting Simultaneous Versions for Software Evolution Assessment. *Journal of Science of Computer Programming*, 12 2010.

[26] Y. A. Liu and S. D. Stoller. Querying complex graphs. In *Proceedings of the Eighth Intl Symposium on Practical Aspects of Declarative Languages (PADL*, pages 16–30. Springer-Verlag, 2006.

[27] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[28] R. Ramesh and W. Chen. Implementation of tabled evaluation with delaying in prolog. *IEEE Trans. on Knowl. and Data Eng.*, 9:559–574, July 1997.

[29] K. T. Tekle, M. Gorbovitski, and Y. A. Liu. Graph queries through datalog optimizations. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 25–34, New York, NY, USA, 2010. ACM.

[30] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000.

[31] V. Uquillas Gómez, A. Kellens, J. Brichau, and T. D'Hondt. Time warp, an approach for reasoning over system histories. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 79–88, New York, NY, USA, 2009. ACM.

[32] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[33] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, Belgium, January 2001.