Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Using Inductive Logic Programming to Derive Software Views

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Andy Kellens
Promotor: Prof. Dr. Theo D'Hondt
Co-promotor: Dr. Tom Tourwé
Adviseur: Johan Brichau
Juni 2003

Samenvatting

Software is niet statisch. Telkens er nieuwe eisen gesteld worden aan een applicatie of een nieuwe technologie in gebruik wordt genomen moet de implementatie van een programma aangepast worden. Wanneer een programmeur de opdracht krijgt deze aanpassingen te maken aan een applicatie is het belangrijk dat hij voldoende begrijpt hoe de software werkt. Deze informatie wordt hem verschaft door de documentatie van de software. Hier echter kunnen er problemen opduiken: vaak lopen documentatie en implementatie niet meer synchroon en kan de programmeur de documentatie niet meer gebruiken om te weten te komen hoe de applicatie in elkaar zit. De oorzaak van dit probleem is te vinden bij de hedendaagse ontwikkelingsomgevingen: ze bieden niet de mogelijkheid om gemakkelijk documentatie te maken die mee zal evolueren wanneer de implementatie veranderd.

In deze thesis stellen we de **Software Views Inducer** voor. Deze tool laat de gebruiker toe om op een simpele manier documentatie te creeëren uit Smalltalk code. Bovendien is deze documentatie robuust: wanneer er aanpassingen in de broncode worden gemaakt kan de documentatie semi-automatisch bijgewerkt worden. We toetsen de kwaliteiten van onze tool door te demonstreren hoe we met onze tool documentatie kunnen maken voor "design patterns" en tonen aan dat deze documentatie synchroon blijft wanneer we de implementatie van de patronen aanpassen.

Abstract

During its lifetime software has to evolve to meet new requirements or to work with new technology. It is important for a developer who has to change the implementation of a piece of software to have good insights into the application. To this extent, having high-quality documentation is invaluable. This documentation can also be a source of problems: in many cases it happens that the documentation is no longer synchronized with the implementation which renders it useless to a developer who has to make adaptations to the implementation. We can situate the cause of this problem with modern day development environments: they do not offer support for *easily* creating documentation that is *robust* when the implementation changes.

In this dissertation we propose the **Software Views Inducer**, a tool for creating documentation from Smalltalk code. Our tool does not only allow to easily create this documentation by simple drag & drop operations, but also offers documentation that is robust with respect to changes: whenever the implementation is adapted, the documentation can be semi-automatically brought back up-to-date. We validate these claims by using our tool to create documentation for design patterns and show that our documentation remains up-to-date when changes to these design patterns are made.

Acknowledgements

I would have never finished this dissertation without the help of a lot of people. Therefore I would like to express my gratitude towards:

Prof. Theo D'Hondt for promoting this thesis.

Tom Tourwé and Johan Brichau for being excellent advisors: not only did they come up with the subject of this dissertation but also guided me throughout the whole process. They helped me at every step of the way from giving suggestions while I was implementing the tool to proof-reading and correcting the mistakes in this document. Without their help this dissertation would have never seen the daylight.

Kris Gybels for proof-reading this document and giving valuable comments.

The researchers at the Programming Technology Lab and my fellow thesis-students for providing a fun and motivating environment to work in.

All my friends who made these past four years at the university a very pleasant experience and who made it easier for me to write this dissertation by distracting me once in a while.

The *Vrije Universiteit Brussel* and *Departement Informatica* for providing an excellent education.

Last but not least my parents for supporting me all these years and giving me the possibility to obtain a higher education.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Proposed Solution . . . 3
1.3 Outline of the dissertation . . . 3

2 Software Documentation Techniques **5**
2.1 Software Classifications . . . 5
2.1.1 Example of the usage of Software Classifications . . . 6
2.2 Intentional Software Views . . . 8
2.2.1 Documenting the Visitor Pattern . . . 9
2.3 Conclusions . . . 10

3 Logic Programming and Logic Meta Programming **12**
3.1 Logic Programming . . . 12
3.2 Logic Programming with SOUL . . . 13
3.2.1 Syntax of SOUL . . . 13
3.2.2 Using Smalltalk code in SOUL . . . 16
3.2.2.1 Smalltalk terms . . . 16
3.2.2.2 Smalltalk clauses . . . 16
3.2.3 Quoted terms . . . 17
3.3 Logic Meta Programming . . . 17
3.3.1 Definition . . . 17
3.3.2 Applications . . . 17
3.4 Logic Meta Programming with SOUL . . . 18
3.4.1 Representing Smalltalk entities in SOUL . . . 18
3.4.2 An Example . . . 19
3.4.3 Library for Code Reasoning . . . 19
3.4.4 An application of LiCoR . . . 20
3.5 SOUL and Intentional Views . . . 21
3.6 Summary . . . 23

4 **Inductive Logic Programming** **24**
4.1 Logic Programming Theory 24
4.1.1 Terminology 24
4.1.2 The syntax of logic programs 25
4.1.3 Properties of logic languages 27
4.1.4 Model theory 27
4.1.5 Proof Theory 28
4.2 Inductive Logic Programming 30
4.2.1 Definition of Inductive Logic Programming 30
4.2.1.1 An example 31
4.2.2 Properties of Inductive Logic Programming Algorithms 32
4.2.2.1 Top-down vs. Bottom-up 32
4.2.2.2 Representation of the background knowledge 33
4.3 Induction Algorithms 34
4.3.1 FOIL 34
4.3.1.1 Creating Candidate Literals 35
4.3.1.2 FOIL_Gain 36
4.3.2 Relative Least General Generalization 37
4.3.2.1 Definitions 38
4.3.2.2 Anti-unification 38
4.3.2.3 Computing the Least General Generalization 39
4.3.2.4 Relative Least General Generalization 39
4.3.2.5 An example with a lot of redundant literals . 39
4.3.2.6 Reducing the size of the clauses 40
4.3.3 Summary 42

5 **Software Views Inducer** **43**
5.1 Description of the tool 43
5.2 Making classifications and classifying entities 44
5.3 Choice of Induction algorithm 45
5.3.1 Implementation of RLGG 46
5.3.2 Extraction of the background information 48
5.4 Example usage of the Software Views Inducer 50
5.5 Summary 52

6 **Experiments** **53**
6.1 Inducing rules for documenting design patterns 53
6.1.1 Design Patterns 53
6.1.2 The Visitor Design Pattern 54
6.1.2.1 The pattern 54
6.1.2.2 The experiment 54
6.1.2.3 Inducing a more general rule 57
6.2 Abstract Factory Design Pattern 58
6.2.1 The pattern 58

6.2.2 The experiment 59
6.2.3 The Observer Design Pattern 62
6.2.3.1 The Pattern 62
6.2.3.2 The experiment 63
6.2.4 Discussion 65
6.3 Using Software Views Inducer to evolve the documentation . 67
6.3.1 Visitors 67
6.3.2 AcceptMethods 67
6.3.3 Discussion 68
6.4 Conclusion 68

7 Related Work **70**
7.1 Co-evolution of documentation and implementation 70
7.2 Code Browsing Approach 71
7.3 Using Machine Learning Techniques in Software Engineering 72

8 Conclusions **74**
8.1 Problem Summary 74
8.2 Contributions 75
8.3 Future work 76

A Background Information Framework **78**
A.1 Implementing the subclass background feature 79
A.2 Overview 80

B Implementation of Relative Least General Generalization **82**
B.1 The main predicate of the induction algorithm 82
B.2 Constructing the hypothesis 83
B.3 Anti-unification 84
B.4 Reduction of clauses 85
B.4.1 Removing the examples out of the body 85
B.4.2 Functional Reduction 85
B.4.3 Negative based reduction 86
B.5 Other predicates 86
B.5.1 coversEx 86
B.5.2 samePredicate 87

List of Figures

2.1 UML diagram of the visitor pattern 6

3.1 The representational mapping in SOUL 18
3.2 The different layer of LiCoR 19

4.1 Derivationtree of the example 29
4.2 a visual representation of the hypothesis space 32
4.3 the FOIL algorithm in pseudo-code 34
4.4 the clause so far and the literals considered by FOIL for addition to the clause 36
4.5 the functional graph of the example 41

5.1 a screenshot of the StarBrowser with 3 Learned Classifications 44
5.2 a screenshot of the classification of our example 50
5.3 a screenshot of the inductionwindow 51

6.1 UML class diagram of the Soul Visitor 55
6.2 a UML case diagram of the Abstract Factory design pattern . 58
6.3 a UML case diagram of the SOUL factory 59
6.4 a UML class diagram of the Observer design pattern 62
6.5 UML class diagram of the buffer example 63

A.1 a UML class diagram of the implementation of background features 79

List of Tables

2.1 A comparison of Software Classifications and Intentional Views 10

3.1 The layers with their groups and some predicates 20

4.1 The $FOIL_Gain$ calculated for two possible literals 37
4.2 A summary of the discussed ILP approaches 42

5.1 The applicability of the background information 49

6.1 the execution times of the induction algorithm for each induced classification 65

Chapter 1

Introduction

This thesis is about the problems that may occur when the documentation and the implementation of software are no longer synchronized. We will introduce a new technique that allows the creation of documentation that is robust with respect to changes in software while keeping it simple for a developer to create it. To accomplish this we are going to use some techniques from machine learning.

1.1 Problem Statement

Software is not static. During the lifetime of an application, changes are constantly necessary to fix bugs, make the software compatible with new technology, implement new or changed requirements, . . . Whenever changes to the application have to be made, a developer needs to have sufficient insights into the implementation to be able to execute the necessary adaptations. The original software is almost never written by a single programmer. In most situations the application is developed by a complete software-engineering team. For this team, it is considered good practice to keep extensive documentation of the written software. It is not unthinkable that the developer who has to make the changes was not a member of the original engineering team. Even if the developer was, it is unlikely that he knows every little detail of the implementation. To help the developer gain the necessary insights, the documentation of the piece of software is important.

This documentation can be the subject of a lot of problems. In an extreme case it can have gotten lost over time and the developer has to start from scratch to understand how the application works. In most cases the documentation is no longer synchronized with the implementation. If a developer uses this outdated documentation, then he may obtain faulty insights into the software which makes it impossible to make the necessary changes. This can happen for a number of reasons. In the early phases of the development

process, the software engineering team starts with creating a blue-print of the application (this is called the design). This design is considered an important part of the documentation. It gives a view of the software by describing which entities construct the software and how they work together. When the programmers start writing the application, the design is used to direct the implementation. While implementing the software, it may happen that the ideas from the design appear not to suffice to make the program work. The developers will encounter places where they need to make (often ad-hoc) changes to the design in order to fix these problems. When this happens, updating the design documentation is often forgotten or neglected.

Once the documentation is out of sync with the implementation a cascade of problems will arise. If a developer starts with an application with already outdated documentation and makes changes to it without reflecting these changes back into the documentation, then over time the quality of the documentation will diminish even more until at a certain point in time the documentation is useless. This is quite a large problem with respect to building software since it makes it difficult to produce a system that is easy to adapt and that keeps being easy to adapt after it has been changed a couple of times.

One could argue that the cause of this problem is situated with the developers who are lazy and negligent since they do not update the documentation every time they change the implementation. In fact this is more a consequence than a cause of the problem. The job of updating the documentation is tedious for the developer and can be in some way automated. The real problem is the way software gets documented: composing high-quality documentation and, especially, keeping it up-to-date with the implementation are not a part of the development process. Most modern integrated development tools do not offer the developer a way to easily create documentation that keeps being consistent when the implementation is changed.

A lot of research effort has already been put in creating a system for better documentation. We can distinguish two different kinds of tools:

- A first kind of tool allows a developer to create documentation easily from the source code but it does not offer possibilities to keep that documentation automatically synchronized with the implementation. This kind of tool also lacks the possibility of expressing the intention behind the documentation: even if the system is documented correctly and the documentation is up-to-date, it sometimes remains hard for a developer to understand what is meant by a certain piece of documentation. An example of this approach is creating UML diagrams with tools like *Rational Rose*.

- A second kind of tools are robust with respect to changes in the implementation and allow the expression of the intention behind the documentation, but they require a developer to express the design in a special kind of language or other formalism. This process can be error-prone and quite tedious since it requires the programmer to have extensive knowledge about the application. A good example of this technique is *Intentional Views* [MMW02a], which we will discuss in the next chapter. Most tools in this category are academic research: there are no industrial tools that allow the co-evolution of design and implementation.

1.2 Proposed Solution

In this dissertation we will introduce a new documentation tool that helps a developer to create documentation out of source code that is easy to keep in sync with the implementation. We want to create a tool that:

- Offers an easy way to document software: the developer can create documentation from the source code by means of simple operations (eg. drag & drop) without having to know about the complex implementation details.
- The obtained documentation is more robust with respect to changes in the implementation and will evolve together with the source code.

We accomplish this by offering a tool with the following functionality:

- The developer manually documents the software with the tool.
- The tool will extract information out of the documentation that gives a description of the structures and relationships in the source code. The tool then uses this information to update the documentation when changes in the implementation occur.

To extract this information out of the source code we are going to use some techniques from the area of Machine Learning [Mit97]. More specifically, we are going to apply Inductive Logic Programming (ILP) to our problem. ILP is a method that allows the creation of first-order logic rules out of examples plus background information. It is a mature technique that has proven to be useful in fields like molecular biology and also in other disciplines of software engineering.

1.3 Outline of the dissertation

In the rest of this document we are going to explain our technique and we are going to validate it by means of a few experiments. The structure

of this dissertation is the following: in chapter 2 we will take a look at two other documentation techniques and we will discuss their benefits and disadvantages. We will take a look at how we can improve on them. Chapter 3 gives an introduction to Logic Programming and the logic language SOUL. It will also take a look at Logic Meta Programming and give a few examples of it by means of SOUL. We start chapter 4 with an overview of the more theoretical background concepts behind logic programming to allow us to give an introduction to Inductive Logic Programming. We will also take a look at a few ILP algorithms. In chapter 5 we will take a look at the Software Views Inducer, the tool we built in the context of this dissertation. We will also briefly discuss its implementation and design decisions. Chapter 6 will describe the experiments we have conducted in order to validate our technique. We will take a look at some related work in chapter 7 and we will finish this dissertation with chapter 8 where we will draw conclusions from the experience we gained during this research. We will take a look at some future work which might be done to improve our technique and make it more applicable to real-life situations.

Chapter 2

Software Documentation Techniques

In this chapter we take a look at two different techniques for creating documentation for a piece of software, namely *Software Classifications* and *Intentional Software Views*. We discuss the advantages and disadvantages of both techniques and take a look at how we can use the advantages of both for creating a good documentation technique.

2.1 Software Classifications

Software Classifications is an approach introduced by De Hondt [DH98] in his dissertation. The idea behind this approach is to offer a developer an easy way to create documentation that can be used to understand the software. To this extent the notion of a *Software Classification* is used. Such a classification is a simple container which is used to hold software entities as its items. The same entity can be an element of multiple classifications. Software Classifications is more then a model for representing documentation, it is also a technique for creating that documentation. There are four different methods for classifying software artifacts, as discussed in De Hondt's work:

- **Manual classification** The developer manually puts software entities in a classification. This is the simplest way to create classifications.
- **Virtual classifications** The items of the classification are obtained by using a feature of the development environment. An example of this in Smalltalk is creating a classification out of a given protocol. The classification will then for instance contain all the methods of that protocol.
- **Classification with advanced navigating tools** A developer often

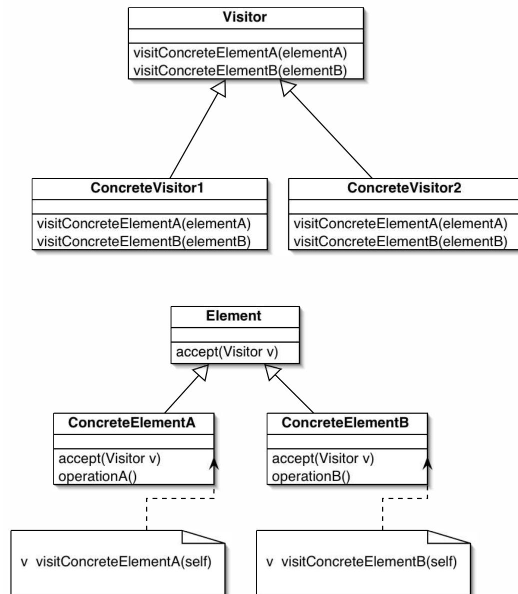

Figure 2.1: UML diagram of the visitor pattern

wants to create a classification containing elements with a certain relationship. Special navigation tools (like for instance class browsers) can be used to guide the developer through the code based on certain relationships. While browsing the code the developer then can decide which elements belong to a classification.

- **Classification by means of method tagging** The idea behind this way of creating classifications is that a developer knows the methods where he/she makes changes to the code. We can make the information about these places implicit by tagging the methods to which changes are made. A classification can then be generated out of these tags.

2.1.1 Example of the usage of Software Classifications

Now let us take a look at how we can use Software Classifications to create documentation for a piece of code. As an example we are going to document the *Visitor design pattern* [GHJV95] by means of classifications. The visitor design pattern is a behavioral pattern that is used to create a looser coupling between a hierarchy of objects and the operations that are defined on that hierarchy. Since we will also use this example again in a later chapter, we will take a more in-depth look at it. If we look at figure 2.1 we can see a UML class diagram of the pattern. The object hierarchy on which we want to implement an operation is represented by the abstract class

`Element` and its subclasses `ConcreteElementA` and `ConcreteElementB`. The operations on the hierarchy are implemented by the `Visitor` class and its subclasses. These subclasses of `Visitor` implement a method for each type of Element in the hierarchy (we can see the `visitConcreteElementA` and `visitConcreteElementB` methods). This method implements the behavior of the operation on that specific `Element`. The subclasses of `Element` all implement an accept method. This accept method takes a `Visitor` as its argument and will call the method on the `Visitor` corresponding with the kind of `Element`. This is the so-called double dispatch: the executed method will not only be selected by dispatching on the kind of `Visitor`, but is also dependent on the kind of `Element`. The advantage of this pattern is that adding new operations can easily be done by subclassing the `Visitor` class and implementing the behavior in the methods.

To document this pattern using Software Classifications we create the following classifications:

- AbstractVisitor
- ConcreteVisitors
- AbstractProduct
- ConcreteProducts
- AcceptMethods

We then will put software entities in the corresponding classifications. For example, we put all the classes that implement an action of the element hierarchy in the *ConcreteVisitors* classification. All the methods on elements of the hierarchy that implement the acceptance of a visitor and will call the corresponding method on that visitor will be put in the *AcceptMethods* classification,...

The obvious advantage of this technique is that creating the documentation is easy: we manually classified the elements of a Visitor pattern. In the tools that implement Software Classifications this can be done by drag & drop. Software Classifications also have a few major disadvantages:

- An important aspect of documentation tools is wether the created documentation is robust to changes: every time the implementation changes, the documentation should change too. If we take a look at the robustness of the documentation we created we see that, if we add for instance a new *ConcreteVisitor* or delete a *ConcreteProduct*, we will have to manually update the elements of the classification. This is exactly the kind of problem we want to avoid when making changes to a piece of software.

- Also, we do not know if we have documented all the correct elements of a classification. It might happen that we have forgotten to classify a few *ConcreteProducts*. This may lead to incomplete documentation.

- Design patterns are well-known and widely understood software structures so a developer who looks at the classifications we made can get a good idea of what is meant with the documentation. However, if we document lesser known things with Software Classifications, the technique does not offer a way to give clear insights into the intention of the classifications. In such a case, the developer would have to browse the elements of a classification and try to reveal what they have in common to be able to understand the piece of software. This is not always a trivial task.

2.2 Intentional Software Views

Intentional Software Views [MMW02a] are an extension to the software classifications we described above. The main difference between the two techniques is that intentional views will use a **description** of the software entities that belong to the classification instead of just enumerating all the elements in the classification. An interesting feature of Intentional Views is that they allow us to create multiple views for the same classification. This allows us to create better documentation since we can use these multiple views to check the mutual consistency of our views. We then can use this information to validate that the views are correct. Before we are going to use Intentional Views to create documentation for the Visitor pattern, we are going to make the reader a bit more familiar with the concept by means of a small example. Suppose we want to document the set {1,2,3,4,5}. We define a view by providing a high-level description of the items in the view. As we already mentioned earlier, we can have multiple descriptions for the same concept. For our simple example we could have for instance the following descriptions:

- The integer larger than 0 and smaller than 6.

- The difference of the set {1,2,3,4,5,6,7} and the set {6,7}

- ...

Notice that if we use the two descriptions above to generate the elements in the view, we would obtain in both situations the same set. Now that we have provided the reader with an intuition for Intentional Views, let us take a look at how we can use them to document the Visitor pattern.

2.2.1 Documenting the Visitor Pattern

Now let us apply Intentional Views to create documentation of the Visitor pattern. For every classification we created in the section about Software Views, we will provide a View for which we give an executable description.

- **AbstractVisitor** Class from which all the ConcreteVisitors inherit. This class specifies the interface for the ConcreteVisitors (a method corresponding to every ConcreteProduct).
- **ConcreteVisitors** Classes that inherit from the AbstractVisitor and that implement an operation on the elements of the hierarchy.
- **AbstractProduct** Abstract class for the elements of the hierarchy.
- **ConcreteProducts** Classes that inherit from the AbstractProduct class and that implement a method for accepting a Visitor. These classes are the elements of the hierarchy on which we want to define operations.
- **AcceptMethods** These methods are implemented on the ConcreteProducts. They implement the double-dispatch:, given a ConcreteVisitor, they will call the method corresponding with the ConcreteProduct on which they are implemented on the Visitor.

If we want to know which software elements belong to the view, we can use the description to calculate them out of the source code. We can notice the following advantages of this approach:

- The documentation we presented here is robust with respect to changes. If we for instance add a new operation to the hierarchy by implementing a new *ConcreteVisitor*, and we re-calculate the elements then the new class will appear as an element of the view. With Intentional Views, it is easy to keep the documentation up-to-date with the implementation.
- Although we are not sure that the description we provided will include all the intended elements,the chance that the resulting documentation is incomplete will get smaller. If we can provide a good description of a view, we can not erroneously forget a software entity. By using multiple descriptions for the same view, and checking that every description results in the same set of elements, we can limit this problem even further.
- One of the problems of Software Classifications is that the intention behind the documentation got lost. We do not have this problem with Intentional Views. If a developer takes a look at the description, he/she will immediately understand what is meant with it: the description reveals the intention behind the view very well.

	Software Classifications	Intentional Views
Easy to document	X	
Complete		X
Robust documentation		X
Intention present in documentation		X

Table 2.1: A comparison of Software Classifications and Intentional Views

Intentional Views also have a few downsides:

- In this example we have provided the descriptions in natural language. In practice, a developer will have to write a description in some kind of formal language (in a later chapter, we will take a look at an example of such a language). Besides from the fact the developer has to learn this language, writing this description can be error-prone, as is with writing any computer program. If the developer does not give a correct description of the elements he/she intends to be in the view, then the resulting documentation will be faulty.
- The developer needs to have extensive structural knowledge about the elements of the view that is being described in order to write down the high-level relationships between these elements. In a lot of cases this will make it very hard to write a description since the relationships between the elements are not always as clear to see. Intentional Views also leads to a chicken-or-the-egg situation: in order to create documentation that will help us to better understand a software system, we already have to know a lot about the software we are trying to write a description for.

2.3 Conclusions

If we take a look at table 2.1 we see a comparison between Software Classifications and Intentional Software Views. Both approaches are interesting techniques for creating documentation but they differ in the quality of documentation and the ease of creating it. If we take a look at the comparison we can see that both approaches are in fact complementary. In this dissertation we will show how we can create a documentation tool that unites the advantages of both Software Classifications and Intentional Software Views. The tool we will introduce allows the user to easily create documentation without having to possess extensive knowledge about the relationships between the elements of the documentation. Also, the documentation is more robust with respect to changes in the implementation and allows a developer to see the intention behind the documentation without too much difficulties. In

this chapter we only discussed these two approaches for creating documentation since they are relevant for describing the tool we developed. We did not encounter any industrial tools which are integrated in the development environment and allow the creation of robust documentation. In chapter 7 we will discuss other (academic) approaches for solving the problem of outdated documentation

Chapter 3

Logic Programming and Logic Meta Programming

In this chapter we take a look at the *Logic Programming* paradigm. By means of a logic programming language,*SOUL*, and some examples, we will explain the basic concepts of such a language. Furthermore we discuss the topic of *Logic Meta Programming.*

3.1 Logic Programming

Before we take a look at how we can write programs in a logic programming language, we are going to discuss the logic programming paradigm. In order to do this, let us compare it with some other programming paradigms:

- **Imperative programming** The typical property of imperative languages is that programs written in them have some sort of state. A program is written by specifying a number of steps which, at time of execution, manipulate that state of the program. Programs in both *procedural* as *object-oriented* languages are generally written in imperative style. Examples of this paradigm are C++, C, Pascal, Java, ...

- **Functional programming** This paradigm can best be compared with mathematical functions. The language allows the construction of functions which transform a given entity and will result in a new entity. A program is written by specifying a chain of transformations. LISP and Scheme are both examples of functional programming languages (if we not consider the destructive operations). An example of a pure functional language is Haskell.

- **Logic Programming** The Logic Programming paradigm is based on first-order predicate logic. Programs in a logic language are written

by specifying the base knowledge that is available about a problem and the relationships between this knowledge as so-called *facts*. The part of the program that will derive new information out of these facts consists out of *rules*. These rules are used to deduce new facts out of already existing ones. Examples of such languages are PROLOG and SOUL.

We will now take a closer look at Logic Programming (LP), the paradigm we are interested in. One of the advantages of LP is that the programs written in it are easy to understand. Programs written in a logic programming language specify **what** is needed to be computed instead of **how** it has to be computed. In the following section we will take a look at the basic concepts of a logic language by looking how logic programs are written in such a language: SOUL.

3.2 Logic Programming with SOUL

Many implementations of logic languages exist, of which PROLOG [DEDC96] is probably the most famous [Fla94].
The *Smalltalk Open Unification Language* (SOUL) is a PROLOG-like logic language developed by Roel Wuyts at the *Programming Technology Lab* in the context of his PhD-research [Wuy01]. SOUL is implemented in the object-oriented language Smalltalk [GR89].

3.2.1 Syntax of SOUL

We are going to show the syntax of SOUL by means of some examples. Consider the following situation: we have some information about a set of people and know who is the parent of who. Suppose we want to write a small program that will calculate the grandparent relationship between those people. We can express this as the following SOUL program:

```
parent(jim,bob).
parent(bob,julie).
parent(bob,eric).
grandParent(?x,?y) if
      parent(?x,?z),
      parent(?z,?y).
```

The first three lines of our example express some base knowledge we have about our example (namely that jim is a parent of bob, that bob is a parent of julie,...). We call this information **facts**. The last three lines of our example form a **rule** that defines the grandparent relationship. Variables in this rule start with a question mark (?). Notice that this rule does not say

how we have to compute the grandparent relationship. It gives a definition of it: someone ($?x$) is the grandparent of someone ($?y$) if there is another person ($?z$) such that person $?x$ is the parent of $?z$ and $?z$ is the parent of $?y$. As the reader will remark, this logic program is a very intuitive definition of the problem we want to express. If we want to know who is a grandparent of who we would pose a query to the SOUL interpreter. In this case the query would look like:

```
if grandparent(?x,?y).
```

The logic language will then process the query (in a later chapter we will take a brief look at how this is done) and will output:

```
{?x → jim, ?y → julie}
{?x → jim, ?y → eric}.
```

Above we can find the variable bindings that the logic language will return. Not only a single solution is returned. Instead the query will result in a set of variable bindings for every possible solution. Every such binding exists out of a variable name and a value for that variable. If we take a look at our example we see that if we take as value for `?x` `jim` and for `?y` `julie`, then this pair would be a correct result of the `grandParent` relationship: if we look at our example we see that `julie` is a grandchild of `jim`. Now that we have shown the basic notions of LP with SOUL, we will explore some other features of the language. SOUL also has a native data structure: the list. The following example shows how we can implement the append of two lists.

```
append(<>,?Y,?Y).
append(<?X | ?Xs>,?Y,<X | Z>) if
      append(?Xs,?Y,?Z).
```

The first rule expresses the base case: the append of the empty list (<>) to another list $?Y$ is the list $?Y$. The second rule is the recursive definition of the append: we can append two lists by taking every element but the first one from the first list and appending it to the other list. This recursive step can be seen in the body of the second rule where the append predicate will call itself. Since the first list of the append will get smaller every step, we will reach the base case at some time. Notice that lists are written down between $<$ and $>$ and the elements are separated by a comma. We also see a list of the form `<?X | ?Y>`. The part before the | is bound to the first element (the head) of the list, the part after it is bound to a list containing the rest of the elements (the tail). Let us illustrate this with an example: `<1,2,3>`. If we unify this list with `<?X | ?Y>` then $?X$ will bound to 1 and $?Y$ will be bound to `<2,3>`. As the reader already may have noticed,

the append program does not consist out of a single rule but has two rules who describe the problem. SOUL will try every rule when constructing a solution. These rules will be applied in order of specification (from top to bottom). If we would pose a quey like `if append(<1>,<2>,?x)` then SOUL will try to apply the first rule (the base case). Since our query and the head of the first rule can never be the same (the $<>$ can never match the $< 1 >$) the second rule will be tried. This second rule can be matched with the query and the body of the rule will be called.

Lists are not the only data-structure we can use in SOUL. SOUL also supports the notion of *functors*.

```
successor(0,s(0)).
successor(?x,s(?x)).
```

To explain this example we are going to define our own notation of natural numbers and the successor relationship on those number. We represent a number unary: the number 3 is represented by s(s(s(0))). We say that $s(?x)$ is the successor of $?x$ (so we add a 1 to a number by adding an extra s to the representation). For example, the successor of s(s(0)) (the number two) is s(s(s(0))) (three). If we take a closer look at the two rules that describe the successor, we see that they contain function symbols in the arguments of the rule. These function symbols are called *functors*. Functors can be compared with normal predicates, except that they never get evaluated and thus only have a structural meaning.

```
minimum(?x,?y,?x) if smallerThen(?x,?y),!.
minimum(?x,?y,?y).
```

This last example is a implementation of a rule that computes the minimum of two numbers. Suppose we want to find the minimum of 3 and 5. We would pose the query `if minimum(3,5,?x)`. As we already discussed with the append example, all the matching rules will be tried when finding a solution so the variable $?x$ in this example will bind to 3 and $?y$ to 5. But if we run this query in SOUL, we would only get the value 3 as a result. This is because of the cut-operator ”!” which we can find in the first rule. The effect of this operator is that when it is encountered none of the other possibilities of the rule with the same head (the part of a rule before the ”if”) will be tried while proving the query. Also all the alternatives of the literals before the cut will be pruned. If we take a look at our example, SOUL will first try the first rule (which matches) and the body of that rule will be evualted. We see that `smallerThen(3,5)` will succeed and that we reach the cut. The effect of the cut is that all the alternatives of the minimum rule (in this case only the second rule) will not be tried and that the example only has one

output, namely the number 3. Cuts are mostly used in logic languages to optimize the programs written in them by canceling out alternatives of a rule. A programmer should always be weary when using cuts, since in some cases they change the pure logic meaning of the program. This example is one of those cases: if we omit the cut-operator then the minimum predicate will have a totally different output. We explained the cut operator here since its understanding is important when we will discuss the implementation of our induction algorithm in a later chapter.

3.2.2 Using Smalltalk code in SOUL

3.2.2.1 Smalltalk terms

SOUL offers a certain symbiosis between itself and the underlying Smalltalk environment in which it is implemented. The programmer can make use of Smalltalk entities from within a SOUL program by using *Smalltalk terms* [DMBM02]. Smalltalk terms are represented by Smalltalk code (with possible logic variables) denoted between '[' and ']'.

```
getFirstOutCollection(?collection,?first) if
      equals(?first,[?collection at:  1]).
```

The above predicate takes a Smalltalk collection as input and will bind the value of the first element of that collection to the variable ?first. When the predicate is called the ?collection variable should be bound to a Smalltalk object (namely an object that implements the `at:` selector). The value of the ?first variable is also a Smalltalk object. This example shows that we can use Smalltalk entities in SOUL programs and that Smalltalk terms can contain logic variables which will be substituted in the Smalltalk code.

3.2.2.2 Smalltalk clauses

We can extend the idea of using Smalltalk terms in a logic program to using *Smalltalk clauses*. These clauses do not differ much from Smalltalk terms. The only difference between the two is that Smalltalk clauses do not need to appear in a logic clause. Instead they can be used as such a clause. This property implies that they have to evaluate to a boolean.

```
greaterThan(?x,?y) if
      [?x > ?y].
writeString(?string) if
      [Transcript show:?string.true].
```

The two examples above use Smalltalk clauses. Notice that with the second

predicate (`writeString`) we explicitly have to specify the `true` at the end of the clause since the Smalltalk clause has to evaluate to a boolean.

3.2.3 Quoted terms

The last peculiarity of SOUL we will discuss are *Quoted terms*. Quoted terms can be compared to strings with the exception that every occurrence of a variable will be replaced by the binding of that variable (much like the quotations in Scheme). Quoted terms are denoted between '{' and '}'.

```
generateStPrint(?class,?code) if
     equals(?code,
            {Transcript show:?class asString}).
```

The above predicate will generate a string representation of Smalltalk code for printing the name of a class on the Transcript. This shows a great use of Quoted terms: code generation. Soul will not evaluate the Smalltalk code in the quoted term but will substitute the values of the variables in the term. If we for instance launch the query `if generateStPrint([Object],?code)` then the variable ?code will bind to {Transcript show:Object asString}.

3.3 Logic Meta Programming

3.3.1 Definition

We define Logic Meta Programming (LMP) as [PRO]:
the use of a Logic Programming Language at Meta level to reason about and manipulate programs built in some underlying base language. With this we mean in our context that we are going to write logic programs that will reason about and adapt programs written in an object oriented language. LMP is a technique that is developed at the *Programming Technology Lab*(PROG) where research is being done on how LMP can be used to create state-of-the-art software development support tools.
The logic programming part of this definition should be clear by now. We say that LMP is a meta programming approach since the support tools are at the meta level with respect to the source code they reason about or make changes to.

3.3.2 Applications

LMP has already been used for a lot of different applications in the context of creating development support tools, but we can put all of them in one of the following five categories:

Figure 3.1: The representational mapping in SOUL

- Verification of source code (eg. conformance checking, coding conventions [MMW01],design models [Wuy98], architectural description [Men00a]
- Extraction of information out of source code (eg. code metrics [MD01]
- Transformation of source code (eg. refactoring,translation,evolution [MT01].
- Generation of source code [Wuy01]
- Aspect Oriented Programming [Bri00] [DVD99]

3.4 Logic Meta Programming with SOUL

As we already discussed in a previous section, a strong interaction is possible between SOUL and the language it is implemented in: Smalltalk. This is not a coincidence. In fact, SOUL was designed to be a LMP-language.

3.4.1 Representing Smalltalk entities in SOUL

Before we can show how meta programs can be written in SOUL, we should take a look at the way Smalltalk terms are represented in SOUL. Most of the Smalltalk entities can be represented by a logic representation of their parse tree. For instance, the Smalltalk statement `self add: 1` will be translated into the SOUL term `<send(self,#add:,1)>`. A Smalltalk entity like a class, which does not have a parse tree representation is represented by a functor with five arguments: the class, its name, the names of its arguments, the names of its temporary variables and the statements of the class.

SOUL has to have access to the Smalltalk entities. To achieve this a meta-language interface (MLI) is used (see figure 3.1). Every time SOUL needs a Smalltalk element, it will call the MLI that will return the SOUL representation of the element. The usage of an MLI allows that SOUL can easily be ported to use a different meta-level language.

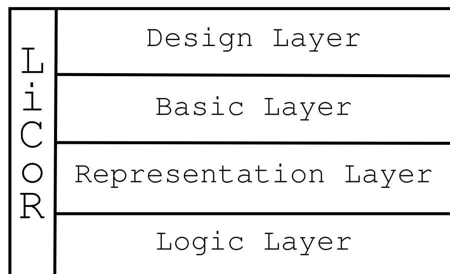

Figure 3.2: The different layer of LiCoR

3.4.2 An Example

In the following simple example will show how we can write meta programs using SOUL. The given rule can be used to obtain all the subclasses of a given class.

```
subclass(?subclass,?class) if
      member(?subclass,[?class allSubclasses]).
```

This predicate uses a Smalltalk term to compute a collection of all the subclasses of `?class` and then applies the member predicate to that collection. The effect of the `member` predicate when it is applied on a collection is that its output variable will be bound to all the items of that collection.

3.4.3 Library for Code Reasoning

SOUL is equipped with a large collection of predicates for meta programming. This collection is called the *Library for Code Reasoning* or LiCoR for short. LiCoR is designed as a set of layers (see figure 3.2) [Wuy01]. The predicates in each layer are grouped by functionality and use only the predicates from the lower layers. Let us take a look at these layers:

- **Logic Layer** This layer implements all the basic logic functionality. We can find predicates here for adding, substracting,... of numbers. For list manipulation like append, length,... For doing type checking on arguments of predicates (like the `var` predicate that checks wether something is a variable) For adding (assert) and deleting(retract) facts and rules from the logic database and for doing basic logic programming operations like pattern matching.

- **Representational Layer** Since we want to reason in SOUL about Smalltalk code, we need to represent Smalltalk entities in SOUL. This is done by the predicates in the representational layer which convert Smalltalk entities into SOUL entities (this process is called reification).

layer	group	some predicates
logic layer	arithmetic list handling type checking repository handling pattern matching	add,sub,greatThan,... append, length, member, flatten, var ,atom,ground assert,retract patternMatch, ...
representational layer	base predicates	class, method,hierarchy
basic layer	parse tree traversal typing flattening code generation accessing auxiliary	isSendTo, assignmentStatements, classUsed, returnStatments instVarTypes, collectionElementType classChain, flattenedMethod generateClass, removeClass methodName,methodClass, methodStatements rootClass,understands,abstractClass
design layer	prog. conventions design patterns	accessor,mutator compositePattern,visitor, abstractFactory

Table 3.1: The layers with their groups and some predicates

The predicates in this layer communicate with the MLI. It contains predicates for getting information about a class (class predicate), a method (methodName, methodArguments,...) , a parse tree of a method body (methodStatements), ...

- **Basic Layer** This layer contains a set of auxiliary predicates which factor out commonly used functionality when reasoning about code. Let us discuss a few predicates from this layer. `isSendTo` is used to retrieve information about which methods are sent in a part of a parse tree. The `instVarTypes` returns the types of the instance variables of a class,...

- **Design Layer** Here are predicates for programming conventions, design patterns, ... In this layer we can find the `visitor` predicate, which can be used to check wether a structure of classes is an instance of the visitor pattern.

All these layers are also subdivided in a number of groups which contain predicates with some common functionality. For a short overview of the layers with their groups and most important predicates, see table 3.1.

3.4.4 An application of LiCoR

We already discussed the Visitor design pattern in section 2.1.1. Now let us write a SOUL program that expresses the relationships between the classes

and methods that make up an instance of the visitor design pattern [Wuy01].

```
visitor(?visitor,?element,?accept,?visitSelector) if
1.          class(?visitor),
2.          classImplements(?visitor,?visitSelector),
3.          class(?element),
4.          classImplementsMethodNamed(?element,?accept,?acceptBody),
5.          methodArguments(?acceptBody,?acceptArgs),
6.          methodStatements(?acceptBody,
                        <return(send(?v,?visitSelector,?visitArgs))>),
7.          member(variable([#self]),?visitArgs),
8.          member(?v,?acceptArgs).
```

We added numbers in front of every line so that we can explain the example more easily. The lines 1 and 3 check wether the binding of the ?*visitor* and ?*element* variable are classes. Line 2 uses the classImplements predicate to check wether the `Visitor` implements the `visitSelector` (in the UML diagram this corresponds to checking wether the `Visitor` implements `visitConcreteElementA` and `visitConcreteElementB`). Line 4 and 5 extract the body and arguments from the method implemented on `Element` with as name the binding of ?*accept*. Lines 6, 7 and 8 check wether this body consists out of a statement which sends the message ?*visitSelector* to a visitor with "self" as an argument.

The program we obtained can be used for a whole range of applications (which we will not discuss any further, since it would lead us outside the scope of this dissertation):

- Detecting instances of the design pattern in a piece of source code
- Verifying that an instance of a pattern is still consistent with its implementation
- Generating a template implementation of an instance of the pattern

3.5 SOUL and Intentional Views

In chapter 2 we discussed Intentional Software Views. We mentioned that if we want to define a view, we have to express a description of that view in a formal language. We already showed that SOUL offers a lot of functionality for writing programs that reason about other programs. Now this is exactly the kind of programs we want to write if we want to express the description of an Intentional View. In chapter 2 we created a few views for documenting the Visitor design pattern. Here we will write down the descriptions of those views with SOUL. In the previous section we already wrote a SOUL

program that expresses the Visitor pattern. To write SOUL rules for these Intentional Views, we will use variations of this program. Let us take a look at the description of two views: *ConcreteVisitors* and *AcceptMethods*. Writing a SOUL program for the other views is similar to the programs we provide here.

- ```
  ConcreteVisitors(?class) if
        class(?class),
        AbstractVisitor(?abstractvisitor),
        subclass(?class,?abstractvisitor).
  ```
  Although the rule we provide here is quite naive (we say that every subclass of an AbstractVisitor is a ConcreteVisitor), it is a quite good description of the ConcreteVisitors view.

- ```
  AcceptMethod(method(?class,?selector)) if
        class(?class),
        ConcreteProducts(?class),
        classImplementsMethodNamed(?class,?selector,?methodbody),
        methodArguments(?methodbody,?acceptargs),
        methodStatements(?methodbody,
                  <return(send(?v,?visitorselector,?visitargs)>),
        member(variable([#self]),?visitargs),
        member(?v,?acceptArgs).
  ```
  The rule we provide here is very similar to the rule we created in the previous section. We say that a method is an AcceptMethod if it is implemented on a ConcreteProduct and if it has a double-dispatch (this information is encoded in the literals that express that the body of the method exists out of one statement that exists out of the return of sending a message to the visitor with `#self` as argument.

Now let us take a look at what we can do with these rules. If we want to calculate all the elements of an Intentional View, we simply pose a SOUL query. If we would for instance like to know all the accept methods, we would launch the query `if AcceptMethod(?methods)`. The rule we provided can also be used to let the documentation evolve whenever the implementation changes. Suppose we would implement a new *ConcreteVisitor* and we would re-calculate the elements in the *ConcreteVisitors* view, then the rule would also detect the new visitor: our description of the view is independent of the implementation of the visitor. Using the SOUL description for a view results in more robust documentation. Also, if we take a look at the rules, it is relatively easy to see what is meant by them. The intention behind the documentation is clear to a developer who reads the rules. The above example also illustrates the major disadvantage of Intentional Software Views: in order to create a description for a view, a developer has to know the language SOUL. Also the developer has to know a lot about the high-level

relationships in the software in order to be able to express this information in SOUL.

3.6 Summary

In this chapter we have studied the basic principles behind logic programming languages by means of the SOUL language. We discussed the notion of Logic Meta Programming and took a look at how we can write logic meta programs with SOUL and the LiCoR library. In the context of our research, we will use Logic Programming as the programming paradigm in which we implement the Induction algorithm (which we will discuss in the next chapter). Logic Meta Programming is important for our tool since we need to extract information out of the source code and also since we want an expressive formalism for describing our documentation. We also showed the relationship between SOUL and Intentional Software Views. We argumented that SOUL is an excellent medium for writing down the description of a view. At the end of chapter 2 we stated that we want to make a tool that allows the easy creation of robust documentation. We can now address the problem we are going to solve in our tool in a more concrete manner. The greatest disadvantage of Intentional Views is that the developer manually has to provide a description for a view and thus has to know about SOUL and about the structural information of the elements in the view. In our tool we are going to extract a SOUL program out of a classification automatically and use this program as the intentional description for the view. In the next chapter we will discuss how we can extract a logic program out of a collection of software artifacts.

Chapter 4

Inductive Logic Programming

In the previous chapter we discussed the topic of logic programming by means of the language SOUL and a set of examples. In this chapter we take a look at the theoretical foundations of logic programming. With this theory as background we discuss the topic of *inductive logic programming* (ILP), a technique from the domain of machine learning. Finally we take a deeper look at two examples of ILP algorithms:*FOIL* and *Relative Least General Generalization*.

4.1 Logic Programming Theory

This section gives a short introduction to logic programming theory. We do not give an in-depth overview of the subject but limit ourselves to the concepts that are useful for explaining Inductive Logic Programming and ILP-algorithms. For a more extensive treatment of the subject we refer to [Fla94] and [Llo87].
We start this section by introducing some terminology we will use in the rest of this section. We will then take a look at the syntax of the building blocks of logic programs: Horn Clauses. Since we are not only interested in what logic programs look like but also want to know what they mean we finish this section by taking a look at the semantics of logic programs by taking a brief look at model theory and proof theory.

4.1.1 Terminology

In the course of this chapter the reader will encounter a few terms related to logic programming theory and predicate logic. In this subsection we give the definition of a few relevant terms. It is not the intention that the reader looks at these terms now, but that he refers to this list whenever a term is

encountered that is not all clear.

Predicate A predicate consists out of a predicate symbol (a constant) and a number of arguments (the arity of the predicate). A predicate has a truth value true or false. Take a look at the predicate `sum(1,2,3)`. The predicate symbol here is `sum`, the arity is 3 and if we interpret the predicate as "the third argument of the predicate is the sum of the first two arguments" then the truth value is `true`.

Truth value of a clause A Horn clause can be true or false. This value is dependent on the truth values of the predicates in the clause.

And,or and implication The $\wedge$ is the *and*-operator. The clause A $\wedge$ B is true if and only if A and B are both true. The $\vee$ is called the *or*-operator. The clause A $\vee$ B is true if either A or B are true. Finally we have the implication ($\rightarrow$). The clause A $\rightarrow$ B is true if A and B have both the same truth value or if A is false and B is true. We can rewrite the implication $A \rightarrow B$ as the clause $\neg A \vee B$.

soundness We say that a logic program is sound if everything we can deduce from it is true

completeness A logic program is complete if it covers all positive examples

consistency If a logic program does not cover any of the negative examples, we say it is consistent.

derivation, deduction Clause C_2 is derivable from clause C_1 ($C_1 \vdash C_2$) if we can get clause C_1 from clause C_2 by applying rewrite operators to C_2.

logic consequence Clause C_2 is a logic consequence of C_1 ($C_1 \models C_2$) if every model of C_1 is a model of C_2.

4.1.2 The syntax of logic programs

In the previous chapter we gave a practical look at how logic languages work by studying the SOUL language. Here we will take a more formal look at logic programs. We define programs in a logic programming language as a collection of Horn clauses. The following grammar gives a formal definition of such a Horn clause. To define the grammar we will use the following conventions: productions between [and] may be omitted. For example $a[b]$ produces the string a or the string ab. A $*$ means that we may repeat the production zero or more times. If we take for instance i^*, this can produce the strings like the empty string but also i,ii,iii,... The | symbol is equivalent with an 'or'. If we encounter for example $a|b$, this can produce two strings namely the string 'a' or the string 'b'.

clause := head [← body]
body := atom
body := [atom [∧ atom]*]
atom := predicate[(term [,term]*)]
term := variable | constant | list
variable := ? identifier
constant := identifier
identifier := ”a single word”
list := < [term]* >

The best way to illustrate this syntax is by means of an example. Consider the following set of clauses to express the *grandparent* relation:

```
grandparent(?x,?y) ← parent(?x,?z) ∧ parent(?z,?y).
parent(?x,?y) ← father(?x,?y).
parent(?x,?y) ← mother(?x,?y).
father(jim,bob).
mother(ellen,louise).
father(bob,louise).
mother(mia,ellen).
```

Notice the similarities between the Horn Clauses and the syntax of SOUL. Now let us see how the concepts we have used to define a clause can be mapped onto our example.

- **clause** In our example `parent(?x,?y) ← father(?x,?y)` is a clause.
- **atom** `parent(?x,?z)` and `father(jim,bob)` are examples of an atom. Atoms are predicates which consist out of an identifier(here parent and father) and an arbitrary number of terms: the arguments.
- **terms** As we already saw in the grammar, a variable and a constant are instances of a term.
- **variable** `?x, ?,y` and `?z` are examples of variables.
- **constant** In our example `jim, bob` and `louise` are constants.
- **list** Although there are no lists in our example, we are still going to give an example of a list since it is used in later sections. `<1,2,3>` represents the list containing the numbers 1, 2 and 3.

We can have two different kinds of clauses: rules and facts. An example of a rule is `grandparent(?x,?y) ← parent(?x,?z) ∧ parent(?z,?y)`. Rules consist of a head (the conclusion) and a body (the preconditions). A fact

is a Horn clause without a body (like for instance `father(bob,louise)`. A fact is always true (more on truth values in the section about model theory). A special case of a clause is the empty clause `false` ← `true`. The empty clause is represented in most literature by the symbol □.

4.1.3 Properties of logic languages

Now that we have introduced the basic syntactical concepts of a logic language, we are going to discuss a few properties of these languages that are important for our further discussion.
We say that a clause or atom is **grounded** if it contains no variables (eg. parent(mia,louise)).

Let $Variables$ be the set of all the variables in a logic program P and let $Literals$ be the set of all the terms that occur in a clause C of a logic program P. A **substitution** $C\theta$ is a mapping $Variables \rightarrow Literals$ in which we change every occurence of variable $?x$ in C into literal l. We note this as: $\theta = \{?x/l\}$. For example: C = parent(jim,?x), $\theta = \{?x/bob\}$, $C\theta =$ parent(jim,bob). We say that a substitution θ is a **unifying substitution** of clauses C_1 and C_2 if $C_1\theta = C_2\theta$. So $\theta = \{?x/jim, ?y/bob\}$ is a unifying substitution of father(?x,bob) and father(jim,?y).

4.1.4 Model theory

We have already discussed the syntax of the logic language consisting out of Horn clauses. Now let us take a look at the semantics. In order to make it easier to determine the truth value of a Horn clause, let us rewrite the clause such that the implication of removed. Consider the following general Horn clause:
$H \leftarrow L_1 \wedge L_2 \wedge \cdots \wedge Ln$. In order to know its meaning, we want to assign a truth value to this clause by assigning a value to each of the literals. We are going to rewrite it by replacing the implication with a disjunction (the terminology in section 4.1.1 shows how this is done).
We then get: $H \vee \neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_n$. We see that this last clause is true if the head is true or if the negation of one of the literals from the body is true. The reader can verify that this is correct by comparing the truth values of our clause with those of the implication. If we now want to know wether a Horn clause is true we associate with every L_i a truth value and check wether the entire clause is true or false.

Before we take a look at the semantics of our example from the previous section, we have to introduce a few new concepts.
The **Herbrand Universe** $\mathcal{U}_P$ of a logic program P is the set of all the grounded terms in the program P. In case of the example from the previous

section $\mathcal{U}_P$ = {jim,bob,louise,mia}.

We define the **Herbrand Base** $\mathcal{B}_P$ of a program P as the set of all the grounded atoms in P. The elements of the Herbrand Base are all possible combinations of predicates and constants. The Herbrand Base of our example is:
$\mathcal{B}_P$ = {grandparent(jim,jim),$\cdots$, parent(jim,louise), mother(louise,louise),$\cdots$}.
It is easy to see that the Herbrand Base can be quite large.

We can make a mapping between the elements of the Herbrand Base and the values {true, false}. We call this mapping the **Herbrand Interpretation** $\mathcal{I}_P$ of a program P. Since we would have to specify for every element of a very large set if it is true or false, we are going to use the inverse relation in practice: we only write down the elements of $\mathcal{B}_P$ of which we say that they are true (formally this is written down as $\mathcal{I}_P^{-1}$(true)). For the rest of the elements of the Herbrand base we assume that they are mapped to false.

Finally, we say that $\mathcal{M}$ is a model for a program $\mathcal{P}$ if $\mathcal{M}$ is a subset of interpretation $\mathcal{I}$ and that all the clauses of $\mathcal{P}$ are true with respect to $\mathcal{M}$. Applied to our example we can find the following models:
$\mathcal{M}_1$ = {grandparent(jim,louise),father(jim,bob),father(bob,louise),
mother(mia,louise)}
$\mathcal{M}_2$ = {mother(mia,louise),mother(ellen,mia),grandparent(ellen,louise)}
$\mathcal{M}_1$ and $\mathcal{M}_2$ are not all of the models of $\mathcal{P}$. There can be many more models (in some logical languages there can even be an infinite number of models). Note that we can conclude from the two models that jim and ellen are grandparents of louise.

4.1.5 Proof Theory

Calculating the models from the Herbrand Base is not a realistic approach for finding correct interpretations of a logic program. It takes too much time to compute and in some cases, when the Herbrand Base is infinite, it is impossible. Instead we want to use proof theories that allow us to derive new, correct clauses from a program by using deductive operators. Resolution is the operation we use to do this. We define resolution [Rob65] as follows:

$$\frac{\begin{array}{c} P \vee L \\ \neg L \vee R \end{array}}{P \vee R}$$

The part above the horizontal line is the begin situation, the part beneath it is the conclusion. We illustrate resolution by means of an example:

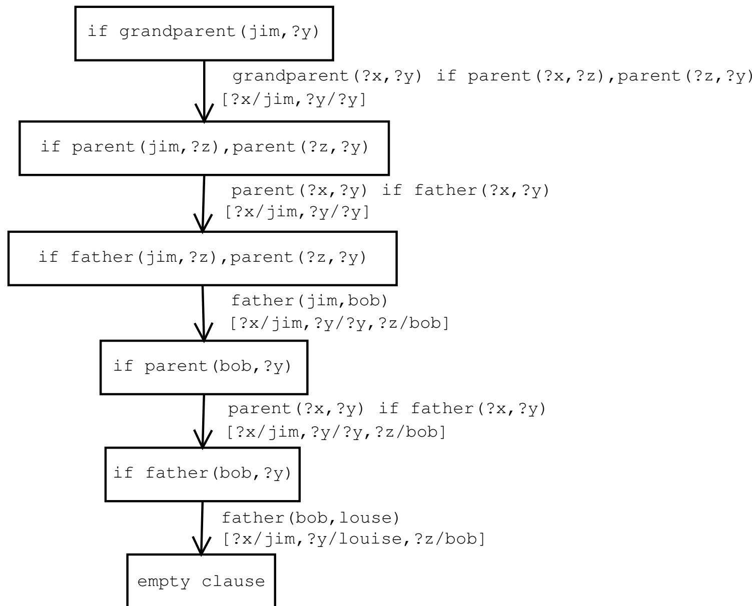

Figure 4.1: Derivationtree of the example

Clause 1: `grandparent(?x,?y)` $\leftarrow$ `parent(?x,?z)` $\wedge$ `parent(?z,?y)`.
Clause 2: `parent(?x,?y)` $\leftarrow$ `father(?x,?y)`.
Clauses 1 and 2 are equivalent to the following set of clauses (when we rewrite the implication):
Clause 1: `grandparent(?x,?y)` $\vee$ $\neg$`parent(?x,?z)` $\vee$ $\neg$`parent(?z,?y)`.
Clause 2: `parent(?x,?y)` $\vee$ $\neg$`father(?x,?y)`.
Notice the literals $\neg$`parent(?x,?z)` from clause 1 and `parent(?x,?y)` from clause 2. We can find a unifying substitution {?z/?y} so that the two literals are equal (not considering the negation in one of them). When we apply this substitution to the clauses we can rewrite clause 2 and get the following set of clauses:
Clause 1: `grandparent(?x,?y)` $\vee$ $\neg$`parent(?x,?z)` $\vee$ $\neg$`parent(?z,?y)`.
Clause 2: `parent(?x,?z)` $\vee$ $\neg$`father(?x,?z)`.
Now take:
P = `grandparent(?x,?y)` $\vee$ $\neg$`parent(?z,?y)`.
R = `father(?x,?z)`
L = `parent(?x,?z)`.
If we now apply our definition of resolution we get as result the following clause: `grandparent(?x,?y)` $\vee\neg$`father(?x,?z)` $\vee\neg$`parent(?z,?y)`. We can turn this clause back into a Horn clause by introducing an implication: `grandparent(?x,?y)` $\leftarrow$ `father(?x,?z)` $\wedge$ `parent(?z,?y)`.
Resolution is sound: if we can obtain a clause C out of a program P by means of resolution, then we can also say that C is a logic consequence of P

(formally this is written down as $P \vdash C \rightarrow P \models C$).

Now we are going to take a look at how we can use resolution to find answers to a logic query. Consider the following technique: suppose we want to prove C. Instead of trying to reach C by using resolution we are going to try to prove ¬C. If we obtain the empty clause $\Box$ while proving ¬C, then we have reached a contradiction (the empty clause can be interpreted as false ← true, which can never be true) and we can conclude that C is valid. If C contains variables, then we can use the substitutions of the proof to get the values for which C holds. This technique to construct proofs is called *proof by refutation.* Note that we can do this since resolution is sound: if we apply the resolution operator to a clause, then the new obtained clause is a logic consequence of the original one. This is a necessary condition since we can not construct a correct proof if we are not sure that every step in the proof is also correct.

Now let us take a look at how we would apply this to our example. Figure 4.1 shows a complete refutation tree for finding all the persons $?y$ who have jim as a grandparent. We start with the query "if grandparent(jim,?y)" (the "←" of the Horn clause is represented by the word "if" in the figure). This clause is equivalent to the Horn clause: `false ← grandparent(jim,?y)`. We can read this clause as "there exists no ?y so that jim is the grandparent of ?y" or less formal as "jim has no grandchildren". By doing a few resolution steps we come to the empty clause which means we have reached a contradiction and have proven the oppositive of the clause namely that "jim does have grandchildren". At each step the unifying substitutions have also been supplied. If we take a look at these substitutions we can see that every binding for $?y$ is a possible solution of the query. In this case is louise the only grandchild of jim.

4.2 Inductive Logic Programming

4.2.1 Definition of Inductive Logic Programming

We should start this section by stating the kind of problems that inductive logic programming tries to solve. A good definition can be found in [BG95].

Given:

- A set of possible programs $\mathcal{P}$
- A set of positive examples $\mathcal{E}^+$
- A set of negative examples $\mathcal{E}^-$
- A logic program B so that $\exists\ e^+ \in \mathcal{E}^+ : B \not\vdash e^+$

Then: find a logic program P $\in \mathcal{P}$ such that the program B $\cup$ P is complete and consistent.

The program B that we give the induction algorithm as an input is the *background knowledge* we have about the problem: it is a collection of rules and/or facts which express the knowledge we already know about the problem. We state that at least one of the positive examples is not a logic consequence of the background information. If this would not be true, then we would already have a program P that covers all the positive examples (it is clear that P = B). The set $\mathcal{P}$ is often called the *hypothesis space*: it is an infinite set that contains all the possible, correct Horn clause programs. The program P we want to find is an element of the hypothesis space that extends the background program B and covers all the positive examples $\mathcal{E}^+$ while not covering any of the negative ones $\mathcal{E}^-$.
If we compare induction with deduction (deduction is the process of deriving new clauses out of existing ones as described in section 4.1.5) we see that they share the following property: let T be a logic program and E a consequence that we can deduce out of T, then E is a logic consequence of T.
Formally this is written down as: T$\models$ E. In the context of ILP we can rephrase this as B $\cup$ H $\models E^+$ and B $\cup$ H $\not\models E^-$ with B the background knowledge, E^+ and E^- the sets of positive and negative examples and H the hypothesis we want to induce. If we want to express the relationship between induction and deduction we can say that induction is the inverse of deduction.

4.2.1.1 An example

As an example, suppose we want to induce a program for expressing the granddaughter relationship.

$\mathcal{P}$ = the collection of all correct Horn Clauses.
$\mathcal{E}^+ = \{$ `grandDaughter(sharon,victor),`
`grandDaughter(julie,victor)` $\}$
$\mathcal{E}^- = \{$ `grandDaughter(ellen,victor)` $\}$

```
B = { father(bob,tom), father(victor,ellen)
       father(bob,sharon),female(sharon),
       mother(ellen,julie),mother(ellen,sharon),
       female(ellen),female(julie), ...  }
P = { grandDaughter(?x,?y) ← female(?x),father(?z,?x),father(?y,?z),
      grandDaughter(?x,?y) ← female(?x), mother(,z,?x),father(?y,?z)
       ... }.
```

The set $\mathcal{E}^+$ consists out of a number of examples of the relationship for which the truth value is true (they are correct instances of the grandDaughter re-

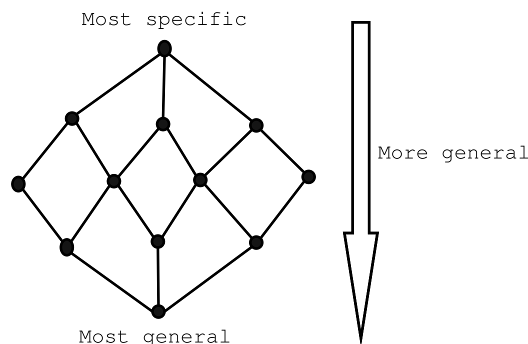

Figure 4.2: a visual representation of the hypothesis space

lationship). $\mathcal{E}^-$ contains one example for which the relationship is false. In the collection B we can find all the background information about the examples we have: it contains facts which represent information about persons and the relationship between these persons. B is the logic program we want to extend so that it also has the notion of the granddaughter relationship. The program P that is obtained by induction contains the rules that can be used to express the granddaughter relationship. Notice that all the positive examples are covered by the program P $\cup$ B while the program does not cover any of the negative examples.
In this section we did not yet specify how we can induce clauses. The following sections will give an explanation on how this is done. In these sections we will reuse the grandDaughter example that is described above or a variation on it.

4.2.2 Properties of Inductive Logic Programming Algorithms

In this section we will briefly take a look at two properties of ILP algorithms: top-down vs. bottom-up and the representation of background knowledge.

4.2.2.1 Top-down vs. Bottom-up

As we already mentioned earlier the idea behind ILP is to find a hypothesis that matches our problem by searching the Hypothesis Space for it. If we take a look at how most algorithms do this, we can classify them in two groups: top-down and bottom-up algorithms.

- Top-down algorithms start with the most general possible hypothesis (a hypothesis which states that everything is true) and will make this hypothesis more specific in every step of the algorithm.

- Bottom-up algorithms do the oppositive of this: they start with the most specific rule (a rule which covers no example) and will make it more general.

Notice that the elements of the Hypothesis Space (as depicted in figure 4.2) form a lattice: for every couple of hypothesis there exists another hypotheses that is more general or more specific than the two hypothesis. This property will be important for one of the induction algorithms we will discuss later on.

4.2.2.2 Representation of the background knowledge

We can represent the background knowledge the algorithm takes as an input in two possible ways: extensional and intensional.

- **Extensional** background knowledge is represented by a collection of facts. These facts define a model of the background we want to use in our algorithm. Suppose, given a set of examples, we want to induce the grandparent relationship as discussed in section 4.1.2. Our positive and negative examples will be expressed by means of facts like `grandparent(jim,louise)`, `grandparent(bob,jim)`, ... The background information would be:
  `B = { father(jim,bob), mother(ellen,louise), father(bob,louise), mother(mia,ellen), parent(jim,bob),parent(ellen,louise), parent(mia,ellen), ...}`.
  If we take a look at the background we also see that all the elements are facts.

- **Intensional** background knowledge differs from extensional in the fact that it is a description of some of the knowledge instead of an enumeration of facts. The background information is represented by a collection of Horn clauses. Applied to our example this gives the following background:
  ```
  B = {parent(?x,?y) ← father(?x,?y),
      parent(?x,?y) ← mother(?x,?y),
      father(jim,bob), mother(ellen,louise), mother(mia,ellen),...}.
  ```
  The background information does not consist solely out of rules: some of the background information can not be expressed as a rule, but are instead data that is written down as facts.

Note that we can easily create the extensional background out of the intensional. This can be done by generating all the facts that correspond with the results of the rules in the background information, given the facts in the background as input.

```
FOIL(TargetPredicate, Predicates, Examples)
 1  Pos ← Positive Examples
 2  Neg ← Negative Examples
 3  Learned_rules ← {}
 4  while Pos
 5  do Learn a NewRule
 6      NewRule ← rule that predicts TargetPredicate with no precondition
 7      NewRuleNeg ← Neg
 8      while NewRuleNeg
 9      do Add a new literal to specialize NewRule
10          Candidate_literals ← generate candidate new literals, based on Predicates
11          Best_literal ← argmax Foil_Gain(L,NewRule) with L ∈ Candidate_literals
12           add Best_literal to preconditions of NewRule
13          NewRuleNeg ← subset of NewRuleNeg
14           that satisfies NewRule preconditions
15      Learned_rules ← Learned_rules + NewRule
16      Pos ← Pos − {members of Pos covered by NewRule}
17  return Learned_rules
```

Figure 4.3: the FOIL algorithm in pseudo-code

4.3 Induction Algorithms

In this section we will have a look at two different approaches for inducing Horn clauses. The first approach is FOIL, a top-down algorithm. The second one is relative least general generalization, which is a bottom-up approach.

4.3.1 FOIL

The first algorithm we will take a look at is FOIL [Qui90]. The pseudo-code for the algorithm can be found in figure 4.3. FOIL is a quite naive algorithm that starts with a general rule and will add literals to that rule until it no longer covers any of the negative examples. It will create a (sometimes extremely large) collection of literals which it will consider for addition at each step. To actually choose which literal gets added, a heuristic is used. If we take a look at the algorithm we see it consists out of two loops. The outer loop will add a new rule to the set of hypothesis *Learned_rules*. It starts with the most specific hypothesis (nothing is true) and will generalize it in each step. It searches the hypothesis space in a bottom-up fashion. The outer loop will keep adding new rules until the set of rules covers all the positive examples.
The inner loop will construct the new rules. To do this it starts with the most general rule (the rule that states that everything is true: for example

`parent(?x,?y)` states that everybody is the parent of somebody) and will add new literals to it until none of the negative examples are covered. This piece of the FOIL algorithm does a general-to-specific (top-down) search of the hypothesis space. We are now going to take a more detailed look at how FOIL creates candidate literals and we will also discuss the performance measure *Foil_Gain* which it uses to select the best literal that is added to the rule.

4.3.1.1 Creating Candidate Literals

When making the current rule more specific FOIL creates, a set of new literals which may be considered for adding to the rule.
Suppose the current rule is of the form: $P(?x_1, ?x_2, \ldots, ?x_k) \leftarrow L_1, \ldots, L_n$.
The following literals L_{n+1} are then considered for adding:

- $Q(?v_1, ?v_2, \ldots, ?v_r)$ with $Q \in Predicates$ and where $\forall\ ?v_i$ with $1 < i < r$ are either new variables or variables that can be already found in the rule. Also $\exists\ ?v_j$ with $1 < j < r$ such that $?v_j$ is a variable that is already part of the rule. The set $Predicates$ consists out of all the predicate names that are considered for adding to the rule. This is the place where we can use the background information we have of the problem in finding a solution: we supply the predicates that appear in the background information as the elements of $Predicates$.
- $Equal(?x_j, ?x_k)$ with $?x_j$ and $?x_k$ variables that are present in the rule.
- The negation of the literals from the two above forms.

To show how this works in practice, consider the following example [Mit97].
Suppose we want to learn the *GrandDaughter* relationship. We are given a set of predicates FOIL can use (in this case $Female$ and $Father$, the predicates which occur in the background knowledge) and we have a set of positive and negative examples. We would start with the most general rule $GrandDaughter(?x, ?y) \leftarrow$. This rule implies that every $?x$ is a granddaughter of $?y$. In figure 4.4 we can see a subset of the literals that FOIL considers to add. Suppose that FOIL choses $Father(?y, ?z)$ as the next literal to add. The rule will then become $GrandDaughter(?x, ?y) \leftarrow Father(?y, ?z)$. In the next step FOIL will consider all the literals from the previous step plus also a few new ones and their negation (see figure 4.4).
Suppose that FOIL choses $Father(?z, ?x)$ in this step. During the next iteration FOIL will then choose the literal $Female(?y)$. The hypothesis will then become: $GrandDaughter(?x, ?y) \leftarrow Father(?y, ?z), Father(?z, ?x), Female(?y)$.
As the reader can see, this rule is a correct definition of the granddaughter relationship. The rule does not cover any negative examples so FOIL can

	Clause so far	Considered Literals
1	`grandDaughter(?x,?y)` ←	equal(?x,?y), female(?y), female(?x), father(?x,?y) father(?y,?x), father(?x,?z) . . . , ¬female(?x) ¬father(?x,?y), . . .
2	`grandDaughter(?x,?y)` ← `father(?y,?z)`	all from first step + female(?z), equal(?z,?x), equal(?z,?y),. . . + negation of these literals
3	`grandDaughter(?x,?y)` ← `father(?y,?yz),` `father(?z,?x)`	all from 2nd step + female(?w), equal(?x,?w), equal(?w,?w), . . . + negation of all these literals

Figure 4.4: the clause so far and the literals considered by FOIL for addition to the clause

remove the positive examples this rule covers and can start over again with the remaining examples to create another rule.

4.3.1.2 FOIL_Gain

Now let us take a look at how FOIL decides which literal gets added each iteration. FOIL uses a performance measure called *FOIL_Gain* to select the literal that is added to the rule. *FOIL_Gain* is defined by the following equation:

$$FOIL_Gain(L, R) = t\left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0}\right) \tag{4.1}$$

$FOIL_Gain(L, R)$ calculates the performance of adding literal L to rule R. Let R′ be the rule that is created by adding literal L to rule R. To measure the performance of a rule, FOIL will generate all the possible bindings of a rule. A binding of a rule is a combination of all the constants with all the variables in a rule. A binding is called positive if there exists a corresponding fact in the background data.

- p_0 are all the positive bindings in rule R.
- n_0 are all the negative bindings in rule R.
- p_1 are all the positive bindings in rule R'.
- n_1 are all the negative bindings in rule R'.

$Female(?y)$	$Father(?x, ?z)$
$t = 1$	$t = 1$
$p_0 = 1$	$p_0 = 1$
$n_0 = 15$	$n_0 = 15$
$p_1 = 1$	$p_1 = 1$
$n_1 = 15$	$n_1 = 80$
$Gain = 0$	$Gain = -2,33$

Table 4.1: The *FOIL_Gain* calculated for two possible literals

- t are all the positive bindings of R which are still positive after adding L to R.

This definition will become clearer to the reader if we apply it to our example. Suppose the input data consists out of the following facts:
$GrandDaughter(Sharon, Victor)$, $Father(Bob, Sharon)$
$Father(Bob, Tom)$, $Female(Sharon)$, $Father(Victor, Bob)$
FOIL will start with the most general rule: $GrandDaughter(?x, ?y) \leftarrow$.
Since the constants in our example are Victor, Sharon, Bob and Tom we can generate bindings of the form $\{?x/Bob, ?y/Victor\}$, $\{?x/Tom, ?y/Bob\}, \ldots$.
Of these 16 possible bindings only $\{?x/Victor, ?y/Sharon\}$ is a positive one (since $GrandDaughter(Sharon, Victor)$ is the only fact that corresponds with a binding), the 15 other are negative.
In table 4.1 we can see a table with the values for *FOIL_Gain* for the $Female(?y)$ and $Father(?x, ?z)$ as L and $GrandDaughter(?x, ?y) \leftarrow$ as R.
We want to maximize the information we can gain by adding a literal so we choose the candidate literal with the highest Foil_Gain. In this example FOIL will choose $Female(?y)$ as the next literal to add.

4.3.2 Relative Least General Generalization

The other technique we will take a look at is called *Relative Least General Generalization*. It is based on the observation that induction is in fact the opposite operation of deduction and that inverse deduction operators can be used to induce clauses. Plotkin [Plo70] [Plo71] was the first to notice that generalization could be used as such an inverse deduction operator.
Relative Least General Generalization works by taking two examples and applying a generalization operator on them so that we obtain a clause that is strictly more general than the two examples. This clause suffices in a lot of cases to also cover other positive examples. This process is repeated until no more positive examples remain.

4.3.2.1 Definitions

If we want to induce clauses by generalization it is important that the clause we find is not overly general. If we for instance always would take as the generalization of two clauses the most general clause ("everything is true"), we would not be able to induce useful rules. In fact, given two clauses c_1 and c_2 we want to find the clause c which is the most specific clause that is more general than c_1 and c_2. So we should take a look at a few definitions which allow us to determine which of two clauses is the most general/specific. Let us take a look at the following definition:
θ-subsumption: Clause C θ-subsumes (or is more general than) a clause D if there exists a substitution θ such that $C\theta$ is a subset of D ($C\theta \subseteq D$).
Example: The clause $element(?x, ?v) \leftarrow element(?x, ?z)$ θ-subsumes the clause $element(?x, <?y|?z>) \leftarrow element(?x, ?z)$ with $\theta = \{?v/ <?y|?z>\}$.
There exists a relation between θ-subsumption and logical consequence, namely: if C θ-subsumes D then $D \models C$ (we will not prove this property since it is out of the scope of this dissertation).

Notice that the hypothesis space as depicted in figure 4.2 forms a lattice. We can see that for every pair of clauses their exists a unique clause that is minimal more general than the two clauses. This leads us to the following definition:
Least general generalization: C is the least general generalization (lgg) of D if C θ-subsumes D and for every other clause E such that E θ-subsumes D it is also the case that E θ-subsumes the clause C.
If we want to compute the *lgg* C of a set of clauses S we can do this by computing the *lgg* of every clause in S.

4.3.2.2 Anti-unification

Let us define the anti-unification of two terms as follows.
Term C is the anti-unification of terms C_1 and C_2 if we can find substitutions θ_1 and θ_2 such that C $= C_1\theta_1 = C_2\theta_2$ and θ_1 and θ_2 are so chosen that for all other θ_i and θ_j holds that $C_1\theta_i$ and $C_2\theta_j$ are more general than C.
What we concrete try to obtain by anti-unifying two terms is a term that shares all the commonalities of the two terms and generalizes the differences between them. An example will make this definition easier to understand:

$C_1 \equiv 2 * 2 = 2 + 2$
$C_2 \equiv 2 * 3 = 3 + 3$
$C \equiv 2 * X = X + X$
$\theta_1 = \{2/X\}$
$\theta_2 = \{3/X\}$.

When we will take a look at the implementation of *rlgg* in a later chapter, we will discuss more detailed how we can compute the anti-unification of two terms.

4.3.2.3 Computing the Least General Generalization

We are now going to take a look at how we can compute the lgg of two clauses.
Suppose we have the clauses $C_1 \leftarrow A_1, A_2, \ldots, A_n$ and $C_2 \leftarrow B_1, B_2, \ldots, B_m$. We define the clause C which is the lgg of the two clauses as:

- The head of clause C is obtained by calculating the anti-unification of the heads of the original clauses,namely C_1 and C_2.
- The body C is constructed by anti-unifying $\forall$ A_i with $1 < i < n$ with $\forall$ B_j with $1 < j < m$.

4.3.2.4 Relative Least General Generalization

We now have a method for generalizing a set of clauses. In practice this is not very useful since we also want to incorporate some background knowledge while creating a generalizing clause. To solve this problem we are going to make use of the *relative least general generalization* (or rlgg for short). The rlgg of two positive examples is the lgg of the examples with respect to a (partial) background model B. We can express this more formally as:

$$rlgg(e_1, e_2, B) = lgg(e_1 \leftarrow B_\wedge, e_2 \leftarrow B_\wedge).$$

Notice that this approach for inducing clauses does a specific-to-general search of the hypothesisspace (bottom-up).

4.3.2.5 An example with a lot of redundant literals

Suppose we have the following set of examples:

```
append(<1,2>,<3,4>,<1,2,3,4>), append(<a>,<>,<a>)
append(<>,<>,<>), append(<2>,<3,4>,<2,3,4>)
```

When we compute the rlgg of `append(<1,2>,<3,4>,<1,2,3,4>)` and `append(<a>,<>,<a>)` with the set of examples as background information we obtain the following clause:

```
append(<?x|?y>,?z,<?x|?u>) ← append(<2>,<3,4>,<2,3,4>),
      append(?y,?z,?u),append(<?v,?z,<?v|?z>)
      append(<?k|?l>,<3,4>,<?k,?m,?n|?o>),append(?l,?p,?q),
      append(<>,<>,<>),append(?r,<>,?r),append(?s,?p,?t),
      append(<?a>,?p,<?a|?p>),append(?b,<>,?b),append(<a>,<>,<a>),
      append(<?c|?l>,?p,<?c|?q>),append(<?d|?y>,<3,4>,<?d,?e,?f|?g>),
      append(?h,?z,?i),append(<?x|?y>,?z,<?x|?u>),
```

```
append(<1,2>,<3,4>,<1,2,3,4>).
```

As the reader may notice, the above rule contains a lot of unnecessary literals. This problem is also discussed in [MF90] we is stated that the clauses that are induced by generalization can contain an extremely large number of (redundant) literals. If we have a background model M and n examples then the maximum number of literals in the created clause can be $|M|^n + 1$. A sufficiently large background model and/or a lot of examples can make the creation of the clause intractable [Bun88]. Besides from the obvious deletion of examples (literals without variables in general) from the clause, a few other methods also have been proposed for reducing the number of literals in the induced clause.

4.3.2.6 Reducing the size of the clauses

The first method for limiting the number of literals in the clauses is *ij-determination* [MF90]. The idea behind this technique is to put a (weak) limitation on the hypothesis language: the clauses that are inducible are limited in the way that a maximum depth and degree is set on the variables that appear in the clause. This limitation prevents a combinatorial explosion in the number of literals in an induced clause. We are not going to discuss the theory behind this technique in more detail since this would lead us out of the scope of this dissertation. Instead we will try to explain the main concept of ij-determination by means of a few examples.
Take a look at the clause `double(?A,?B)` $\leftarrow$ `plus(?A,?A,?B)`. We say that this clause is 12-determinate: the variable $?B$ is dependent from 2 values, namely two times the variable $?A$. We say that the variable $?B$ is at depth 1 since the variable $?A$ on which it is dependent occurs in the head.
The clause `grandfather(?A,?B)` $\leftarrow$ `father(?A,?C), father(?C,?B)` is 21-determinate: the variables $?B$ and $?C$ are both only dependent on a single variable, therefore the degree of the clause is 1. The depth of this clause is 2 since the variable $?B$ is dependent on the variable $?C$ which has degree 1. If we put a limitation on the depth (i) and the degree (j) of the inducible clauses, we can decrease the number of literals that appear in the body of the induced clause.

The clauses we obtain through induction can also be *negative-based reduced.* Suppose we have a clause $\texttt{A} \leftarrow B_1, \ldots, B_n$. We search the first literal B_i from $B_1, \ldots, B_n$ so that the clause $\texttt{A} \leftarrow B_1, \ldots, B_i$ does not cover any negative examples. We then reduce the clause to $\texttt{A} \leftarrow B_i, B_1, B_{i-1}$ in the same manner and iterate the process until further reduction does not make the size of the clause smaller .

The last method of clause reduction we are going to discuss is called *func-*

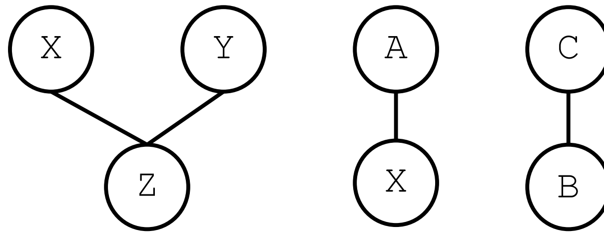

Figure 4.5: the functional graph of the example

tional reduction. Shapiro [Sha83] was the first to add an additional constraint to Horn clauses by describing the arguments of a clause as input or output variables. Take for example the literal `plus(?x,?y,?z)`. We then say that ?x and ?y are input variables and ?z is an output variable. We can use this idea to reduce clauses. We do this by creating a functional and/or-graph out of the unreduced rlgg. We will represent this and/or-graph as a set of Horn clauses. For every literal l in the body of the clause with input variables $v_1, \ldots, v_n$ and output variables $u_1, \ldots, u_m$ we create a set of Horn clauses $\{(v_1 \leftarrow u_1, \ldots, u_m), \ldots, (v_n \leftarrow u_1, \ldots, u_m)\}$. For every input variable i in the head of the clause we add a fact i. If we search this graph we can obtain a set of variables which we can use to reduce the induced clause. Let us demonstrate the use of functional reduction by means of an example. Suppose that after applying relative least general generalization we obtain the following clause for the grandfather relationship:

```
grandfather(?X,?Y) ←
      father(?X,?Z),
      father(?Z,?Y),
      father(?X,?A),
      father(?B,?C).
```

We see that only the first two literals of the body are necessary, the last two are redundant. The father predicate expresses the "... is the father of ..." relationship. We say that the mode of the father predicate is father(out,in): the first variable is an output variable, the second one an input variable. When we create the Horn clauses according to the way we described above we get the following set of clauses:

```
?Z ← ?X
?Y ← ?Z
?A ← ?X
?C ← ?B.
```

These clauses correspond to the graph in figure 4.5. If we now take a

	FOIL	RLGG
Search direction	Top-down	Bottom-up
Search approach	Hyp. space search	Inv. Deduction
Ex. implementation	FOIL	GOLEM

Table 4.2: A summary of the discussed ILP approaches

look at the graph and we take all the nodes starting from the nodes that represent the variables from the head of the clause, we see that only the variables $?X, ?Y$ and $?Z$ are necessary for computing the grandfather relationship and thus can we prune the last two literals of the clause. We then obtain the following clauses as the result:

```
grandfather(?X,?Y) ←
      father(?X,?Z),
      father(?Z,?Y).
```

4.3.3 Summary

In this chapter we have discussed the theoretical background behind logic programming. We also looked at Inductive Logic Programming and gave an overview of a few approaches for ILP, namely FOIL and relative least general generalization. Table 4.3.3 shows an overview of these approaches and their properties. We are going to use ILP in the context of our research to extract a description out of the documentation and use that description to make sure that the documentation can evolve whenever the implementation does.

Chapter 5

Software Views Inducer

In this chapter we will take a look at *Software Views Inducer*, the tool we created in the context of this dissertation. We show how we can use the techniques like ILP and LMP which we discussed in previous chapters in order to develop a tool for creating documentation. We will discuss the implementation of this tool and take a look at an example usage of it.

5.1 Description of the tool

In the Software Views Inducer tool we combine the advantages of both software classifications [DH98] and Intentional Views [MMW02a]. By making use of software classifications we want to make the creation of documentation out of a Smalltalk image easier for the developer. Instead of having to write hard SOUL rules, a developer can document a system by simple drag & drop operations. The tool then extracts an intentional description out of the software entities the developer has classified. Whenever there are changes made to the software, this description can be used to semi-automatically update the documentation. Our tool also makes it easier for a developer to understand the high-level structures in the documentation since they are made explicit in the description.

- The developer documents the software by creating classifications and classifying software entities that belong to these classifications.
- The tool will analyze the software artifacts in the classifications and will generate background information facts for these artifacts. To do this the logic meta programming capabilities of SOUL will be used. In a work-floor ready tool, we would allow the tool to analyze the elements of a classification on a whole range of features (like for instance subclasses, message sends,...) without the developer having to specify which ones are considered. In our implementation of the tool we allow a developer to choose which kinds of features are used when

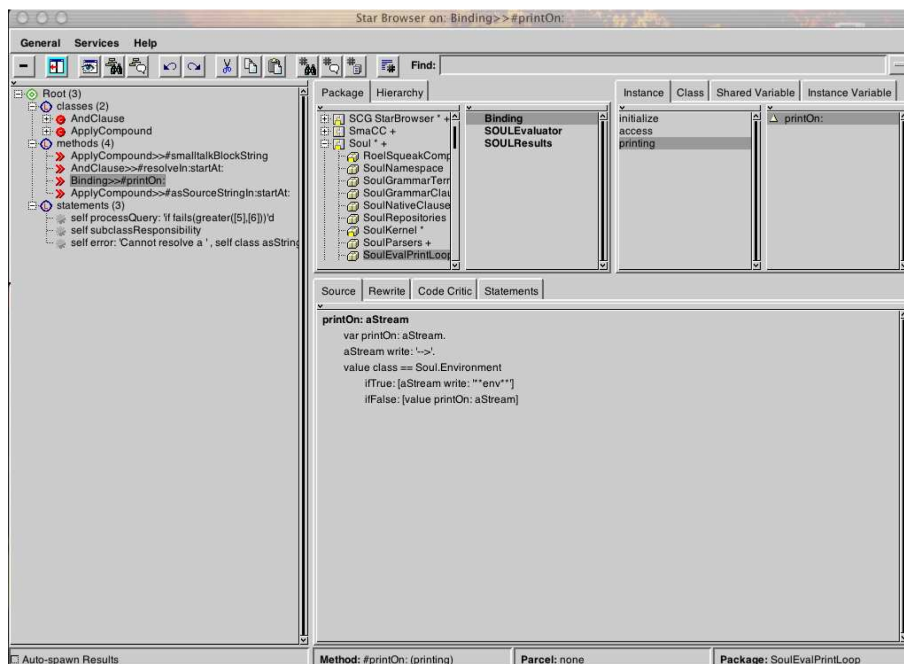

Figure 5.1: a screenshot of the StarBrowser with 3 Learned Classifications

analyzing the software entities. This is done for performance reasons and for allowing us to better control the experiments.

- The tool creates a set of logic rules by means of inductive logic programming that describe the software entities in such a classification with respect to the types of background information the developer has provided.
- Whenever the source code is changed, these rules can be used to calculate which software artifacts belong to the classification and thus keep the documentation and the implementation synchronized.

The Software Views Inducer is nowhere near being a work-floor-ready tool, but it is stable enough and offers enough functionality to allow us to conduct some experiments. The following sections give a high-level view of some of the design decisions we had to make while implementing the tool and will give an example of the usage of it.

5.2 Making classifications and classifying entities

The classifications we use in our tool do not differ conceptually from the ones that are used in software classifications [DH98] as we discussed in chapter 2.

Different tools already have been created that implement software classifications. For creating the classifications in the Software Views Inducer we have extended such an existing tool, namely Roel Wuyts' *StarBrowser* [Wuy]. The StarBrowser allows the easy creation of classifications and implements the classification of software entities out of the Smalltalk image by means of drag & drop. The reason why we chose to extend the StarBrowser is that it is more than just a tool: it is a complete framework for writing tools that make use of classifications. We extended the StarBrowser with a new kind of classification: the *Learned Classification*. Although the StarBrowser allows the classification of almost all types of items that may occur in a Smalltalk image, we restrict the types of items we can put in such a classification. We only allow items that can be relevant for software documentation:

- Classes
- Methods
- Statements from the body of methods

It is clear that when creating documentation we want to classify classes and methods. We also included the classification of statements, but we are not going to use them in our experiments. Statements might be useful when documenting cross-cutting concerns, but this is outside of the scope of this dissertation. Other Smalltalk entities like protocols, categories, ... can also be interesting as documentation but we can omit these since they are basically nothing more than a container of classes or methods. The StarBrowser already supplies the functionality for classifying classes and methods. Since the functionality for classifying statements is not implemented in the StarBrowser, we had to adapt the StarBrowser and the Refactoring Browser (this is the standard object browser in VisualWorks [Cin], the Smalltalk environment we use). Figure 5.1 shows a screenshot of the StarBrowser with three Learned Classifications. On the left hand side of the screenshot the reader can see the classifications (in this case: *classes*, *methods* and *statements*). Each of these classifications contain a certain number of software entities from within the Smalltalk image. On the right hand side a class browser is opened on the currently selected item.

5.3 Choice of Induction algorithm

As we already stated above, we want to use an ILP algorithm to create a set of logic rules out of the items of a classification and the background information. In section 4.3 we discussed a few approaches to ILP. We are now going to take a look at which approach best fits our problem. First let us take a look at our situation. We have:

- A relatively small amount of positive examples. The developer who creates documentation will only classify a limited number of software artifacts. The developer expects from the tool that it will find a rule that also detects the other elements of the classification.

- A small or usually non-existent amount of negative examples. It is not natural for a developer to have to say which parts of the software do not belong to a certain part of the documentation. The only way our tool might obtain negative examples is that the rule obtained out of the classification also covers some faulty software artifacts and that the developer will mark them as such.

- A large collection of background information about the examples.

If we now take a look at the different ILP algorithms we can make the following observations:

- **FOIL** uses the collection of negative examples to guide its search of the hypothesis space. If we recall how the FOIL algorithm works, we see that the algorithm will keep adding candidate literals with the highest FOIL_Gain until the clause no longer covers a negative example. When there are no negative examples available, which is common in our situation, FOIL will only be able to induce the most general clause (namely that everything matches the target predicate).

- **Relative Least General Generalization (RLGG)** uses the set of positive examples to induce the clauses and therefore fits our needs better. As already discussed in section 4.3.2.5, the amount of candidate literals in the clause grows exponentially with the number of examples and the size of the background information. This influences the overall performance of the RLGG approach since a lot of candidate literals are considered for adding to the rule and we have to prune the resulting rules due to this large number of (redundant) literals.

The key factor in our decision of an ILP algorithm is the fact that we do not have (a lot of) negative examples. Therefore relative least general generalization is the only inductive logic programming technique we discussed which we can apply to our situation.

5.3.1 Implementation of RLGG

We did not start from scratch when implementing the RLGG algorithm. In [Fla94] an implementation of RLGG can be found. This implementation puts a limitation on the clauses that can be induced. It only allows restricted clauses: clauses for which all the variables that occur in the body of the clause also occur in the head. Although this limitation significally increases

the performance of the algorithm by decreasing the possible number of candidate literals, it makes it impossible to induce clauses like the grandfather relationship (`grandfather(?x,?y) if father(?x,?z), father(?z,?y)`) because the variable $?z$ which is necessary to express the grandfather predicate does not appear in the head of the clause. Since this puts a serious restriction on the kind of clauses that are inducible, we changed the implementation so that the restriction no longer applies. As we discussed in 4.3.2.5, the clauses which are generated by relative least general generalization contain a lot of redundant literals. We solve this problem by pruning the obtained clauses by means of *functional reduction* and *negative reduction.* We will give a more high-level overview of the implementation of the induction algorithm. For the complete source code and a discussion of it we refer to appendix B. Our induction algorithm works as follows:
The developer provides the tool with the following input:

- The positive examples in the classification
- Optionally: some negative examples.
- A set of background information which should be taken into account by the induction algorithm and which will extract facts that express this information out of the positive examples.

The algorithm will then follow the next steps:

1 Create, given the examples, facts that represent the background information applied to the examples. Let us explain this by means of an example. Suppose that the developer has classified a class $ClassA$. Since our tool takes into account all the subclasses of classes in a classification as background information, it will calculate all these subclasses. Suppose for our example that $ClassA$ has one subclass namely $ClassB$. Our tool will extract this knowledge by means of the SOUL predicate *subclass* and generates a fact `subclass([ClassA],[ClassB])`. This is the extensional background information for the relative least general generalization.

2 The algorithm will take the first two positive examples and calculate the relative least general generalization of the examples with respect to the background information.

3 The clause will be reduced by means of functional and negative based reduction.

4 The positive examples which are covered by the reduced clause will be removed.

5 If there are no positive examples left, then return the set of clauses. Else repeat from step 2.

5.3.2 Extraction of the background information

The induction algorithm needs as an input a collection of facts which express the background information we have about the examples in the classification. The choice of background information is very important: it will determine which predicates will be used in the induced rules and will thus have a large influence on the overall quality of the rules we find. To extract this background information out of the examples we are going to analyze them. We do this by making use of the facilities SOUL and LiCoR offer us. Our tool will, given an input example, use SOUL predicates to determine the information that holds for this example and to generate a fact that expresses this information. The Software Views Inducer considers the following SOUL predicates when analyzing the examples in the classification:

- **classInNamespace(?class,?namespace)** Every class in Smalltalk belongs to a namespace.
- **protocolmethod(?class,?selector),?protocol)** A developer can group a number of methods implemented on the same class in a protocol.
- **instVar(?class,?variable)** The instance variables of a class.
- **subclass(?subclass,?class)** All the classes that inherit from a class.
- **classImplementsMethodNamed(?class,?selector)** The selectors of the methods implemented on a class.
- **methodArguments(?class,?selector,?arglist)** The arguments of a method.
- **methodStatements(?method,?statements)** The statements in the body of a method.
- **superclass(?superclass,?class)** The class from which a certain class inherits.
- **classInCategory(?class,?category)** All the classes in Smalltalk belong to a category.
- **methodWithAssignment(?method,?assignment)** All the statements in the body of a method which make an assignment to a variable.
- **variablesUsed(?method,?variables)** All the variables that are referenced in the statements of a method.
- **methodWithSend(?method,?receiver,?message,?arguments)** All the messages that are sent in the body of a method.

	class	**method**	**statement**
namespace	X		
protocols	X	X	
inst. vars.	X		
subclasses	X		
method names	X	X	X
arguments		X	
statements	X	X	X
superclass	X		
category	X	X	
var. assign.		X	X
var. used		X	X
message sends		X	X
method calls		X	
overr. methods		X	
hierarchy	X		
other class.	X	X	X

Table 5.1: The applicability of the background information

- **isSendTo(?receiver,?message,?method)** Methods who get called from within other methods.
- **overriddenMethod(?method,?overriddenmethod)** Methods who override other methods in the class hierarchy.
- **hierarchy(?root,?class)** All the superclasses of a class except Object.
- **other classifications** In certain cases it is interesting to find the links between a number of classifications. To allow our tool to detect these links, we can also include the elements of the other classifications as background information while inducing a rule.

This set of types of background information is not very large, but as we will see later on when we are conducting a few experiments, it suffices for inducing rules which are good enough. It is not a limitation of our tool since the tool offers a framework for creating new kinds of background information (see appendix A for more information about this framework). As we already discussed, we can have 3 kinds of elements in the Learned Classifications: classes, methods and statements. It is clear that not all of the background information as described above is applicable to each kind of classification element. The subclass relationship for instance can not be applied to a statement, nor can method arguments be applied to a class. Table 5.1 gives an overview of the applicability of all of the background information we have

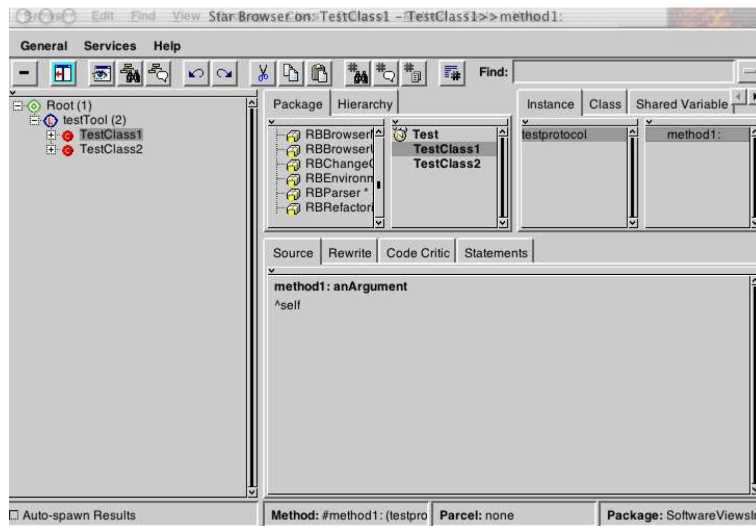

Figure 5.2: a screenshot of the classification of our example

implemented in the tool. If the user of the tool selects a kind of background information that is not applicable to the items of the classification, then this information will be ignored.

5.4 Example usage of the Software Views Inducer

In this section we give an example of how we can use the Software Views Inducer. It is not the intention of this section to demonstrate which rules we can learn out of a classification of software entities nor is it the intention to demonstrate the evolution of documentation. We show the workings of the Software Views Inducer so that the reader can get insights in how the experiments in the following chapter were conducted. Consider the following (trivial) example: the classes TestClass1 and TestClass2 which share the following properties:

- An instance variable `var1`.
- A method `method1` which does a return of *self* and is part of the protocol `testprotocol`.
- Both classes are part of the `Test` namespace and are in the category `SoftwareViewsInducerTest`.

We start with creating a Learned Classification named 'testTool' and classify the two classes as the elements of the classification (see screenshot 5.2). We right-click on the classification and chose the 'Induction' option. The

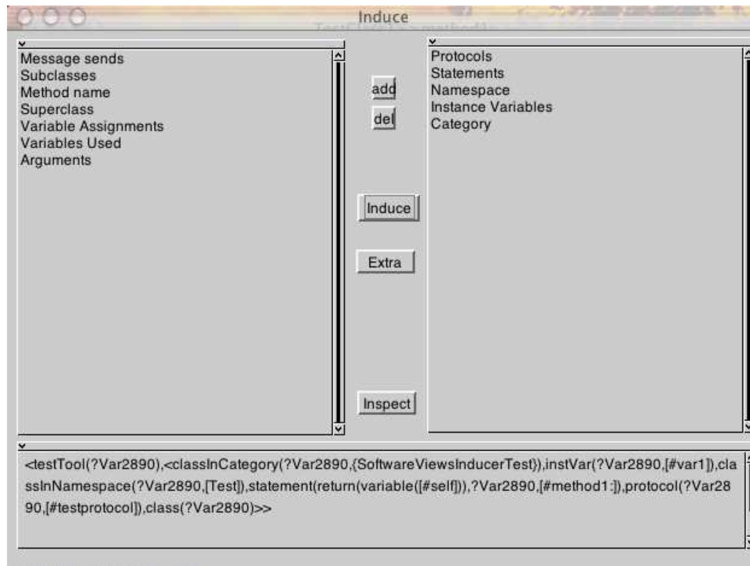

Figure 5.3: a screenshot of the inductionwindow

Induction Window will now open on the classification and we select the background information *namespace, protocol, statements, instance variables and category* and click on the induce button to let the system induce a rule for the examples. We get the following rule (also see the screenshot 5.3):

```
testTool(?element) if
      classInCategory(?element,{SoftwareViewsInducerTest}),
      instVar(?element,[#var1]),
      classInNamespace(?element,[Test]),
      statement(return(variable([#self])),?element,[#method1:]),
      protocol(?element,[#testprotocol]),
      class(?element)
```

This rule is the same as the one we can find in the screenshot, except that we renamed the variables so that it is easier to read (the induction algorithm will not introduce variables with a logic name like ?*element*, but will instead use variable names like ?*var*030). If we use the testTool predicate we obtained out of the induction algorithm to pose a query, we get two possible bindings for ?element namely TestClass1 and TestClass2, which are the elements of the classification we started out with.

5.5 Summary

In this chapter we have introduced the Software Views Inducer, the tool we created in the context of this dissertation. This tool wants to provide a trade-off between the advantages of Software Classifications as they are used in the work of De Hondt [DH98] and the advantages of Intentional Software Views [MMW02a]. We have taken a look at our situation and have argumented why relative least general generalization is the best algorithm to solve our problem. We also showed how we can use logic meta programming in our tool to extract the background information out of the documentation. We also gave an overview of which kinds of background information are supported by the tool. The goal of our tool is to help a developer to easily create documentation that is robust when the software changes. In the next chapter we are going to validate these claims by conducting and discussing a few experiments.

Chapter 6

Experiments

In this chapter we validate the claims we made in the previous chapter. We will start with showing that the software views inducer is able to derive a set of logic rules from a classification and that these rules contain the intentional information about the classified software entities. We will then take a look at a small case study of how we can use the software views inducer to create robust documentation and how to evolve the documentation when the implementation changes.

6.1 Inducing rules for documenting design patterns

The first aspect of our technique we want to research is the quality of the rules that can be induced from a classification. In our experiments we will document a few design patterns [GHJV95]. We choose design patterns since they are widely used and generally easy to understand. They are interesting to take a look at since they impose high-level structural relationships between the elements of the pattern. In our experiments we will try to extract these relationships out of the documentation. Also, design patterns already have been used as a way to document software, so inducing rules out of instances of a pattern can result in useful documentation.

6.1.1 Design Patterns

Design Patterns originated from an engineering point of view on software construction. When a problem is encountered while writing a piece of software, the developers can apply some sort of template solution in a lot of cases. This can best be compared with an engineer who has to design a bridge: although it is a specific problem the engineer will make use of standard constructions and materials which are applicable to bridge building. A design pattern is a construction a software developer can use for solving

a particulate problem. A design pattern consists out of a name, a problem domain to which it is applicable, the structural description of the pattern and the consequences of applying it. It is clear that we can only express the structure of the pattern in our documentation. This structure can be described by a set of roles and the collaborations between these roles. The roles are the participants (usually classes or methods) of a design pattern. With collaborations we mean how the participants of a design pattern work together to solve the problem. For example, the messages that the participants send to each other are part of the collaborations.

6.1.2 The Visitor Design Pattern

6.1.2.1 The pattern

In this section we will take a look at the visitor design pattern and how it is structured and works. We already used the visitor pattern in section 2.1.1 as an example of an application of LiCoR. The visitor pattern is used to allow a looser coupling between an object hierarchy and the operations on that hierarchy. Now let us take a closer look at the different roles which belong to the visitor pattern:

- **Visitor:** All the concrete Visitors inherit from this class.
- **ConcreteVisitor:** Subclasses of Visitor which implement an operation on the hierarchy. For every element of the hierarchy, a separate method is implemented on the ConcreteVisitor which implements the behavior for that Element.
- **Element:** Abstract class from which all the ConcreteElements inherit.
- **ConcreteElement:** These classes implement an accept method which will accept a ConcreteVisitor and call the correct method on that Visitor.

Every different operation on the object structure (which consists out of *ConcreteElements*) is implemented on a different *ConcreteVisitor*. An operation is applied to a structure by calling the accept method on the structure with the visitor as argument. The accept method will then call the method corresponding with itself on the visitor with 'self' as an argument (this is called the double dispatch).

6.1.2.2 The experiment

As for an experiment, we are going to document an instance of the visitor pattern. In the implementation of SOUL a visitor is used to define the operations on the objects which represent Soul terms (see figure 6.1). If we fill in the roles of this instance then the *Visitor* is `SimpleVisitor`, *Element*

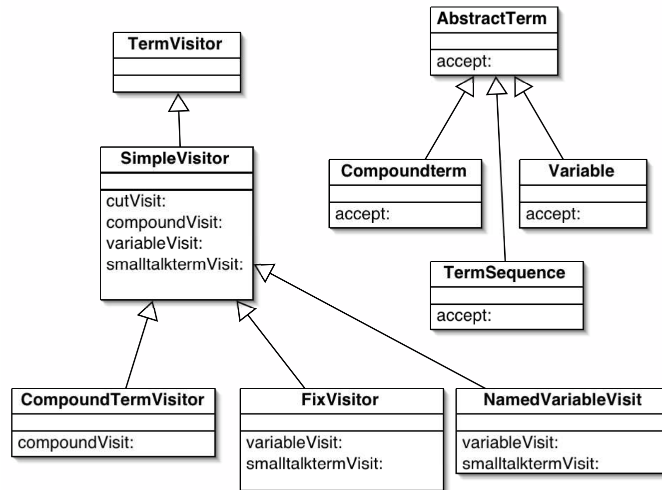

Figure 6.1: UML class diagram of the Soul Visitor

is `AbstractTerm`, the subclasses of `SimpleVisitor` are *ConcreteVisitors* and the subclasses of `AbstractTerm` are *ConcreteElements*. In light of this experiment, we create two classifications containing the following software entities which can be found in figure 6.1:

1. a classification named **Visitors** which contains the *ConcreteVisitors*, namely the classes `CompoundTermVisitor`, `FixVisitor` and `NamedVariableVisitor`.
2. a classification named **acceptMethods** containing all the methods on the *ConcreteElements* (these all implement a selector `#accept` and are implemented on the *ConcreteElements*).

If we use as background for the Visitors classification all the possible kinds of background information that are applicable to classes, we get the following rule:

```
Visitors(?class) if
    class(?class),
    classInNamespace(?class,[Soul]),
    classInCategory(?class,{Soul-Kernel},
    hierarchy(?class,[Soul.TermVisitor]),
    hierarchy(?class,[Soul.SimpleVisitor]),
    methodOverridden(?class,method([Soul.SimpleVisitor],?overriddenmethod)).
```

The rule above gives a description of the elements in the classification. It expresses that every class in the Soul namespace and Soul-Kernel category that overrides methods on `SimpleVisitor` and that is somewhere in the

inheritance chain `SimpleVisitor - TermVisitor` is an element of *Visitors*. Not only did we extract information about the location of the class in the image (namespace and category), we also extracted structural information: all the visitor classes are part of the same inheritance hierarchy (the hierarchy predicate will detect all the common superclasses and thus detects that all the Visitors have two common parents) and override methods on one of their common superclasses.
Let us take a look at the quality of this rule. If we would extend the SOULVisitor with a new kind of visitor and we do not violate the naming conventions (putting the new visitor in the Soul-Kernel category and in the Soul namespace), then our rule is able to detect the new visitor. Our rule is not overly general: since it makes use of this naming information, the rule will not detect any *Visitor* classes from other instances of the design pattern. When we discussed Intentional Views we manually supplied a rule for describing the Visitors view. If we look back at that predicate we can see that the rule we induced here shows many resemblances with the rule written by a human. Notice that the rule we induced is not a general rule for detecting Visitors. It can only be used to detect the *Visitors* that are part of the SoulVisitor.
Now let us take a look at the second classification. The rule we get when inducing while limiting the background information to protocol, method name, message sends, arguments, statements, variables used and method calls is:

```
acceptMethods(method(?class,[#accept:]))  if
1.       isSendTo(?class,[#accept:],?visitorselector),
2.       statement(return(send(
          variable(aVisitor,?visitorselector,<variable(self)>))),
          ?class,[#accept:]),
3.       argument(?class,[#accept:],1,variable(aVisitor)),
4.       methodWithSend(?class,[#accept:],send(variable(aVisitor)),
          ?visitorselector,<variable(self)>),
5.       classImplements(?class,[#accept:]),
6.       protocol(method(?class,[#accept:],visitor).
```

Let us explain the meaning of this rule. Literals 1 and 2 express that the statement of the body of an acceptMethod contains a message send to a variable named `aVisitor` with as one argument `self`. Literal 3 states that the method only has one argument and that this argument is called `aVisitor`. In literal 5 we can see that the acceptMethod is implemented on a selector named `#accept:`. Finally, literal 6 expresses that the method is implemented on protocol *visitor*.
We can see that the induction algorithm extracted the information based on naming conventions (using a variable `aVisitor`, implementing on pro-

tocol visitor,...) as well as the structural information: we can see that the double dispatch is also present in the rule. Notice that this rule again shows close resemblance with the rule for the visitor pattern we manually wrote at the end of section 2.1.1 and used in the Intentional View. We should also remark again that the rule we found here is only useful for the SoulVisitor: it is not general enough for detecting acceptMethods of other instances of the design pattern. The rule contains a bit of redundancy: the information that the accept method calls a visitorselector method with itself as argument is encoded multiple times in the rule. This duplication in the rule is the result of overlapping background information. The statement predicate will detect that all the methods in the classification implement a statement that expresses the sending of a message with `self` as argument. The methodWithSend predicate will detect a sort-like commonality of the items in the classification. Otherwise from making the rule a bit longer and perhaps harder to read, this redundancy does not introduce any problems.

6.1.2.3 Inducing a more general rule

In the previous section we showed that our tool is able to induce rules that cover the intention of the visitor design pattern pretty well. We also discovered that our rules are only applicable to one certain instance of the design pattern: since we only provided examples from that instance of the pattern, instance-specific information was extracted out of the examples. In the context of software documentation, this is not a disadvantage: we only wanted to create documentation for the SoulVisitor. So if we would obtain a rule that is capable of detecting eg. accept methods of other instances of the visitor pattern then it would not be useful for documenting the SoulVisitor. To demonstrate that our tool is capable of inducing more general rules than the ones we obtained in the previous section, we conducted the following experiment:
We created a classification named *generalAcceptMethods* and classified accept methods of two instances of the visitor pattern. To do this we used the accept methods of the SoulVisitor and the accept methods of the StarBrowserVisitor (the StarBrowser also uses a visitor pattern in its implementation). We obtained the following rule:

```
generalAcceptMethods(method(?class,?selector)) if
     argument(?class,?selector,[1],variable(?visitor)),
     methodWithSend(?class,?selector,send(variable(?visitor),
           ?visitorselector,<variable([#self])>)),
     classImplements(?class,?selector),
     classInCategory(?class,?category),
     statement(return(send(variable(
           ?visitor),?visitorselector,<variable([#self])>)),
```

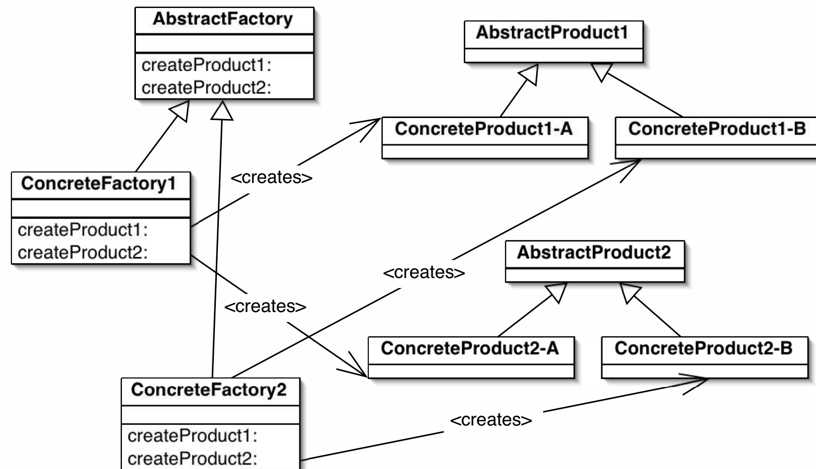

Figure 6.2: a UML case diagram of the Abstract Factory design pattern

```
        ?class,?selector).
```

The rule above looks a lot like the rule from the previous section, with the exception that it no longer contains instance-specific information. If we use the rule above to calculate the elements in the classification, we obtain **all** the accept methods in the Smalltalk image. Although we argumented that the rule from the previous section has a lot in common with the rule we manually wrote, we can now conclude that this more general rule resembles the hand-written rule even more closely.

6.2 Abstract Factory Design Pattern

6.2.1 The pattern

The Abstract Factory pattern is used when in a piece of software it is important that different families of objects can be used. The best example of this is the use of GUI toolkits (like Qt or GTK). The developer wants to create an application that is independent of the graphical toolkit that is used. This can be done by abstracting the way how objects of the toolkit are instantiated. If we take a look at the class diagram in figure 6.2 we see how this pattern is used. We can distinguish the following roles:

- **AbstractFactory:** This abstract class specifies an abstract method for every different kind of element that has to be produced.
- **ConcreteFactory:** These subclasses of AbstractFactory implement methods that return an instantiation of a ConcreteProduct.

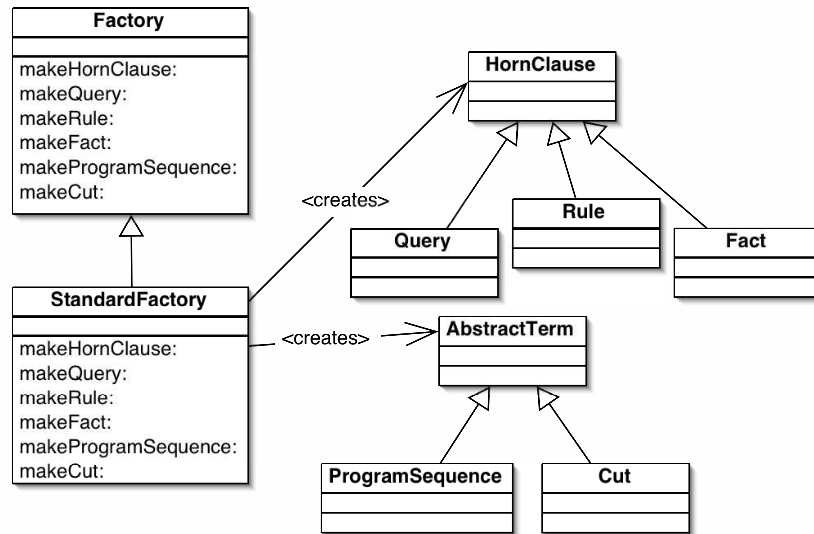

Figure 6.3: a UML case diagram of the SOUL factory

- **AbstractProduct:** The abstract class from which the products which are instantiated by a ConcreteFactory inherit from.
- **ConcreteProduct:** The implementation of the products of the ConcreteFactory.

If we look at the class diagram again, we can see that we have two families of objects: family one which consists out of `ConcreteProduct1-A` and `ConcreteProduct1-B`, family two out of `ConcreteProduct2-A` and `ConcreteProduct2-B`. Suppose now if we want to use the first family of objects, we instantiate a `ConcreteFactory1` whose methods will initialize object from the first family. If we want to use the second family, we use an instance of `ConcreteFactory2`. Notice that we can write our code which uses the products independently from the chosen family.

6.2.2 The experiment

For the experiment we are again going to take a look at an instance of the design pattern. If we take a look at the implementation of SOUL, we see that a factory is used to create objects which are related to Horn clauses. In the class diagram (6.3) we see that only one family of objects is used. Notice that for this one family two root classes or used (we have two instances of the *AbstractProduct* role, namely `AbstractTerm` and `HornClause`). Since we have only one family of objects, we also have only one *ConcreteFactory* (`StandardFactory`) which inherits from the *AbstractFactory* (`Factory`). To

create documentation for this instance of the design pattern we create two classifications:

1. A classification named **concreteProducts** which contains all the *ConcreteProducts*. In our example this are the classes which inherit from `AbstractTerm` and `HornClause`.

2. A classification with as name **factoryMethods** in which we put methods implemented on `StandardFactory`. For every kind of *ConcreteProduct* there is such a method. This method returns an instance of that *ConcreteProduct*.

If we induce a set of rules for the concreteProducts classification with as background knowledge protocols, instance variables, namespace, method names, class hierarchy and category we obtain the two following rules:

```
concreteProducts(?class) if
     class(?class),
     classInCategory(?class,{Soul-GrammarClauses}),
     classInNamespace(?class,[Soul]),
     hierarchy([Soul.HornClause],?class),
     protocol(?class,[#printing]).

concreteProducts(?class) if
     class(?class),
     classInCategory(?class,{Soul-GrammarClauses}),
     classInNamespace(?class,[Soul]),
     hierarchy([Soul.AbstractTerm],?class),
     protocol(?class,[#printing]),
     classImplements(?class,[#printing:]),
     classImplements(?class,[#printOn:]).
```

The first rule expresses that every class in the hierarchy of `HornClause` which is in the Soul namespace and the Soul-GrammarClauses category and which implements the `#printing` protocol is a concreteProduct. If we take a look at the second rule we can see that it expresses almost the same thing, with the exceptions that all the classes are in the `AbstractTerm` hierarchy. The first thing we notice is that our tool detects two possible rules for the concreteProducts classification. If we take a look at the diagram in figure 6.3, we see that there are two root classes (*AbstractProducts*) for the products of the factory, namely `HornClause` and `AbstractTerm`. Our tool will see that some of the *ConcreteProducts* are subclasses of the first *AbstractProduct* and some are subclasses of the second one and thus produces two rules.
If we look at the rules we see that they do not contain much structural

information. This is what we excepted: other than a common superclass and some naming conventions, *ConcreteProducts* generally have not a lot in common. Still, the rules we found here are not worthless when having to evolve the documentation. If a developer makes changes to the implementation or adds a new product, the naming conventions like for instance the fact that all products in our example implement a protocol `#printing:` will help keeping the documentation up-to-date. Although naming conventions are not as robust as structural information, they are invaluable when creating software that is easy to understand. Notice again that the rule we found here is not a general rule for detecting *concreteProducts* but is limited to one instance of the the pattern: the SOULFactory.
The rule which is obtained by using our induction tool on the *factoryMethods* classification is more interesting. As background information we use the namespace, category, method names, message sends, assignments, statements and overridden methods. We also include the elements of the concreteProducts classification as background information. The rule we get is:

```
factoryMethods(method([Soul.StandardFactory],?selector)) if
     factoryProduct(?product),
     statement(return(variable(?product)),
          [Soul.StandardFactory],?selector),
     classImplements([Soul.StandardFactory],?selector),
     methodOverridden([Soul.StandardFactory],?x,
          method([Soul.Factory],?selector)).
```

This rule expresses that every method implemented on `StandardFactory` which returns a factoryProduct is a factoryMethod.
If we take a look at the above rule, we can see that a lot of structural information about the factory methods is present:

- The methods on the ConcreteFactory override a method on the AbstractFactory.
- All the methods exist out of one statement which returns a variable ?*product*.
- This variable ?*product* is a factoryProduct (for this predicate we induced a rule in the beginning of the experiment).

The rule we obtained fits the intuitive definition we would give of a *factoryMethod*: a method that is implemented on a *factory* and that returns a *factoryProduct*. An interesting property about the rule we induced here is that it puts a link between two classifications. The `factoryMethods` predicate detects that every method in the classification returns an element of another classification: the *factoryProducts* classification for which we induced

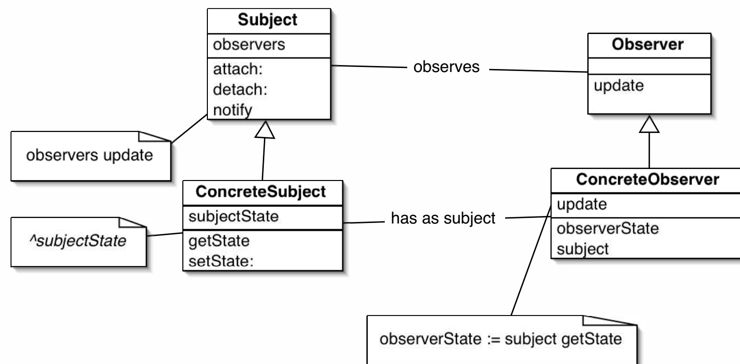

Figure 6.4: a UML class diagram of the Observer design pattern

a rule earlier in this section. In contrast from the rule for *factoryProducts*, this rule is almost entirely based on structural information instead of naming conventions. As is with the rule for *factoryProducts*, this rule is limited to one instance of the factory design pattern namely the Soul Factory. This is a consequence of the fact that we only documented artifacts of this instance. With the visitor pattern we showed that it is possible to induce a more general rule out of multiple instances of a design pattern. Of course, the same thing can be done with the factory pattern (we omit this experiment since it is almost identical to the visitor experiment).

6.2.3 The Observer Design Pattern

6.2.3.1 The Pattern

The last pattern we will take a look at is the Observer Design Pattern. It is used in situations where objects are dependent on the state of other objects. A good example of an observer is the Model-View-Controller architecture as is used in many cases to construct the interactions between an application and the graphical user interface. The interface of the application only wants to change when the internal state of the underlying application changes. We can distinguish the following roles in the pattern:

- **Subject:** This is the class from which all the ConcreteSubjects inherit. It contains a list of observers and offers the facilities for observer management.
- **ConcreteSubjects:** These classes implement the behavior of the application and call the notify method every time their state changes.
- **Observer:** This class is the interface of the ConcreteObservers.

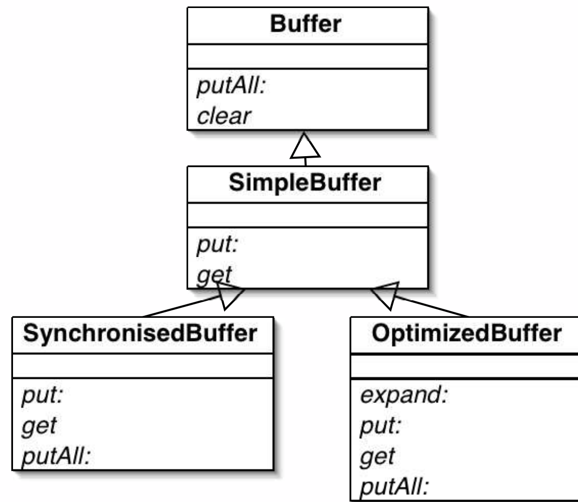

Figure 6.5: UML class diagram of the buffer example

- **ConcreteObserver:** These classes are interested in the state of a ConcreteSubject. They implement the update method which synchronizes the state of the Observer and the Subject.

A UML class diagram can be found in figure 6.4. Whenever a *ConcreteObserver* wants to observe a *ConcreteSubject*, it will use the attach method on the subject to register itself as an observer. Whenever the Subject makes a change to its state, it will call the notify method. This method will send an update message to all the observers which will synchronize their state with the state of the *Subject*.

6.2.3.2 The experiment

This experiment differs from the previous ones: we do not want to make classifications in order to document an instance of the pattern. Instead we are going to induce a rule which expresses the places where a state change in the *Subject* occurs. The state changes in a *Subject* are the tricky part when implementing the observer pattern: every time when a method makes a change to the internal state of the object, the notify message has to be called in order to signal the *Observers* of the change. It is easy to see that this can be very error prone since every method which makes a state change has to call the notify method and when this is forgotten one time, it can lead to an inconsistency between *Subject* and *Observer*. It also makes the code which implements the *Observer* pattern harder to understand and debug. For the experiment (that is also described in [TBKG03]), consider a buffer that is implemented as is shown in figure 6.5. In the diagram we have only

written down methods that do a state change. If we create a classification named *stateChanges* and classify all the methods on our implementation of a buffer, we can induce the following rules:

```
1.  stateChange(method(?class,?selector)) if
     classImplementsMethod(?class,?selector),
     statement(assign(variable(?var),?expression),
          ?class,?selector),
     instVar(?var,?class).
2.  stateChange(method(?class,?selector)) if
     classImplementsMethod(?class,?selector),
     statement(send(variable(content),#addFirst:, ?expression),
          ?class,?selector).
3.  stateChange(method(?class,?selector)) if
     classImplementsMethod(?class,?selector),
     statement(send(variable(content),#removeLast:, ?expression),
     statement(return(?exp),?class,?selector),
          ?class,?selector).
4.  stateChange(method(?class,?selector)) if
     classImplementsMethod(?class,?selector),
     statement(?statement,?class,?selector),
     statement(send(variable(#self),?message, ?args),
          ?class,?selector),
     stateChange(method(?class,?message)).
```

Now let us take a look at these four rules:

- Rule 1 expresses that a method that assigns a value to an instance variable is a statechange.
- Rules 2 and 3 state that methods which send the `#addFirst:` or `#removeLast:` selector to the variable **content** make state changes.
- In rule 4 we can find that a method that recursively calls a state changing method is also a state changing method.

The rules are able to describe the intention of a state change (especially rules 1 and 4 which describe the assignment to a variable or the recursive call of a state changing method). Notice that they are general enough to detect state changes in classes outside of the buffer example. Rules 2 and 3 on the other hand are too restrictive to do this: they hard-code the fact that two methods in the buffer implementation send a message to the content variable. Rule 3 also expresses that method has to contain a return statement. The reason why we obtain these two rules which can not be used to detect state changing in general is because they are specific to our implementation of the buffer.

	seconds SOUL	seconds PROLOG
Visitors	80	4
acceptMethods	7	0,7
concreteProducts	70	3,6
factoryMethods	18	1,2
AdapterMethods	1	0,2
AdapteeMethods	2	0,5
stateChange	90	5

Table 6.1: the execution times of the induction algorithm for each induced classification

6.2.4 Discussion

The rules as shown in the experiments are literally the output of the algorithm. The only change we made to them was renaming the variables in order to make the rules more human readable. After studying the induced rules we remarked that the information they contain is a mix of:

- Naming conventions
- Structural information

The ideal information we want to extract for creating robust documentation are the high-level structural relationships between the elements in the classification. Since we include background information like protocol names, method names, the category of classes,... our tool also detects naming conventions. One can consider this a limitation of our tool. If the developer who makes changes does not follow these conventions, then the changes will not be reflected back into the documentation. In practice however, it is good programming style for a developer to respect naming conventions since they make it easier for other programmers to understand the code. Since our tool can extract these naming conventions out of the source code, they become an active part of the documentation. In some cases (like the *concreteProducts* classification) there is no real structural information that can be induced out of a classification and the naming conventions are the only documentation we have got.
We opted for creating documentation for design patterns since they are a well-understood example and contain a lot of high-level structures. From the experiments we can conclude that our rules are able to capture the intention behind the elements of a classification. The rules sometimes contain a little bit of redundancy, but this does not change the overall quality of the rules much. In case of the visitor and factory pattern we induced rules that are instance specific. They are too restrictive to be used to detect instances of the design patterns but are good as documentation for one instance of the

pattern. By obtaining rules for the state changes in the observer pattern and by inducing a rule for the *generalAcceptMethods* we showed that our tool is also capable of detecting more general rules.

Now let us discuss the disadvantages of our tool and some problems we have encountered. In the first column of table 6.1 we can see execution times of the induction algorithm implemented in SOUL for each classification. As the reader can see, this is one of the largest weaknesses of our tool. When the amount of background information increases, the time needed to calculate the rules for the classification increases from a couple of seconds to over a minute. We tried to solve this problem by using the PROLOG logic kernel instead of SOUL. This is more of a workaround than a solution since the integration between the PROLOG kernel and our tool does not work optimally. Using the PROLOG kernel reduced the execution time of for instance the *stateChange* classification from 90 seconds down to 5 seconds. As a consequence of this speed problem we also had to put some limitations on our induction algorithm.
The resulting rules we obtain out of a classifcation are dependent on the order of the examples in the classification. In the experiments we have put the examples in such an order that the rules were optimal. This is not a problem of relative least general generalization or ILP in general. Implementations of induction algorithms like for instance GOLEM [MF90] solve this problem by computing the rlgg of one example with respect to a set of other examples and choosing the best rule. If we would want to implement this in SOUL, the number of rlgg-operations would increase significantly. This would make our algorithm a lot slower which would render it impossible for us to induce rules in an acceptable amount of time.
We also noticed that the quality of the rules is dependent on the number of elements in the classification and the size of the background information. If we do not provide enough examples for our algorithm then the rules we obtain are overly general of specific. The other problems we addressed with our tool are a consequence of the implementation. Too general or too specific rules are a common problem of machine learning approaches. We can alleviate this problem by turning our documentation tool into a semi-automatic system, but this has not been tested yet.

Although we have showed here that the rules we obtain out of the Software Views Inducer contain relevant and non-trivial information about the software we are trying to document, we have not yet used these rules to evolve the documentation of a piece of software. In the next section we are going to take a look at how we can do this.

6.3 Using Software Views Inducer to evolve the documentation

In this section we will take a look at an example of how we can use the Software Views Inducer to evolve the documentation of a piece of software when changes to the implementation have been made. We will use a simple case study: we already induced rules for the *Visitors* and the *AcceptMethods* classification of the SOULVisitor. We will now adapt the SOULVisitor and see if our rules are able to detect the changes.

6.3.1 Visitors

First let us start with checking which visitors are detected with the rule we induced in the previous section. If we launch the SOUL query `if Visitors(?element)` we get the following bindings for the variable ?element:

- Soul.FixVisitor
- Soul.CompoundTermRenamingVisitor
- Soul.VariableAndUnderscoreVariableVisitor
- Soul.NamedVariableVisitor

We now extend the SOULVisitor by adding a new class `Soul.TestVisitor`. We implement this class by:

- Creating a subclass of `SimpleVisitor`
- Implementing a few visitmethods on this class so that it does dummy operations with the elements.
- We follow the coding conventions: the class is defined in the Soul namespace and the Soul-Kernel category.

If we now recalculate the elements of the *Visitors* classification then the `Soul.TestVisitor` appears in the results.

6.3.2 AcceptMethods

We are now going to do a similar operation with the *AcceptMethods*. First we compute all the accept methods by using the *AcceptMethods*-predicate. This results in:

```
method(Variable,#accept:)
method(UnderscoreVariable,#accept:)
method(UnaryMessageFunctor,#accept:)
method(SmalltalkTerm,#accept:)
method(CompoundTerm,#accept:)
method(Constant,#accept:)
method(method(UppedObject,#accept:)
method(DelayedVariable,#accept:)
method(Cut,#accept:)
method(TermSequence,#accept:)
method(QuotedCodeTerm,#accept:)
method(UppedObject,#accept:)
method(KeywordFunctor,#accept:)
```

Note that the methods we find here are not only the ones we classified when we created the rule, but also other correct methods which are detected by our rule.
For checking wether the rule can detect changes in the implementation we are going to adapt the SoulVisitor in two ways:

- Creating a new concrete product class `TestProduct` that follows the coding conventions and implements an accept method.
- Removing the accept method on the `Cut` class.

If we run the *AcceptMethods* predicate again we will have a new accept method `method(TestProduct,#accept:)`. Also, the accept method on the `Cut` class is no longer a part of the elements of the classification.

6.3.3 Discussion

With this little experiment we show that the rules we have induced with the Software Views Inducer are able to evolve the documentation whenever changes are made to the implementation. In the two experiments we conducted here we adapted the implementation of the SOUL language and checked wether our rules were able to detect those changes. We can conclude that our rules detect the correct elements of the classification before we made changes. By using the rule we were also able to bring the documentation back up-to-date.
We have not tested our approach on a large system with a huge class hierarchy. This is due to the fact that our current implementation lacks the performance to conduct experiments like this in a reasonable amount of time. Notice that the possibility of updating the documentation is related to the quality of our rules: if the rules are too specific then they will not be able to detect changes in the implementation. If our rules are too general then it might happen that we falsely detect elements as part of a classification.

6.4 Conclusion

The goal of this chapter was to show that the rules we obtain with the Software Views Inducer are rich enough to reveal the intention behind the documentation and are useful for keeping documentation and implementation synchronized. In the first part of the chapter we looked at the quality of the rules we obtain by induction and can conclude that the rules extract information that is based on naming conventions and structural properties (like inheritance information, message calls, statements,. . .). The rules we obtained in most cases showed similarities with hand-written rules for the same problem and in general succeed reasonably well in revealing the intention behind the classification.

In the second part of the chapter we used the rules we induced for the Visitor pattern to evolve the documentation when the implementation changes. We can conclude that the rules we obtain with our tool are able to correctly update the documentation whenever changes are made to the implementation. In this chapter we also discussed the problems we encountered with our tool, namely the speed issues with SOUL and the problem with the order of the artifacts in the classification. We also noticed that the quality of our rules is strongly dependent on the number of examples and the background information. This is not a consequence of our implementation but rather a problem of machine learning techniques. As we already discussed, we believe that a semi-automatic tool can help alleviate this problem.

Chapter 7

Related Work

In this chapter we will take a look at related research which addresses the same problem as we do in this dissertation.

7.1 Co-evolution of documentation and implementation

In chapter 2 we already discussed Software Classifications [DH98] and Intentional Software Views [MMW02a], which are an extension to Software Classifications. Besides from these two approaches, research has been done on how documentation and implementation can be co-evolved. In his PhD thesis, Wuyts [Wuy01] not only introduces the logic meta programming language SOUL and the predicate library LiCoR which we have used in the implementation of Software Views Inducer, he also uses SOUL to direct the co-evolution of design and implementation. The design of a piece of software can be considered a part of the documentation. We can look at it as an abstraction of the implementation. If we could make a causal link between the design and the implementation, we could make sure that they are synchronized. Wuyts solves this problem by manually expressing the design as a SOUL program. Every time changes are made to the implementation, we can bring the design back up-to-date by using this logic program. This work is closely related with our tool. The Software Views Inducer extends the idea of this work by inducing the SOUL rules which are used to evolve the documentation.

Tourwé [Tou02] proposes another interesting technique for co-evolving design and implementation which makes use of LMP. The design of a program can be documented by means of design patterns. Every design pattern consists out of a number of meta-patterns. A collection of high-level transformations has been defined on these meta-patterns which can be used to evolve the software and update the documentation by checking for broken

design structures.

The last LMP solution we discuss here is [Men00b]. Mens proposes an expressive language for describing the architecture of a piece of software and the mapping onto the software entities in the implementation. This language allows the definition of multiple views on the architecture of the system. Each such *architectural view* concentrates on a given part of the structure of the software. Along with the language, an algorithm is specified that can be used to check the conformance between the implementation and the architectural views. The differences which are found by this check can then be used to update the documentation.

Reflexion models [MNS95] are a somewhat different approach for keeping documentation and implementation in sync. Reflexion models offer a language to describe the design of a piece of software. They will return the differences between that design description and the actual implementation of the application. These differences can then be used to synchronize the design with the implementation.

7.2 Code Browsing Approach

Another approach for making adaptations in a poorly documented piece of software are code browsing tools. These tools allow the developer to easily understand which pieces of code belong together to implement some functionality such that making changes becomes easier. Their main goal is to make it possible for a developer to browse cross-cutting code: code that can not be put in one module of the system, but instead is scattered throughout the software. A first tool that was introduced was Aspect Browser [GKY99]. This tool allows the developer to browse the code making use of coding conventions like method names and variable names (doing a text-based analysis). It is clear that when the coding conventions are only partially followed (or not at all), that the Aspect Browser is not very useful. The Aspect Mining Tool [HK01] overcomes this disadvantage by not only browsing the code based on the naming conventions, but also doing a type-based analysis. Text-based and type-based analysis are complementary techniques. The Aspect Mining Tool showed that using the combination of both can lead to interesting results.

Robillard and Murphy introduced another technique: the Feature Exploration and Analysis Tool (FEAT) [RM02]. They propose a formalism for expressing the cross-cutting code in a software system: Concern Graphs. The nodes of such graphs are the classes, methods and fields of the application; the edges express the relationships between the nodes: method calls,

read and writes of variables, class hierarchy relationships,... Also the mapping from the Concern Graph onto the source code is defined. The FEAT tool uses a Concern Graph to allow the developer to browse the source code. To this extent a set of operations (like expanding all the methods in a class, returning all the dependencies of a node,...) are implemented which the developer can use to walk through the Concern Graph and thus browse the code.

The last code browsing tool we will take a look at is JQuery [JDV03]. JQuery wants to unite the advantages of hierarchy browsers and query languages. Hierarchy browsers allow the navigation of the code for a particular relationship (a class hierarchy browser for instance shows the inheritance relationship). The downside of hierarchy browsers is that they are rather limited in terms of expressiveness and that a developer has to switch tools to get a different view on the software. Query languages (of which SOUL might be considered an example) do not have this limitation. The problem with query languages is the impossibility of getting all the information a developer wants with a single query, so the developer writes a query, looks at the result, writes another query and so on. The exploration path which connects these queries gets lost and the developer will soon loose an overview about the software. JQuery combines the functionality of hierarchy browsers and query languages by implementing a tool that offers hierarchical browsers build on top of a query language.

7.3 Using Machine Learning Techniques in Software Engineering

We are not the first to use machine learning techniques in the field of software engineering. In fact, for over two decades research has been conducted how techniques from artificial intelligence, which already have proven their use in other fields, can be applied to the software development process.

Zhang and Tsai give an overview of most of this research in [ZT02]. Since there are many different approaches to Machine Learning, we will limit ourselves to taking look at how Inductive Logic Programming can be used throughout the software development process. Cohen and Devanhu use ILP in the process of software quality prediction [CD97]. They made a comparative study of which ILP method best can be used when predicting software faults for C++ programs.

A large part of the software engineering process is the maintenance phase. During this phase the software is adapted to meet new requirements the client imposes, to use other technologies and to correct bugs. This last part,

when faults are removed out of the software, is called the corrective maintenance. Like with all other phases of the development process, a cost has to be determined for the bug fixing. Although experienced maintainers are able to give an accurate estimation of the cost, the whole process remains informal, error-prone and poorly documented and thus hard to replicate. [dALM98] takes a look at how ILP (and Decision Tree Learning) can be used to learn models which can be used for cost estimation.

When having to make a change to an application, one of the biggest problems a developer may encounter is the need to understand how the application works. We already took a look at code browsing tools that can be used for this extent. Another approach for understanding a piece of software is proposed in [Coh95] where inductive logic programming is used to extract the specifications (which can bring insights into the software) out of the source code.

Another important aspect of developing software is testing the application to check wether it functions correctly. This is mostly done by writing a collection of programs which check if the output of a part of the software corresponds with the expected value. This can be a tedious job: if the intended output is changed or it new functionality is added the test cases have be to updated. [BG96] proposes a system where ILP is used to derive test cases out of a program.

Chapter 8

Conclusions

8.1 Problem Summary

During its lifetime a piece of software gets changed a lot of times due to maintenance tasks or new needs that are imposed on the application. When a developer has to make changes, a decent understanding of the system is necessary and thus high-quality documentation is needed. However, in a lot of cases, the required documentation can suffer from a number of problems. One of the most common problems is that whenever a programmer makes changes to the implementation of a piece of software, updating the documentation is often omitted of forgotten. This will result in documentation that is out of sync with the implementation and thus useless for a developer who has to understand a system in order to adapt it. The cause of this problem is the way documentation is created nowadays: it is not an integrated part of the development process and development tools almost never offer decent support for it. Two interesting ways of documenting an application are *classifications* and *views*. They allow a developer to create documentation for a piece of the code by grouping software entities. Approaches that make use of these techniques can be classified as:

- Tools that put the emphasis on making it easy for a developer to create documentation. The downside of these approaches is that the documentation that is created is not very robust with respect to changes in the source code and also that it is not always clear what the original developer intended with it. e.g. *Software Classifications* [DH98]

- Tools that are robust and encapsulate the intention behind the documentation rather well, but imply that the developer has extensive knowledge about the software that is documented. They also require the developer to express the documentation in some kind of language, which can be error prone and in many cases hard to do. e.g. *Intentional Software Views* [MMW02a].

8.2 Contributions

In this dissertation we have introduced the *Software Views Inducer* documentation tool. This tool unites the benefits of both approaches we mentioned above. It is based on the StarBrowser [Wuy], an implementation of the ideas proposed in [DH98]. This allows our tool to offer a developer an easy way to document an application by creating classifications and putting software entities in these classifications by means of simple drag & drop operations. The documentation that is created with the StarBrowser is not robust with respect to changes and does not have a description of its intention. The *Software Views Inducer*, however, creates documentation that is robust and that gives a good description of the elements in the classification by extracting an intentional SOUL description out of the classification. To offer this functionality we used *relative least general generalization*, an algorithm from the domain of *Inductive Logic Programming*, to extract logic rules written in the language SOUL from the classifications the developer has made. These rules describe high-level relationships between the elements of the classification and thus provide the developer with clear information about the artifacts that are documented. More importantly, these rules are then used to update the documentation whenever the implementation changes.

We have validated our claims by conducting a few experiments with the *Software Views Inducer*. As a small case study we used the tool to create documentation for instances of design patterns. We opted to use design patterns since they are well-understood and contain a large number of high-level structural relationships. A first set of experiments were executed to check wether the rules we obtain out of instances of patterns are able to reveal the intention behind the documentation. These experiments showed us that our tool is able to detect naming conventions as well as structural information (like for instance inheritance information, message calls,...). We also took a look at the information that is contained in the obtained rules. We can conclude that these rules are in a lot of cases very similar to hand-written rules for the same concept or give a good description of the intuition someone might have of the design patterns.
We also conducted a second experiment to validate the claim that the rules we extracted can be used to evolve the documentation of a piece of software. To do this we used the rules we obtained for an instance of the visitor design pattern from the first set of experiments. We used these rules to calculate the elements which belong to the classifications and compared these elements with the software artifacts that were detected after we made some changes to the implementation. From this experiment we can conclude that our tool is able to create documentation that is more robust with respect to changes in the implementation.

Although we can conclude from our research that the *Software Views Inducer* can be successfully used to easily create documentation that is robust and we feel that we have proven the claims we made in this dissertation, we also encountered a few problems. Our implementation of the ILP algorithm is rather limited. Putting the items in the classifications in a different order can result in the induction of a totally different set of rules. This is not a general problem of algorithms which are based on *rlgg* like GOLEM but is rather a consequence of the speed problems we encountered with SOUL. Changing our implementation in such a way that the order of the items would not make any difference would imply a considerably larger amount of generalization operations which would make it impossible to conduct experiments in a reasonable amount of time.
We should also be careful that the rules we obtain are not too general or too specific. Although we did not have encountered any problems as such in our experiments, we are aware that they might happen. These problems become less probable if the number of software artifacts in the classifications and the number of background information increase. We could make the number of examples larger by making our tool semi-automatic and requiring input from the user from time to time. We can make sure that our tool has enough background information by using a larger number of predicates for analyzing and extracting this information out of the examples.

8.3 Future work

Besides from improving the implementation of our ILP algorithm and extending the tool with new kinds of background information, we can also propose a few other topics which may be interesting to research in the context of this work.

When writing logic programs, the order of the clauses in the body of the rule is important for the overall performance of that rule. In our current implementation of the ILP algorithm we do not make any assumptions about the performance of the rule we induce. Since we want to use this rule to calculate the elements of a classification, it might be interesting to make sure that the rule is optimal.

For now the tool is used by creating a classification, classifying software artifacts and calculating and using rules for evolving the documentation. We could also adapt the tool so that the developer is guided while creating the documentation so that the process of classifying the artifacts becomes incremental. While the developer is classifying software entities the tool could create a rule in the background and calculate all the artifacts which

are covered by the rule and offer this list to the developer as a set of possible items also to consider to add to the classification. This technique would make it even easier for a developer to document a software system and would help minimizing the possibility of obtaining too general or too specific rules.

A number of SOUL rules have been written manually to detect all the instances of a design pattern in a Smalltalk image. The problem with these rules is that they are based on the text-book form of the design pattern and do not detect an instance of the design pattern if the developer deviated too much from this form. Although it is too much work to write rules that detect all these variations of a pattern by hand, we could try to do this by applying ILP to the problem.

In this dissertation we only used our tool to create documentation for object oriented style software. It might also be interesting to see how we can apply our tool for documenting cross-cutting concerns as in Aspect Oriented Programming and using these rules to learn crosscuts or pointcuts.

We only considered ILP as the learning algorithm to extract information out of classification. Perhaps the use of other learning techniques (like for instance analytical learning [Mit97]) can result in better or more interesting results.

Appendix A

Background Information Framework

In this appendix we are going to take a look at how the framework for background information in the Software Views Inducer works. This framework allows a developer to easily add a new kind of background information. We are going to demonstrate its functionality by showing step by step how we can implement the *subclass* background feature. The class diagram in figure A.1 shows the base class of the framework,*AbstractFeature* and the methods that are relevant for adding new kinds of background information. We will start by saying which methods need to be implemented on a background feature class in order to have it work and what the methods are supposed to do

- **name** This method returns a string that is the name of the background information
- **getFeatureClass:** This method implements the behavior for the feature on classes
- **getFeatureMethod:** This method implements the behavior for the feature on methods
- **getFeatureStatements:** This method implements the behavior for the feature on statements
- **createFeature:element** In this method the representation of the background fact is generated.

If the *getFeatureClass:*, *getFeatureMethod:* or *getFeatureStatements:* methods do not get overridden then the standard implementation (the feature is not applicable to this kind of element) will be used.

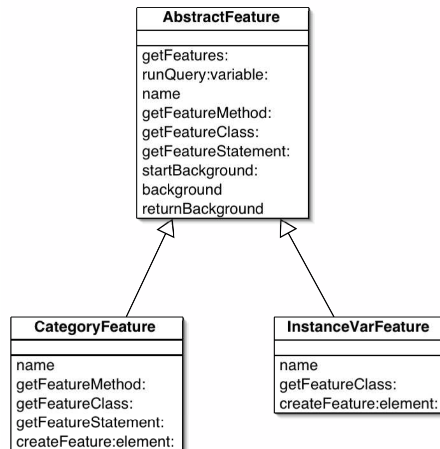

Figure A.1: a UML class diagram of the implementation of background features

A.1 Implementing the subclass background feature

As an example we will now guide the reader through the implementation of the *subclass* background information. We will start by creating a subclass of the *AbstractFeature* class which we will call **SubclassFeature**. In this class we implement the *name* method to return the string 'Subclasses'. The next method we will implement is *createFeature:element:*. This method has two arguments: the first is a s string containing information about the background, the second one is the classification element to which it is related. The method has to return a representation of the fact that the induction algorithm uses. The framework offers the following methods on the *AbstractFeature* class for easily doing this:

- **startBackground:** This method has as an argument the predicate name we want to use in the output fact. It will start a new background fact.
- **addClass:** Adds the argument (a class) of the method to the list of arguments of the fact.
- **addSelector:**
- **addLiteral:**
- **returnBackground** A method that will return the generated fact.

For our subclass example we want to generate facts that look like `subclass([RootClass],[Subclass])`. The code for the createFeature method

looks like this:

```
createFeature: aSubclassString element:aClass
  self startBackground:'subClass'.
  self addClass:aClass.
  self addLiteral:aSubclassString. "this is already a string"
  ↑self returnBackground
```

What this method does is start the creation of a fact with as name `subClass`. Add the class of the element in the classification of which we are calculating the background information as the first argument. Then add the subclass of this class as the second argument of the fact. Subclass is only applicable to classes, so the only method left for us to define in order to get subclass information of the classes in a classification as background information while inducing is `getFeatureClass:`. This method has to return a collection of background facts. A standard method to do his has been supplied and can be used while implementing the *getFeatureClass:*method namely *createFeatures:element*. The *createFeatures:element:* method takes as input the element of the classification and the output of the query and will use the *createFeature:element* method that is overridden by the developer to generate a collection of facts. Since SOUL is used a lot to extract background knowledge out of the elements, an abstraction (the `runQuery:variable` method) has been created which lets the implementor of the background knowledge use SOUL queries easily. This method has two arguments: the first argument is a string containing the SOUL query, the second argument is the name of the variable which will contain the output of the query. Let us take a look at how we would implement the *getFeatureClass:* method:

```
getFeatureClass: aClass
   | subclasses |
   subclasses := self runQuery:
         'if subclass([',aClass asString,'], ?subclass' variable:'subclass'.
  ↑self createFeatures: subclasses element:aClass.
```

We now have implemented the *subclass* background information. The next time a developer uses the Software Views Inducer, *subclass* will appear in the list of available types of background information.

A.2 Overview

We finish the appendix by giving a summary of how a developer can implement a new type of background knowledge:

1. Create a subclass of *AbstractFeature*.

2. Override the name method on this class to return the name of the background information.

3. Override the createFeature:element: method in order to create a logic fact for the background information. Use the abstractions like startBackground: addLiteral:, addClass:, ... to do this.

4. Override the getFeatureClass:, getFeatureMethod: and getFeatureStatement: methods. Use the createFeatures:element: method in the body of these methods to return a collection of facts. If a SOUL query is needed, the runQuery: method canbe used.

Appendix B

Implementation of Relative Least General Generalization

In this appendix we will take a look at the actual implementation of relative least general generalization in SOUL. We will start this overview of the code with the most top-level predicates and will work our way down until we get at the low-level implementation details.

B.1 The main predicate of the induction algorithm

```
 induceRlgg(?Poss,?Negs,?Model,?Clauses) if
     append(?Poss,?Model,?BG),
     covering(?Poss,?Negs,?BG,<>,?Clauses).

covering(<>,?N,?M,?H,?H) if !.
covering(?Poss,?Negs,?Model,?H0,?H) if
     constructHypothesis(?Poss,?Negs,?Model,?Hyp),!,
     removePos(?Poss2,?Model,?Hyp,?NewPoss),
     covering(?NewPoss,?Negs,?Model,<?Hyp | ?H0>,?H).
covering(?Poss,?Negs,?Model,?H0,?H) if
     append(?H0,?Poss,?H).
```

The *induceRlgg* predicate is the interface to the algorithm. It is given a set of positive examples, a set of negative examples and some background knowledge and it will return a set of clauses which cover the examples. Note that we append the positive examples to the background information and use this extended version of the background in the algorithm. This is done so that we can induce recursive rules. The *covering* predicate will construct a hypothesis and will remove the positive examples from this hypothesis. It will then recursively call itself with the remaining positive examples as argument. When no new hypothesis can be constructed, the *covering* predicate

will append all the remaining positive examples to the output clauses of the algorithm: these examples can be considered as exceptions.

B.2 Constructing the hypothesis

```
constructHypothesis(<?E1,?E2 | ?Es>,?Negs,?Model,?Clause) if
     rlgg(?E1,?E2,?Model,?Cl),
     reduce(?Cl,?Negs,?Model,?Clause),!.
constructHypothesis(<?E1,?E2 | ?Es>,?Negs,?Model,?Clause) if
     constructHypothesis(<?E2 | ?Es>,?Negs,?Model,?Clause),!
```

constructHypothesis will create a new hypothesis out of the first two positive examples and the background model and will reduce this new clause. If no new clause can be constructed with the first two examples, then *constructHypothesis* will try again with the second and the third example.

```
rlgg(?e1,?e2,?m,<?head | ?body>) if
     antiunify(?e1,?e2,?head,<>,?s10,<>,?s20),
     rlggBodies(?m,?m,<>,?body,?s10,?s1,?s20,?s2,?v).

rlggBodies(<>,?b2,?b,?b,?s1,?s1,?s2,?s2,?v) if !.
rlggBodies(<?l | ?b1>,?b2,?b0,?b,?s10,?s1,?s20,?s2,?v) if
     rlggLiteral(?l,?b2,?b0,?b00,?s10,?s11,?s20,?s21,?v),
     rlggBodies(?b1,?b2,?b00,?b,?s11,?s1,?s21,?s2,?v).

rlggLiteral(?l1,<>,?b,?b,?s1,?s1,?s2,?s2,?v) if !.
rlggLiteral(?l1,<?l2 | ?b2>,?b0,?b,?s10,?s1,?s20,?s2,?v) if
     samePredicate(?l1,?l2),
     antiunify(?l1,?l2,?l,?s10,?s11,?s20,?s21),
     !, rlggLiteral(?l1,?b2,<?l | ?b0>,?b,?s11,?s1,?s21,?s2,?v).
rlggLiteral(?l1,<?l2 | ?b2>,?b0,?b,?s10,?s1,?s20,?s2,?v) if
     not(samePredicate(?l1,?l2)),
     rlggLiteral(?l1,?b2,?b0,?b,?s10,?s1,?s20,?s2,?v)
```

Rlgg will compute the least general generalization of two examples with respect to the background model. It will create a clause with as head the anti-unification of the two examples. The body of the new clause will consist out of the anti-unification of all the literals in the background model with, again, all the literals in the background. This will result in a set of clauses which express the commonalities between the input examples.

B.3 Anti-unification

```
 antiunify(?T1,?T2,?T) if
     antiunify(?T1,?T2,?T,<>,?S1,<>,?S2),!.
1.  antiunify(?T1,?T2,?T1,?S1,?S1,?S2,?S2) if
     [?T1 asString = ?T2 asString],!.
2.  antiunify(?T1,?T2,?V,?S1,?S1,?S2,?S2) if
     subslookup(?S1,?S2,?T1,?T2,?V),!.
3.  antiunify(?T1,?T2,?T,?S10,?S1,?S20,?S2) if
     nonvar(?T1),
     nonvar(?T2),
     functor(?T1,?F,?N),functor(?T2,?F,?N),!,
     createfunctor(?Temp,?F),
     antiunifyargs(?N,?T1,?T2,?T,?Temp,?S10,?S1,?S20,?S2).

4.  antiunify(?T1,?T2,?V,?S10,<subst(?T1,?V)|?S10>,?S20,<subst(?T2,?V)|?S20>)
     if newVar(?V),!.

antiunifyargs(0,?T1,?T2,?TNew,?T,?S1,?S1,?S2,?S2) if
     equalsStructureList(?TNew,?T),!.
antiunifyargs(?N,?T1,?T2,?T,?Temp,?S10,?S1,?S20,?S2) if
     greater(?N,0),
     sub1(?N,?N1),
     argat(?N,?T1,?A1),
     argat(?N,?T2,?A2),
     antiunify(?A1,?A2,?A,?S10,?S11,?S20,?S21),
     addargument(?Temp,?A,?NewT),
     antiunifyargs(?N1,?T1,?T2,?T,?NewT,?S11,?S1,?S21,?S2),!
```

Antiunify will create, given two terms, a new term that is minimal more general than the two given terms. This predicate has for every separate case a different rule.

1. The two terms are equal. The anti-unification is then equal two the terms.

2. The two terms are literals for which there already exists a variable.

3. The two terms are function symbols. A new functor will be created with the same function symbol and as arguments the anti-unification of the arguments of both terms.

4. The two terms are not equal, are not functors and there does not exist a variable yet that is the anti-unification of the terms. A new variable will be introduced.

B.4 Reduction of clauses

```
reduce(<?Head |?B0>,?Negs,?M,<?Head,?B>) if
    nonVarElements(?B0,?M,?B1),
    reduceFunctional(?Head,?B1,?B2),
    reduceNegs(?Head,?B2,<>,?B,?Negs,?M).
```

The *reduce* predicate will reduce the obtained clauses by first eliminating all the literals without variables in the body of the clause, then by functional reduction of the clause and finally by negative based reduction.

B.4.1 Removing the examples out of the body

```
nonVarElements(<>,?Model,<>).
nonVarElements(<?L | ?Rest>,?Model,<?L | ?Other>) if
    not(varElement(?L,?Model)),!,
    nonVarElements(?Rest,?Model,?Other).
nonVarElements(<?L | ?Rest>,?Model,?Other) if
    nonVarElements(?Rest,?Model,?Other).
```

nonVarElements will remove all the literals from the body which do not contain variables (eg. the examples). These literals will not add any knowledge to the rule.

B.4.2 Functional Reduction

We are not going to discuss the SOUL code of functional reduction here. In the chapter about Inductive Logic Programming, the matter of functional reduction has already been discussed thoroughly and giving an overview of the code would not help bringing a better understanding of the subject. When using functional reduction it is important that the correct modes for the predicates have been specified. For each predicate we want to use in the induced rules we have to express which variables are input variables and which ones are output variables. Let us take a look at a few examples:

- `mode(statementIn,<out,in,in>)` Normally the statement predicate looks like `statementIn(?statement,?class,?method)`. We see that the class and the method are the input and that the statement is the output of the predicate.

- `mode(classInNamespace,<in,out>)` The classInNamespace(?class,?namespace) predicate returns the namespace of a class, given a class as input. It is clear that the class is the input and the namespace is the output.

If there is no mode specified for a predicate, the algorithm will not remove any literals from the body of the clause which contain that predicate: the

algorithm assumes then that it is not safe to reduce those literals.

B.4.3 Negative based reduction

```
reduceNegs(?H,?B,?In,?B,<>,?M) if !.
reduceNegs(?H,<?L | ?B0>,?In,?B,?Negs,?M) if
     append(?In,?B0,?Body),
     not(coversNeg(<?H,?Body>,?Negs,?M,?N)),!,
     reduceNegs(?H,?B0,?In,?B,?Negs,?M).
reduceNegs(?H,<?L | ?B0>,?In,?B,?Negs,?M) if
     reduceNegs(?H,?B0,<?L | ?In>,?B,?Negs,?M).
reduceNegs(?H,<>,?Body,?Body,?Negs,?M) if
     not(coversNeg(<?H,?Body>,?Negs,?M,?N))
```

The *reduceNegs* predicate will try to remove a literal from the body by checking if the rule without that literal does not cover a negative example. If none of the negative examples are covered by the reduced rule, then the literal that was removed was redundant in the clause. To allow this kind of reduction, we need some negative examples. If no negative examples are available, this kind of reduction will be skipped.

B.5 Other predicates

In this section we will take a look at some predicates which are used in the implementation of rlgg and which are not trivial to implement.

B.5.1 coversEx

```
coversEx(<?Head,?Body>,?Example,?Model) if
     try(and(equals(?Head,?Example),
          checkBody(?Body,?Model))).

checkBody(<>,?Model).
checkBody(<?First | ?Rest>,?Model) if
     grounded(?First),
     member(?First,?Model),!,
     checkBody(?Rest,?Model).
checkBody(<?First | ?Rest>, ?Model) if
     member(?X,?Model),
     equals(?First,?X),
     checkBody(?Rest,?Model).
```

We say that example is covered by a rule by trying every substitution that

equals the head of the rule with the example and by checking if all the literals in the body of the rule also occur in the model. This predicate is used by the *removePoss* predicate and in negative based reduction.

B.5.2 samePredicate

```
samePredicate(?t1,?t2) if
     equalsStructureList(?t1,<?functor | ?args1>),
     equalsStructureList(?t2,<?functor | ?args2>).
```

We do not take a look at this predicate because of its functionality (checking wether to terms have the same predicate symbol), but because of its implementation. In the source code for rlgg we have to construct and manipulate SOUL terms in a few places. The implementation of *samePredicate* shows how we can do this by using the *equalsStructureList* predicate, which is part of SOUL.

Bibliography

[BG95] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From machine learning to software engineering.* MIT Press., 1995.

[BG96] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Trans. Software Engineering and Methodology*, 5(2):119–145, 1996.

[Bri00] J. Brichau. Declarative composable aspects. In *OOPSLA Workshop:Advanced Separation of Concerns*, 2000.

[Bun88] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[CD97] W. Cohen and P. Devanbu. A comparative study of inductive logic programming for software fault prediction. In *14th International Conference on Machine Learning*, 1997.

[Cin] Cincom. Cincom visualworks. website. http://www.cincom.com/scripts/smalltalk.dll/.

[Coh95] W. Cohen. Inductive specification recovery: understanding software by learning from example behaviours. *Automated Software Engineering*, 2(2):107–129, 1995.

[dALM98] M. de Almeida, H. Lounis, and W. Melo. An investigation on the use of machine learned models for estimating correction costs. In *International Conference on Software Engineering*, pages 473–476, 1998.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual.* Springer-Verlag, 1996.

[DH98] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems.* PhD thesis, Vrije Universiteit Brussel, 1998.

[DMBM02] W. De Meuter, J. Brichau, and K. Mens. *Soul Manual*, 2002.

[DVD99] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *Meta-level architectures and reflection, Second International Conference, Reflection '99*, pages 250–272. Springer-Verlag, 1999.

[Fla94] P. Flach. *Simply Logical - Intelligent Reasoning by Example.* John Wiley and Sons, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKY99] W.G. Griswold, Y. Kato, and J.J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems - OOPSLA 99*, 1999.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80, The Language.* Addison-Wesley, 1989.

[HK01] J. Hannemann and G. Kiczales. overcoming the prevalent decomposition in legacy code. In *Workshop on Advanced Separation of Concerns, International Conference on Software Engineering*, 2001.

[JDV03] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development 2003*, 2003.

[Llo87] J.W. Lloyd. *Foundations of Logic Progamming.* Springer-Verlag, 1987.

[MD01] T. Mens and S. Demeyer. Evolution metrics. In *Int. Workshop Principles of Software Evolution, Vienna*, 2001.

[Men00a] K Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, 2000.

[Men00b] K Mens. *Automating architectural conformance checking by means of logic meta programming.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2000.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First Conference on Algorithmic Learning Theory*, 1990.

[Mit97] T.M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[MMW01] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Int. Conf. Software Engineering and Knowledge Engineering*, 2001.

[MMW02a] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Int. Conf. Software Engineering and Knowledge Engineering*, pages 289–296. ACM Press, 2002.

[MMW02b] K. Mens, T. Mens, and M. Wermelinger. Supporting software evolution with intentional software views. In *Int. Workshop Principles of Software Evolution*, 2002.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT 1995, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[MT01] T. Mens and T. Tourwé. A declarative evolution framework for object-oriented design patterns. In *Int. Conf. Software Maintenance*, 2001.

[Plo70] G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Plo71] G.D. Plotkin. A further note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6. New York:Elsevier, 1971.

[PRO] Programming Technology Lab PROG. The prog dmp swiki. website. http://prog.vub.ac.be:8080/DMP.

[Qui90] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RM02] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference On Software Engineering 2002*, 2002.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Sha83] E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

[TBKG03] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. paper submitted to the European Smalltalk Users Group Conference 2003, 8 2003.

[Tou02] T. Tourwé. *Automated Support For Framework-Based Software Evolution.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[Wuy] R. Wuyts. The starbrowser. website. http://www.iam.unibe.ch/ wuyts/StarBrowser/.

[Wuy98] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS USA '98*, pages 112–124. IEEE Computer Society Press, 1998.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

[ZT02] D. Zhang and J. Tsai. Machine learning and software engineering. In *14th IEEE Internation Conference on Tools with Artificial Intelligence*, 2002.