Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Incorporating Dynamic Analysis and Approximate Reasoning in Declarative Meta-Programming to Support Software Re-engineering

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

Door: Coen De Roover
Promotor: Prof. Dr. Theo D'Hondt
Begeleider: Kris Gybels
Mei 2004

Abstract

Software re-engineering is a complex process in which a maintainer is confronted with the challenging task of understanding the design of an existing application of which the documentation is often non-existent or no longer up-to-date.

To support the software engineer in this precarious undertaking, many design recovery tools based upon declarative meta-programming techniques have been proposed, but most of them are only able to reason about a program's structural architecture instead of about the exhibited run-time behaviour. This document explores how behavioural analysis can complement structural source code analysis in the detection of software patterns in the broadest sense.

Furthermore, interesting software patterns often describe inherently vague concepts which cannot be expressed efficiently in classical logic programming languages. Many software patterns are in addition described by such overly idealised logic rules that real-life variations on the implementation of these abstract patterns can no longer be detected. In this dissertation, we will therefore also study the use of approximate reasoning techniques to overcome these common problems.

Acknowledgements

I would like to take this opportunity to express my gratitude towards all the people who supported me tremendously throughout the writing of this dissertation and without whose help I would have never finished it:

Prof. Theo D'Hondt for promoting this thesis.

Kris Gybels who came up with the subject and guided me along every step through implementation and writing to proofreading which must have consumed much of his valuable time. The excellent advice and helpful comments always kept me on the right track even when I didn't longer believe I would be able to finish on time. I have also very much enjoyed the various related discussions we had.

Wolfgang De Meuter and Roel Wuyts for their input during my initial presentation on approximate reasoning techniques.

Johan Brichau and Andy Kellens for proofreading the first drafts.

The researchers at the Programming Technology Lab for enduring my colourful presentations.

The people from Infogroep for providing an abundance of tempting distractions.

My parents for their never-ending support and for providing the opportunity to obtain a higher education at the excellent facilities of the Vrije Universiteit Brussel.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis

In this dissertation, we will support two claims. First of all, the application of declarative meta-programming to the detection of software patterns which are inherently vague to describe or expressed in an overly idealised manner requires appropriate language support for approximation by the declarative technique that is applied.

Furthermore, behavioural information obtained through a dynamic analysis of a program's execution complements structural information obtained through a static analysis of a program's source code. Each specific analysis has its own benefits and deficits, but combined they provide a wealth of information which allows more complex patterns to be expressed that are of use in the entire software re-engineering process.

1.2 Context

The research context for this work is situated in the domain of software re-engineering which covers the analysis of existing applications in order to recover a lost design from existing source code or locate deficits in its implementation. After the initial design recovery and deficit identification phase, the program maintainer performs source code refactorings which transform the application to an equally functional state where bugs have been resolved and the system's design has been improved. An application often endures several iterations of the re-engineering process.

Automatic detection of well-known software patterns for which a common vocabulary has been developed by the software engineering community, can significantly improve a maintainer's understanding of the application under investigation:

- Design patterns [GHJV94] are prototypical examples of software patterns describing elegant architectural building blocks which often recur in proven software designs.
- Tools supporting software pattern detection can also be employed in the verification and enforcement of minimalistic software patterns capturing an organisation's programming style and conventions.

- The complexity of the identification phase of the software re-engineering process can be greatly reduced by the automatic detection of software patterns describing proven indicators of suboptimal design and bad implementation practices also known as "bad smells" [FBB+99].
- Behavioural collaborations and interactions through messages sent between instances are, together with the classification of the UML binary class relationships based on object lifetime and exclusiveness properties, examples of software patterns that go beyond syntactical properties of the system under investigation to improve a maintainer's understanding of its run-time properties.

Software maintainers often want to define additional software patterns to describe and verify particular aspects of the mental model they individually developed of the application's inner workings. Declarative meta-programming, in which one writes programs reasoning about other programs using a declarative programming language, allows a flexible specification of various software patterns in a powerful domain-specific language.

1.3 Motivation

There are however two problems with current declarative meta-programming approaches to software re-engineering. First of all most tools are limited to software patterns describing structural properties of an application's source code while many interesting patterns describe the run-time interactions between instances. A second problem comprises the use of crisp all-or-nothing declarative reasoning schemes while humans employ a much more forgiving way of thinking as evidenced by the vague classification boundaries exhibited by most software patterns.

We will begin our discussion with the kind of problems for which the solution we propose comprises introducing behavioural information obtained through dynamic analysis in the software re-engineering process.

Inability to Capture Behavioural Aspects The use of static information implies that only the structural aspect of a software pattern can be expressed. This is fine for patterns that are very architecture-centric and thus can be easily described in terms of class hierarchies and methods. However, it is much harder to express a programming pattern that primarily describes how its entities collaborate with each other.

Patterns are Strongly Tied to Source Code When exclusively relying on a static source code analysis, declarative rules can often only find one particular implementation of a pattern. Additional rules will have to be defined for other common implementations of the same pattern. This is particularly clear when expressing the "accessor method" software pattern which verifies whether a class accesses its instance variables only through accessor methods. In addition to simply returning the variable's value, there are getting method implementations which incorporate lazy initialisation of the accessed variable. While static rules must explicitly handle multiple implementation possibilities, dynamic rules could simply check whether the getting method returned the value of the instance variable it was supposed to wrap.

Control and Data Flow Information is Difficult to Obtain Due to the nature of static source code analysis some information is difficult to obtain. This includes lifetime and exclusiveness properties of instances encountered during a program's execution. Polymorphism, late binding and inheritance in object-oriented languages make it even difficult to predict the actual method that will be invoked following a message send.

As dynamic analysis techniques reason about execution traces of an application, their most notable downside can be quickly identified. The easily obtained control and data flow information is only correct with respect to one particular execution history. Static analysis techniques on the other hand provide incomplete information that is correct for every execution of the program. A more detailed and balanced overview of the benefits and disadvantages of both analysis techniques is presented in section 2.1.3.

The problems we describe below are shared by all declarative meta-programming approaches whether they incorporate run-time information or not. Although these problems can be overcome by clever engineering of the declarative rules with flexibility in mind, it is in the interest of clarity that rules are kept as succinct and expressive as possible. Therefore, support for approximate reasoning is needed in the base declarative programming language. After all, domain specific languages need to be tailored to the domain in which they are applied.

Vague concepts It can be verified that many software patterns expressed in a declarative meta-programming language describe inherently vague concepts Typical examples of vague software patterns are the aforementioned bad smells such as *"too many instance variables"* or *"too many parameters"* where classification boundaries are vague: when the user sets a boundary limit of 10 variables, he will likely consider a class with 9 instance variables almost as bad as a class with 10 instance variables.

Overly idealised pattern descriptions Another frequently recurring problem is that most rules describing software patterns are overly idealised. They work fine for direct translations of the abstract models they describe, but often fail on real-life concrete implementations of the same models.

Intolerance for small discrepancies A related problem comprises tolerance for small discrepancies between a wanted solution and the program facts at hand. If we are searching for a method called `acceptVisitor:` we should also tolerate method selectors with small typing errors such as `accceptVisitor:` or selectors that are semantically equivalent such as `visitorAccept:`.

Approximate reasoning also allows the discovery of software patterns using different different heuristics that are each weighted by the certainty we have in it. In our approach where dynamic and static information can be combined, this feature allows us to select the source of information that is most appropriate for solving the problem at hand. For instance, when a rule needs to know when a class gets instantiated, it can obtain this information from a dynamic and a static analysis source. A dynamic analysis can provide this information with high fidelity unless no instance of the class was created in a particular program run. In that case, the static analysis can verify that there are no instance creation methods sent to the class. Solutions to a pattern thus often need to be weighted on a case-per-case basis.

1.4 Validation

We will validate our claim about the complementary relation between static and dynamic analysis in software re-engineering by implementing a library for reasoning about a program's behaviour which complements the existing library for structural source code reasoning developed at the Programming Technology Laboratorium of the Vrije Universiteit Brussel.

We have also implemented an extension of the base declarative meta-programming language supporting approximation in rules and program facts. We will demonstrate with an example of each of the commonly occurring problems in declarative meta-programming how approximate reasoning provides a solid background to overcome each problem. These examples represent only an initial exploration of the applicability of approximate reasoning to support the software re-engineering process, but they are encouraging nonetheless.

1.5 Organisation of the Dissertation

We will begin with a detailed overview of the background information on each of the two pillars of our approach.

Declarative meta-programming and its application in static and dynamic software analysis is discussed in chapter 2 together with recent developments in software re-engineering approaches incorporating this technique.

Chapter 3 consists of theoretical background on modelling vague concepts and of an overview of existing approximate reasoning techniques.

In chapter 4, we will describe how we incorporated dynamic analysis and approximate reasoning into SOUL, an existing tool for program reasoning based on structural source code analysis.

The benefits in a software re-engineering setting of our new dynamic analysis model together with our new logic programming language supporting approximate reasoning, will be detailed in chapter 5.

Chapter 6 contains the conclusions of our work.

Chapter 2

Declarative Meta-Programming Approaches to Software Re-engineering

As we will investigate in this chapter, software re-engineering is a complicated process in which humans can be assisted in each of the different steps it comprises by programs capable of reasoning about other software systems.
We will start this discussion by detailing the software re-engineering process and continue our investigation with an overview of systems for automated re-engineering employing a declarative domain-specific programming language suitable for reasoning about other programs.

2.1 Introduction

2.1.1 Software Re-engineering

We will first identify the different phases in the software re-engineering process in which declarative meta-programming can be of help.

Software systems continue to evolve even after their shipment day. Bugs need to be fixed and new functionality has to be introduced to cope with new or changed requirements. An extremely large amount of an application's lifetime is dedicated to software maintenance.

This immense maintenance task is an iterative process in which the maintainer is first confronted with the challenging task of understanding a program's behaviour and its structure even when documentation is non-existent or no longer up-to-date.

The process in which the high-level design concepts of a software system are rediscovered from source code and other low-level program artefacts is called *design recovery*. It goes beyond reverse engineering an application to study its components and their interactions with the outside world. It is an inherently challenging task since the original abstract concepts of the design are diffused throughout the implementation. Further-

more, it often involves massive and complex systems. Therefore it should come as no surprise that many techniques have attempted to facilitate and automate this process.

When the maintainer has formed a mental model of the high-level design concepts by abstracting away from implementation details and as such has formed a basic understanding of the program, he still has to identify the locations in the source code which have to be modified. This phase centres around the detection of *design defects*.

It is not until this point that relatively safe modifications to the source code can be applied. These modifications ideally *transform* a system from one working state to another while introducing new functionality or fixing existing shortcomings.

2.1.2 Declarative Meta-Programming

We will now discuss the concepts involved in declarative meta-programming and the various declarative programming paradigms that can be used to reason about other programs.

2.1.2.1 Definitions

Meta-Programming *Meta-programs* are regular applications able to manipulate and reason about other programs. Examples of such programs are compilers, programming style checkers and tools that extract design information from an existing applications.

Meta-programming thus involves the implementation of said programs. As always, more expressive domain-specific languages can be created to facilitate this process with special language constructs and higher-level features. These are called *meta-programming languages*.

Declarative Programming In classical *procedural* programming languages, programmers specify exactly **how** the solution to a problem is to be found in step-by-step algorithmic descriptions.

In contrast, *declarative* programming languages allow the problem itself to be specified so programmers can concentrate on **what** the problem is. The programming language will find a solution by itself, depending on the declarative programming paradigm incorporated by the language.

Declarative Meta-Programming As declarative languages are highly expressive mediums, they are particularly suited for meta-programming purposes in which one usually has to reason about a fair amount of application date.

2.1.2.2 Declarative paradigms

The following is an overview of the declarative techniques used in the meta-programming approaches to software re-engineering discussed in this chapter.

Logic programming In the logic programming paradigm, a query is considered a theorem of which a proof has to be found. A proof is constructed by logically inferencing new facts from the already available facts in the information base using

rules specified by the programmer. These rules are multi-directional in nature as they define mathematical relations between variables: a single rule defining the $<$-relation can be used to verify that $a < b$, but also to generate for instance all $a < 5$ and even all $2 < b$.

More information about the prototypical implementation, Prolog, can be found in [Fla94].

Constraint programming A constraint satisfaction problem (CSP) is defined by three groups of entities:

- a set of variables
- a set of finite or infinite domains associated with each variable
- a set of constraints restricting the values variables can take on instantaneously

A solution to a CSP assigns each variable a value from its domain while respecting the restrictions defined by the constraints. Constraint satisfaction solvers often use logic programming languages to describe these constraints.

An introduction to programming with constraints can be found in [MS98]

Pattern matching Pattern matching languages aren't exactly programming languages as they aren't even remotely Turing complete, but they can be used to describe information very succinctly or to find recurring patterns in large data. Therefore, this are also often used in meta-program contexts.

A well-known pattern matching language is the regular expressions language [MY60].

2.1.3 Dynamic and Static Program Analysis

Tools can rely on static information, dynamic information or a combination of both to aid a maintainer in the re-engineering process. In [Ric02], an overview of the problems associated with each type of analysis is given.

Static analysis In a static program analysis, only information extracted from the program's source code is considered. This kind of information is best suited for the retrieval of structural information like the basic object-oriented entities forming the application's architecture.

There are however drawbacks associated with this technique. First of all, correctly identifying collaborations between classes is hard as the receiver's class type determines exactly which method is invoked. Often, there are many possible candidates due to polymorphism and dynamic binding in object-oriented programming. Inheritance poses another problem: parts of a method's behaviour may be defined in superclasses. This is even more serious in dynamically typed languages (such as Smalltalk) where even the base type of methods and variables is only known at run-time. Statically obtained information about the control flow in an object-oriented system is thus bound to be incomplete.

The granularity of the obtained information restricts the types of reasoning that are possible. We can distinguish information pertaining to classes, ancestor and associative relations between classes, method names and full parse trees of the

statements in individual methods. It may be more difficult to generate code using only coarse-grained information. It is possible to use high-level UML-like [Fow97] meta-models[1] [AACGJ01], but also to rely only on the basic structural object oriented entities like class, superclass and defined methods [Ric02]. Other techniques incorporate entire parse trees in their source models [Wuy01].

Dynamic analysis In a dynamic analysis, information is obtained from program execution traces. This information is best suited to model an application's behaviour such as the interactions between class instances. This information can help in understanding the relationship between an application's source code and its run-time behaviour.

The data obtained by dynamic analysis is always correct: the type of objects is unambiguous and so is the identity of the method that is executed upon a certain message invocation. It also provides information that is impossible to obtain statically such as the number of class instances or the amount of times a method was invoked.

When interpreting dynamically-obtained information, the maintainer should however always be aware of the fact that this information is only valid for one of many different program executions. The control flow in an application might change drastically when other user input is provided.

This problem can also be seen as a certain advantage over static analysis. Very often, it is not necessary to know everything about the application at once. An engineer rather wants to understand instead how the interaction between run-time instances supports a particular functionality. Through careful selection of the classes about which run-time information is to be gathered[2] and by restricting a program's execution to a well-defined scenario that must be analysed, one can successfully focus the investigation on specific program parts. This doesn't necessarily imply that the maintainer already knows everything about the application under investigation since, at the end of each iteration of the design recovery phase, he is able to refine the mental model he formed about the inner workings of the application by limiting his investigation to those parts that aren't clear yet.

When it is in contrast necessary to reason about the entire program's run-time behaviour, the relevance of the executed scenario can be verified with a code coverage analysis to make sure that the invoked methods cover a minimal percentage of a program's source code. Unit test programs (whose use is advocated in Extreme Programming) of larger software frameworks define excellent scenario's for this purpose.

As with static analysis, the granularity of the source meta-model restricts the possible software engineering uses of dynamic information. Fine-grained models [GDJ02] contain information about variable assignments, returned values and object lifetime and can therefore be used to debug a program's on the control flow level. Coarse-grained models [Ric02] only contain method invocation information and are better suited to reason about the higher-level collaborations in large systems.

[1] In this context, a meta-model defines how the static information is modelled.

[2] In general, the chosen classes are altered on a byte code or source code level so that their execution can be traced. This process is called *instrumentation*

Finally, there are mainly two variants of dynamic analysis. In the first one, the application is executed entirely during which trace information is collected in an execution trace history. When the program has finished, all events recorded during its execution can be reasoned about. This form of analysis is called *post-mortem* analysis and is for instance used in [Ric02]. Its major drawback is the enormous amount of information that is generated, especially when fine-grained meta-models are used.

The other form of analysis is *ad-hoc* analysis in which the reasoning process steers the execution of the analysed program. The program is paused and control is given to the analysing process when a particular execution event (such as a method invocation) is encountered. The entire execution trace is never available. Only the current event can be reasoned about, but future events can be requested. This technique is used in [GDJ02].

2.2 Survey Of Existing Declarative Approaches

This section gives an overview of the existing techniques that can be used in the entire reverse engineering process and especially focuses on declarative meta-programming techniques which also incorporate dynamic information obtained from program execution traces.
We will start with a discussion of the on static information relying SOUL, followed by a discussion of the work of Tamar Richner which is based solely on dynamic information. We will end with the work of Guéhéneuc which combines both types of information.

2.2.1 Declarative Reasoning about the Structure of Object-Oriented Systems

This section summarizes and discusses the logic meta-programming work of Roel Wuyts, Kim Mens, Isabel Michiels, Theo D'Hondt and others performed at the *Programming Laboratory of the Vrije Universiteit Brussel* in Brussels.

It was one of the first works to introduce a logic language for meta-programming purposes in the object-oriented programming paradigm. It provides meta-programs written in SOUL the entire application's source code to perform a static analysis on which can then be used in all phases of the re-engineering process from pattern detection to source code generation. Interested readers may consult [DDVMW00, Wuy98, MMW01, Wuy01] and [WD01].

2.2.1.1 Motivation

We will use SOUL in our program understanding setting for expressing logic rules which describe the architecture of design patterns, but it was initially conceived to keep co-evolving software artefacts such as design, implementation and documentation synchronised.

After the initial shipment of an application, the link between the artefacts describing its architecture and the actual implementation often weakens (when modifications to the source code aren't reflected in the documentation) or even dissolves (when legacy

documentation is lost).

This makes it harder for maintainers to correctly understand the implications of a local implementation change –will it not affect other vital parts of the system?– or to correctly identify all parts in the source code that will have to be modified when a major architectural change has been introduced. Furthermore, little tools exist to enforce programming patterns (like a company's coding conventions) throughout the entire program in a consistent way.

In [Wuy98] the common source of these problem is identified to be the *"incapability to express high-level structural information in a computable medium that is used to extract implementation elements"*.

2.2.1.2 Approach

A system's architecture is described at the meta-level using a logical programming language called SOUL [WD01]. It can be applied in the software development process by allowing users to search for code that matches user-defined programming patterns, to specify other rules which can be used to detect violations of these patterns and to generate code for a specific pattern. The proposed techniques can be used to keep co-evolving software artefacts synchronised.

2.2.1.3 Source Model

Static information from full-fledged parse trees is used to model the system's implementation. However, instead of having large logic repositories filled with facts mimicking the application's source code, the *symbiosis* between the meta-model's implementation language and the implementation language of the analysed source code –which fully supports *reflection*– enables the direct use of source code objects in the application's model.

a) Meta-Model Smalltalk parse trees are *reified* to the declarative meta-level by mapping them to logical predicates (not facts) defined by the meta-model:

- *class(?class)*: represents classes in source code
- *superclass(?super, ?sub)*: states that ?*super* is the superclass of ?*sub* [3]
- *instVar(?class, ?iv)*: represents instance variables in a class
- *method(?class, ?m)*: represents a class' methods whose parse trees are modelled themselves using predicates representing literals, variables, assignments, return statements, message sends and block statements

b) Extraction As mentioned already before, the source model isn't explicitly stored in the logic repository. Therefore, the constructs described by the meta-model aren't simple facts but implemented as logic rules which use SOUL's symbiosis with Smalltalk to reify source code entities on demand.

[3] Variables in SOUL are preceded by question marks

An example of such a rule is the predicate *class*(?*c*) which, due to the multi-way directionality of logic languages, can be used to either obtain a list of classes in the system or to verify whether a given class is present in the system:

```
class(?c) if
  var(?class),
  generate(?class, [SOULExplicitMLI current allClasses]).
class(?c) if
  atom(?class),
  [SOULExplicitMLI current isClass: ?class]).
```

This approach allows an analysis of a program's implementation without keeping track of an explicitly reified duplicate of the source code. Combined with the reflexive capabilities of the base language, it is possible to analyze every aspect of the Smalltalk system.

The ability to execute arbitrary Smalltalk code is essential in the definition of these predicates so this and other features of SOUL will be discussed in the following section.

2.2.1.4 Smalltalk Open Unification Language

A variant of Prolog, called SOUL, is used as the declarative meta language. This logic programming language provides a symbiosis with the object-oriented Smalltalk which allows arbitrary Smalltalk code to be executed within logic rules. As a result, a form of *symbiotic reflection* where both the meta-language and the base language (which can be implemented in different programming paradigms) are able to analyze and alter the other's implementation is created.

The language construct providing this symbiosis is called the *symbiosis term*, or in the context of SOUL, the *smalltalk term*. This logic construct allows the execution of arbitrary Smalltalk expressions –which may contain logic variables for parameterisation– during a logic proof. Therefore, it is necessary to define a transformation between the entities in each language.

To evaluate a *smalltalk term* in a logic rule written in SOUL, each logic variable used in the term has to be transformed to a Smalltalk object. After this substitution, the *smalltalk term* can then be evaluated as a regular Smalltalk expression (actually, a block closure) whose result will have to be wrapped in a logic entity again so it can be used in a logic proof. The *class* predicate defined in the previous section is an example of the usage of the *smalltalk term*.

As SOUL can be used to reason about any Smalltalk object, it can also reason about its own implementation classes. This property is called *introspection*. However, since the logic repository and the current set of logic variable bindings of a smalltalk term are accessible during the evaluation, smalltalk terms can in addition influence their own evaluation. This property is called *reflection*.

Reflection allows an interpreter to extend its own implementation and as such is used in SOUL to implement higher-order logic constructs like the *assert* predicate which adds a clause to the logic repository:

```
assert(?clause) if
  [?repository addClause: ?clause]
```

2.2.1.5 LiCoR: a Library for Code Reasoning

SOUL's library for code reasoning, called LiCoR, is organized as a layered set of rules where each layer only uses the predicates defined in the layers below it. Four layers can be identified:

Logic layer This layer defines the lowest-level logic programming constructs which are normally provided by the implementation's libraries. It is in the interest of reflection that as many core functionality as possible is defined in the language itself: predicates for arithmetic, list handling and repository control are thus defined in this layer. Examples are the *greaterThan*, *ground* and *append* predicates:

```
ground(?X) if
    [?X isGround]

greaterOrEqual(?N,?M) if
    comparable(?N,?M),
    [?N >= ?M]

append(<>,?List,?List).
append(<?ElFirst|?RestFirst>,?Second,<?ElFirst|?Rest>) if
    append(?RestFirst,?Second,?Rest)
```

Representational layer This layer defines the predicates used to reify the object-oriented entities of the base-language to the logic meta-level. It contains definitions for the predicates discussed in the section about the meta-model. Methods are reified as a logic representation of their entire parse tree:

```
$method(?class,
        ?name,
        arguments(?paramlist),
        temporaries(?varlist),
        statements(?stats))
```

The layer thus supports predicates for describing each base-level statement that may occur in the source code such as literals, message sends and blocks.

Basic layer This level provides a level of abstraction over the low-level predicates of the representational layer. It contains among others accessing predicates like *methodName, methodStatements* and predicates for common parse tree traversals like *returnStatements* and *classesUsed*.

```
implementedSelectors(?class,?selectors) if
  findall(?selector,classImplements(?class,?selector),?selectors)
```

It also contains the definition of general predicates supporting code generation of which *generateMethodInProtocol* is the cornerstone. The ?*quotedTerm* representation of a method's parse tree may be retrieved using the *methodSource(?methodParseTree, ?source)* predicate.

```
generateMethodInProtocol(?quotedTerm, ?class, ?protocol) if
  atom(?protocol),
  existingClass(?class),
  sound(?quotedTerm),
  [(?class compile: ?quotedTerm sourceString classified: ?protocol) = nil]
```

Design layer The top-most layer is used to store the logic rules (created by the maintainer or already included with the system) describing programming patterns such as coding conventions and design patterns. This application of the declarative meta-programming language will be the next topic of discussion.

a) Declaratively Codifying Programming Patterns In general, the programming patterns that are declaratively codified in the design layer support the software engineering process in three distinct ways. This will be illustrated with examples[4] using the *Getting Method* best practice pattern [Bec96]:

> *How do you provide access to an instance variable?* Provide a method that returns the value of the variable. Give it the same name as the variable.

In each of the following use cases for a declaratively codified pattern, certain building blocks can be reused. In case of the *gettingMethod* predicate this is true for the $gettingMethodStats < return(variable(?V)) >, ?V)$ fact which states that a getting method consists of a single statement returning a variable. To support more complicated getting method implementations (which could for example use lazy initialisation), other basic method statements building blocks have to be defined.

Pattern detection The pattern rules can be used in a straightforward manner to check whether some given classes comply with a pattern, but also –due to the multi-directionality property of logic predicates – to search for pattern occurrences in the source code.

The *Getting Method* pattern detection rule below states that the method must have the same name as the instance variable it is wrapping and that its statements must match those defined by the *gettingMethodStats* fact:

```
gettingMethod(?C,?M,?V) if
  classImplementsMethodNamed(?C,?V,?M),
  instVar(?C,?V),
  gettingMethodStats(?Stats,?V),
  methodStatements(?M,?Stats).
```

Pattern violation detection Pattern violations rules must be hand-coded which means that a maintainer has to think of every possible way in which a pattern can be violated. In the case of the *Getting Method* best practice pattern, this is the

[4] Taken from [MMW01]

case when a class directly sends messages to an instance variable without using accessor methods wrapping that variable. The rule that checks for this kind of violation is given below:

```
accessingViolator(?C,?M,?V,?Msg) if
   instVar(?C,?V),
   method(?C,?M),
   not(gettingMethod(?C,?M,?V)),
   isSendTo(?C,?M,variable(?V),?Msg).
```

Pattern code generation The pattern's building blocks also support code generation for programming patterns using the symbiosis with Smalltalk and the latter's reflexive powers to compile methods on the fly.

```
generateAccessorCode(?C,?V) if
   instVar(?C,?V),
   not(classImplements(?C,?V)),
   gettingMethodStats(?Stats,?V),
   generateMethod(method(?C,?V,<>,<>,?Stats)).
```

The rule first checks whether an instance variable with the given name exists in the class definition and whether a method with that name exists not yet. It then uses the *generateMethod* predicate discussed in the previous section to compile the new method using the source code provided by the *gettingMethodStats* fact.

2.2.1.6 Evaluation

a) Use of Static And Dynamic Information The DMP techniques implemented with SOUL rely only on static information. Also, they do not model an application's design, but instead use the actual source code of the system to reason about its implementation and especially about the structure of its source code.

As Smalltalk is a dynamically typed programming language, obtaining type information for expressions is hard. Special typing rules are defined in the basic layer which guess the type of an expression by looking for classes whose method dictionary matches all messages invoked on the expression but these are hardly satisfactory as many Smalltalk classes implement methods with the same selector such as `initialize`, `do:` and `accept:`.

b) Use of Logic Programming The techniques are implemented using a logic programming language called SOUL. The author argues that logic programming languages offer distinct benefits over other techniques (pattern matching, for example) used in reasoning about meta-information [Wuy01]:

- Logic programming languages have pattern matching abilities built-in which makes them very suitable for finding specific nodes in abstract syntax trees.
- Predicates in logic programming languages describe relations between their arguments. A solution will be found for omitted arguments due to the multi-directionality of these relations.

- Logic programming languages are Turing-complete and support many powerful programming constructs like recursion. This in contrast to techniques which use SQL.

c) Use of Language Symbiosis The symbiosis between the meta-level and base-level language in which SOUL is implemented, allows both to reason and use entities from the other. It also removes the need for a separate logic representation of the entities from an analysed program at the meta-level and allows a straightforward manipulation of the program's source code. This already sets soul apart from more conventional techniques.

d) Use of Reflection Smalltalk's reflective capabilities [FJ89] enable SOUL to reason about any component in the entire system while a form of lightweight reflection is used by SOUL itself to allow the logic repository and binding environment in the evaluation of *smalltalk terms* to be modified from within SOUL predicates.
SOUL can be extended naturally using this reflective capability as is shown in the implementation of its higher-order predicates.

e) Scalability Industrial experiments were conducted to check whether the used techniques scale up well. From these experiments it was concluded that querying the source code using a logic meta-programming language is feasible although some queries might take longer than an hour. The performance can however be improved since the system was originally designed with extensibility in the head.

2.2.2 A Query-Based Approach to Recovering Behavioural Design Views

This section summarises and discusses the recent reverse engineering-oriented work of Tamar Richner-Hanna and Stéphane Ducasse of the *Institut für Informatik IAM in Bern*.
It is one of the first works demonstrating the feasibility of declarative event analysis from program execution traces. It uses dynamic information to aid in the first phase of the re-engineering process: understanding a program's behaviour. Interested readers may consult [RDW98, RD99] and [RD01] for further reading.

2.2.2.1 Motivation

Most of the preexisting tools that rely on dynamic information for the recovery of behavioural models only offer verbose, fine-grained views of information gathered during a program's execution. Most of them simply plot on a time line all the message invocations that occurred during the application's lifetime which often results in massive sequence diagrams that are hard to interpret.
In order to present the large amounts of low-level information in a more abstract form that is of help to programmers trying to understand a program's behaviour, these approaches primarily have to rely on visualisation, navigation and clustering techniques.
Most tools furthermore only offer a fixed set of views. However, not all kinds of views are equally apt at aiding in specific phases of the program understanding process. Compact high-level views, for instance, aid in forming general concepts about a program's

run-time behaviour at large. Focused low-level views, on the other hand, are invaluable for understanding object interactions supporting specific program functionality.
In order to overcome the above problems, a new approach is proposed which enables maintainers to tailor the generated views to the software re-engineering task at hand.

2.2.2.2 Approach

The proposed approach is query-based: in order to understand a program, the developer iteratively launches queries about the program's behaviour and by interpreting the obtained results either refines or reviews its conceptions.
The following definitions play a vital role in this iterative process:

Source model The dynamic and static information base about the software system that is being reverse engineered.

Perspective A declarative specification of the aspects from the source model the engineer is interested in. Engineers query the source model using perspectives to obtain a new view.

View A view of the source model consists of those elements which meet the declarative specification of a given perspectives. A view represents the response to a query.

The analogy with photographing a scene is used to explain the relationship between perspective and view: the reverse engineering tool can be thought of as a camera, a perspective as a camera lens, a view as a photograph and a source model as the scene that is to be photographed. Different lenses can be mounted to obtain different effects.

2.2.2.3 Source Model

The source model contains both static and dynamic information about the program's behaviour. This data is modelled using the constructs provided by the meta-model and saved as Prolog [Fla94] predicates defined in the representation layer of the declarative framework.

a) Meta-Model

Static Meta-Model Static information is modelled using the FAMIX [TDDN00] meta-model. It provides a language-independent extensible representation of object-oriented code and is used by a variety of refactoring tools. There is also plug-in support for language-specific extensions. As a sufficient base for testing and performing refactorings is provided, the meta-model is quite rich: classes, methods, attributes, inheritance, method invocation and attribute access candidates[5] are all supported. Only the basic object-oriented concepts of *class*, *superclass* and *method* are however used in the experiments, so a much simpler meta-model probably suffices.

Dynamic Meta-Model A program execution is modelled as a list of run-time events which are numbered according to *sequence order (SN)* and *stack level (SL)*. There are two event types: method invocation events and method exit events.

[5] One-to-many relationships due to polymorphism

```
send(49,7,'EllipseFigure',139,'EllipseFigure',139,'fillColor').
send(50,8,'EllipseFigure',139,'Drawing',685,'fillColor').
send(51,9,'Drawing',685,'FigureAttributes',4426,'fillColor').
send(52,9,'Drawing',685,'FigureAttributes',4426,'fillColor').
send(53,7,'EllipseFigure',139,'Drawing',685,'compositionBoundsFor:').
```

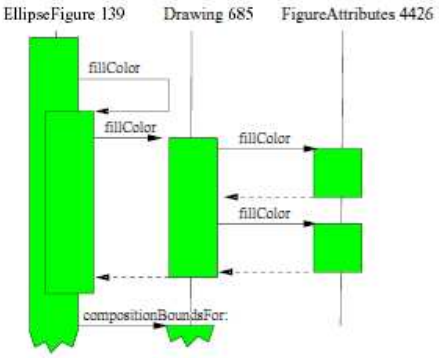

Figure 2.1: Example sequence diagram and corresponding run-time events

As shown in Table 2.1, method exits aren't represented as facts in the representational layer of the declarative framework since they can be deduced from the *stack level (SL)*.

send(SequenceN, StackLevel, Class1, Instance1, Class2, Instance2, Method) an instance Instance1 of Class1 invokes Method on instance Instance2 of Class2. SequenceN is the sequence number of the event, and StackLevel is the stack level of the method call.

indirectsend(SequenceN, StackLevel, Class1, Instance1, Method1, Class2, Instance2,Method2) an instance Instance1 of Class1 sends the message Method1, which is unobserved. The next observed invocation is the execution of method Method2 on instance Instance2 of Class2.

Table 2.1: Dynamic meta-model entities as predicates

Figure 2.1[6] shows an example sequence of method invocation events.

We must instrument the source code of interesting methods so that their execution will be observed and logged in the execution trace. A call to an uninstrumented method will not be observed. As the used instrumentation method [BFJR98] allows selective instrumentation of single methods, this implies that there are two kinds of send events: direct sends and indirect sends. A direct send event occurs when an instrumented method is invoked by another instrumented method while an indirect send event occurs when an instrumented method is invoked by an uninstrumented method.

[6]Taken from [Ric02]

b) Extraction Both static and dynamic information is stored as Prolog [Fla94] facts. Static information is obtained by a static analysis of the program's source code using the tools provided by the FAMIX [TDDN00] meta-model, while dynamic information is extracted by instrumenting the interesting methods using the Method Wrappers technique discussed in [BFJR98] and running a typical scenario to obtain an execution trace.

2.2.2.4 Concept View Recovery

The following definition is given [Ric02]:

> Concept views present the user with a view of the software as a set of components and connectors. The semantics of the components and the connectors are defined by the engineer: components are created by grouping together static elements of the software, such as classes or methods; connectors are defined by expressing a dynamic relationship between these static elements, such as invocation or creation relationships. Concept views thus accommodate a range of different views.

a) The Declarative Framework The declarative framework, supporting the definition of perspectives, is organised as a layered structure where each layer makes use of the layers below it like in [Wuy01]. The following layers can be identified:

Representation layer This layer consists of the static and dynamic information gathered about the program and is modelled after the source meta-model. It corresponds to a declarative description of the source model as Prolog facts.

Base layer Rules making use of the dynamic and static information provided by the representation layer are defined in this layer. They can be roughly divided into two categories:

- rules giving semantics to the connectors in a view by defining a relationship between two source model entities: inheritance relationships, method invocations, ...
- rules clustering source model entities into components: class categories, ...

Auxiliary layer The auxiliary layer contains all rules defined by the user who is iteratively trying to obtain a view that answers his question. It may contain application-specific predicates (a clustering predicate which groups all classes within a certain class category, for example) and general reusable predicates (like design pattern[GHJV94] detection rules).

Perspective layer In this layer the predicates required for creating concept views are defined. Although querying the source model with the above predicates may already reveal some interesting information, a more global understanding of the system can be obtained by interpreting these high-level views.

A view V is a set of components C and connectors R corresponding to a directed graph. There will be a directed edge from component C_1 to component C_2 if there is at least one element e_1 in C_1 and one element e_2 in C_2 for which a relationship r holds.

Prolog is used to declaratively specify the predicate *C* which induces a partitioning of the source model and to specify the binary relationship *r* which needs to hold between two source elements in order for an edge to be drawn between the components each element is assigned to. These two predicates completely describe the perspective through which the developer wants to view the source model.

Views can then be shown by using the second-order predicate *createView*[7].

For understanding the invocations between class categories in the HotDraw[Joh92a] [Joh92b] framework, a view [8] can be created by invoking *createView(invokesClass, allInCategory)*. The perspective is then defined as show in Figure 2.2.

```
          composedView(sendsTo,allInCategory).

r rule8 :  sendsTo(Class1,Class2).

C rule9 :  allInCategory(Category,ListOfClasses) :-
               setof(Class,class(Class,Category),ListOfClasses).
```

Figure 2.2: Definition of a perspective

The resultant view is shown in Figure 2.3. Each node in the graph corresponds to a HotDraw class category and each directed edge from *A* to *B* means that at least one instance of a class in category in *A* invokes a method on an instance of a class in category *B*.

b) Implementation The above approach is implemented in a Tool called "Gaudi" which uses Prolog for declarative reasoning and DOT (a graph layout tool part of the open source Graphviz project) for displaying the calculated views.

2.2.2.5 Collaboration View Recovery

Since in this technique the source model is directly used and collaborations within it are discovered by pattern matching, little logic reasoning is performed so we will describe it only briefly.
The following definition is given [Ric02]:

> Such a view presents the dynamic information as a collection of class collaborations. The goal here is to understand how instances collaborate at runtime to carry out a certain functionality by abstracting from similar execution sequences to a collaboration. These abstractions are created by applying pattern matching to the execution trace the role of the engineer here is to specify what he or she considers to be similarity in execution sequences by modulating the pattern matching criteria. This gives semantics to the notion of collaboration.

[7] Different predicates are provided for the creation of labelled views, ordered views, ...
[8] Taken from [RD99]

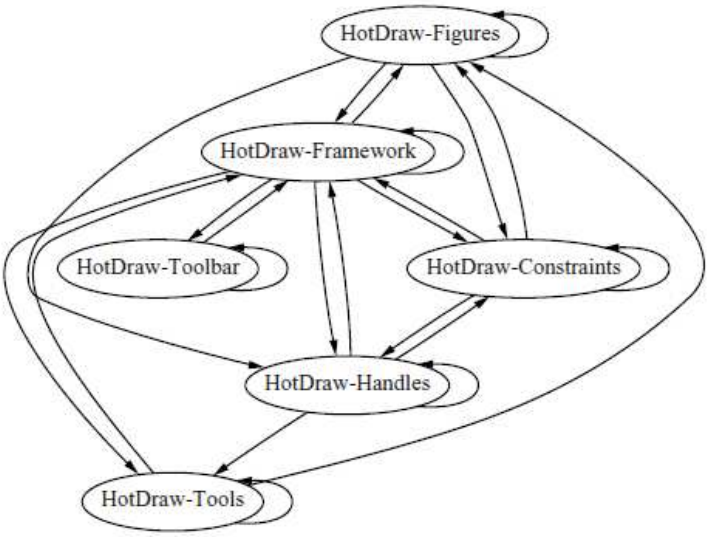

Figure 2.3: Example concept view

A developer searches for collaborations in a program's execution trace by modulating the pattern matching parameters of the tool. Each method invocation event (which records the basic information about the sender, receiver and method name) in the execution trace gives rise to a sequence of method invocations from the beginning of the invocation to its final return. These call trees are called *collaboration instances*.
Searching for *collaboration patterns* boils down to a search for equivalent collaboration instances. These patterns can be seen as an approximation of the high-level design concept of collaboration while the methods involved in such a collaboration pattern correspond, to a degree, to the high-level design concept of a role.
A developer can query the system for particular collaborations by using the multi-way[9] predicate: *sendInCollab(Sender, Receiver, Method, Collab* which holds when a collaboration instance in the collaboration pattern *Collab* invokes *Method* on an instance of *Receiver*.
A graphical front-end is provided for this query in the form of a tool which also allows for an collaboration instance to be displayed as an interaction diagram using the "Interaction diagramming-tool" from [BFJR98].

a) Pattern matching Pattern matching settings determine how the system abstracts from a *collaboration instance* to a *collaboration pattern* and can thus determine what *collaboration instances* are considered similar enough to be grouped together into *collaboration patterns*.
Three modulation options are offered:

Information about an event The following matching options can be chosen for the information conveyed by a single event:

- sender: ignore, match on object identity, match on sender class

[9]Meaning that a solution will be found for uninstantiated variables in the query

- receiver: ignore, match on object identity, match on receiver class, match on name of class defining method
- invoked method: ignore, match on method name, match on method category name

Events to exclude Events can be excluded from the trace whenever their depth of invocation with respect to the entire execution trace or with respect to the first invoked method of the collaboration pattern are above a given limit. Self sends –often of little importance– can be ignored too.

Structure of the collaboration instance It is possible to ignore the ordering and nesting of *collaboration instances* by treating them as sets of events instead of trees.

Richner concludes that the best results are obtained when the execution trace is treated as a set since matching on tree structure appears to be too restrictive for design recovery. The most useful modulation parameters are the maximum invocation depth and single event information which allows for polymorphism likes the class defining a method.

b) Implementation The pattern matching is based upon hashing: the call tree is traversed bottom-up and each invocation event node in the tree is assigned a hash-value.

2.2.2.6 Evaluation

In [Ric02], Richner concludes the following about concept view recovery:

> Our work on concept view recovery showed that perspectives provide for *simple view specification*, and that the logic programming framework supports *extensible view specification*. The case studies described showed that the views obtained are *succinct*, and that they enable us to answer many *behavioural questions* at both a *high-level* of component relations and a *low-level* of object interactions. The case studies also demonstrated an iterative recovery process *guided by the developer*.

a) Use of Static And Dynamic Information An experiment in concept view recovery was conducted in which the connectors of views only used static information from the program's source code. More specifically, the static invocation information from the FAMIX model was used instead of the dynamically retrieved method invocation events. This resulted in cluttered views since the number of candidate receivers of a message send is very large due to polymorphism.
Also, in general, very few static information is needed: only the basic object-oriented concepts like class hierarchies and methods are needed to obtain meaningful high-level views.

b) Use of Logic Programming This work has shown that logic programming can be used to describe and reason about dynamic information and as such can form a basis for design recovery tools.

c) Use of Pattern Matching Pattern matching is used to group together similar, but not necessary identical, method invocations in order to detect a form of large-scale collaborations.

d) Scalability Generating views of the queries about the moderately-sized use cases presented in [Ric02] all had response times well under 5 minutes.
Although the tool has not been tested on large systems, it is expected that it will scale quite well under the assumption that the developer will never want to know everything about the system, but rather only needs information about how a specific functionality is supported by the program's behaviour.
Focusing the investigation is thus extremely important and already begins in choosing which methods to instrument and which to leave uninstrumented. A good choice can be made by browsing through the source code. This focused program-slicing is inherent to systems using dynamic analysis. The approach also handles large amounts of information by letting the developer choose the size of the wanted view of the source model by specifying perspectives with a coarser component clustering.

2.2.3 Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects

This section summarises and discusses the recent re-engineering-oriented work of Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence and Narendra Jussien of the *École des Mines de Nantes*.
The work relies on a combination of no less than 3 declarative meta-programming paradigms and aspires to assist in all phases of the re-engineering process. It combines the use of coarse-grained static information with a dynamic analysis of binary class relationships.
The interested reader can consult [GDJ02, GJ01, Gué02, GAA01, AACGJ01] and [Gué03] for further reading.

2.2.3.1 Motivation

Design patterns can not only be used to describe the architecture of (legacy) applications, but also to detect and modify parts of the implementation that differ slightly from these well-known micro-architectures. As distorted versions may be indications of a poor implementation or the unfitness for the given problem, the authors make a case for the relationship between design patterns and design defects.

2.2.3.2 Approach

This work tries to aid in the entire re-engineering process by applying various declarative meta-programming techniques. A meta-model is proposed in which design patterns and program source code can be formally described. In order to help in rediscovering a program's architectural design, these models can be visualised. A constraint satisfaction problem (CSP), whose domain covers the application's architecture and whose constraints correspond to the entities in a given pattern and the relationships between them, is formulated. The solution to this problem is generated by an explanation-based constraint solver indicating which constraints needed to be relaxed in order for a solution to be found and thus identifies the shortcomings in the implementation. As each relaxed constraint is related to a transformation rule, the respective source code can then by transformed to better comply with the architectural design prescribed by the pattern in order to improve its quality.

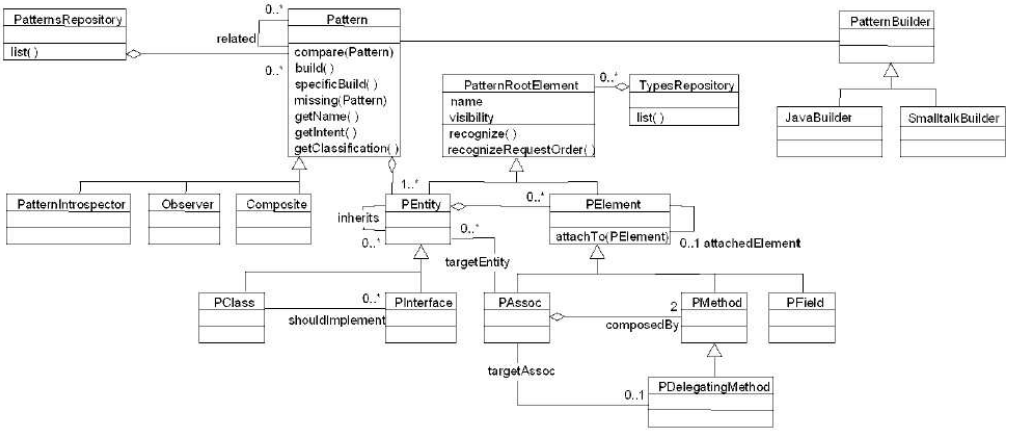

Figure 2.4: Overview of the classes in the Pattern Description Language meta-model.

2.2.3.3 Source Model

The generic micro-architectures of the design patterns given in [GHJV94] address re-occurring problems in object-oriented programs and do this in a manner independent from possible application domains. The model describing the entities in the design patterns' architecture is called an *abstract model* since none of the classes, methods or relationships have been applied to a specific problem yet. Models representing the application's architecture are called *concrete models* instead. Both models are described using the same set of constructs defined in the meta-model that is discussed in the next section.

a) Meta-Model A fully reifyable meta-model called the Pattern Description Language (PDL), is used to model the participants and their relations in a design pattern or an application's architecture as first-class Java entities which can be manipulated as are ordinary classes.
In order to create a model defining the semantics of a design pattern, one has to instantiate the classes provided by the meta-model. A partial overview of the available classes is given in Figure 2.4 [10].

Pattern Design patterns are instances of this class' descendants. It keeps track of a collection of entities, which correspond to the participants in the design pattern's description, and also provides general code generation and constraint formulating services to its subclasses. As such, the meta-model uniformly handles instantiation and detection.

PEntity Entities, which are either classes or interfaces, consist in turn of *PElement* instances.

PElement The PElement class represents collaborating instance variables, methods and associations in the pattern.

[10] Taken from [GAA01]

The following source code is an excerpt from an example given in [GAA01]. It declares the *Component* participant in the *Composite* design pattern. This declaration takes place in the constructor of the *Pattern* subclass.

```
component = new PInterface("Component")
operation = new Pmethod("operation")
component.addPElement(operation)
this.addPEntity(component)
```

The abstract model of the design pattern is concretised to the application's needs by instantiating the *Pattern* subclass and assigning each pattern participant its corresponding name in the application. This is handled by the *PatternsBox* tool.

b) Extraction

Static analysis Static information is extracted with the now discontinued CFParse binary class file parser toolkit from IBM. Extracting architectural information from the program's binaries is possible due to the one-to-one relationship between the JVM bytecodes and Java class definitions.

Dynamic analysis Maintainers using the PTIDJ (Pattern Trace Identification, Detection and Enhancement For Java) part of the re-engineering tool set can invoke the Caffeine tool to query the system to check their conjectures about its runtime behaviour. Example uses involve counting the number of times a specific method invocation pattern is encountered and checking whether a particular class is a singleton. Such queries are formulated in the logic programming language Prolog.

Furthermore, dynamic information is automatically used in the verification of the exact nature of binary class relations; an analysis often too costly or difficult to perform statically since instance lifetime and exclusiveness are involved. A composition relationship, for example, defines a subsumption constraint between the lifetime of instances of the whole and instances of the part. Instances of the part must also be exclusively contained by the whole alone.

Trace model A program's execution is modelled with execution events –covering both the control and data flow in the application – described by the Prolog predicates shown in figure 2.5[11]. The program's data flow is registered by the *fieldAccess*, *fieldValue* and the return value aware *methodExit* events.

Binary class relationship information obtained from the dynamic analysis is added to the architecture's model described by in the Pattern Description Language.

In contrast to the work discussed in section 2.2.2, Caffeine does not perform a *post-mortem trace*: it knows the current event and can request information about possible future execution events, but doesn't keep track of previously encountered events.

[11] Taken from [GDJ02]

Java execution event	Definition and parameters
fieldAccess(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be read.
fieldModification(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>, <Unique ID of the new object assigned to this field>, <Fully qualified name of the class of the new object>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be modified. The ID of the object to be assigned to the field is <Unique ID of the new object assigned to this field>. For visualization purpose, the fully-qualified name of the object class is given.
classLoad(<Class name>, <Event unique ID>)	The program requires the class <Class name>.
constructorEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the newly created instance>)	A new instance of class <Declaring class name> is being created. This instance is uniquely identified by <Unique ID of the newly created instance> for the rest of the program execution.
finalizeEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the being-finalized instance>)	The instance of class <Declaring class name> uniquely identified by <Unique ID of the being-finalized instance> is being finalized.
methodEntry(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is called on the instance uniquely identified by <Unique ID of the receiver instance>.
Identical definitions for dual events: classUnload; constructorExit; and, finalizerExit.	
Definition changes for event methodExit:	
methodExit(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>, <The value returned by this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is completing its call on the instance uniquely identified by <Unique ID of the receiver instance>. The returned value is provided in <The value returned by this method>.

Figure 2.5: Prolog predicates used to describe execution events in Caffeine

Implementation The Prolog engine JIProlog runs in a separate thread alongside the application that is being traced. Using the Java platform Debug Infrastructure (JDI), Prolog queries steer the program's execution with the *nextEvent* predicate which resumes the analysed program's execution until the next desired event is encountered.
Events are generated by wrapped entities in the instrumented code: return values are replaced by calls to the *CaffeineWrapper.methodReturnedValueWrapper* method which simply records and returns any given value while finalizer calls to the *System.exit(int)* method are replaced by calls to the *CaffeineWrapper.caffeineUniqueExit* method.
Instrumentation is once again performed with the CFParse tool. Because of the importance of correct lifetime information in the classification of binary class relationships, a separate thread is continuously making calls to the Java garbage collector to make sure instances are destroyed from the moment they are no longer needed in the application.

Usage In [Gué02], the following simple example illustrates the use of the *nextEvent* predicate in counting the number of times a certain "startTest" method is invoked:

```
query(N, M) :-
  nextEvent( [generateMethodEntryEvent], E),
   E = methodEntry(_, startTest, _, _, _),
   N1 is N + 1,
   query(N1, M).
query(N, N).

main(N, M) :- query(N, M).
main(N, N).
```

The analysed program is allowed to continue until a *methodEntry* event is encountered whose method name is then required to be "startTest". This will increment the counter.

2.2.3.4 Declarative Framework

PaLM [JB00], an explanation-based constraint satisfaction problem (CSP) solver, is used to solve a problem consisting of the following parts:

Variables Each problem has a set of variables associated which represent the object-oriented entities in the design pattern's abstract model.

Constraints A set of constraints is used to describe the relationships between the entities in the design pattern.

Domains The domain of each variable covers the entities in the application's concrete model.

The problem itself is described in the high-level CSP Claire [CL96] programming language.

A solution to this problem identifies in a program's source code architectural entities identical or similar to the micro-architecture put forth by a design pattern. Distorted solutions to the problem identify source code entities whose relationships only satisfy a subset from the problem constraints. The minimal subset of constraints to which an architecture must adhere is determined either by the user or by the weights associated with each constraint. Unsatisfied constraints are pointers for design defects which should be remedied.

The built-in constraints govern relationships ranging from inheritance (*StrictInheritanceConstraint* $A < B$, stating that class A must be the superclass of class B), over invocation knowledge (*RelatedClassesConstraint* $A \triangleright B$ stating class A invokes a method of class B) to instance variable types (*PropertyTypeConstraint* $B : A.f = B$ stating that the field f in class A must be of type B).

In [GAA01], a CSP definition for the Composite design pattern is given, of which the following declaration of the problem's variables is an extract:

```
[problemForCompositePattern() : PalmEnumProblem ->
  ...
  let pb := makePalmEnumProblem("Composite Pattern",
                                length(listC),
                                length(listC)),
      leavesTypes := makePtidejIntVar(pb, "LeavesType", ...
      leaves := makePtidejIntVar(pb, "Leave", ...
      composites := makePtidejIntVar(pb, "Composite", ...
      components := makePtidejIntVar(pb, "Component", ...
```

A new problem is declared whose domain is the number of entities in the program architecture (*length(listC)*) since classes are enumerated. This is immediately followed by the declaration of the variables representing the three entities in the design pattern: *Leaf, Composite, Component*.

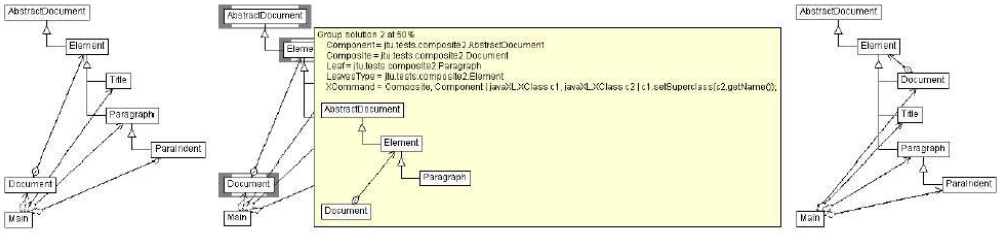

Figure 2.6: Interaction with the PTIDJ tool

A number of constraints are then posed to the problem. Amongst others is the *Strict-InheritanceConstraint* on the composite and component variables to which a weight of 50 is attached:

```
post(pb,
     makeStrictInheritanceConstraint( "Composite,Component |
                                          javaXL.XClass c1,
                                          javaXL.XClass c2 |
                                          c1.setSuperclass(c2.getName());",
                                      composites, components),
     50)
```

JavaXL [AA01] transformation rules are associated along with each constraint to correct the source code of the classes that failed to comply.

2.2.3.5 Transformation Rules

JavaXL or Java Extended Language [AA01], a source-to-source Java transformation engine, takes care of the source code modifications suggested by the explanations provided by a distorted CSP solution.
Figure 2.6[12] shows the interactions with the graphical front-end to the tool set with on the left the original application's architecture. In the center a solution to the Composite problem with a 50% compliance is shown together with the proposed transformation rules. The right side of the figure presents the application's architecture when the proposed refactoring is performed.
When making modifications to the code, JavaXL always tries to make only the most necessary alterations so the original author's coding conventions and comments are preserved. If a single field's visibility needs to be changed, no other modifications are performed in contrast to most transformation tools which regenerate the entire program's code.

2.2.3.6 Evaluation

a) Use of Static and Dynamic Information As the design patterns are detected mostly on the basis of static information, this approach is best for detecting design patterns which rely heavily on the structural definition of their micro-architecture. The behavioural aspect of design patterns is minimised which may limit this technique's

[12]Taken from [GAA01]

generic applicability with respect to the detection of behavioural design patterns which require statically undecidable information.

Dynamic information is however used for a detailed classification of binary class relationships based on lifetime and exclusiveness properties. A clear distinction is made between associations, aggregations and compositions. This dynamic information is incorporated in the application's model, but isn't really used in a stand-alone fashion.

b) Use of Logic Programming Though it is possible for a user to query a system's dynamic behaviour using the Caffeine tool, the possibilities are somewhat limited since it is based on a ad-hoc trace instead of a post-mortem trace where information about past events is available too. This kind of information may however not be required in an investigation of isolated, local program behaviour. As such, letting Prolog steer the execution of the analysed program can still bring other interesting information to attention.

c) Use of Reifyable Meta-Models The Pattern Description Language provides a unified way to code generation for and detection of design patterns. Using this meta-model, design patterns can be formalised as first-class programming entities which can further be manipulated. Models using PDL however only incorporate relationships between structural entities such as classes and interfaces, not between instances at run-time.

d) Use of Explanation-based Constraint Programming This approach has proven that constraint satisfaction problems offer, in combination with an explanation-based constraint solver, a user-friendly way of detecting even distorted versions of design patterns.

The explanations of the system can be used to exactly pin-point the differences in the source code and even to refactor design defects. The latter is in contrast to techniques which use logic programming (like [Wuy01]) where all possible ways in which a pattern may be violated have to be explicitly coded in advance in a separate logic rule. Such (unmodified) systems cannot explain why there's no solution to the logic rule either.

e) Use of Transformation Rules It may not always be possible to associate a simple transformation rule to every constraint. Furthermore, small deviations from a design pattern may be due to domain-specific requirements. Therefore, the proposed transformations may not always improve the actual source code.

f) Scalability In [GAA01] it is stated that mapping the meta-model over a large application is a slow and fragile process which requires a large amount of memory. The detection of the design patterns once the meta-model has been created is however quite fast as constraint satisfaction problems are already in industrial use.

2.3 Conclusions

This chapter introduced the software re-engineering process and compared the benefits of dynamic and static analysis therein. It also provided an overview of the recent (circa 2000) declarative meta-programming approaches to software re-engineering. The presented works were selected because of their original solutions to the problems in the field.

They differ on the use of dynamic versus static information, how this information is extracted, the source meta-models describing this information and their granularity. Another axis of comparison is the incorporated declarative programming paradigm, and how well it deals with distorted or violated programming patterns, the amount of language symbiosis offered, and the extensibility and expressiveness of the meta-level and last but not least the phases of the re-engineering process that are supported.

Meta-programming applications developed in SOUL have proven that logic meta-programming can indeed be used to reason about a program's source code and by doing so aid in the software engineering process. These applications are limited to statically obtained information and are primarily used to reason about an application's architecture. SOUL itself will serve as the basis for our experiments with dynamic program analysis as its existing library for structural code reasoning can be used to check whether both are complementary in a software re-engineering setting. The symbiosis with Smalltalk will furthermore allow advanced reasoning and inspection of the run-time instances involved in the execution of the program under investigation.

Richner's work is very interesting as it is one of the first dynamic analysis tools that go beyond mere visualisations of a program's run-time behaviour and as such presents a major contribution to the field of behavioural reverse engineering using declarative languages and dynamic information.

Guéhéneuc's work is interesting from two perspectives. It is a largely statical approach to software re-engineering, but can optionally incorporate dynamic information to refine binary class relationships in UML diagrams. It also proposes explanation-based constraint logic programming to overcome the semantic gap between implementations in a specific domain and abstract models of design patterns. A fully automated process will probably never be available, but the proposed techniques are however a major step towards that end. We will tackle the same problems in our work using dynamic analysis and approximate reasoning in contrast.

Chapter 3

Approximate Reasoning and Fuzzy Logic Programming Techniques

3.1 Introduction

3.1.1 Approximate Reasoning

As our world is pervaded with vague and uncertain information, incorporating some form of human-like common sense reasoning is necessary for automated reasoning systems. In real-life situations, formal logical deduction seems unable to draw intelligent conclusions from axiomatically stored information. Indeed, it seems impossible to model statements such as *"Peter is possibly quite tall"* in classical systems that only allow crisp and absolutely correct information to be processed.

A more intuitive, but still well-defined and machine implementable, method for conceptualising and solving real world problems is thus required. Instead of using a two-valued logic where the only possible truth values are absolute truth and absolute falsehood, it seems natural to develop a many-valued logic where propositions with varying degrees of truth are allowed. This chapter will explore some of the various techniques that emerged over the last decades to facilitate approximate reasoning under vague and uncertain information.

3.1.2 Uncertainty and Vagueness

In general, a source of information gives rise to two kinds of uncertainty: the subject of the information can be fuzzy or vague, and the statements about the subject can be uncertain. The outcome of a throw with a die is uncertain, but it will always be a precise number. On the other hand, outcomes might not always be sharply defined in the real world: "sunny" and "clouded" are vague specifiers and the distinction between these two concepts is blurry.

Uncertainty Uncertainty exists when we're unable to determine whether a statement is entirely true or false. This is most often due to a lack of information about

the state of the world. An example of an uncertain statement is "The next throw of this die will be 4". This can be considered as a crisp proposition which may either be true or false, but we can only bet on which will be the case.

We might however be able to provide an estimate of the truth of such a statement in a numerical form, or try to produce sensible conclusions even when we're lacking certain information.

In this chapter we will concentrate on the former approach in which we will try to associate a probability or degree of belief in the statement that the sentence is true. More formally, the probability of a crisp statement ϕ can be seen as the truth-value of the vague statement "ϕ is probable". [HG]

We will concentrate on techniques for modelling uncertainty other than the purely probabilistic techniques since associating correct probabilities to facts modelling a program's source code and determining their interdependency doesn't seem to bring us closer to goal of introducing common-sense reasoning techniques.

Vagueness Vagueness is introduced when a statement contains a gradual predicate whose meaning is vague and depends on the context. In other words, we have an imprecise statement under complete information. An example of such a predicate is "warm". Clearly, what a person perceives as warm weather in mid-winter differs from warm weather in mid-summer and even then it is a very subjective and individual assessment of a given temperature. Therefore, a statement involving vague predicates is often not entirely true nor false and thus uncertain.

As we will see, determining the truth degree of a statement "X is Y" can be approached from two different mathematically sound viewpoints. On one hand, the value of X can be entirely known at which point we can determine the degree of truth by measuring how well it matches the concept defined by Y. On the other hand, the only information available about X may be the fact that it is Y. In that case Y can be considered as constraining the actual possible values of X.
Considering the certainty of a statement as a vague predicate in se is more close to the first viewpoint while the second viewpoint lends itself to treating uncertainty more as in a flexible probability-like theory coined possibility theory whose proponents insist that uncertainty and vagueness need to be considered as completely different phenomena.

The concept of truth-functionality or truth-compositionality provides another dimension to catalogue fuzzy systems: can we determine the truth value of a formula by the truth of its constituents?
It can be argued that uncertainty intuitively doesn't behave compositional. Given a formula φ with the same uncertainty as $\neg\varphi$ and a fixed function F which determines the uncertainty of a conjunction, the uncertainty T of a formula $\varphi \wedge \varphi$ is $T(\varphi \wedge \varphi) = F(T(\varphi), T(\varphi)) = F(T(\varphi), T(\neg\varphi)) = T(\varphi \wedge \neg\varphi)$ under the assumption of compositional uncertainty, which isn't very intuitive.
The inapplicability of truth-functionality for handling uncertainty is well-known in probability theory where it is impossible to determine the probability of two events happening concurrently when we have no information about the independence of the individual events. Thus, we should never interpret degrees of truth as probabilities.
The truth degree of vagueness can however be seen as a truth-functional concept. In the most popular choices for truth-functional valuation functions τ this might mean that $\tau(\varphi \wedge \neg\varphi) >= 0$ which covers statements such as "It is warm: yes and no" where the

truth degree of the constituents is for example 0.5.

Handling the concepts of uncertainty and vagueness equally well in one formalism is rather difficult. We will only be able to associate a degree of truth to the proposition "Peter is rather tall" when a precise (and some might argue artificial) meaning has been associated with the vague concept "rather tall". This exemplifies why most of the systems discussed in this chapter are either better suited for handling uncertainty or for handling vagueness depending on the particular viewpoint from which they originated.

3.2 Fuzzy Sets and Logic

Fuzzy sets were proposed by Zadeh [Zad65] in the mid-sixties as a mathematical framework for capturing fuzzy concepts without sharp boundaries that originate from human-like descriptions of systems. It provides a mathematical foundation for specifying and reasoning about imprecise information. As such, fuzzy set theory can be considered as the inspiration for most of the systems described in this chapter.

The problem of conceptualising vague information isn't new and in fact has been discussed in antiquity as testified by the Sorites paradox, derived from the Greek word *sores* meaning *heap*, which originally referred to the following puzzle attributed to Eubulides of Miletus: "Would you describe a single grain of wheat as a heap? No. Would you describe two grains of wheat as a heap? No. You must admit the presence of a heap sooner or later, so where do you draw the line?". The problem can be situated in the inapplicability of classical induction to a vague concept φ: the classical interpretation of the implication $\varphi(x) \rightarrow \varphi(x+1)$ causes the paradox. In [Hyd02] an extensive historical and philosophical treatment is given of this entire family of paradoxes.

Fuzzy set theory offers one solution to this paradox by allowing various degrees of "being a heap" through its extension of the classical all-or-nothing binary-valued set membership function to a gradual real-valued one.

3.2.1 A Generalised Characteristic Function

3.2.1.1 Fuzzy Extension

Apart from the extensional enumeration of all elements ($\{0,2,4\}$) and the intensional definition of the common properties of the elements ($\{2x \in \aleph \,|\, x < 3\}$), a conventional crisp set A is defined by its characteristic function μ_A which associates 1 to elements of the universe U belonging to A and 0 to the elements out of A:

$$\mu_A : U \rightarrow 0,1$$
$$\mu_A(x) = \begin{cases} 0 \leftrightarrow x \notin A \\ 1 \leftrightarrow x \in A \end{cases}$$

The characteristic function $\mu_A$of a fuzzy set A takes on values in the real interval $[0,1]$ for every element in the universe U thus admitting intermediate membership values. The value $\mu_A(x)$ represents the degree of membership of the element x to the fuzzy set A and can also be seen as the degree of truth that the proposition $x \in A$ is true. As the endpoints of the interval $[0,1]$ still represent respectively full set exclusion and inclusion, this is an extension of the characteristic function of conventional sets.

$$\mu_A : U \rightarrow [0,1]$$

$$\mu_A(x) = \begin{cases} 0 \leftrightarrow x \notin A \\ 1 \leftrightarrow x \in A \\ 0 < \alpha < 1 \leftrightarrow x \in A \text{ to the extent } \alpha \end{cases}$$

A fuzzy set is still considered as a collection of elements sharing a common feature and thus it still describes a certain concept. However, a fuzzy set doesn't introduce a complete dichotomy in the universe. In addition to being completely compatible or incompatible with the concept defined by the set, elements of the universe can also share the features described by the set to a certain degree. The fuzzy set OLD, for example, assigns a high membership degree to 70-year olds and a low membership degree to toddlers.

The intensional definition of a fuzzy set A with respect to the universe U is then formally defined as the set of tuples: $\{(x, \mu_A(x)) | x \in U, \mu_A(x) \in [0,1]\}$. A fuzzy set is described as the union of all the elements in the universe graded by the membership function, or in Zadeh's original notation for a finite universe: $A = x_1/\mu_A(x_1) + \ldots + x_n/\mu_A(x_n) = \sum_{i=1}^{n} x_i/\mu_A(x_i)$ where $+$ stands for union.

3.2.1.2 Assigning Membership Degrees

We may now interpret the values of μ_A as membership degrees or equivalently degrees of truth, but we haven't considered yet what it means to assign a membership degree of only 0.7 instead of 0.8.

Fuzzy sets and the derived fuzzy logic have been particularly successful in the domain of control systems where one has to react in a fuzzy manner to changes in input readings. In such a setting with exact sensory input, a precise meaning can be associated with the membership degrees of, for example, a certain temperature to the fuzzy set WARM. Examples of such membership functions are given in the next section.

In expert systems which model and reason with human-originated information, this is however often impossible and numbers might seem to be assigned rather arbitrarily to truth degrees. To overcome this problem, the unit interval $[0,1]$ (inspired by the fact that crisp membership functions have a range $\{0,1\}$) can be simply interpreted as an ordinal scale instead which may also be more abstract, with the single constraint that it is a totally ordered set in order to ensure a smooth membership grade.

One might still object that this approach makes it difficult to compare values assigned by different persons, but one should keep in mind that vagueness is context-dependent and subjective in se. Also, as not all objects in the real world can be compared with each other, the same is true for fuzzy concepts. To overcome this problem, an L-fuzzy set variant (L from lattice) has been proposed [Gog67] for handling incomparable information where the range of the membership function is a partially ordered set.

3.2.1.3 Common Membership Functions

In most control applications, it is sufficient to specify the general shape of the membership function together with some parameters (notation from [DP80]):

Open right shoulder

$$\Gamma(x,\alpha,\beta) = \begin{cases} 0 & x < \alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha \leq x \leq \beta \\ 1 & x > \beta \end{cases}$$

This function is shaped as the left side of a trapezium with the right side open.

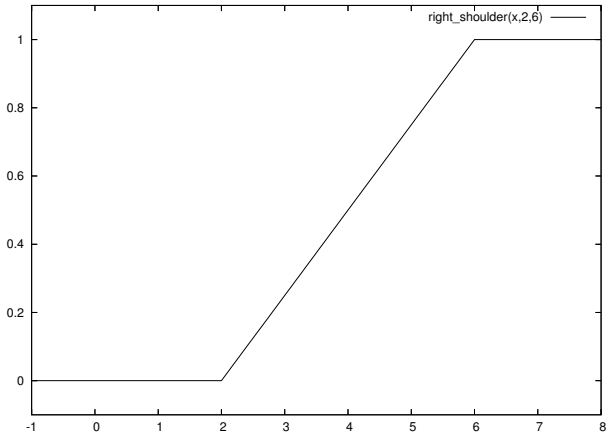

Figure 3.1: Plot of the open right shoulder membership function $\Gamma(x,2,6)$

Open left shoulder

$$L(x,\alpha,\beta) = \begin{cases} 1 & x < \alpha \\ (\beta-x)/(\beta-\alpha) & \alpha \leq x \leq \beta \\ 0 & x > \beta \end{cases}$$

This function is shaped as the right side of a trapezium with the left side open.

Triangular shape

$$\Delta(x,\alpha,\beta,\gamma) = \begin{cases} 0 & x < \alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha \leq x \leq \beta \\ (\gamma-x)/(\gamma-\beta) & \beta \leq x \leq \gamma \\ 0 & x > \gamma \end{cases}$$

The shape of this function is a triangle centred around β with the left and right endpoints situated at α and γ respectively. Its support (see the vocabulary section) is the triangle's base minus the endpoints: (α,γ). In a way, it models the approximate neighbourhood of x to β.

Trapezoidal shape

$$\Pi(x,\alpha,\beta,\gamma,\delta) = \begin{cases} 0 & x < \alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha \leq x \leq \beta \\ 1 & \beta \leq x \leq \gamma \\ (\delta-x)/(\delta-\gamma) & \gamma \leq x \leq \delta \\ 0 & x > \delta \end{cases}$$

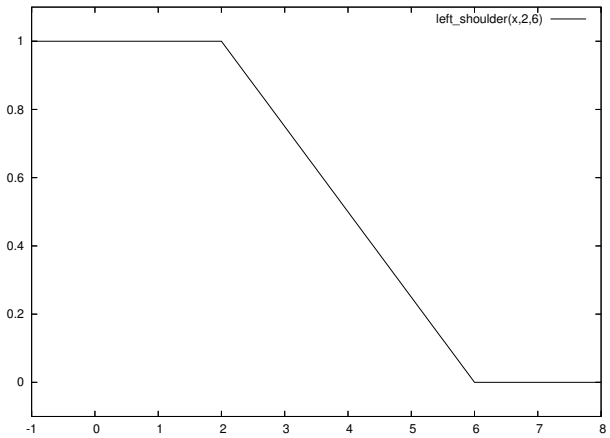

Figure 3.2: Plot of the open left shoulder membership function $L(x,2,6)$

The shape of this function is a trapezium with the top horizontal defined by β and γ while its left (right) endpoint is defined by α (δ). In a way, it models the approximate neighbourhood of x to the interval $[\beta,\gamma]$.

Other examples of often used member functions are sigmoidal and gaussian shaped membership functions offering even more gradual transitions.

Although the above functions have well-understood mathematical properties, one isn't obliged to use any of them. In fact, many membership functions are defined in a simple relational manner: $\mu_{around_2}(0) = 0.2, \mu_{around_2}(1) = 0.7, \mu_{around_2}(2) = 1, \mu_{around_2}(3) = 0.7, \ldots$

3.2.1.4 Common Vocabulary

The following are definitions of re-occurring vocabulary in fuzzy set literature.

Empty set The empty fuzzy set $\emptyset$ is defined as the set with $\forall x \in U : \mu_\emptyset(x) = 0$. It is a subset of ever fuzzy set.

Largest set The largest fuzzy set 1 in the universe U is defined as the set with $\forall x \in U : \mu_1(x) = 1$. Every fuzzy set is a subset of 1.

Core The core of a fuzzy set A is the crisp set $\{x|\mu_A(x) = 1\}$.

Bandwidth The bandwidth of a fuzzy set A is the crisp set $\{x|\mu_A(x) \geq 0.5\}$.

Support The support of a fuzzy set A is the crisp set $\{x|\mu_A(x) > 0\}$.

α-cut The α-cut of a fuzzy set A is the crisp set $A_\alpha = \{x|\mu_A(x) \geq 0\}$. If $\alpha_1 > \alpha_2$ then $A_{\alpha_1} \subset A_{\alpha_2}$.

Normality A fuzzy set A is called normal iff $\exists x \in U : \mu_A(x) = 1$.

Subsets A fuzzy set A is a subset of a fuzzy set B iff $\forall x \in U : \mu_A(x) \leq \mu_B(x)$.

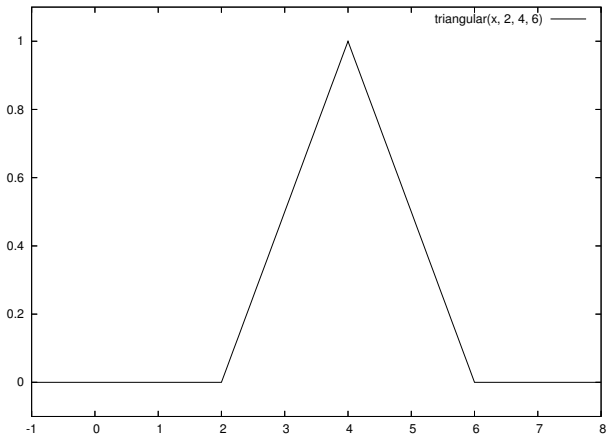

Figure 3.3: Plot of the triangular membership function $\Delta(x,2,4,6)$

Equality Two fuzzy sets A and B are equal iff $A \subset B$ and $B \subset A$ or, equivalently, $\mu_A(x) = \mu_B(x)$.

Linguistic variables A linguistic variable is a variable whose value is a linguistic expressions modelled as a fuzzy set. An example is the variable 'temperature' whose value (given a precise input) can be one of the following: very cold, cold, moderate, warm, very warm.

They play an import role in the fuzzy control sub-domain, where they feature in fuzzy if-then rules of which the antecedent and the conclusion consist of linguistic variables.

The original definition by Zadeh is the following quintuple[Zad75a]:

$$\langle X, T(X), U, G, M \rangle$$

- X is the name of the variable
- $T(X)$ is the term set of X e.g. $\{verycold, cold, moderate, warm, verywarm\}$
- U is as usual the universe of discourse
- G is the syntactic rule (or grammar) generating terms from $T(X)$
- M is the semantic rule assigning a fuzzy set to each term from $T(X)$

Hence, we can also consider a linguistic variable A as a function with a crisp domain D_A of input values that will be mapped to a fuzzy set.

Fuzzy relation Given two universes of discourse U and V, a fuzzy relation R is a fuzzy set in their product space $U \times V$ with an associated membership function $\mu_R(u,v)$.

3.2.2 Set-Theoretic and Logical Operations

By now we are able to model expressions such as "Temperature is High", but we would also like to combine fuzzy sets in order to model expressions like "(Temperature is

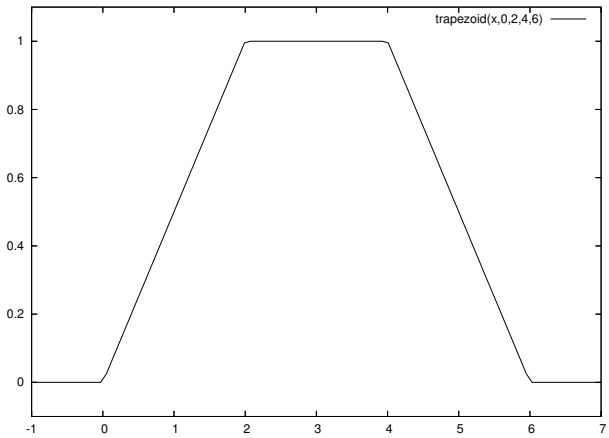

Figure 3.4: Plot of the trapezoidal membership function $\Pi(x, 0, 2, 4, 6)$

High) and (Pressure is Low)". By the natural analogy of sets and propositions, we will introduce many-valued logics at the same time.

3.2.2.1 Fuzzy Extensions of Conventional Set Operations

As we introduced fuzzy sets as an extension of conventional crisp sets (or alternatively conventional sets are special cases of fuzzy sets), we would like our set-theoretic operations to behave in the conventional way meaning they need to be closed (their results are again a fuzzy set) and their results coincide at least on the boundaries of the unit interval $[0, 1]$.

In the next section we will introduce the general class of operations which conform to these requirements, but we will start our discussion with the operations originally proposed by Zadeh as these are still the most popular choices:

- Intersection: $\mu_{A\cap B}(x) = min(\mu_A(x), \mu_B(x))$
- Union: $\mu_{A\cup B}(x) = max(\mu_A(x), \mu_B(x))$
- Complement: $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$

These definitions coincide with the natural set operations for the boundaries of the unit interval, but one can object that intersection is pessimistically modelled since augmenting the truth value of the largest of the two operands doesn't change the outcome of the operation at all.

Obviously, since we are now allowing intermediate values between absolute truth and absolute falsity, the classical law of the *excluded middle* can no longer be valid: $A \cap \bar{A} = \emptyset$ is only true when $A = \emptyset$ and $\bar{A} = 1$. Similarly, the (fuzzy variant) of the law of non-contradiction $\mu_{A\cup\bar{A}} = 1$ is only true under the same conditions. As pointed out before, these deviations are necessary to handle truth degrees. De Morgan's laws are still valid.

3.2.2.2 Logical Algebra with Triangular Norms and Co-norms

The requirements on the set-theoretic operations (logical connectives) can be summarised in the notions of a triangular norm (t-norm) and triangular co-norm (also called an s-norm) [SS63] which correspond to intersection (conjunction) and union (disjunction) respectively. They are in effect generalisations of the original set-theoretic operations put forth by Zadeh.

T-norm A t-norm is a function $* : [0,1] \times [0,1] \rightarrow [0,1]$ which is commutative, associative, non-decreasing in both arguments with 1 as the unit element and 0 as the zero element. It is used to model intersections and conjunctions.

1. *Commutativity:* $x * y = y * x$
2. *Associativity:* $(x * y) * z = x * (y * z)$
   Associativity allows t-norms to be extended to multiple arguments.
3. *Non-decreasing monotonicity:* $\forall x < x', y < y' : x * y \leq x' * y'$.
   This requirement expresses the requirement that if we augment the degree of truth of one of the constituents of a logical formula its degree of truth should at least be as large as that of the original formula. In addition, a t-norm is called strict when it is strictly increasing in both arguments.
4. *Boundary conditions:* $1 * x = x, 0 * x = 0$
   This requirement gives the function its conjunctive semantics on the unit boundaries.

We will primarily be interested in continuous t-norms as they are suitable for modelling the gradual aspect of fuzzy logic.

As we will see in a moment, choosing a particular t-norm for conjunction semantics actually also fixes the semantics for disjunction and implication. The following basic t-norms are hence named after the multi-valued logic to which they give rise:

- *Gödel* $x \wedge y = min(x,y)$ This t-norm has the same operational semantics as those proposed by Zadeh and is suitable for calculating upper bounds on degrees of truth where one assumes the scenario in which one statement subsumes the other.
  This particular t-norm is the only idempotent one: $x \wedge x = x$ (a proof is given below).
- *Łukasiewicz* $x \wedge y = max(x + y - 1, 0)$ This t-norm is suitable for calculating lower bounds on degrees of truth where one assumes the scenario in which statements are unrelated.
  For this t-norm we have $c \wedge c < c (c \notin 0, 1)$.
- *Product* $x \wedge y = x \cdot y$ This is the classical product operator from probability theory in which one assumes total independence of the arguments.
  The multi-valued logic branch resulting from this t-norm has only recently been studied in [HGE96].

These three t-norms are called *basic* since a theorem [Lin65] states that all other t-norms are either isomorphic to the Łukasiewicz and Product t-norms or equal to the Gödel minimum t-norm or a combination of the above.

There is a certain ordering among t-norms as $\forall$ t-norms

$$t : t(x,y) \leq min(x,y)$$

since $t(x,y) \leq t(x,1) \leq x$ follows from monotonicity and the boundary condition while we can derive $t(x,y) = t(y,x) \leq t(y,1) \leq y$ in the same manner from commutativity. Hence, $t(x,y) \leq min(x,y)$.

Whenever $t_1(x,y) \leq t_2(x,y) \forall x,y \in [0,1]$ we say that t_1 is the weaker t-norm and t_2 is the stronger. The strongest t-norm is thus $min(x,y)$.

This result also allows us to prove that the Gödel conjunction is the only idempotent t-norm as $a = t(a,a) \leq t(a,b) \leq min(a,b)$ at least for $a < b$. However, due to commutativity we can also derive $a = t(a,a) \leq t(a,a) = t(b,a) \leq min(b,a)$ so this statement is true for any a and b.

Moreover, it can also be shown that the t-norm *Drastic product* is the weakest t-norm. It is defined as follows:

$$a \wedge b = \begin{cases} min(a,b) & \text{if } max(a,b) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Co-norm A t - co-norm or s - norm is a function $+ : [0,1] \times [0,1] \rightarrow [0,1]$ obtained by De Morgan's law (given a suitable negation operator) from a t-norm $*$ in the following way: $x + y = \overline{x * y} = 1 - ((1-x) * (1-y))$ under the usual semantics of negation.

Consequently, the same requirements for a t-norm are in order except for the boundaries which are defined as $x + 0 = x, 1 + x = 1$. Co-norms are used to model unions and disjunctions.

The basic co-norms are:

- *Gödel* $x \vee y = max(x,y)$
- *Łukasiewicz* $x \vee y = min(1, x+y)$
- *Product* $x \vee y = a + b - a \cdot b$

An ordering similar to the one in t-norms exists for co-norms: $\forall$ co-norms $s : s(x,y) \geq max(x,y)$ since $s(x,y) \geq s(x,0) \geq x$ follows from monotonicity and the boundary condition while we can derive $s(x,y) = s(y,x) \geq s(y,0) \geq y$ in the same manner from commutativity. Hence, $s(x,y) \geq max(x,y)$ is the weakest t-co-norm.

The strongest t-co-norm is *Drastic Sum* defined as:

$$a \vee b = \begin{cases} max(a,b) & \text{if } min(a,b) = 0 \\ 1 & \text{otherwise} \end{cases}$$

Negation In the above definition of a co-norm, we have already used the negational semantics put forth by Zadeh. The minimal requirement for a negation operation is however to switch the boundaries of the unit interval: $\neg(0) = 1$ and $\neg(1) = 0$ and to be non-decreasing in its argument.

This restriction is too relaxed for co-norms to be defined in terms of t-norms and complementation since it doesn't restrict semantics on the interval between the

unit endpoints. In light of this, $\neg : [0,1] \rightarrow [0,1]$ can be demanded to adhere to involutiveness constraint $\neg\neg x = x$.

The negation defined by Zadeh $\neg x = 1 - x$ is thus a suitable choice and can be seen in a more formal manner as the result of the formula: $\neg\varphi = \varphi \rightarrow_{Ł} 0$ in which $\rightarrow_{Ł}$ is the Łukasiewicz implication. In contrast, the negation operator $\neg(0) = 1, \neg(x \in (0,1]) = 0$ obtained from Gödel's implication $\rightarrow_G$ doesn't adhere to the involutiveness restriction.

Implication will be the subject of the next section.

It can be proved [Kle82] that every t-norm t is distributive to the *max* co-norm operator: $t(max(a,b),c) = max(t(a,c),t(b,c))$. If we thus want an idempotent t-norm as well as distributivity, our choice is limited to Gödel semantics.

3.2.2.3 Implication Operators

Many semantics can be given to the fuzzy variant of implication, but we will start our investigation by recalling the truth table of the implication in binary propositional logic:

φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1

Standard strict implication This implication is based on the intuitive extension to many-valued logics of the following observation:

$$\varphi \rightarrow \psi \iff \begin{cases} 1 & \varphi \leq \psi \\ 0 & \text{otherwise} \end{cases}$$

This operator is very sensitive to small changes in its input: given a φ with a truth degree of 0.4, $\varphi \rightarrow \psi$ will be 1 when ψ is 0.4, but 0 when ψ is 0.39.

Residuated implication We also have the following residuation equation:

$$\varphi \rightarrow_* \psi \geq \chi \iff \varphi * \chi \leq \psi$$

where $*$ is a continuous t-norm which now uniquely determines its residuated implication $\rightarrow_*$:

$$\varphi \rightarrow_* \psi = max\{\chi \in [0,1] | \varphi * \chi \leq \psi\}$$

The residuated implications corresponding to the three basic t-norms are:

- *Gödel*

$$\varphi \rightarrow \psi \iff \begin{cases} 1 & \varphi \leq \psi \\ \psi & \text{otherwise} \end{cases}$$

- *Łukasiewicz*

$$\varphi \rightarrow \psi \iff \begin{cases} 1 & \varphi \leq \psi \\ 1 - \varphi + \psi & \text{otherwise} \end{cases}$$

- *Product*

$$\varphi \to \psi \iff \begin{cases} 1 & \varphi \leq \psi \\ y/x & \text{otherwise} \end{cases}$$

Kleene-Dienes implication This operator is obtained from the equation $\varphi \to \psi = \neg\varphi \vee \psi$ using the definition of union and negation from Zadeh:

$$\varphi \to \psi = max(1-\varphi, \psi)$$

or from Łukasiewicz:

$$\varphi \to \psi = 1 - x + y$$

Mamdani implication This one is often used in expert systems and is simply defined as the minimum:

$$\varphi \to \psi = min(\varphi, \psi)$$

Larsen implication This implication is also primarily of use in expert systems and is simply defined as the product:

$$\varphi \to \psi = \varphi \cdot \psi$$

Implication in two-valued logic can no longer be seen as a special case of a many-valued logic when Mamdani or Larsen implication is chosen since $0 \to 0$ gives 0. They can be considered primarily for engineering applications such as expert or control systems where performance matters and the case in which rule antecedents are false is irrelevant.

3.2.2.4 Deductive Systems of Many-Valued Logics

The previous sections described the different possibilities for the truth valuations of logical connectives. A basic many-valued propositional logic system based on continuous t-norms can now be developed using appropriate axioms and modus ponens (given φ and $\varphi \to \psi$ we can derive ψ) as the deduction rule for constructing proofs. The older Gödel logic and Łuckasiewicz logic can be seen as extensions of this basic fuzzy logic by Hájek [H98].
We will omit axiomatisation and completeness proofs for these logics as they are beyond the scope of this document. A good overview is provided in [HG].

Basic fuzzy logic is completely based on a t-norm $*$ and its residuated implication $\to_*$ with the conventional logic connectives defined as follows:

- $\varphi \wedge \psi = \varphi * (\varphi \to_* \psi)$
- $\varphi \vee \psi = ((\varphi \to_* \psi) \to_* \psi) \wedge ((\psi \to_* \varphi) \to_* \varphi)$
- $\neg\varphi = \varphi \to_* 0$
- $\varphi \leftrightarrow \psi = (\varphi \to_* \psi) * (\psi \to_* \varphi)$

The axiomatisation of basic fuzzy logic is the usual:

1. $(\varphi \to_* \psi) \to_* ((\psi \to_* \chi) \to_* (\varphi \to_* \chi))$
2. $(\varphi * \psi) \to_* \varphi$

3. $(\varphi * \psi) \rightarrow_* (\psi * \varphi)$
4. $(\varphi * (\varphi \rightarrow_* \psi)) \rightarrow_* (\psi * (\psi \rightarrow_* \varphi))$
5. $(\varphi \rightarrow_* (\psi \rightarrow_* \chi)) \rightarrow_* ((\varphi * \psi) \rightarrow_* \chi)$
6. $((\varphi * \psi) \rightarrow_* \chi) \rightarrow_* (\varphi \rightarrow_* (\psi \rightarrow_* \chi))$
7. $((\varphi \rightarrow_* \psi) \rightarrow_* \chi) \rightarrow_* (((\varphi \rightarrow_* \psi) \rightarrow_* \chi) \rightarrow_* \chi$
8. $0 \rightarrow_* \varphi$

It is interesting to note how the original Gödel (1933) [Goe33] and Łukasiewicz (1970) infinitely-valued logics can be obtained from the corresponding basic continuous t-norms:

Łukasiewicz In this logic system, we can add the axiom $\neg\neg\varphi \rightarrow_* \varphi$ as a result of the convolutiveness of the negation corresponding to this t-norm as we have seen before.

Gödel We obtain Gödel logic by adding the axiom $\varphi \rightarrow_* (\varphi * \varphi)$ which states that the Gödel t-norm is idempotent.

Product The axioms for a logic based on the product t-norm are the same as the basic axioms extended by the axiom $\neg\neg\varphi \rightarrow_* ((\varphi \rightarrow_* \varphi \cdot \psi) \rightarrow_* \psi \cdot \neg\neg\psi)$ which was proved in [Cin01].

3.2.2.5 Fuzzy Set Product and Fuzzy Relational Composition

Fuzzy Set Product The product of n fuzzy sets $A_1, A_2 \ldots, A_n$ with each A_i a fuzzy subset of their universe of discourse U_i is defined by means of a t-norm $*$ as the fuzzy set on the universe of discourse $U_1 \times U_2 \times \ldots \times U_n$ with the following membership function:

$$\mu_{A_1,A_2\ldots,A_n} = *(\mu_{A_1}, \mu_{A_2}, \ldots, \mu_{A_n})$$

Fuzzy Relational Composition For crisp relations $S \subset X \times Y$ and $R \subset Y \times Z$, composition is defined by

$$(x,z) \in S \circ R \iff \exists y \in Y | (x,y) \in S \wedge (y,z) \in R$$

This definition is generalised to fuzzy sets by changing $\wedge$ to a t-norm $*$ and $\exists$ to the supremum:

$$\mu_{S \circ R}(x,z) = sup_{y \in Y} * (\mu_S(x,y), \mu_R(y,z))$$

3.2.2.6 Linguistic Hedges

The above many-valued logics can be considered as fuzzy logic in the narrow sense which provides a formal background to reasoning with a graded truth. Fuzzy logic in the broader sense concerns approximate reasoning techniques for drawing imprecise conclusions from imprecise antecedents.

From this viewpoint, fuzzy set modifiers or hedges play an important role in the interpretation of human-originated description of concepts as they allow modifying the semantics of set membership functions by linguistic terms such as *very*, *fairly* and *more or less*. The following are the most important categories that can be distinguished within fuzzy set modifiers (an overview is given in [KC99]):

Concentration $\mu_{concentration(A)} = (\mu_A)^p$ where $p > 1$. Often, one takes $p = 2$.

The membership function of a concentrated set will lie within that of the original fuzzy set. They have a common support and they share the same membership values on the boundaries of the unit interval. Overall, the vagueness of the resulting set is decreased.

An example of a linguistic hedge that can be modelled by concentration is the adverb *very*.

Dilation $\mu_{dilation(A)} = (\mu_A)^p$ where $p \in]0,1[$. A common value for p is $1/2$.

The membership function of a dilated set will lie outside the one of the original fuzzy set. They have a common support and they share the same membership values on the boundaries of the unit interval. Overall, the vagueness of the resulting set is increased.

Somewhat is an example of a linguistic hedge that can be modelled by dilation.

Intensification

$$p > 1, \mu_{intensification(A)}(x) = \begin{cases} 2^{p-1}(\mu_A(x))^p & 0 \leq \mu_A(x) \leq 0.5 \\ 1 - 2^{p-1}(1-\mu_A(x))^p & \text{otherwise} \end{cases}$$

Intensification will increase the vagueness where it was already high and decrease it where it was low.

An example of a linguistic hedge that can be modelled by intensification is for example *indeed*.

Fuzzyfication This fuzzy set operator has the opposite effect from intensification.

$$\mu_{fuzzyfication(A)}(x) = \begin{cases} (\frac{\mu_A(x)}{2})^{\frac{1}{2}} & 0 \leq \mu_A(x) \leq 0.5 \\ 1 - (\frac{1-\mu_A(x)}{2})^{\frac{1}{2}} & \text{otherwise} \end{cases}$$

Translation $\mu_{translation(A)(x)} = \mu_A(x+\alpha)$

Translation provides an alternative way to model for example the adverb *very* with $\alpha = 0.5$, given a fuzzy set with an increasing member function.

Recently, the study of linguistic hedges in L-fuzzy set theory (see the discussion in section 3.2.1.2) has begun in [MDC01].

3.2.3 Fuzzy Process Control

One step further away from fuzzy logic in the narrow sense is approximate reasoning with "fuzzier" inference rules than those of the many-valued logics described so far. This notion has primarily flourished in our next topic of discussion: the domain of fuzzy control systems.

3.2.3.1 Process Control

Fuzzy process control was the first practical application of fuzzy set theory and refers to the modelling of mechanical processes as a collection of simple fuzzy if-then rules with imprecise premises and imprecise conclusions.

It has had many successful commercial applications including air condition regulation, cruise control and even motion detection in video camera's where a distinction needs to be made between moving objects and motion caused by instable cameraman hands. The above examples all require gradual output changes when their input is altered and their complexity often hinders a precise statement of the causal connection between input x and output y values. If it was possible to describe the causal connection between x and y as a function $y = f(x)$, we could use *regular modus ponens* to regulate the process:

premise	$y = f(x)$
fact	$x = x'$
consequence	$y = f(x')$

When the causal relation between the input and the output is only partially or pointwise known, fuzzy process control allows the system to be described as a collection of fuzzy if then-rules with linguistic variables X and Y:

$rule_1$	if X is A_1 then Y is B_1
$rule_2$	if X is A_2 then Y is B_2
...	...
fact	X is A
consequence	Y is B

A typical example of the use of such fuzzy if-then rules is that of controlling the sway of a crane transporting large containers: the experience built up by human crane operators can be translated effortlessly to rules while the it poses many problems from the classical engineering perspective.

3.2.3.2 Fuzzy Control Reasoning System

Designing a fuzzy control system generally consists of the following steps:

Fuzzification This is the basic step in which one has to determine appropriate fuzzy membership functions for the input and output fuzzy sets and specify the individual rules regulating the system.

Inference This step comprises the calculation of output values for each rule even when the premises match only partially with the given input.

Composition The output of the individual rules in the rule base can now be combined into a single conclusion.

Defuzzification The fuzzy conclusion obtained through inference and composition often has to be converted to a crisp value suited for driving the motor of an air conditioning system, for example.

3.2.3.3 Inference for Approximate Reasoning

Zadeh identified [Zad75b] some inference rules common to human-like approximate reasoning for the above scheme:

Entailment Choose for A the intensification *very* as an example.

premise	X is A
fact	$A \subset B$
consequence	X is B

Projection Choose $R(X,Y) = equal(7,4)$ for an example.

premise	X,Y have a relation $R(X,Y)$
consequence	X is $\Pi_X(R)$

premise	X,Y are in a relation $R(X,Y)$
consequence	Y is $\Pi_Y(R)$

Compositional Rule of Inference This is the most important rule defined by Zadeh and can be seen as a generalisation of classical modus ponens which is of practical use in forward inferencing systems for approximate reasoning:

premise	if X is A and Y is B then Z is C
fact	X is A' and Y is B'
consequence	Z is C'

Herein, the fuzzy rule can be seen as the fuzzy relation $A \times B \rightarrow C$. Furthermore C' is a relation composed of a factual matching and an implication:

$$C' = A' \times B' \circ (A \times B \rightarrow C)$$

Which would give with *min* as the t-norm of choice in the fuzzy set product and fuzzy relational composition the following membership function:

$$\mu_{C'} = sup\ min\{min(\mu_{A'}, \mu_{B'}), (min(\mu_A, \mu_B) \rightarrow \mu_C)\}$$

where we still have to choose an implication operator. As performance is important in fuzzy control systems, popular choices are the Mamdani (min) and Larsen (product) implication:

- *General Modus Ponens with Mamdani Implication*

  $$\mu_{C'} = sup\ min\{min(\mu_{A'}, \mu_{B'}), min(min(\mu_A, \mu_B), \mu_C))\} = sup\ min\{\mu_{A'}, \mu_{B'}, \mu_A, \mu_B, \mu_C\}$$

- *General Modus Ponens with Larsen Implication*

  $$\mu_{C'} = sup\ min\{min(\mu_{A'}, \mu_{B'}), (min(\mu_A, \mu_B) \cdot \mu_C)\}$$

3.2.3.4 Combining Individual Rule Results

The overall behaviour of the system is modelled by taking an aggregation of the individual rule results. Usually, union interpreted as *max* is chosen for this task.

3.3 Fuzzy Logic Programming

The axiomatised propositional many-valued logics presented in section 3.2.2.4 incorporated modus ponens as the notion of proof. We will now discuss an entirely different kind of logics, denoted as fuzzy definite clausal logics, which embrace a resolution rule as the notion of proof.
Extending the classical resolution rule to handle weighted clauses initiated a plethora of *"Fuzzy Prolog"* systems which differ on the chosen form of fuzzification: some only support fuzzy facts, while others allow fuzzy predicates and some support fuzzy rules in addition.

The development of a satisfactory resolution rule for the predicate versions of the deductive logics discussed in section 3.2.2.4 is the topic of much ongoing research. An overview is given in the introduction of [Als01].
The generalisation of Herbrand's theorem to these logics, one of the resolution principle's foundations which states that a set of clauses is unsatisfiable if their ground versions are unsatisfiable in propositional logic, has only recently been proved in [NP00].
Another difficulty poses the non-distributiveness of conjunction and disjunction operators which hinders algorithmisation of the refutation procedure.
Therefore, we will only consider *"Fuzzy Prolog"* for our declarative meta-programming system.

3.3.1 Fuzzy Logic Programs

3.3.1.1 Syntax

Generally, a fuzzy program is a set Π of fuzzy clauses each weighted by a real $\vartheta \in]0,1]$:

$$\alpha : \vartheta \leftarrow \beta$$

where the atom α is called the head of the clause and the body β is a conjunction of atoms $\beta_1, \ldots, \beta_n$ with $n \geq 0$.
It should be noted that some systems disallow clauses to be weighted. We will interpret the weight ϑ as the truth degree of the conclusion of the rule given the truth of β.
Furthermore,

- An *atom* $p(t_1, \ldots, t_n)$ consists of an *n*-nary *predicate* symbol *p* and *n terms* t_i.
- A *predicate* symbol starts with a lowercase letter.
- A *term* is either a *variable*, a *constant* or an *n*-ary *functor* $f(t_1, \ldots, t_n)$ consisting of a *function symbol* and *n* terms t_i.
- A *function symbol* starts with a lowercase letter.
- Traditionally a *variable* symbol starts with a capital while a *constant* symbol starts with a lowercase letter. Free variables are considered to be universally quantified.

While atoms and terms might look the same, there's an important difference as atoms can be assigned truth values while terms cannot. A fuzzy fact $\alpha : \vartheta \leftarrow$ or $\alpha : \vartheta.$ is a

fuzzy clause with $n = 0$. A fuzzy query is a clause of the form $? : \vartheta \leftarrow \beta$ where ϑ is allowed to be instantiated.
In the following sections, the semantics and refutation procedure of this *"Fuzzy Prolog"* will be developed analogous to that of crisp Prolog.

3.3.1.2 Model Semantics

Herbrand universe U The Herbrand universe U_Π of program Π is the set of all ground terms (a grounded term doesn't contain any variables) constructed from the functors and constants in the program. An arbitrary constant can be chosen when the program lacks constants.

Herbrand base B_Π A Herbrand base B_Π is the set of ground atoms constructed from Π's predicates with elements of Herbrand Universe U_Π as arguments.

Herbrand interpretation I A Herbrand interpretation I is the tuple $\langle B_{\Pi_I}, \tau_I \rangle$ where $B_{\Pi_I} \subset \beta_\Pi$ is a subset of the Herbrand base and $\tau_I : B_{\Pi_I} \rightarrow]0,1]$ is a truth valuation function. Under this definition, elements of B_{Π_I} are considered to be more or less true.

Herbrand model of a program An interpretation $I = \langle B_{\Pi_I}, \tau_I \rangle$ is a model of a program Π if it is a model of each clause in Π.

Herbrand model of a clause An interpretation $I = \langle B_{\Pi_I}, \tau_I \rangle$ is a model of a clause $\alpha : \vartheta \leftarrow \beta$ if and only if

- $\alpha \in B_{\Pi_I}$ whenever $\forall 0 \leq i \leq n : \beta_i \in B_{\Pi_I}$
  When $n = 0$ we only require $\alpha \in B_{\Pi_I}$.
  This means that each ground atom $\alpha \in B_{\Pi_I}$ is logically entailed by the program.
- $\tau_I(\alpha) =\rightarrow ((\tau_I(\beta_1, \ldots \beta_n), \vartheta)$
  This definition is based on modus ponens (which is used backwards by the resolution refutation):

$$\frac{\begin{array}{l}\beta : b. \\ \alpha : a \leftarrow \beta\end{array}}{\alpha : x}$$

  Given a query $? : x \leftarrow \alpha$ our operational semantics will be to use the above backwards in order to obtain the quantifaction x of its answer:

$$x =\rightarrow (b, a)$$

The least Herbrand model captures a program's meaning as it contains all ground atoms from the Herbrand base which are logically entailed by the program.

Substitution θ A substitution is a mapping of *variables* to *terms*. An example substitution is $\{[X/1]\}$ which substitutes 1 for X. A substitution $\theta = \{[X_1/t_1], \ldots [X_n/t_n]\}$ is applied to a formula φ by simultaneously replacing all variables X_i by the right component t_i of their corresponding equation $[X_i/t_i]$ resulting in a new formula $\varphi\theta$.

Query Answer A substitution θ is a correct answer to a query $? : x \leftarrow \alpha$ with respect to a program Π if and only if every model I of Π ($I \models \Pi$) is also a model of $\alpha\theta$ ($I \models \alpha\theta$).

3.3.1.3 Fix-point Semantics

The least Herbrand model of a program Π is the fix-point of the immediate consequence operator T_Π applied on a starting $I = \langle B_{\Pi_I} = \emptyset, \tau_I(x) = 0\rangle$, that is I is the least Herbrand model if and only if:

$$T_\Pi(I) = I = T_\Pi(\ldots(T_\Pi(T_\Pi(\langle B_{\Pi_I} = \emptyset, \tau_I(x) = 0\rangle)))))$$

where the immediate consequence operator T_Π is defined as:

$$T_\Pi(\langle B_{\Pi_I}, \tau_I\rangle) = \langle B'_{\Pi_I}, \tau'_I\rangle$$

$$\begin{array}{rcl} B'_{\Pi_I} & = & \{\alpha\theta : \theta \text{ is a grounding substitution,} \\ & & \exists\alpha : \vartheta \leftarrow \beta_1, \ldots \beta_n \in \Pi, \forall 1 \leq i \leq n, \beta_i\theta \in B_{\Pi_I} \\ & \text{or} & \exists\alpha : \vartheta.\} \end{array}$$

$$\tau'_I(x) = \begin{cases} max(sup\{\rightarrow (\tau_I(\beta_i), \vartheta) : \alpha \text{ is a clause }\}, sup\{\vartheta : \alpha \text{ is a fact }\}) & x = \alpha\theta \\ \tau_I(x) & \text{otherwise} \end{cases}$$

3.3.1.4 Operational Semantics

We are now ready to define the operational semantics which are equivalent to the semantics we discussed earlier.
An SLD-resolution tree for a program Π and a query goal $? : \vartheta \leftarrow \alpha$ consist of nodes $\langle A, \sigma, \tau\rangle$ where A represents a possibly empty goal set, σ is the set of substitutions applied so far and τ is a truth assigning function.
Then, given a node $\langle a \cup A, \sigma, \tau(x)\rangle$, its successor is defined as:

- $\langle A\theta, \sigma \circ \theta, \tau(a) = \vartheta\rangle$
  iff $a' : \vartheta \leftarrow\in \Pi$ and θ is the most general unifier $MGU(a', a)$
- $\langle (A \cup B)\theta, \sigma \circ \theta, \tau(a) =\rightarrow (\tau(B), \vartheta)\rangle$
  iff $a' : \vartheta \leftarrow B \in \Pi$ and θ is the most general unifier $MGU(a', a)$
- ε
  iff none of the above rules apply

Solutions to the query $? : \vartheta \leftarrow \alpha$ consist of branches in the SLD-resolution tree starting from the root node and ending in the empty goal node containing the correct answer substitutions:

$$\{: \tau(\alpha\sigma) \leftarrow \alpha\sigma | \langle \alpha, \varepsilon, true\rangle \rightarrow^* \langle \emptyset, \sigma, \tau(x)\rangle\}$$

The above inference procedure is a conservative extension of regular SLD-resolution (i.e. their solutions collapse on the boundaries of the unit interval) in which S stands for the *selection rule* which selects from a goal set the goal which will be resolved in the next step (Prolog uses a left-to-right selection rule). L emphasises the *linear* shape of the obtained proof trees while D stands for *definite* clauses.

As the described inference method might give rise to infinite trees (for instance in the advent of recursive predicates), resolution and fuzzy resolution aren't *complete*. This means that not every logical consequence of a Prolog program can be derived using

resolution.

Regular Prolog is *sound* under the condition that unification incorporates an occur check which will be discussed in greater detail in the following section. This means that every conclusion derived by resolution is a logical consequence of the program. Lee [Lee72] was the first to prove the equivalent soundness of fuzzy resolution using only clauses whose truth value lies within the interval $[0.5, 1]$ for inference.

3.3.2 Similarity-Based Unification

Until now, we have mainly concentrated on expressing uncertainty present in the domain knowledge. Approximate reasoning adds to this some fuzziness in the reasoning process itself, possibly by extending the unification procedure to allow partial matches.

3.3.2.1 Classical Unification

The above resolution procedure relies for its resolvents on a function *MGU* which returns the most general unifier between two clause heads if it exists. We will need some additional definitions:

More General Substitution A substitution θ is more general than a substitution σ if and only if their exists a substitution λ for which $\theta\lambda = \sigma$.

Unifier A substitution θ is a unifier for two atoms φ and ψ if and only if $\varphi\theta = \psi\theta$.

Most General Unifier A unifier θ of two atoms φ, ψ is the most general unifier of φ, ψ if and only if it is more general than any other unifier of φ and ψ.

The most general unifier (MGU) of two atoms (starting with the same predicate symbol and of the same arity) $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ can be calculated from the set of equations $S = \{t_1 = s_1, \ldots, t_n = s_n\}$ with the following algorithm:

Choose non-deterministically one equation from S_i and apply one of the following rules depending on the form of the equation until the algorithm fails or no more changes are applied to S_{i+1}:

- $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \triangleright$ *replace equation by the equations* $\{t_i = s_i : \forall 0 < i \leq n\}$
- $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m), f \neq g, m \neq n \triangleright$ *stop with failure*
- $x = x \triangleright$ *remove the equation*
- $t = x, t$ *is not a variable* $\triangleright$ *replace by* $x = t$
- $x = y, x$ *is a variable different from* y *and* x *occurs at least once more in* $S_i \triangleright$ *we perform an occurrence check by failing whenever* X *occurs in* y *and otherwise we replace all occurrences of* x *by* y *in every other equation*

The above algorithm either fails to unify two atoms or returns their most general unifier.

3.3.2.2 Weak Unification

A natural extension of the unification algorithm above allows for two atoms to be unified even when they are syntactically different, but can be considered semantically or syntactically similar up to a certain degree.

Similarity can be expressed by a similarity relation $R : U \times U \rightarrow [0,1]$ which is a binary fuzzy relation defined on the universe U for which the following properties are required to hold (generalisation of the concept of an equivalence relation):

- *reflexive* $\forall x \in U : R(x,x) = 1$
- *symmetric* $\forall x,y \in U : R(x,y) = R(y,x)$
- *transitive* $\forall x,y,z \in U : R(x,z) \geq supR(x,y) * R(y,z)$ where $*$ is a t-norm with the definition of fuzzy relational composition from section 3.2.2.5 in mind.

A similarity relation R between function and predicate symbols is used in [Ses02] to allow approximate inferences when the exact unification procedure fails.
The following algorithm finds a weak λ-unifier for two atoms $p(t_1,\ldots,t_n), q(s_1,\ldots,s_n)$ of equal arity. The similarity relation R associates a unification degree of λ with this unifier.
Choose non-deterministically one equation from S_i and apply one of the following rules depending on the form of the equation until the algorithm fails or no more changes are applied to S_{i+1}:

- $f(t_1,\ldots,t_n) = g(s_1,\ldots,s_n)$ *where* $R(f,g) > 0 \triangleright$ *replace equation by the equations* $\{t_i = s_i : \forall 0 < i \leq n\}$ *and set* $\lambda = min(\lambda, R(f,g))$.
- $f(t_1,\ldots,t_n) = g(s_1,\ldots,s_m)$ *where* $R(f,g) = 0$ *or* $m \neq n \triangleright$ *stop with failure*
- $x = x \triangleright$ *remove the equation*
- $t = x, t$ *is not a variable* $\triangleright$ *replace by* $x = t$
- $x = y, x$ *is a variable different from y and x occurs at least once more in* $S_i \triangleright$ *we perform an occurrence check by failing whenever X occurs in y and otherwise we replace all occurrences of x by y in every other equation*

The unification degree of the most general unifier calculated by this algorithm can be used further on in a "Fuzzy Prolog" system supporting approximate reasoning and reasoning with uncertainty.

3.3.2.3 Fuzzy Unification Based on Edit-Distance

The above weak unification algorithm can only unify the symbols of predicates and functors of the same arity, but does allow a semantic similarity relationship to be used. A different approach [GS00] incorporates the *Levensthein* or *edit distance* to determine the similarity of predicates and terms. This is a purely syntactical distance measure which allows predicates and terms of different arity to be compared.

As the term implies, the *edit distance* is defined on strings as the minimal number of add, replace or delete operations to transform one string into the other. More formally,

the edit distance $e(a.A, b.B)$ of two string $a.A$ and $b.B$ (where a and b denote the first characters of the strings) can be defined recursively in the following manner:

$$e(A,\varepsilon) = e(\varepsilon,A) = |A|$$

$$e(a.A, b.B) = min \left\{ \begin{array}{ll} e(A, b.B) + 1 & , \\ e(a.A, B) + 1 & , \\ e(A,B) & \text{if } a = b \end{array} \right\}$$

where ε denotes the empty string.

The *normalised edit distance* $ne(A,B)$ between two strings is introduced in order to be able to compare the edit distance of two short strings with that of two long strings:

$$ne(A,B) = \frac{e(A,B)}{max(|A|,|B|)}$$

The performance of the straightforward recursive definition of e is unfortunately exponential both in the size of the first string n and in the size of the second string m. Fortunately, the complexity can be reduced to $O(nm)$ using dynamic programming techniques [AJ74].

The similarity relationship associated with the normalised edit distance can be defined as $R(x,y) = 1 - net(A,B)$.

The Fury system [GS00] incorporates a generalisation of string edit distance to Prolog predicates and functors in the interest of fuzzy unification. Although predicates can be assigned truth values while functors cannot, there's no syntactic difference between the two and they are thus treated alike by the distance relationship.
The following is a recursive definition of the fuzzy unification *et* between $f(t_1,\ldots,t_n)$ and $g(s_1,\ldots,s_n)$ [GS00]:

- $et(t,\varepsilon) = \langle size(t), [], size(t)\rangle$ Omitting a term is allowed with a penalty equal to the size of the term.
- $et(x,y) = \langle 0, [x/y], 0\rangle$ given $var(x), var(y)$ Same as normal unification.
- $et(x,t) = \langle 0, [x/t], 0\rangle$ given $nonvar(t)$ and x doesn't occur in t Same as normal unification with occurs check.
- $et(f,g) = \langle e(f,g), [], max(|f|,|g|)\rangle$ Atomic symbols match with a penalty equal to their edit distance. The size of this comparison equals the maximum symbol length.
- $et(f(t_1,\ldots,t_n), g(s_1,\ldots,s_n)) =$
  $et(f,g) \oplus min \left\{ \begin{array}{l} et((t_2,\ldots,t_n),(s_1,\ldots,s_n)) \oplus et(t_1,\varepsilon), \\ et((t_1,\ldots,t_n),(s_2,\ldots,s_n)) \oplus et(s_1,\varepsilon), \\ et((t_2,\ldots,t_n),(s_2,\ldots,s_n)) \oplus et(t_1,s_1) \end{array} \right\}$
  where $\langle p,s,n\rangle \oplus \langle p',s',n'\rangle = \langle p+p', s\cdot s', n+n'\rangle$.

  This definition is a cruder version of the edit distance for strings in that entire arguments must be omitted instead of just single string characters.

Its result is a triple consisting of the *penalty* we have to pay for the unification of the two expressions and denotes the number of mismatches. The second entity contains the *unifier* calculated by the procedure while the last entity denotes the *maximum size* of pairwise node comparisons along the traversal. It will be used as a normalising factor for the penalty.

The size of an expression is defined as follows:

$$\begin{aligned} size(\varepsilon) &= 0 \\ size(f) &= |f| \\ size(f(t_1,\ldots,t_n)) &= |f| + \sum_{i=1}^{n} size(t_i) \end{aligned}$$

The normalised edit distance over Prolog trees is then defined as the pair

$$net(f(t_1,\ldots,t_n), g(s_1,\ldots,s_n)) = \langle \frac{p}{n}, s \rangle$$

where

$$et(f(t_1,\ldots,t_n), g(s_1,\ldots,s_n)) = \langle p, s, n \rangle$$

As an example $net(inseticide(baygon), insecticide(baygon, b9)) = \langle \frac{3}{19}, [] \rangle$.

When the result of $net(\alpha,\beta) = \langle 0, s \rangle$, s equals the classical $MGU(\alpha,\beta)$.

3.3.2.4 Alternative Fuzzy Unification Methods

As always with fuzzy systems, many other methods have been proposed to introduce similarity-based unification. The above were prototypical examples of unification based on a semantical similarity measure (Sessa's method) and a purely syntactic similarity measure (edit-distance based unification). In this section we will briefly sketch two alternative approaches.

a) Likelog The Likelog logic programming language proposed in [AF99] performs a unification even in cases where a classical unification method would fail by associating a set of constants that would make the unification successful if they were considered similar.

This set is called a *cloud* and consists of elements considered pairwise similar under a similarity relation R. The *codiameter* $\mu(X)$ of a cloud X represents the degree up to which the cloud can be considered as a singleton and is defined as

$$\mu(X) = infimum_{x,y \in X} R(x,y)$$

where $\mu(\emptyset) = 1$.

The crispness degree $\xi(Z)$ of a *system of clouds*, simply a finite set Z of clouds, is defined as $\xi(Z) = infimum_{X \in Z} \mu(X)$. A system of clouds is called *compact* if none of its subsets overlap (i.e. their intersection is non-empty).

The notion of a substitution is then extended from a mapping of variables to constants to a mapping of variables to clouds of constants. A substitution θ is called a λ-substitution if for every substituted variable, the codiameter of the cloud it is mapped to exceeds λ. When unifying atoms $\alpha = f(t_1,\ldots,t_n)$ and $\beta = g(s_1,\ldots,s_n)$ from an equation set S containing equations of the form $\alpha = \beta$, the unification degree of S under a substitution θ is defined as

$$infimum_{\alpha=\beta\in S} R(\theta(\alpha),\theta(\beta))$$

The unifier $U(S)$ of a set of equations S is defined as the substitution with the largest unification degree.

This is calculated by the unification algorithm by transforming each equation $\alpha = \beta$ into two systems of clouds $\{f,g\}$ and $\{t_1 \sqcup s_1,\ldots,t_n \sqcup s_n\}$ where $t \sqcup s$ is a union operator extended to clouds:

$$t \sqcup s = \begin{cases} t \cup s & \text{if } t,s \text{ are clouds} \\ \{t\} \cup s & \text{if } t \text{ is a variable and } s \text{ is a cloud} \\ \{s\} \cup t & \text{if } s \text{ is a variable and } t \text{ is a cloud} \\ \{t,s\} & \text{if } t,s \text{ are variables} \end{cases}$$

A substitution θ_Z on a system of clouds $Z = \{M_1,,M_n\}$ is then defined in the following way (where C is the collection of ordinary constants):

- $\theta_Z(x) = x$ if $x \notin M_1 \cup \ldots \cup M_n$
- $\theta_Z(x) = M_i \cap C$ if $x \in M_i$ and $M_i \cap C \neq \emptyset$
- $\theta_Z(x) = y$ if $x \in M_i, M_i \cap C = \emptyset$ and y is a variable in M_i

Although the unifiers in the Likelog systems are similar to those discussed in section 3.3.2.2, the algorithm itself differs in the use of the codiameter of clouds for the unification degree. It also keeps track of these clouds so the end-user is made aware of the similarities under which the unification and thus also a logical derivation holds.

An example of these clouds is given in [AF99] from which the following program is an extract:

```
horror(drakula).
best_seller(Book) :-
    interesting(Book),
    recent(Book).
recent(Book) :-
    published(Book, Y),
    Y > 1997.
```

As `interesting` is left undefined, no answer to the query `?- best_seller(X)` can be given using classical unification. Under the fuzzy unification algorithm, we can however obtain an answer to this query using a similarity relation R which equals the identity relation for constants and defines the predicates *horror* and *interesting* to be similar up to a degree of $0.7 : R(horror, interesting) = 0.7$.

The answer consists of a tuple containing the variable substitutions $\{[X/dracula]\}$ and the cloud $\{horror, interesting\}$ which allows us to consider dracula a best seller under the conditions that horror and interesting are assumed similar. The codiameter of this cloud, in this case 0.7, determines the confidence degree of the answer substitution.

b) Fuzzy Unification of Fuzzy Constants Another variant of the unification algorithm is proposed in [TL98] where normalised, trapezoidal fuzzy sets denoted by a linguistic term and linguistic variables (defined in the same way as in section 3.2.1.4) are added as new kinds of terms in a logic programming language. To support the unification of fuzzy sets A and B, a similarity relation $SD(A,B)$ between fuzzy sets A and B with cores A' and B' respectively (see section 3.2.1.4) is defined as the mean value of the membership function of the fuzzy set B on the interval $[min(A'), max(A')]$. It gives a measure of the inclusion degree of the set A' into the set B'.

Also, a distinction between general and specific knowledge is made: the former consists of terms occurring in the facts of the program while the latter consists of the terms occurring in the heads of the program's rules. The truth value of `buy(a)` is intuitively higher in the second program than in the first program as the concept `about_27` (from the specific knowledge of the second program) is included in the concept of `between_25_and_30` (from the general knowledge of the second program) while the reverse isn't true.

```
price(a, between_25_and_30) : 1.
buy(x) : 0.7 :- price(x, about_27).

price(a, about_27) : 1.
buy(x) : 0.7 :- price(x, between_25_30).
```

More details on the (quite complex) unification of linguistic variables and fuzzy constants based on this idea can be found in [TL98] from which the above examples were taken.

3.3.3 A Mini-Survey of Fuzzy Logic Programming Systems

The previous section described a generalisation of the many forms of fuzzy resolution and unification one might find in a fuzzy Prolog system. As we will now briefly describe some of the existing systems together with their peculiarities, it will become clear that there are many ways to fuzzify a logic programming language and that there is no general agreement on what is the best way to do it. We can only try to develop a language which suits the application domain best.

3.3.3.1 Fuzzy Ciao Prolog

We will begin our discussion with the Fuzzy Prolog proposed in [VGoH02] since it differs from the others in at least three important ways:

- Truth values aren't modelled as reals in the unit interval, but as a finite union of sub-intervals on the unit interval. It can be seen as a response to the objections against an overly precise membership function of fuzzy sets (since one exact number must be assigned to each element) or an overly precise degree of truth in fuzzy logic.
- These intervals are represented as constraints on $\Re$ which can be solved efficiently by the Ciao Prolog constraint logic programming system. This approach doesn't alter the regular inference system of Ciao Prolog, but uses its original constraint solving capabilities to model partial truth.

- Rules can't be assigned a degree of truth so their degree of truth is determined entirely by the conjunction that makes up their body. In order to generalise among the many possible truth valuations of a conjunction, aggregator operators are introduced which subsume triangular norms for conjunction and triangular co-norms for disjunction. The user can thus choose on a rule-per-rule basis whether he would like to use the minimum or the product for the truth value of the rule body.

The following example directly expresses the fact that a certain plush `bear` is soft up to a degree of truth between 0.8 and 0.9 and that the *min* operator is to be used in order to determine the truth of the `loved_by_children` predicate:

```
soft(bear, C) :~
   C .>=. 0.8,
   C .=<. 0.9.

loved_by_children(X, C) :~ min
   soft(X, C1),
   nice_color(X, C2).
```

At run-time the rule `loved_by_children` is expanded to a regular Ciao Prolog rule where `minim` is an auxiliary predicate calculating the minimum of a list of values:

```
loved_by_children(X, C) :-
    soft(X,C1),
    nice_color(X, C2),
    minim([C1, C2], C),
    C .>=. 0, .=<. 1.
```

3.3.3.2 More Conventional Systems and Their Extensions

Lee's paper on the fuzzy resolution rule [Lee72] gave rise to a myriad of fuzzy logic programming languages, of which the *Prolog-ELF* system [IK85] was the first. All of these languages use variants of the fuzzy resolution rule discussed in section 3.3.1.4.

A possible extension is to only consider candidate clauses for inference if their truth degree exceeds a user-specified cut-off value. In some cases this cut-off value can be specified on a rule-per-rule basis. When this cut-off value is fixed to 0.7, Lee's original soundness result holds.
Another natural extension collects all possible solutions to a query and only returns those with the highest truth value.
Other extensions allow fuzzy numbers as truth degrees for facts and rules.

The system closest to the general resolution rule is the *f-Prolog* system [LL89, LL90] where minimum is used to compute the truth value of the body of a rule which is then multiplied by the weight associated with the head of the rule.

While most of these systems use min-max and product logic, the system described in [KK94] is based on Łuckasiewicz logic instead.

Some basic optimisation strategies for a fuzzy Prolog which only allows weighted facts are described in [FSCdS94]. Solutions of subqueries can be cached and reused in subsequent proofs while a form of α-β pruning known from game trees is adapted for the calculation of truth values. In effect there's no use considering truth values of alternative solutions in a disjunction when it is already known that a certain value is the maximum. Off course, in a more general setting, the goals themselves still need to be verified.

Systems incorporating some form of fuzzy unification were already presented in the previous section.

3.4 Conclusion

In this chapter, we have detailed the theoretical background on approximate reasoning. We began our discussion with the cornerstone of existing techniques for modelling vague concepts: fuzzy sets which are based on a natural extension of the set membership function to allow gradual membership. We provided the reader with common terminology and extensive background on fuzzy set-theoretical and the corresponding logical operations.

Fuzzy process control was exemplified as the most popular and successful application of fuzzy set theory.

In sight of possible applications in declarative meta-programming, we broadened our horizon with a detailed account on existing fuzzy logic programming languages in which the ordinary resolution procedure is extended to incorporate partial truths. We also explored the theoretical possibilities for extending the unification procedure based on semantical or syntactical similarity measures.

We concluded the chapter with a mini-survey on existing fuzzy logic programming languages.

Chapter 4

Extending SOUL's Declarative Framework

In chapter 2 we have explored the various ways in which declarative meta-programming can aid in the software re-engineering process by providing reasoning facilities for analysing the structure of a program's source code as well as its run-time behaviour. However, none of these approaches have language support for approximate reasoning while, as we will see in chapter 5, the ability to express vague concepts and tolerance for near-perfect solutions can increase the expressiveness of pattern detection rules and even solve some commonly occurring problems in declarative meta-programming.

This chapter will detail the two pillars of the approach we developed: an extension of SOUL's base logic programming language for approximate reasoning and a library for reasoning about a program's behaviour using dynamic analysis which complements the existent library for structural program analysis called LiCOR.

4.1 Library for Dynamic Program Analysis

Following SOUL's organisation of the the LiCoR library for analysing a program's structure by its source code (see section 2.2.1.5), we have imposed an equivalent layered structure on our library for analysing a program's behaviour from execution traces. Each layer from this structure depends on the layers underneath.

4.1.1 Logic layer

As this layer is in common with SOUL's LiCOR library for reasoning about source code, we refer to section 2.2.1 for more information. To summarise, the lowest-level logic programming constructs normally provided by the libraries of a logic programming language are implemented in this layer.

4.1.2 Representational Layer

As we have seen in section 2.1.3, traces of a program's execution can be modelled as an ordered set of run-time events. To support a declarative analysis of such execu-

tion traces, the representational layer defines logic predicates which reify the dynamic events from an execution trace to the logic meta-level.
In order to allow a *post-mortem* as well as an *ad-hoc* analysis (these two variants of dynamic analysis were discussed in 2.1.3), different definitions of these predicates are provided for each variant.
As such, they describe the same source model (an execution trace) using the same meta-model (the predicates from the representational layer), but their definition differs in the way they extract the information for the source model from the application. In the case of a *post-mortem* analysis, each event from the execution trace is transcribed as an individual logic fact while in the case of an *ad-hoc* analysis, the same predicates will be implemented as logic rules exploiting the symbiosis of SOUL with Smalltalk to extract dynamic information from a running program on a need-to-know basis.

4.1.2.1 Source Model

We will analyse a program's behaviour using execution traces modelled as an ordered collection of run-time events.
Three types of run-time events are distinguished:

Method invocations This type of event occurs whenever a method is invoked. The instance sending the message, the receiving object, the method's selector and the method's arguments are recorded.

Variable assignments This type of event occurs whenever a value is assigned to a variable. The registered values are the object containing the variable, the method invocation in which the assignment happened, the assigned value and the name of the altered variable.

Method exits This type of event occurs whenever control returns from a method. We record the method invocation we exit from and also the returned value.

Our model is thus rather fine-grained: it allows data flow information to be recorded in addition to control flow information. In contrast to the simple coarse-grained model deployed in Richner's analysis tool (discussed in section 2.2.2), it keeps track of the arguments passed to methods and the values they return. Not unlike the extremely fine-grained model deployed in the Caffeine tool by Guéhéneuc, Douence and Jussien (discussed in section 2.2.3.3), we also log all variable assignments. This allows us to keep track of the ever changing object states. Since Smalltalk handles classes as objects and class instantiations as messages sent to these class objects, we however don't need separate events for class instantiations. The additional data flow information allows more complex behavioural patterns to be detected and associations between classes to be categorised.

Another difference with Richner's source model is that we only allow entire classes to be traced instead of individual methods. Furthermore, we don't make the distinction between indirect (i.e. a traced method is called from within an untraced method) and direct (i.e. a traced method is called from within another traced method) method invocations.

In general, we should be aware that while fine-grained models allow a great level of detail in an application's control and data flow, they also produce large amounts of

Listing 4.1: Source code extract of the `Test` class.

```
Test class>>new
  ^(super new) initialize

Test>>initialize
  treeRoot := Test2 new

Test>>foo
  | bar |
  Transcript show: 'text'.
  bar := 'another text'.
  ^bar
```

information. The sheer volume of run-time events gathered may limit the practical applicability of a post-mortem analysis, but as a lightweight ad-hoc dynamic analysis variant may spot possibly interesting classes, the size of the execution history can be kept to a minimum by only tracing those candidate classes.
Therefore, we include both variants of dynamic analysis in contrast to Richner's tool which relies solely on the post-mortem analysis variant and the caffeine tool which deploys an ad-hoc dynamic analysis.

In the context of software re-engineering, execution traces are obtained by executing a scenario consisting of user actions or message sends specifically chosen in such a way that only run-time information relevant to the examination of a particular part of the program's behaviour is generated. In addition, we aren't required to trace the entire program, but can limit our investigation to those classes we are interested in.

We implemented a `Trace` class that turns tracing on for classes given as arguments to its `trace:` method. This is the same for ad-hoc and post-mortem analysis up to an additional `adhoc` notification that has to be sent to the `Trace` class before beginning the execution of a scenario.

To illustrate, we will generate an execution trace of the `Test` class shown in listing 4.1. This is the only class we will be tracing so no information about the behaviour of other classes will be included in the resulting execution trace.
An extended sequence diagram mimicking the execution trace of a message `foo` sent to a newly created instance of `Test` (i.e. the scenario consists of the statement '`Test new foo`') is shown in figure 4.1.

As execution traces may change drastically with different user input, determining a suitable scenario is crucial to the successful investigation of a particular part of program's behaviour. They should focus and limit the execution trace to information that is necessary to detect the requested patterns and shun the generation of excess data as this will significantly slow down the detection process. Ideally, execution scenarios invoke deterministic behaviour; the same program with the same scenario and input should produce the same run-time information. Unit tests are often very suited for this purpose as they should test one particular program feature in a deterministic way.

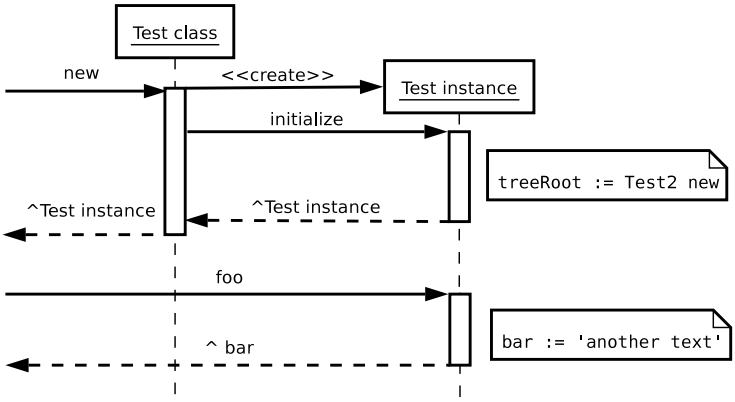

Figure 4.1: Example of a source model represented as a sequence diagram

We generate the run-time events during a program's execution by transforming its source code in such a way that the previously mentioned `Trace` class is notified of each event. This process is commonly called *instrumentation* and our particular implementation is detailed in appendix A.

4.1.2.2 Meta-Model

The events which comprise the execution trace (our source model) are modelled using the Smalltalk class hierarchy depicted in figure 4.2. The `TraceEvent` class forms the root of this hierarchical tree and has a subclass for every type of run-time event: `MethodEntryEvent`, `MethodExitEvent` and `VariableAssignmentEvent`. Each run-time event is provided with a `sequenceNumber` instance variable which denotes its chronological ordering in the execution trace.

Since we are interested in performing a declarative analysis of a program's behaviour, every run-time event can be transcribed to SOUL or Prolog facts using their `toFactOn:` and `toPrologFactOn:` methods respectively. The following predicates form the declarative meta-model used to describe execution traces with:

Method invocation

```
methodEntry(?sequenceNumber,
            ?sendingInstance,
            ?receivingInstance,
            ?receivedSelector,
            ?receivedArguments)
```

Variable assignment

```
assignment(?sequenceNumber,
           ?methodInvocationNumber,
           ?instance,
           ?variable,
```

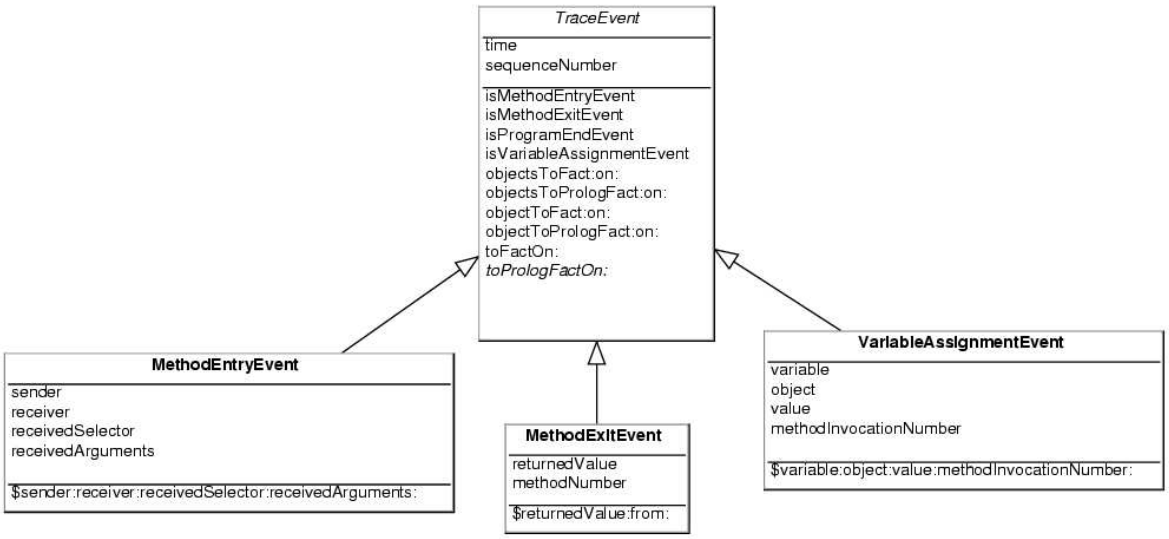

Figure 4.2: UML diagram of the run-time events class hierarchy

```
            ?value)
```

Method exit

```
methodExit(?sequenceNumber,
           ?methodInvocationNumber,
           ?returnedValue)
```

As an illustration the execution trace depicted in figure 4.1 is thus transcribed to the following set of SOUL facts with each object mapped to a unique identifying integer:

```
methodEntry(1, 0, 1, [#new], <>)
  methodEntry(2, 1, 2, [#initialize], <>)
    assignment(3, 2, 2, treeRoot, 0)
  methodExit(4, 2, 2)
methodExit(5, 1, 2)
methodEntry(6, 0, 2, [#foo], <>)
  assignment(7, 6, 2, bar, 3)
methodExit(8, 6, 3)
```

4.1.2.3 Reifying the Source Model

The reification process varies with the form of dynamic analysis that is deployed.

Mapping Objects to Unique Identifiers In an *ad-hoc analysis*, the symbiosis between Smalltalk and SOUL can be exploited to use the actual Smalltalk objects involved in the execution trace as arguments of the above predicates. We could do the same with the *post-mortem* analysis variant, but the equivalent transcription to Prolog

forces us to map each object to a unique object identifying integer. During a post-mortem analysis within SOUL, it is however possible to retrieve the object corresponding to a given integer using the `objectMap(?ObjectNumber,?Object)` predicate, but when using this functionality it is important to bear in mind that the retrieved object has already been manipulated during the entire execution trace and that its values no longer represent the values it had during the run-time event it was involved in at a particular time.

Post-mortem Analysis Under *post-mortem analysis*, the process consists of instrumenting the classes under investigation followed by an execution of a well-determined scenario that focuses the execution trace on interesting object interactions and behaviour. The events generated during the execution of the traced classes are gathered into a knowledge base of logic facts after which the declarative analysis can start.

In other words: we do not start reasoning until the application has ended. This allows advanced (i.e. we can consider multiple candidate solutions when looking for a particular run-time event in the execution history through backtracking) reasoning patterns over the entire run-time behaviour of a program. It however also means that large amounts of data need to be stored and processed which might be overkill and even slash the performance of very simple rules.

Ad-hoc Analysis In the alternative *ad-hoc analysis* variant, the declarative reasoning process runs as a coroutine alongside the application. The execution of the application is interleaved with the evaluation of the logic program which can suspend and resume the execution of the former.

We declaratively request a particular execution event and the execution of the application that is being analysed is continued until this event is encountered. Control then returns to the reasoning process where we analyse the current event and make a request for the following event.

As a consequence, we have at no time during the evaluation of our query a complete execution history to our disposal. The applicability of this approach is thus limited to lightweight rules: since the execution of the program is advanced whenever we backtrack upon a choice for a run-time event we can't access past events in this way and cannot consider alternatives for an event that is requested to prove the body of a rule without advancing the application.

To interweave the evaluation of the application and the logic program, SOUL and the program under execution need to be started in separate processes. We can then query for the next execution event by simply launching a goal with one of the predicates from the meta-model. These predicates are no longer facts but full-grown rules that suspend and resume the application's execution. By using the same predicates for ad-hoc and post-mortem dynamic analysis, we ensure a maximal reuse of common rules.

The following rule illustrates how the `methodEntry` predicate used to model method invocation events now steers the application's execution. Also note how the rule's arguments now contain the actual Smalltalk objects involved in the run-time event:

```
methodEntry(?sequenceNumber,?sendingInstance,
```

```
            ?receivingInstance,?receivedSelector,
            ?receivedArguments) if
        event(?event),
        [?event isMethodEntryEvent],
        equals(?sequenceNumber,[?event sequenceNumber]),
        equals(?sendingInstance,[?event sender]),
        equals(?receivingInstance,[?event receiver]),
        equals(?receivedSelector,[?event receivedSelector]),
        listAsCollection(?receivedArguments,[?event receivedArguments])
```

The rule first asks for a new run-time `event` which must be of the `methodEntryEvent` type. If it is not, the system will simply backtrack and ask for another `event` until one of the correct type is encountered. The events themselves are instances from the classes depicted in figure 4.2 to which we can send regular Smalltalk messages. We use this property in the above rule to dissect an event into its `sequenceNumber`, `sendingInstance`, `receivingInstance`, `receivedSelector` and `receivedArguments` components.

Because the execution of the analysed application is paused when a suitable event is encountered, the information in this event is always accurate and up-to-date. We therefore do not have to map the objects participating in the event to unique integer identifiers and can instead use the actual objects themselves. This is a major advantage over post-mortem traces as we can now access and manipulate the objects in the analysed program during its execution with declarative meta-programming which is of use in advanced debugging sessions.

The application's execution is interweaved with the evaluation of the logic program using the threefold `event` predicate:

```
event(?e) if
        equals(?e,[Tracing.Trace current event]),
        [Tracing.Trace current event isProgramEndEvent],
        !

event(?e) if
        not([Tracing.Trace current event isProgramEndEvent]),
        equals(?e,[Tracing.Trace current event])

event(?e) if
        not([Tracing.Trace current event isProgramEndEvent]),
        [ Tracing.Trace current semaphore signal.
        Tracing.Trace current semaphore wait.
          true],
        event(?e)
```

The first part of the rule defines the obvious case in which the program has ended and no more solutions can be found to this rule. The second part simply unifies the argument of the predicate with the last encountered execution event. When we ask for more solutions to this rule, the third part is invoked which signals a semaphore shared by the application process and SOUL's process. This signal results in the execution of the application being resumed. We also put our own reasoning process to sleep by sending

the wait message to the shared semaphore. When a run-time event is encountered, the exact opposite will happen at the application's side: it will signal the semaphore which results in the reasoning process to be resumed and put itself to sleep again. The entire process is depicted schematically in figure 4.3.

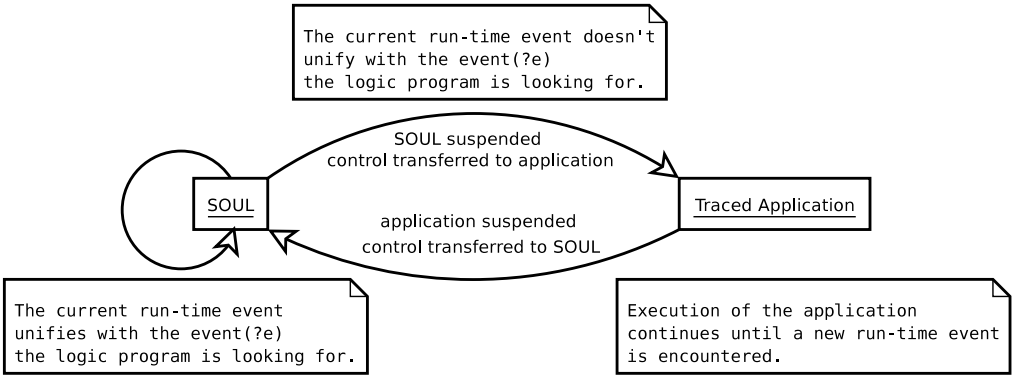

Figure 4.3: Schematic representation of the ad-hoc analysis process.

4.1.2.4 Simple Queries over the Representational Layer

In this section, we will show how the predicates in the representational layer can be used to construct some simple analysis rules and also illustrate the practical differences between post-mortem and ad-hoc analysis. More commonly used rules defined in the basic layer will be detailed later on.

a) Post-mortem Analysis As a simple, yet convincing application of post-mortem dynamic analysis, we will show how instances of the double dispatching pattern can be detected.

The essence of this well-known pattern is depicted in figure 4.4 and its instances are detected –solely relying on dynamic analysis– by the following rule which is satisfied whenever an `?invoker` instance starts a double dispatching method invocation sequence between two objects, `?primary` and `?secondary`, where `?primarySelector` and `?secondarySelector` are the selectors of the first method invocation and the second method invocation (invoked within the call stack of the first method invocation) respectively. We also provide information about the ordering of the method invocations in the execution history through the sequence numbers `?primarySN` and `?primaryExit`.

The implementation is a straightforward translation of the aforementioned sequence diagram:

```
?invoker doubleDispatchesOn: ?primary
         selector: ?primarySelector
         at: ?primarySN
         andOn: ?secondary
         selector: ?secondarySelector
         at: ?secondaryExit if
```

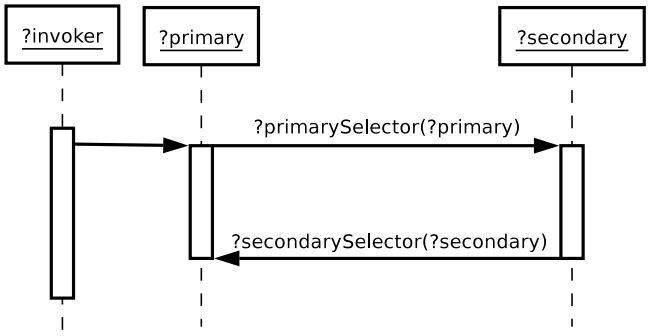

Figure 4.4: Sequence diagram of double dispatching patterns detected by the `?invoker doubleDispatchesOn: ?primary selector: ?primarySelector at: ?primarySN andOn: ?secondary selector: ?secondarySelector at: ?secondarySN` rule.

```
methodEntry(?primarySN,?invoker,?primary,?primarySelector,?primArguments),
methodExit(?primaryExit, ?primarySN, ?),
member(?secondary,?primArguments),
methodEntry(?secondarySN,?primary,?secondary,?secondarySelector,?secArguments),
methodExit(?secondaryExit, ?secondarySN, ?),
member(?primary,?secArguments),
greater(?primaryExit, ?secondaryExit)
```

The application of the rule, in which we map in addition each object identifier to the corresponding Smalltalk object, on the execution trace of the heavily on double dispatching relying Visitor design pattern results in the following answers:

```
[?invoker-->[a Tracing.DDTestInvoker],
 ?primarySN-->[20],
 ?primary-->[a Tracing.DDTest1],
 ?primaryExit-->[30],
 ?secondarySelector-->[#visitDDTest1:],
 ?primarySelector-->[#accept:],
 ?secondary-->[a Tracing.DDVisitor]]

[?invoker-->[a Tracing.DDTest1],
 ?primarySN-->[21],
 ?primary-->[a Tracing.DDTest2],
 ?primaryExit-->[27],
 ?secondarySelector-->[#visitDDTest2:],
 ?primarySelector-->[#accept:],
 ?secondary-->[a Tracing.DDVisitor]]

[?invoker-->[a Tracing.DDTest2],
 ?primarySN-->[22],
 ?primary-->[a Tracing.DDTest3],
 ?primaryExit-->[24],
```

```
?secondarySelector-->[#visitDDTest3:],
?primarySelector-->[#accept:],
?secondary-->[a Tracing.DDVisitor]]
```

This result shows that there exists a double dispatching pattern starting at sequencenumber 20 and ending at sequencenumber 30 which encompasses a second double dispatching pattern starting at 21 and ending at sequencenumber 27 which in turn encloses a final double dispatching pattern starting at 22 and ending at sequencenumber 24.

This answer collection is complete; all double dispatching instances in the execution trace are detected using the above rule under post-mortem analysis. In the next section we will see that this rule is incomplete under ad-hoc analysis which is the main difference between the two dynamic analysis variants.

b) Ad-hoc Analysis When we try to run the double dispatching rule under ad-hoc analysis, we will only detect the first instance of the double dispatching pattern which starts at the method invocation with sequence number 20 and ends with the method invocation at sequence number 29.
To understand why, it helps to keep in mind that the execution of the application is continued until a matching run-time event is encountered. As such, when we satisfy the first `methodEntry` subgoal, its variables get bound which causes the execution of the program to continue until a new `methodEntry` event is encountered that satisfies the existing variable bindings. Therefore, the intermediate double dispatching patterns will never be detected since the execution of the application has already advanced beyond their starting point.

This might seem as a major disadvantage of ad-hoc analysis, but its lightweight nature makes the partial results obtained through a preliminary ad-hoc analysis perfect to limit the size of the execution histories that need to be considered by a post-mortem analysis with. This can for instance be obtained by selectively tracing only those classes that occur in the results of the preliminary analysis.

In general, ad-hoc analysis is unsuited for rules in which we have to backtrack over previous choices for run-time events. A technical solution to this problem comprises using a cache of past events with whose content we could try to satisfy our requests for run-time events before advancing the application to obtain a new event. Such an ad-hoc analysis extended with a cache could be seen as a windowed post-mortem analysis.

An ordinary ad-hoc analysis does however suffice for linear queries which don't require backtracking over past events. In such cases, a full-blown post-mortem analysis is often too costly when large execution traces are involved. Therefore, we will now focus on some possible applications of ad-hoc analysis:

Counting single argument message sends We begin with an example of a simple dynamic rule `allSingleArgumentMessagesToClass(?class, ?messages)` which returns a list `?messages` containing every single-argument message call received by instances of `?class` in the form of the received argument and selector together with the ordering of the invocation event in the execution history.

It is defined as a findall over all `methodEntry` events where the receiver is an instance of the given `class` and the argument list contains only one element:

```
allSingleArgumentMessagesToClass(?class,?messages) if
  findall(<?sn,?selector,?arg>,
          and(methodEntry(?sn,?,?receiver,?selector,<?arg>),
              instanceOf(?receiver,?class)),
              ?messages)
```

The following is the result of the query

```
allSingleArgumentMessagesToClass([Tracing.DDVisitor],?arguments)
```

on an execution trace of a typical Visitor Design Pattern:

```
[?arguments-->< <[23],[#visitDDTest3:],[a Tracing.DDTest3]>,
                <[26],[#visitDDTest2:],[a Tracing.DDTest2]>,
                <[29],[#visitDDTest1:],[a Tracing.DDTest1]>>]
```

Among others we observe that a `visitDDTest3:` message was sent at time 23 with an instance of the `Tracing.DDTest3` class as its single argument.

While the above example can be executed in a post-mortem analysis setting as well, this rule doesn't require all `methodEntry` events of the execution history to be stored. Therefore, the ad-hoc analysis variant avoids in this case an unnecessary memory overhead by interleaving the execution of the application and the evaluation of the logic query.

Declaratively altering program behaviour In the software re-engineering process, the detection of patterns such as design deficits in a program's source code is often followed by a phase in which these errors are corrected by transforming or refactoring its source code. The dynamic equivalent of modifying a program's source code boils down to changing its run-time behaviour. The ad-hoc dynamic analysis variant provides a simple mechanism to do just that. The example in itself is rather artificial but is illustrates that declarative analysis of a program's behaviour supports all phases of the software re-engineering process. In practice, software engineers alter the actual source code of a program to overcome behavioural deficits that were detected during its execution.

Imagine an already existing e-commerce application and that we want to alter its behaviour so that a discount is awarded on the price of a purchase when the user has already made 3 of them. We can achieve this goal using declarative meta-programming and ad-hoc event analysis:

```
offerDiscountAfterThree if
  methodEvent(?,?,?receivingInstance,calculatePrice,?),
  equals(?time,[?receivingInstance time]),
  greater(?time,2),
  [?receivingInstance price: (?receivingInstance price - 10).
   Transcript show: 'Discount awarded'; cr.
   true],
  offerDiscountAfterThree
```

The above rule lets the application execute normally until an invocation of the `calculatePrice` message is intercepted. In that case we verify whether the user already bought more than 2 items. If so, we subtract 10 € from the currently calculated price and show the message "discount awarded" on the transcript. If not, the rule backtracks and the application continues its normal execution.

This results in the following output on the transcript:

```
Time = 1
Price you have to pay = 100
Time = 2
Price you have to pay = 100
Time = 3
Price you have to pay = 100
Discount awarded
Time = 4
Price you have to pay = 90
```

We can thus conclude that ad-hoc declarative event analysis can be used to *alter* an application's behaviour. It turns out the above example is actually quite close to Event-Based Aspect-Oriented Programming (EAOP) in which aspects are triggered by observing program execution events. This approach to Aspect-Oriented Programming is presented in the work of Douence, Motelet, and Südholt [DMS01].

Declarative debugging Another possible use of straightforward ad-hoc analysis queries is declarative debugging. An erroneous program can be executed until a rule which declaratively describes the incorrect behaviour is satisfied. From then on the declarative reasoning process can be used to inspect and continue the program's execution step-by-step. Seen this way, ad-hoc analysis can be used to specify flexible and dynamic breakpoints.

4.1.3 Basic Layer

The basic layer contains auxiliary predicates commonly used in the design layer. They provide a reasonable level of abstraction over the representational layer and also implement low-level dynamic analysis functionality commonly used in the design layer. It contains among others accessing predicates providing abstraction over the dynamic events in an execution trace and functionality for tracking variable assignments, object instantiations and predicates for analysing binary class relations.
We will continue our discussion with some of the most important predicates from the basic layer.

4.1.3.1 Object Instantiation

When trying to understand a program's behaviour, identifying when a particular instance of a class was created is an interesting problem. We can get an answer to this question using the `?instance isInstanceOf: ?class at: ?sequenceNumber` rule which is defined as follows:

```
?instance isInstanceOf: ?class at: ?methodInvocationNumber  if
```

```
methodExit(?,?methodInvocationNumber,?instance),
methodEntry(?methodInvocationNumber,?,?class,?,?),
instanceOfClass(?instance,?class)
```

It states that an `?instance` of a `?class` is created at a particular `?sequenceNumber` when we have a method invoked on `?class` which returns an instance of this class.

This kind of information is hard to obtain statically because deriving an exact control flow graph from the source code is hard. For instance, our rule correctly detects instances that were created by an object Factory while a static rule that only searches for `new` messages fails to detect it or would need to be extended to handle this specific case. The dynamic rule is in contrast only a few clauses long.

Users can use this rule in the query below where we additionally map the integers returned by the rule to a human-readable Smalltalk object:

```
?instance isInstanceOf: ?class at: sequenceNumber,
objectMap(?instance, ?i),
objectMap(?class, ?c)
```

The results returned by this query are of the following form:

```
[?instance-->[2],
 ?c-->[Tracing.DDTestInvoker],
 ?class-->[1],
 ?i-->[a Tracing.DDTestInvoker],
 ?methodInvocationNumber-->[1]]
```

Note that we can limit the amount of results by binding more variables in the query, for instance when we want to restrict our investigation to instances of a particular class or instances generated during a certain method invocation. Also note that we do not restrict ourselves to instances that were directly created in a method invocation (as a static analysis would do), but also consider instances that were created indirectly by intermediate method calls.

In the next chapter, we will show how the accuracy of this rule can be improved by incorporating static information from the existing source code reasoning library in a structural way.

4.1.3.2 Object State Tracking Using Variable Assignments

The previous example used the `objectMap` predicate to map integers in the logical transcript of an execution history to the actual Smalltalk object they represent. However, as a consequence of most-portem analysis, the object returned by this predicate is in the state it was in at the end of the program's execution and not in the state it was in during any other given message invocation. This also implies that if an instance variable is assigned to in the middle of the program and once again at the end of the program, we can only access its latter value even if we are reasoning about events that occurred before the last assignment.

From this observation, the need to know the state of all objects at each moment in time is easily identified. For this reason, we provide the predicate `?variable in:`

`?instance at: ?methodInvocationNumber value: ?value`, which states that a `?variable` related to a particular `?instance` contained a `?value` at the end of the method invocation identified by a `?methodInvocationNumber`. Its implementation involves declaratively calculating the value of a variable from the recorded assignment events and is detailed in appendix B.

Due to the multi-directional nature of logic predicates, we can use this predicate (aside from reasoning about variable values in the design layer predicates) in multiple ways:

- If only the `?instance` variable is bound to a value, we obtain for all the `?variable` names related to that instance a summary of the different values taken on at the end of each method invocation identified by the`?methodInvocationNumber`.

  Among others, a possible use would be to detect unused or potentially dangerous variables: variables which never take on a value other than nil.

- Suppose that an instance variable holds an incorrect value, but that we cannot pin-point the exact method responsible for the error. In that case, we can use the rule to search for all methods in which this faulty value is assigned to the variable.

- Suppose on the other hand that we know the last occurrence of the correct value for the above variable and that we want to know where this correct value is overridden by an incorrect one. For this purpose, we could use the `overridesExternalAssignment` (a sub-goal of the above predicate, see appendix B for more information) which returns the assignments that override another assignment during another method invocation.

- The `overridesInternalAssignment` predicate, another subgoal of the predicate, can for instance be used to count the number of assignments to an instance variable that override another assignment within the same method invocation.

It is clear that this kind of information is extremely difficult to obtain from a static source code analysis.

By only tracking variable assignments we are however ignoring another important source of state information: the collections in an instance. These are normally not assigned values, but extended using *add*-like messages and shrunk using *remove*-like messages depending on the actual kind of collection used. Keeping track of these values would involve writing wrappers for commonly used collection classes which generate events when a element is added to the collection or removed from it. A major downside is the amount of trace information that once again increases enormously so we haven't pursued this path further on.

4.1.3.3 Binary Class Relationships

As was shown by Guéhéneuc in his Ph.D. thesis [Gué03], a declarative analysis of a program's run-time behaviour can be used to categorise binary class relationships:

Association There exists an association relation between classes *A* and *B* if instances of the classes exchange message sends with each other.

Aggregation There is an aggregation relation between classes *A* and *B* if there already is an association relation and class *B* is contained in a variable of class *A*:

```
aggregation(?wholeInstance,?partInstance,?sequenceNumber) if
  directedAssociation(?wholeInstance,?partInstance,?sequenceNumber),
  ?variable isAssignedVariableOf: ?wholeInstance,
  ?variable in: ?wholeInstance at: ?sequenceNumber value: ?partInstance
```

Composition An aggregation relationship can be classified as a composition if the lifetime of the part is entirely encompassed by the lifetime of the whole.

```
composition(?compositeInstance,?componentInstance,?sequenceNumber) if
  aggregation(?compositeInstance,?componentInstance,?sequenceNumber),
  ?componentInstance of: ? instantiatedAfter: ?compositeInstance of: ?
```

The above rules can be used to classify the binary class relationships in programmatically extracted UML diagrams. As this involves lifetime and exclusiveness information, this classification cannot be performed statically.

4.1.4 Design layer

This layer contains predefined predicates for analysing a program's behaviour and design. This layer will not only base its conclusions upon static and dynamic information, but also appeals to approximate reasoning –the second pillar of our framework– for a more expressive and flexible specification of its predicates.
Therefore, we will devote a separate chapter (Chapter 5) to the design layer and describe the features of our base language for approximate reasoning first.

4.2 Declarative Language for Approximate Reasoning

To incorporate approximate reasoning –which forms the second pillar of our approach– in the automated re-engineering process, we have developed a basic fuzzy Prolog. The features it exhibits for dealing with uncertainty and vague concepts are introduced in the following exposition along with concrete examples to make the reader acquainted with its extended syntax.

The language itself is implemented for efficiency reasons as a meta-interpreter in the SWI-Prolog system instead of in SOUL itself.

Basic Weighted Facts and Rules One of the primary language features is the ability to assign weights to rules and facts. We have chosen the backquote operator for this purpose since the more familiar colon is already used for other purposes by the SWI-Prolog system in which the meta-interpreter was implemented.

In the following example, we are absolutely certain that Jan is Rob's father, while we're not so certain at all about Karen being Rob's child. Also, we can't be totally certain about the grandparent rule so it is assigned a truth degree lower than the absolute truth.

```
parent(jan,rob) ‘ 1.
parent(rob,karen) ‘ 0.3.
parent(rob,tom) ‘ 0.5.
```

```
grandparent(X, Y) ‘ 0.9 :-
        parent(X,Z),
        parent(Z,Y).
```

We can now ask our meta-interpreter for the degree of truth of solutions to `grandparent(X,Y)` with the query `prove(grandparent(X, Y), C)`. In this case, we will obtain a certitude of 0.27 for the solution `X = Jan`, `Y = Karen` while we obtain a certitude of 0.45 for Jan being Tom's grandfather.

As can be derived from these results, we're using Gödel semantics for the conjunction operator (i.e. the minimum) and the Larsen implication operator (i.e. multiplication) which are, as we have seen in the previous chapter, popular choices for Fuzzy Prologs.

Combining Crisp and Fuzzy Clauses In the previous example, we explicitly stated that we were absolutely certain about the fact `parent(jan,rob) ‘ 1` by assigning it a certainty of 1. This truth assignment can be omitted for crisp facts and rules as is the case for the `sold` fact and `popular_product` rule in the next example.

We are stating that any product of which more than 10 items are sold must be a popular product, while other products are destined to become popular with a fair certainty given an attractive packaging and good advertising.

```
sold(flower, 15).
nicely_packaged(chips) ‘ 0.9.
well_advertised(chips) ‘ 0.6.

popular_product(X) :-
        sold(X, A),
        A > 10.

popular_product(X) ‘ 0.8 :-
        nicely_packaged(X),
        well_advertised(X).
```

In the example, `chips` is a popular product with a certainty of $min(0.9, 0.6) * 0.8 = 0.48$ while we can be absolutely certain about `flower` being a popular product.

Algebraic Weight Assignments It is also possible to assign a variable to the degree of truth of the head of a rule instead of a real on the unit interval. This variable can then be bound in the body of the rule.

In the following example, this language feature is used to simulate the fuzzy set `about_20`. Its membership function $\Delta(10, 20, 30)$ can be modelled by the following set of rules. They can be used to determine how close a given number is to 20, but not in the other way around; that is to return a number close to 20 up to a certain degree.

```
about_20(20).
```

```
about_20(X) ‘ 0 :-
        X < 10.

about_20(X) ‘ C :-
        X >= 10,
        X < 20,
        C is (X - 10) / 10.

about_20(X) ‘ C :-
        X > 20,
        X =< 30,
        C is (30 - X) / 10.

about_20(X) ‘ 0 :-
        X > 30.
```

When using truth variables in this way, it is important to keep in mind that they are being bound to the degree of truth of the left-hand side of the implication (the head of the rule) and not to the degree of truth of solutions to the entire rule.

In essence we are saying that the head of a rule is true up to a certain degree of truth which we will assign later on in the body of the rule given that this body is absolutely true. If this body is true only up to degree of for instance 0.5, the certainty of solutions to this rule will be half that of the value of the variable bound to the degree of truth of the head of the rule.

Truth of Subgoals Another useful feature allows obtaining the degree of truth of a rule's individual subgoals.

```
successful_product(X) :-
        popular_product(X) ‘ C,
        C >= 0.5.
```

In the above example, this feature is used to limit successful products to those products which are popular up to a degree of 0.5. The solutions will all have a certainty of 1 as the certainty of expressions φ ‘ *C* is defined to be 1 when *C* unifies with the truth degree of φ.

Explicit Truth Assignment on Rule Heads As 1 is the neutral element of Gödel conjunction, the fact that φ ‘ *C* is absolutely certain when *C* unifies with the truth degree of φ can be used to combine the truth degrees of a rule's subgoals in a way that is more suitable to the specific problem at hand.

In the example below, the average of the quantitative part of the first two subgoals is explicitly assigned to the certainty of the rule's head.

```
popular_product2(X) ‘ C :-
        nicely_packed(X) ‘ C1,
        well_advertised(X) ‘ C2,
        C is (C1 + C2) / 2.
```

Again, it is important to note that we are only assigning the certainty of the rule's head. In this case, the degree of truth of the body is simply 1.

Explicit Truth Assignment on Rule Bodies Combined with the possibility of using numbers as regular clauses (their truth degree equals the number they represent), we can use the above feature to actually assign the truth degree of a rule's body and thus also overrule the default interpretation of a conjunction of truth degrees.

It could be argued that using the average of the truth degrees of `nicely_packed` and `well_advertised` is a more balanced measure of the popularity of a product than the minimum. We can express this in the following way:

```
popular_product3(X) ‘ 0.9 :-
        nicely_packed(X) ‘ C1,
        well_advertised(X) ‘ C2,
        C is (C1 + C2) / 2,
        C.
```

The truth degree of each subgoal is 1 except for the last one which is actually the average of the degree of truth of `nicely_packed` and `well_advertised`. Using this rule, we get a certainty of $0.9 \times min(1,1,1,\frac{0.9+0.6}{2}) = 0.675$ for the popularity of chips.

Truth Modifiers Using algebraic expressions and explicit truth assignments on rule bodies or heads we can model linguistic hedges (whose use and purpose we have discussed in section 3.2.2.6) in the following way:

```
absolutely(X) :-
        X ‘ 1.

fairly(X) ‘ C :-
        X ‘ C1,
        C is sqrt(C1).

very(X) ‘ C :-
        X ‘ C1,
        C is C1 ** 2.
```

In addition to unifying degrees of truth with variables, our language also supports using these certainties as input of user-defined predicates. The degree of truth of an expression φ ‘ ψ then equals the truth degree of ψ with the truth degree of φ as its input.

For instance, *C* will be bound to 0.83666 in the query `prove(0.7 ‘ fairly, C)`.

The following rule expresses that a successful product must be a very popular product up to a degree of at least 0.2. It also shows how the backquote operator can be chained:

```
successful_product2(X) :-
        (popular_product(X) ‘ very) ‘ C,
        C > 0.2,
        C.
```

Combining Multiplication and Minimum for Conjunctions We have already shown how the default calculation of truth values of a conjunction can be overridden. There's also some syntactic sugar for another often-used conjunction operator: the multiplication designated by `#`. This will come in handy for the next feature we will discuss.

```
zeropointfour  :-
        1 # 0.4,
        0.9 # 0.5.
```

In the above example, the truth degree of the query `zeropointfour` is calculated as $min(1 \times 0.4, 0.9 \times 0.5) = 0.4$.

Fuzzy Unification of Strings Another language feature allows approximate unification of strings based on their Levensthein edit distance (see section 3.3.2.3). The operator for approximate unification is `~`. It is reasonable to obtain the certainty of a goal with approximately unified arguments by multiplying the unification degree of the arguments with the certainty of the goal itself. This is where the `#` operator comes in handy.

The following mini-program implements a dictionary which is insensitive to spelling mistakes thanks to the fuzzy unification on the first argument of `dictionary_entry`:

```
dictionary_entry('flavour', 'fragrance') ‘ 0.8.
dictionary_entry('flavour', 'taste').

approximate_lookup(X, Y) :-
        dictionary_entry(X1, Y) # X ~ X1.
```

We will find that 'fragrance' and 'taste' are answers to the query `approximate_lookup(flavour, X)` with a certainty of 0.69 and 0.86 respectively.

Fuzzy Unification of Terms Our language also supports approximate unification of prolog terms as discussed in the previous chapter.

In this example, we are trying to unify a term with a small typing error in the functor name at the second level and with the second argument omitted.

```
prove(a(abba(X, Y), Z) ~ a(abb(2), 3), C).

X = 2
Y = _G148
Z = 3
C = 0.667774
```

We can see that the algorithm tries to unify the two terms as good as possible. It is clear that this feature has to be used with caution as we might be able to unify two very distinct terms, albeit with a very low unification degree and with some variables left unbound.

An implicit fuzzy unification for all clauses could be built in the language itself, but we have opted not to do so and leave an explicit choice with the user as this is often not needed in declarative meta-programming for software re-engineering and involves in addition a large run-time overhead.

4.3 Conclusion

We have discussed the two cornerstones of our approach to software re-engineering: the declarative framework for dynamic analysis and the base logic meta-programming language which supports approximate reasoning.

Our dynamic analysis library for reasoning about a program's behaviour is structured in a layered manner similar to the organisation of the LiCOR library for structural code reasoning it is intended to complement.

Our source model consists of execution traces comprising chronologically ordered run-time events which are reified to logic facts. The source model is fine-grained and records method invocation arguments, method return values and variable assignments which allows control and data-flow information to be obtained from the execution traces.

We have implemented two variants of dynamic analysis: post-mortem and ad-hoc analysis where the latter interweaves the execution of the application and the declarative analysis of its run-time behaviour. The former reasons over complete execution histories obtained after the program has ended. Ad-hoc analysis is suitable for simple queries which don't require backtracking over previous choices for run-time events. Its lightweight nature makes it fit for limiting the search space of complex logic rules in the post-mortem analysis variant.

We have shown some examples of low-level ad-hoc predicates for reasoning about a program's behaviour: a rule which counts the number of single argument message invocations and an elementary way to influence the behaviour of an existing application reminiscent of event-based aspect-oriented programming.

We have also discussed examples of useful post-mortem predicates which are able to detect exactly when each instance of a particular class is instantiated, we have shown how to calculate an object's ever-changing state at a random moment in an execution history through variable assignments and we have classified the binary class relations known from UML diagrams. We have also described a rule for detecting the double dispatching pattern. These rules are all examples of information that is very difficult or even impossible to obtain through static source code analysis.

We have concluded the chapter with an exposition of the approximate reasoning features incorporated in our base logic meta-programming language.

Chapter 5

Applying Approximate Reasoning and Dynamic Analysis to Software Re-engineering

In this chapter we will discuss how the predicates from the lower layers of our declarative framework for dynamic analysis can be used to express more complex software patterns such as the Visitor design pattern. We will also show how approximate reasoning, the second pillar of our approach to software re-engineering, can augment the expressiveness of the base declarative meta-programming language thus allowing software patterns to be described by flexible yet succinct rules.

We have identified three commonly re-occurring problems in declarative meta-programming applied to software re-engineering for which support for approximation and vague concepts in the base declarative language promises to be a viable solution. The following sections will introduce each problem with a dynamic or static example that motivates the use of approximate reasoning to overcome the problem.

5.1 Idealisation of Pattern Detection Rules

The first of the three commonly recurring problems we have identified in declarative meta-programming is that of overly idealised declarative rules. We often strive for rules that are as expressive and compact as possible, but –in an attempt to take every conceivable real-world exception into account– often end up with massive rules consisting of multiple conditions only to find out later that yet another discrepancy between the utopic abstract description of a software pattern and a real-life occurrence wasn't covered by this rule.

The Visitor design pattern is a perfect case study for the merits of approximate reasoning in detecting design patterns with compact logic programming rules as it is the prototypical design pattern example with average complexity. It is one of the twenty-three design patterns introduced by the "gang of four" in their famous book on reusing

proven and often re-occurring software designs [GHJV94].

The Visitor pattern is used when many unrelated operations need to be performed on objects of different types in a compound object structure. Its structural architecture is shown in figure 5.1.

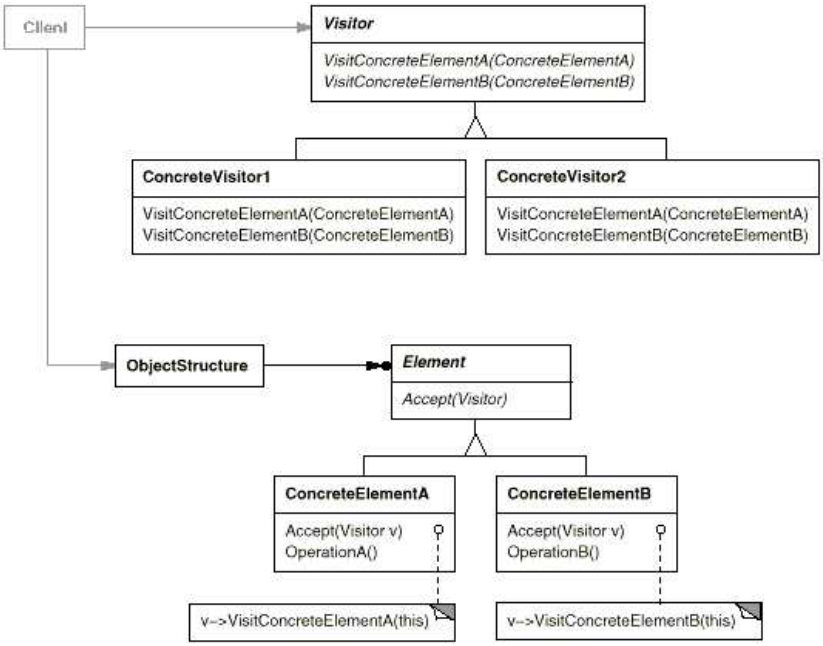

Figure 5.1: The architecture of the Visitor Design Pattern [GHJV94].

The `Visitor` abstract class has a method `visitConcreteElementX:` for each element of type X in the object structure. Instead of cluttering each element in the object structure with their respective part in the definition of a complex object structure operation, concrete implementations of the `visitConcreteElementX:` methods can be given in `Visitor` subclasses to define the operation on the object structure. The objects in the object structure accept a `Visitor` subclass with their `accept:` method and subsequently call the `visitConcreteElementX:` method corresponding with their type on the received visitor. The run-time behaviour of the Visitor design pattern is depicted in the sequence diagram shown in figure 5.2.

This way, all methods defining a specific operation on the object structure are kept together in a separate class instead of ending up intermixed with other operator implementations in the components of the object structure.

Implementing new operations on the object structure thus amounts to creating a new `Visitor` subclass instead of extending the implementation of the object structure components. Extending the object structure itself is however more difficult as every `Visitor` subclass will have to implement a new method for visiting the newly created object structure element.

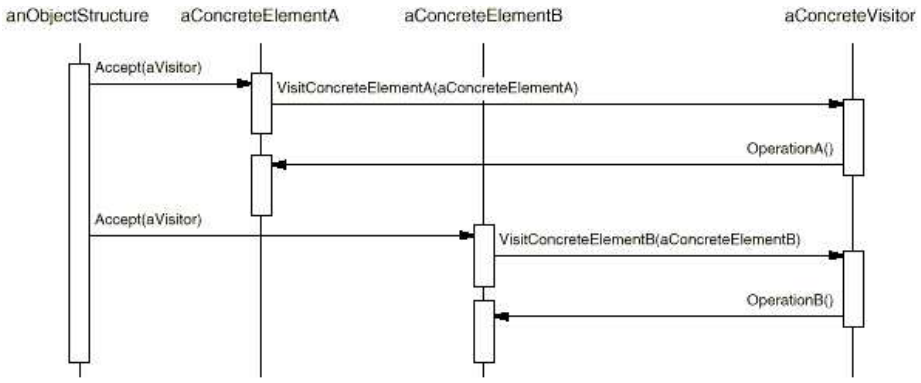

Figure 5.2: The sequence diagram of the Visitor design pattern [GHJV94].

We will first show how the pattern can be detected using a static analysis and identify some of the problems related to this form of program analysis. This type of functionality is particularly helpful to maintainers whose understanding of an unknown legacy system (of which the documentation is often non-existent or no longer in sync with evolved versions) can be significantly increased when a tool informs them about instances of well-known design patterns that were successfully detected in the application's source code.

5.1.1 Static Detection of the Visitor Design Pattern

The use of static information implies that only the structural aspect of a programming pattern can be expressed in a straightforward manner. This doesn't pose any problems for structural patterns which are very architecture-centric and can be easily described in terms of class hierarchies and methods alone. However, it is much harder to express a programming pattern that primarily describes how its entities collaborate with each other.

The Visitor design pattern is a fine example of a behavioural design pattern that can be detected more easily using dynamic analysis. Its architectural structure is shown in figure 5.1. The logic rule below describes this structure in a straightforward manner:

```
visitor(?visitor, ?element, ?accept, ?visitSelector) if
  class(?visitor),
  classImplements(?visitor, ?visitSelector),
  class(?element),
  classImplementsMethodNamed(?element, ?accept, ?acceptBody),
  methodArguments(?acceptBody, ?acceptArgs),
  methodStatements(?acceptBody,
                   <return(send(?v, ?visitSelector, ?visitArgs))>),
  member(variable([#self]), ?visitArgs),
  member(?v, ?acceptArgs).
```

First of all, the rule checks whether the `visitor` variable[1] is a class that implements the `visitSelector` method. The rule then verifies that `?element` implements a method with the body `?acceptBody`. This method will be called with the visitor as its argument which is checked on the last line of the rule. In the body of this method, a message `visitSelector` is consecutively sent back with the visited element as its argument. This is verified by matching its source code with the statements in the `methodStatements` part of the rule.

We are thus heavily relying on the actual implementation of the pattern in the source code with little room for small derivations. The above rule assumes for instance that the descend through the object structure is controlled by the visitor instead of by the object structure itself as the body of the `accept:` method is required to match the `<return(send(?v, ?visitSelector, ?visitArgs))>` statement list exactly.

5.1.2 Dynamic Detection of the Visitor Design Pattern

In the previous chapter, we discussed some of the predicates from the basic layer which offer information that is difficult or even impossible to obtain through a static analysis alone. As we have illustrated, these predicates are by themselves useful to reason about a program's behaviour.

However, in the introduction we also claimed that dynamic and static analysis techniques complement each other. This can be easily verified if we move one layer up in our declarative framework to the design layer. Some design patterns, for instance, are difficult to detect statically while others are difficult to detect dynamically. We will illustrate this now with a dynamic rule to detect Visitor design pattern instances.

In the previous section, we showed how instances of the visitor design pattern can be detected by searching for literal translations of the pattern's architecture in a program's source code. The Visitor design pattern can however also be described by the behaviour it exhibits instead of the implementation of that behaviour. The pattern's sequence diagram describing its run-time behaviour was shown in figure 5.2.

The translation of this sequence diagram to a rule that can be used in a dynamic program analysis results in more expressive and readable definitions of the visitor design pattern. Furthermore, rules incorporating dynamic analysis can ignore differences in the concrete implementation of the visitor behaviour such as whether the visitor or the object structure controls the visitation of the components as they only look for run-time events that are absolutely elementary to the behaviour of the Visitor design pattern.

If we study the dynamic behaviour of the Visitor Design Pattern using the annotated sequence diagram shown in figure 5.3, we can conclude that a recursive double dispatching over instances held by a parent node characterises this pattern's behaviour. The visitation of `anObjectStructure` begins and ends at certain moments in time between which a visitation of the subelements `aConcreteElementB` and `aConcreteElementA` occurs. The latter visitation comprises a third visitation on `aConcreteElementC`.

[1] See section 2.2.1 for a description of the SOUL syntax

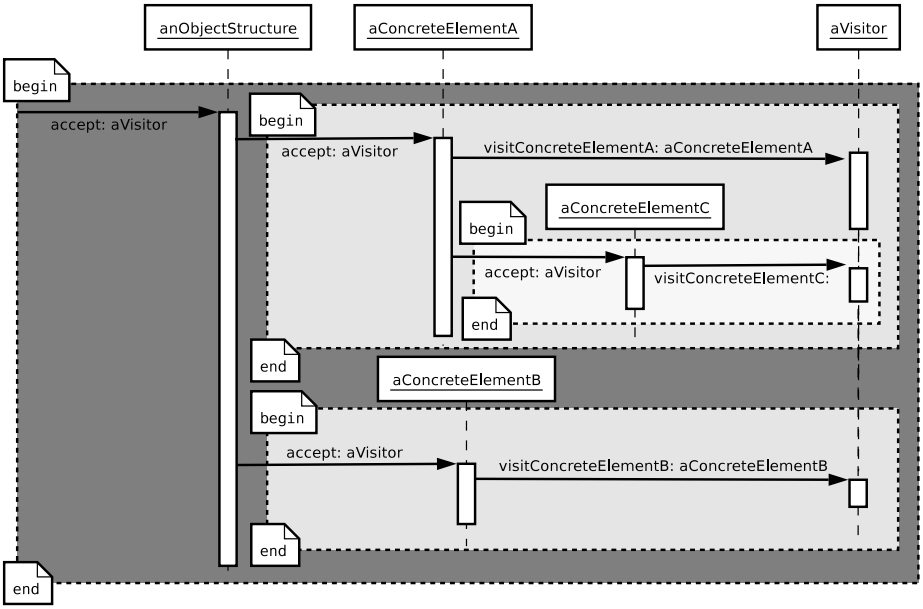

Figure 5.3: An annotated sequence diagram demonstrating the recursive nature of the Visitor design pattern

We recall that the `doubleDispatchesOn:selector:at:andOn:selector:at:/7` predicate from the basic layer states that two consecutive method invocations can be classified as a double dispatching if the receiver of the second message was part of the arguments of the first method invocation and if the arguments of the second invocation contain the receiver of the first message.

The behaviour of the Visitor Design Pattern rule can then be captured by the following succinct rule which is a straightforward translation of the corresponding sequence diagram:

```
?visitor visits: ?composite from: ?begin till: ?end invokedBy: ?invoker if
  ?invoker doubleDispatchesOn: ?composite
           selector: ?acceptselector
           at: ?begin
           andOn: ?visitor
           selector: ?visitselector
           at: ?end,
  (?visitor visits: ?part from: ? till: ? invokedBy: ?)
    forall: (?composite contains: ?part at: ?begin)
```

The rule expresses that an `?invoker` caused a `?visitor` to visit a `composite` from the `?begin` sequence number corresponding with the `accept:` method invocation till the `?end` sequence number corresponding with the matching method exit event if two conditions are met. The first condition captures the double dispatching event of the pattern: there should be a double dispatching between the `?composite` and the `?visitor` in which the first method plays the role of the accept method in the pat-

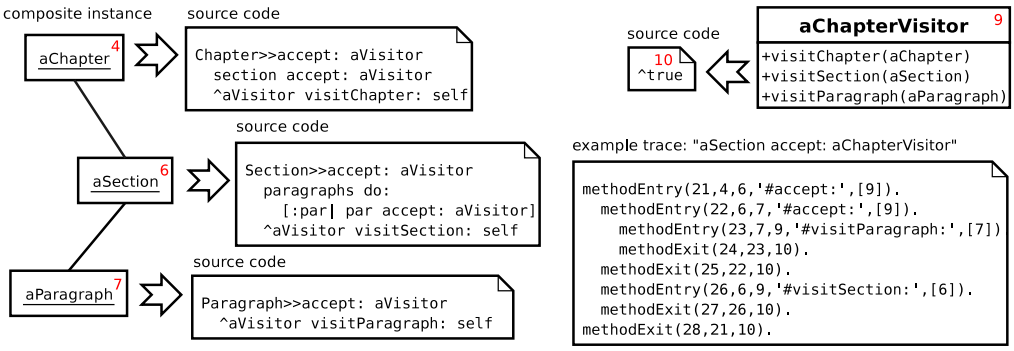

Figure 5.4: Object structure, visitation methods and extract from the corresponding execution trace.

tern –the `?acceptselector` is received by a `?composite` with the `?visitor` as its argument– and the second method behaves as the visit method in the pattern – the `?visitselector` is received by the `?visitor` with the `?component` as its argument. The second condition captures the recursive nature of a visitor: in addition to the presence of the above double dispatching pattern, we also demand that the visitor *recursively* visits all the components of the composite. [2]

The recursion in this rule mirrors the recursion in the pattern's sequence diagram as discussed above. This validates the need in meta-programming for a full logic programming language supporting recursion instead of opting for a lightweight execution history analysis using regular expressions.

The visitor rule behaves as expected on an example trace of a `ChapterVisitor` visiting a composite instance consisting of a `Paragraph` held in a collection contained by a `Section` stored in an instance variable of a `Chapter` object. The execution trace is perfect with respect to the `forall` predicate as every component of the composite tree is visited by the visitor.
This situation is depicted in figure 5.4 which depicts a composite instance consisting of a `Paragraph` instance held in a collection contained by a `Section` instance which in its turn is the value of an instance variable of a `Chapter` object. The figure also shows an extract of the execution history of `ChapterVisitor` instance visiting the section object. Numbers in the instance squares denote the unique identifying integer they are mapped to in the execution trace. The full execution trace is given in appendix C. As the source code extracts show, the iteration through the composite tree is controlled by the tree itself.

The results of the query `?visitor visits: ?composite from: ?begin till:`

[2] The `composite contains: ?part at: ?begin` predicate is, in addition to the instance variable value tracking predicate from the basic layer, composed of a logic transcription of the members contained by collections in each instance at the end of the execution trace. This implies that our current visitor detection rule is limited to composites whose components are known at the end of the execution trace. This limitation can however be overcome in future versions of the declarative framework by tracking additions and removals from collections. This way, collections can be queried for their contents at each moment in the execution history and thus also at the moment of the visitation of their respective parent.

`?end` on the corresponding execution trace given in appendix C are shown below. In addition, we took the liberty to change the object identifying integers to the textual representation of the corresponding instances which can be obtained using the `objectMap/2` predicate:

Composite	Invoker	Begin	End	Visitor
a Chapter	a VisitorInvoker	20	30	a ChapterVisitor
a Section	a Chapter	21	27	a ChapterVisitor
a Paragraph	a Section	22	24	a ChapterVisitor

From these results we derive that the `ChapterVisitor` class (the root of a class hierarchy of visitors for transforming formatted book chapters to for instance plain text files) visits the `Chapter` class beginning with method invocation 20 ending with a method exit at sequence number 30. During this visitation, the visitor also pays a visit to the `Section` class from sequence number 21 till sequence number 27. The `Paragraph` class is visited from sequence number 22 till 24. The recursive nature of the Visitor design pattern is emphasised by the order in which the components are visited: the visitation of the root node ends when the invocation of the visitor on its children has ended. The control over the recursive descend of the composite structure is located in the structure itself which can be derived from the solutions by observing that the `Invoker` variable is always bound to the parent node in visitations originating from higher levels in the structure.

This rule is by nature insensitive to differences in common implementation variants of the visitor as they mostly exhibit the same run-time behaviour. For instance, it is insignificant whether the visitation of composite elements happens through an iteration over elements in a collection (as is the case for the `Section` class) or through a (possibly indirect) call to an instance variable (as is the case for the `Chapter` class).

Although this was an example of the control over the descend in the composite structure being localised in the structure itself, we will see in the next section that our dynamic rule also correctly detects instances of the visitor design pattern where the visitation is controlled by the visitor itself instead of by the visited structure.
As the static rule presented in section 5.1.1 requires the statements of the visitor to be of the literal form `<return(send(?v, ?visitSelector, ?visitArgs))>` there is really no room for the object structure to take on the additional role of controlling the descend through the structure.

In contrast to the static approach that searches for archetypical implementations of the Visitor design pattern architecture, our dynamic approach finds multiple variants of the Visitor design pattern using only one expressive and extremely compact rule. We can conclude that dynamic analysis proves to be a powerful alternative for detecting Visitor design pattern instances using a straightforward and compact translation of the pattern's sequence diagram.

5.1.3 Using Approximation in Overly Idealised Rules

The dynamic Visitor rule from the previous section performs as expected in cases where the visitor descends every instance contained by each node in the composite tree starting from the tree's root. An example of such a situation was the `ChapterVisitor` in

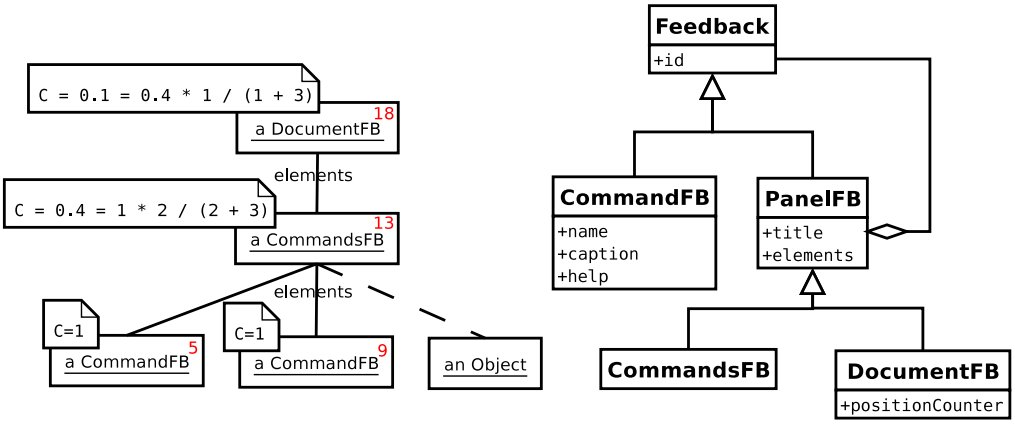

Figure 5.5: An incomplete visitation of an object structure taken from a real-life application.

figure 5.4 where a `Chapter` instance at the top of the tree was visited by recursively visiting the children it contained.

Unfortunately, equally perfect examples of the Visitor design pattern are rather rare in most real-life situations since composite components often contain many auxiliary instance variables or collection elements that are never visited. An example of such a commonly occurring situation is depicted in figure 5.5 where a composite tree contains many objects (detected as elements of collections and non-collection values of instance variables) that aren't visited by the visitor.

The example is taken from the author's licenciate project where a user feedback structure is built containing commands the user can choose from and which will be transformed by a visitor to either an XML or an HTML document depending on the used visitor.

Two kinds of instances are held in each component: instances contained by a collection belonging to the component and instances assigned to an instance variable of the component. The instance variables in each component are shown in the class diagram at the right. The left side depicts an actual instantiation of a user feedback composite consisting of a root `DocumentFB` node whose `elements` instance variable contains a collection comprising one `CommandsFB` instance whose `elements` variable in turn contains three objects of which the last isn't visited by the visitor. The feedback composite will be visited by an `XMLFBVisitor` instance which transforms the visited structure in an XML document suitable for rendering in web browsers.

As depicted in figure 5.5, only the leaves of the composite structure are visited recursively and our visitor rule fails on all but these nodes. This is a bit of a disappointment since we wouldn't like to give up either the expressiveness or the compactness of the original succinct rule by extending its definition with conditions to forgive small deviations. Our visitor rule is a clear example of an overly idealised rule as we require each component of the composite to be visited and since our rule is only relying on dynamic

information, we have no choice but to consider every instance held by a component as its children. This assumption is more often than not false in real-life situations.

Until now, we have evaluated our rule in the usual crisp logic programming environment provided by SOUL. We can however refrain from altering our compact rule by evaluating it in the fuzzy logic programming environment introduced in section 4.2 which includes a fuzzy variant of the `forall/2` predicate which we haven't discussed before. Since this predicate assigns an (albeit lower) degree of truth to `forall/2` predicates whose test query doesn't succeed on all unbound variables, we can close the gap between the real-word concrete Visitor design pattern instance and its idealized abstract description without extending the rule itself.

For completeness' sake, the definition of the fuzzy version of the `forall/2` predicate in the fuzzy meta-interpreter is given below:

```
fuzzy_forall(Query, Test) :-
        findall(SucCert, (Query, Test) ‘ SucCert, Successes),
        findall(TotCert, Query ‘ TotCert, Total),
        calculate_certainty(Successes, Total).

calculate_certainty(S, _) ‘ 1 :-
        S = [].

calculate_certainty(Successes, Total) :-
        minimum(Successes, Min),
        length(Successes, S),
        length(Total, T),
        C is Min * S / T,
        C.
```

The truth degree of answers to this rule `fuzzy_forall(Query, Test)` equals the ratio `Query` solutions satisfying `Test` to the total amount of `Query` solutions multiplied by the minimum of success truth degrees. This factor can be seen as an expansion of a forall to a series of conjunctions.

Using this fuzzy forall predicate in the visitor rule, we can now successfully detect a visitor that visits the composite structure incompletely. The results of the query `prove(visits_from_till(Visitor,Composite,Begin,End, Invoker), C)` on an execution trace of a `XmlFBVisitor` visiting the composite shown in 5.5 are given below.
In contrast to the example in the previous section, this is a visitation where the control over the descend in the composite structure is located in the Visitor. This implies that the `Invoker` variable will always be bound to the Visitor instance in solutions that recursively descended from a visitation originating higher up in the tree.

Certainty	Composite	Invoker	Begin	End	Visitor
0.1	a DocumentFB	a FeedbackTranslator	60	100	a XmlFBVisitor
0.4	a CommandsFB	a XmlFBVisitor	67	97	a XmlFBVisitor
1	a CommandFB	a XmlFBVisitor	74	83	a XmlFBVisitor
1	a CommandFB	a XmlFBVisitor	85	94	a XmlFBVisitor

The last result shows that the visitation on the composite's leaves are 100% successful. The visitation on the `CommandsFB` is however less successful as only 2 of three instances contained in its collection are visited recursively and its 2 non-collection instance variables aren't visited at all. This augments to a certainty value of only $0.4 = \frac{2}{3+2}$. The visitation on the composite's root isn't very successful either as its only collection member is visited for 0.4 and its 3 non-collection instance variables aren't visited. This augments to a degree of truth of $0.1 = 0.4 \times \frac{1}{1+3}$ for the recursive visitation on the entire object structure.

It might seem that the certainty values in the obtained solutions get rather small when large object structures need to be traversed, but this is just a matter of interpretation. As long as this value exceeds the minimum truth value of 0, we can be quite sure that there is in fact a visitor visiting the object structure. The magnitude of the associated truth degree is an indicator for the amount of recursive visitations that are called upon objects contained in the composite structure. When this number is extremely small, this might indicate that there are too many instance variables in a particular class; an indication of a design fault which we will discuss in the next section. Alternative definitions of the fuzzy `forall/2` predicate which aren't based on the minimum truth aggregator, but on an averaging operator return larger quantitative results.

We could also make the forall constraint in the Visitor design pattern detection rule less restrictive by limiting its unbound variables to objects in the composite structure which implement the `accept:` selector, but this requires the combination of static and dynamic information (which we will discuss later on) and would make it impossible to detect less perfect instantiations of the pattern.

Although the example presented in this section can only be considered as the beginning of an exploration of the use of approximate reasoning in the relatively new domain of declarative meta-programming, it already motivates that allowing fuzzy predicates aids in closing the gap between the ideal description of a pattern and its real-world implementation.

5.2 Expressing Vague Software Patterns

Many software patterns express inherently vague concepts which rely on arbitrary boundaries to decide whether a part of a program's implementation adheres or fails the constraints of the pattern. When these constraints are enforced too rigidly, interesting instances that only just failed the strict restrictions of the software pattern may be overlooked. Therefore, built-in support for expressing vague concepts is desirable in declarative meta-programming.

5.2.1 Detecting Bad Smells

Typical examples of vague software patterns are the so called *"Bad Smells"* put forth by Fowler and Beck in their book on software refactoring [FBB+99] as indications of situations in which to apply a refactoring in order to improve overall source code quality.

In her master's thesis [Bra03], Francisca Muñoz Bravo already showed how logic meta-programming can be applied to support the detection of bad smells and to execute their corresponding refactorings.

However, a common problem in expressing bad smells as logic rules is that they often rely on user-specified thresholds which decide whether the bad smell is detected or not. Examples of bad smells suffering from this problem are the *"too many instance variables"* or *"too many parameters"* bad smells. Their classification boundaries are inherently vague: if the user sets a boundary limit of 10 variables, he will likely consider a class with 9 instance variables almost as bad as a class with 10 instance variables.

If we evaluate these logic rules in a crisp logic programming language, many indicators of a bad design may be left undetected. A language for approximate reasoning offers two possible solutions to this problem: we can either replace the crisp comparison operators by fuzzy variants which are flexible in their judgements or we can let the user define their own concept of a large amount of instance variables by providing the membership function of the corresponding fuzzy set. At the heart, both solutions converge but they represent different ways of looking at the same problem.

We fill first discuss the fuzzy comparison operator approach as it doesn't require many changes to the original bad smell detection rule. Our base approximate reasoning language defines a fuzzy greater than or equal operator $y \succeq x$ whose membership function can be represented as an extension of the open right shoulder function $\Gamma(y,t,x)$ (see section 3.2.1.3). It returns the maximum truth value when $y \geq x$ and the minimum truth value when $y \leq t$. The threshold t determines the utter left point at which numbers y can be considered close enough for the comparison with x to succeed. A reasonable choice for most application domains is to let this threshold vary with the magnitude of the number x to which we compare. This way we obtain $7 \succeq 10 : 0$ and $47 \succeq 50 : 0.57$ although 7 is as far from 10 as 47 is from 50.

The `fuzzyGreaterThan/2` predicate is defined in our base declarative meta-programming language where it can be reused in many other rule definitions:

```
fuzzyGreaterThanThreshold(X, Threshold) :-
        Threshold is X - (2 + X // 10).

fuzzyGreaterThan(Y, X) ‘ 1 :-
        Y > X,
        !.

fuzzyGreaterThan(Y, X) ‘ 0 :-
        fuzzyGreaterThanThreshold(X, Threshold),
        Y < Threshold,
        !.

fuzzyGreaterThan(Y,X) ‘ C :-
        fuzzyGreaterThanThreshold(X, Threshold),
        Threshold =< Y,
        !,
        C is (Y - Threshold) / (X - Threshold).
```

And then finally, the rule which detects instances of the *"tooManyInstanceVariables"* bad smell in a fuzzy manner can be simply defined as the original rule with the crisp comparison operator replaced by the above fuzzy greater than predicate:

```
tooManyInstanceVariables(ClassName, AmountOfVariables) :-
        userTreshold(tooManyInstanceVariables, Maximum),
        numberOfInstanceVariables(ClassName, AmountOfVariables),
        fuzzyGreaterThan(AmountOfVariables, Maximum) ‘ notFalse.
```

In the last clause of the rules' definition, we use a linguistic modifier to demand that the result of the fuzzy comparison is at least larger than the minimal degree of truth, but other linguistic hedges can be used such as "very" or "slightly" to intensify or weaken our constraint on the amount of instance variables.

We used the above rule on the VisualWorks refactoring browser package with the fact `userTreshold(tooManyInstanceVariables, 10)` and obtained among others the following results:

```
ClassName = 'CodeModel'
AmountOfVariables = 11
C = 1 ;

ClassName = 'RBMethodNode'
AmountOfVariables = 8
C = 0.333333 ;

ClassName = 'RefactoringBrowser'
AmountOfVariables = 9
C = 0.666667 ;
```

We can see that the `CodeModel` class definitely contains too many instance variables given a user-defined threshold of 10 as also indicated by the original rule using a crisp comparison operator. In addition, our rule also suggests classes containing less instance variables with a lower degree of certainty that were overlooked by the original.

When using the fuzzy comparison operator defined above, the user only has to specify a classification threshold which will then be evaluated in a fuzzy manner. An alternative, but equivalent, approach lets the user specify its own fuzzy membership function (probably shaped like an open right shoulder) and use the corresponding fuzzy set as a constant which has to be unified approximately with the amount of instance variables present in the class.
We didn't implement this form of fuzzy unification since there is still a lot of ongoing research on the best way to implement this functionality (especially on how to unify two fuzzy constants, see for instance [GA98] and [Als01]), but it is interesting to see how the bad smell detection rule could be expressed in such a fuzzy Prolog:

```
tooManyInstanceVariables(ClassName, AmountOfVariables) :-
        userTreshold(tooManyInstanceVariables, Maximum),
        numberOfInstanceVariables(ClassName, AmountOfVariables),
        AmountOfVariables ~ large
```

Although the above rules represent a very elementary application of declarative meta-programming for static source code analysis, they do suggest the applicability of built-in support for approximate reasoning in the meta-programming domain as it provides a standard framework for representing vague concepts.

5.3 Overcoming Small Discrepancies

Often, there are small differences between a wanted solution and the actual program facts at hand. These differences can for instance originate from consistent spelling errors or different naming conventions for method selectors but are insignificant to the broader software pattern that we were trying to detect. A reasonable amount of flexibility is thus required when searching for solutions to queries over the base program facts.

5.3.1 Detecting the Visitor Design Pattern

In the Visitor design pattern detection rule above, we didn't impose any constraints on the selectors involved in the double dispatching pattern. We can however improve the accuracy of the rule by demanding the `AcceptSelector` variable to unify with `accept:` which is the selector the software engineering community seems to have agreed upon. This way, we can eliminate many false positives, but we will at the same time be unable to detect visitor design pattern instances using small variants on this standardised selector.

Similarity-based unification provides a possible solution to this problem as we discussed in section 3.3.2. Our base declarative meta-programming language provides a built-in operator ~ for approximate unification based on a syntactic similarity measure defined over prolog terms: the Levensthein edit distance we discussed in section 3.3.2.3.

We can use this operator to demand the `AcceptSelector` to approximately unify with `accept:` so consistent spelling errors and small naming deviations won't cause our detection rule to fail, but will simply return solutions with lower associated certainty degrees:

```
visits_from_till(Visitor,Composite,Begin,End) :-
    doubleDispatchesOn_selector_at_andOn_selector_at(Invoker,
          Composite,
          AcceptSelector,
          Begin,
          Visitor,
          Visitselector,
          End) #  AcceptSelector ~ '#accept:',
     forall(in_at_value(Variable,Composite,Begin,Part),
            visits_from_till(Visitor,Part,_,_)).
```

Note that we aren't using the regular Gödel interpretation (minimum) for the truth value of the conjunction between the double dispatching and the approximate unification clauses, but are using multiplication instead as denoted by the # operator. This is because a low approximate unification degree of the selector shouldn't dominate the

truth value of the entire visitor detection rule, but only slightly affect our confidence in the appropriateness of the detected double dispatching pattern instance.

The tolerance for naming variations can be exploited for launching flexible search queries over execution traces, but due to the syntactic nature of the similarity relationship deployed by the fuzzy unification operator our Visitor design pattern detection rule is limited to accepting small variations on the general accept selector such as `acceptVisitor:` and `accept:`.

Semantic similarity relationships can unfortunately not be implicitly provided by the declarative base language, but have to be expressed explicitly by the maintainer using user-defined fuzzy similarity predicates:

```
similarSelector('#accept:', '#doVisitor:') ‘ 0.5.
similarSelector('#accept:', '#receiveVisitor:') ‘ 0.7.
```

5.3.2 Detecting Accessor Methods

As we have seen before, dynamic analysis allows maintainers to overcome even very large discrepancies in concrete source code implementations of the same dynamic behaviour.
For instance, the easiest way to detect with one succinct rule the most common implementation variants of the run-time behaviour of an accessor method is to search for methods named after the instance variable whose value they return on method exit:

```
accessor(?selector,?instance,?variable) if
  methodEntry(?sn,?,?receiver,?selector,?),
  equals(?selector,?variable),
  ?variable in: ?receiver at: ?sn value: ?value,
  methodExit(?,?sn,?value).
```

In an attempt to achieve the same order of flexibility in a static analysis, we have experimented with the application of approximate unification based on edit-distance over prolog trees as defined in section 3.3.2.3 to detect approximations of software patterns in method parse trees.
The tolerance for differing functor symbols and omitted functor arguments does make it seem as a good solution for automating the detection of more elaborate variants of the elementary instance variable accessor pattern such as variants incorporating lazy initialisation or access count tracking. The following rule considers accessor method statements as approximations of the elementary accessor form which simply returns the variable's value:

```
accessor(Class, MethodSelector, Var) :-
   mliMethod(Class, MethodSelector,
             'method'(Class, MethodSelector, Args, Temps, Statements)),
             Statements ~ 'statements'(['return'('variable'(Var))]).
```

In this rule we use the fuzzy unification operator ~ to state that the statements of an accessor method need to match the pattern `^ var` approximately. Note the use of the logic variable `Var` in this pattern of statements.

However, this rule proved to perform very poorly in detecting the more elaborate accessor variants which can be explained by the fact that there can only be variation in the actual statements the method parse tree comprises of since symbol and arity of functors describing individual statements are fixed. The approximate unification would still work on the statement list was it not for the edit-distance algorithm working in a left-to-right fashion:

```
prove(statements([return(variable(x))]) ~
      statements([return(variable(X)), complexstuff]), C).
X = x
C = 0.733333

prove(statements([return(variable(x))]) ~
      statements([complexstuff, return(variable(X))]), C).
X = _G159
C = 0.733333
```

In the upper query, we see that the more elaborate statement list approximately unifies with the base statement list up to a degree of 0.73 substituting x for X as expected. However, in the lower query (which only differs from the upper query in the statement order), we see that X remains unbound thus causing many interesting solutions to remain undetected.

As such, a syntactic similarity relation that is indifferent to permutations of method statements is better suited for defining a similarity-based unification procedure in the base declarative meta-programming language. However, more interesting future work comprises the definition of a semantic similarity relation (for use in static analysis approaches) on methods which relies on dynamic analysis to compare the semantics of the run-time behaviour exploited by the methods that need to be compared.

5.4 Other Applications of Approximate Reasoning

5.4.1 Weighting Different Heuristics

As we have seen in the chapter on approximate reasoning, it is possible to assign different weights to rules describing the same predicate. This allows us to express heterogeneous heuristics about software patterns with varying confidence degrees.

5.4.1.1 Combining Static and Dynamic Information

Using the ability to assign weights to rules, it is possible to build a preference hierarchy among rules employing different heuristics to detect for instance the Visitor design pattern.

Since the static analysis rule from section 5.1.1 was able to detect perfect instances of the archetypical implementation of the Visitor design pattern's architecture (with the exception of implementations where the control over the descend of the object structure was located in the structure itself), we can rest assured that when this rule detects a Visitor design pattern instance it is almost certain no false positive. Our dynamic analysis rule on the other hand detects much more variations in the implementation of

the pattern, but there's a slightly increased risk of false positives. Therefore, we could assign the dynamic analysis rule a lower weight than the static rule and combine the solutions of each analysis variant in one unifying rule where each solution is weighted by the data source from which it originated:

```
% static results
visitor(VisitorClass, VisitedClass) :-
  visitor(Visitor, VisitedClass, _, _).

% dynamic results
visitor(VisitorClass, VisitedClass) :-
  visits_from_till(VisitorInstance,VisitedInstance, _, _),
  instanceOf(VisitorInstance, VisitorClass),
  instanceOf(VisitedInstance, VisitedClass).
```

There is already a weight associated with the solutions of the dynamic Visitor detection rule due to the use of the fuzzy forall predicate and the approximate unification on the accept selector. But this is not the only way in which weights can be used to differentiate solutions of rules. We can also construct a preference hierarchy among the different rules we devised earlier as they can be considered implementations of different heuristics to limit the number of false positives:

- Our original rule didn't impose any restrictions on the selectors of the methods involved in the double dispatching operation. Therefore, it is reasonable to assign it an arbitrary weight of merely 0.9.
- We can significantly increase the accuracy of solutions to this rule by binding its `Acceptselector` variable to the `'#accept:'` selector. This represents the most limiting heuristic we considered and as such it is safe to assume this rule has an associated absolute certainty of 1 which will cause Visitor design pattern instances following this common naming convention to be detected with a higher confidence degree.
- The variant of our rule that incorporated approximate unification on the accept selector to keep the ability to detect visitor design pattern instances which don't follow the naming conventions closely could be assigned an intermediary weight of 0.95.

It is important to keep in mind that these rule weights will still be decreased by the degree up to which the visitation of all the objects in the visited composite structure will be visited. The certainty of solutions to the last rule will in addition depend on the unification degree of the accept selector.

The weighting technique can also be applied to incorporate a similar heuristic into the object instantiation rule we discussed in section 4.1.3.1 by extending its definition with clauses that check whether the creational selector is defined in an instance creation protocol:

```
  ...
  selectorInProtocol([?stclass class],?protocol,?selector),
  instanceCreationProtocol(?protocol)
```

5.5 Conclusion

In this chapter we have compared our new library for dynamic analysis with the existing LiCoR library for structural code reasoning.

We have shown how the existing Visitor design pattern rule which relies on static information is only able to detect archetypical concrete source code implementations of the pattern's architecture. We proposed a dynamic rule which detects the Visitor design pattern by looking for instances of the run-time behaviour shared by all common variants of the pattern. This allowed more complex implementations of the Visitor pattern to be detected by one and the same declarative rule. We were for instance able to detect instances of the pattern where the control over the descent in the composite structure is controlled by the structure itself instead of by the visitor. This was an example of a case where the static alternative failed.

We realised that our succinct dynamic Visitor detection rule was an example of an overly idealised declarative rule which is one of three commonly occurring problems with declarative meta-programming in general: handling overly idealised rules, expressing inherently vague software concepts and overcoming small discrepancies between wanted information and the program facts at hand. We gave an example of each of these problems and showed how approximate reasoning support in the base declarative meta-programming language can help these overcome.

Our dynamic visitor detection rule was a fine example of an overly idealised declarative rule and we showed how this problem can be overcome by interpreting the rule in our approximate base language where a fuzzy variant of the forall predicate is available. This way, the same succinct rule could be used to detect imperfect instances of the pattern without having to sacrifice the rule's succinctness.

We also showed how, using approximate reasoning, inherently vague concepts such as bad smells can be expressed in a natural way. The flexible interpretation of the classification boundaries present in many of these software patterns allowed our framework to detect indications of bad design that went undetected in the crisp logical setting.

Another important problem in declarative meta-programming is how to overcome small discrepancies between the facts demanded by a logic rule and the program facts at hand. Approximate reasoning provides similarity-based unification methods to resolve this problem by unifying different, but still similar program facts with a lower degree of certainty. We integrated this feature in our existing Visitor detection rule and identified some of the problems encountered when a purely syntactical similarity relation is used on method parse trees.

Our final exploration of the use of approximate reasoning in a software re-engineering setting comprised combining dynamic and statically obtained information and, more generally, the imposition of a preference hierarchy among rules using different heuristics to detect instances of the same pattern.

As our experiments represent initial explorations on the applicability of approximate reasoning to declarative meta-programming in a software re-engineering setting, we have along the way identified many interesting topics of future work.

Chapter 6

Conclusions

6.1 Summary

In chapter two, we started our discussion with an overview of the broader context of this dissertation: using declarative meta-programming to support the highly complex and iterative software re-engineering process which comprises three main phases: design recovery, identification of design defects and source code transformations to alleviate any discovered defects.

We then set out to describe the existing tools for software re-engineering focusing on recent developments employing logic programming, constraint logic programming and pattern matching. Some of these systems incorporate only static information about an application's source code while others analyse a system's run-time behaviour using dynamic information and a third kind tries to combine both in a rudimentary way.

Static analysis offers complete information about a program's source code, but obtaining correct information is difficult due to polymorphism, late binding and inheritance which are prevalent in today's object-oriented programming languages. Information from a dynamic analysis is however always correct with respect to a chosen execution scenario, but dynamically obtained control and data-flow information may change with the executed scenario.

We hypothesised that a framework in which both forms of analysis are possible opens doors to complex reasoning patterns about source code and run-time behaviour. As such, providing a dynamic alternative to SOUL's existing static reasoning provisions formed one of the two main pillars throughout this dissertation.

We have detailed the theoretical background on approximate reasoning in chapter three starting with the cornerstone of existing techniques for modelling vague concepts: fuzzy sets which are based on a natural extension of the set membership function to allow gradual membership. We continued with a detailed account on existing fuzzy logic programming languages in which the ordinary resolution procedure is extended to incorporate partial truths. We also explored the possibilities for extending the unification procedure based on semantical or syntactical similarity measures. An extension of the Levensthein edit distance for strings to logic terms was given as an example of a

purely syntactical similarity measure.

In chapter four, we discussed the structure of our declarative framework for dynamic analysis. It is organised in a layered manner where predicates in the top layers depend on the predicates defined in the layers below. The representational layer contains predicates for reifying the source model, a program's execution history consisting of ordered run-time events. The source model is in contrast to existing approaches fine-grained, which is necessary to allow complex patterns to be expressed using data flow information obtained from method arguments, returned values and variable assignments.

We have implemented and compared two alternatives for collecting these events. In a post-mortem analysis, the entire application is run during which all run-time events are collected. The reasoning process doesn't start until the program has ended at which point we have the entire execution history at our disposal. In the ad-hoc analysis variant, the execution of the application and the evaluation of the logic program are interleaved as co-routines. The logic program requests a particular run-time event which causes the application to be run until the requested event is encountered. Control then returns to the reasoning process. At no time during the program's execution, we have the entire execution history at our disposal. We can only access the run-time event that was last encountered.

The basic layer contains definitions for often-used queries which offer information that is hard or impossible to obtain statically such as object instantiations, object state tracking or a precise classification of binary class associations.

We concluded the chapter on our new declarative framework with a walk-through of the features our base logic programming language exhibits for supporting approximate reasoning.

In chapter five, we detailed how dynamic analysis allows us to express complex software patterns such as the Visitor design pattern in an alternative way by looking for straightforward translations of their sequence diagrams in a program's run-time behaviour instead of searching for archetypical implementations of their architecture in a program's source code.

We also identified three often re-occurring problems in declarative meta-programming: abstract software patterns described in the form of overly-idealised rules which are too far from concrete real-life implementations, software patterns that describe inherently vague concepts and small discrepancies between a wanted fact and the facts at hand. For each of these problems, we described how our experiments hint that approximate reasoning can provide a rudimentary solution in a theoretical framework.

6.2 Conclusions

In the introduction we have made two claims: one about the need for approximate reasoning support in a declarative meta-programing language and one about dynamic analysis complementing SOUL's already existing library for static analysis. To support these claims, we have developed a declarative framework incorporating both approximate reasoning and dynamic analysis and begun exploring the possibilities of each

pillar from our approach for supporting the software re-engineering process.

First of all, declarative meta-programming can benefit from built-in support for approximate reasoning in the base declarative programming language. We have identified three commonly occurring problems in declarative meta-programming and proposed a rudimentary solution for each of them using techniques from the approximate reasoning domain. Our results are still preliminary since we have only started an initial exploration of all the possibilities, but they are already encouraging nonetheless.

Overly Idealised Rules A first problem is that of logic rules describing software patterns in an overly idealised manner leaving a large discrepancy between the abstract concept described by the rule and the concrete instantiation of this concept in the program's source code or behaviour. Introducing a partial truth for clauses in a rule assists in overcoming this discrepancy as the rule's constituents aren't longer validated on a black-or-white basis.

Our overly idealised Visitor design pattern detection rule failed on incompletely visited composite structures within a crisp logic programming setting, but proved to remain useful when interpreted in our approximate logic programming language which supports a fuzzy `forall/2` predicate.

Vague Software Patterns The second problem is situated in the original application domain of fuzzy set: vague concepts. Some software concepts are inherently vague and thus can't be described in an expressive and straightforward manner in a logic programming language without implicit support for vague concepts. One of the most straightforward examples is that of a pattern containing vague numeric classification boundaries such as many bad smells. We have applied fuzzy comparison operators and fuzzy sets in our definition of the "too many instance variables" bad smell.

Overcoming Small Discrepancies In many declarative meta-programming rules, facts from the source model need to be unified with a particular pattern in which we're interested. Often, while the exact fact we're looking for can't be found in the background knowledge, similar facts may be present. Approximate reasoning provides similarity-based unification as a theoretical framework for handling partial semantic or syntactic similarities defined on logic terms. Straightforward examples are variations on the accept-selector in the Visitor design pattern which are semantically similar but completely different judged on a syntactical basis.

Although not a common problem in declarative meta-programming, approximate reasoning also provides a sound theoretical background for assigning different weights to different rules or heuristics describing the same software pattern. In our setting, this feature allows us to combine static and dynamic information to augment the certainty of a solution obtained through one analysis variant by incorporating additional information from the alternative data source.

The second and most thoroughly explored part of our work involves declaratively reasoning about a program's behaviour using dynamic analysis.

Our experiments have shown that a fine-grained source model allows complex behavioural patterns to be expressed relying on object states, method invocation arguments and method return values. This kind of analysis goes beyond the mere discovery of patterns in execution traces only comprised of method invocation events. Many of these patterns are difficult or even impossible to express statically as object-oriented language features such as polymorphism, late binding and inheritance make it hard to analyse an application's control and data flow exactly from source code.

The basic layer from our framework already provides useful predicates for performing low-level analyses over execution traces. We are for instance able to verify the value returned by complex method chains, detect when a particular value of an instance variable gets overridden by another one and analyse the exact nature of binary class relationships.

We implemented and compared two variants of dynamic analysis: ad-hoc and post-mortem analysis where the latter reasons about a program's behaviour after the program has ended and the former interleaves the execution and analysis of the program.
Our experiments have shown that the ad-hoc analysis variant provides a lightweight alternative for the evaluation of simple rules that don't require backtracking over run-time events such as many of the low-level predicates from the base layer. It also provides a rudimentary way to modulate a program's behaviour in a declarative way.
The post-mortem analysis variant is in contrast suitable for solving complex queries that require the entire execution history at the cost of a high performance and memory overhead.

We have shown that straightforward logic transcriptions of the sequence diagrams associated with design patterns offer flexible alternatives to detecting instances of these patterns in a program's behaviour. These rules aren't limited to detecting archetypical source code implementations of a pattern's structural architecture, but can find many implementation variants sharing the same run-time behaviour. Our dynamic Visitor detection rule was even able to detect instances of the pattern where the control over the object structure descent was located in the composite instead of in the Visitor. This is an example that the static version of the rule failed to detect.

To conclude, we believe that our library for behavioural program analysis exhibits a distinct expressiveness advantage over the existing library for structural source code reasoning as it is more natural to think in terms of the concepts central to object-oriented programming, namely objects interacting with each other through message sends, than to think in terms of class structures and concrete method statements as we are forced to do in static analysis tools.

As our experiments have shown that dynamic analysis is well-suited for reasoning about a program's behaviour while static analysis is fit for reasoning about a program's architectural structure, neither may be omitted from declarative meta-programming tools supporting the software re-engineering process. In addition, we believe after our initial elementary experiments that increasing the expressiveness of the base declarative meta-programming language by including support for approximate reasoning may open doors to a more human-like detection of even more complex and vague software patterns.

6.3 Future Work

This dissertation has focused primarily on the implementation of a general declarative framework incorporating dynamic analysis and approximate reasoning. In addition to an initial exploration of the use of approximate reasoning techniques in declarative meta-programming, we have extensively studied the use of dynamic analysis in the context of software re-engineering and its design recovery phase in particular.

Possible future work includes investigating more specific applications of dynamic analysis in declarative meta-programming and its integration in industrial development tools. A natural extension of the ad-hoc dynamic analysis variant is a declarative debugging tool which allows programmers to declaratively specify dynamic breakpoints on run-time events (such as *"a variable is assigned a value that indirectly originated –possibly through intermediary assignments and message sends– from the return value of a certain method invocation"*) after which either the program could be inspected on an event-per-event basis or continued until another dynamic breakpoint fires.

Another point of further investigation is how ad-hoc analysis could take advantage of the reflective language features of an application's implementation language. In the case of Smalltalk this would allow us to reason about and such low-level concepts as method contexts and call stacks.
A related path to pursue in an extended study of ad-hoc analysis is how a program's behaviour can be altered. It would also involve the implementation of an event-based aspect-oriented programming system where joinpoints could be specified in a declarative manner.

Another interesting research topic comprises collecting run-time events without having to execute fixed execution scenarios. Abstract interpretation [JGS93] could allow the extraction of run-time events without knowing the exact value of the objects involved in method invocations as these are normally provided by the execution scenario. Approximate reasoning could also be needed for further reasoning with these unknowns.

As we have developed our library of predicates for dynamic analysis in SOUL, we are using backward reasoning common to all Prolog-like logic programming languages. It might be interesting to see how forward reasoning performs in our application domain especially when we are faced with large volumes of run-time events. Also, more experiments are needed to evaluate the efficacy and performance of our dynamic analysis library on industrial-sized applications.

Static analysis in SOUL has until now only been applied to class-based object-oriented languages such as Smalltalk and Java, but since dynamic analysis lends itself naturally to express patterns about instances and message sends, it may also lead to the application of declarative meta-programming to prototype-based languages.

As no previous work exists to our knowledge on the application of approximate reasoning to declarative meta-programming, we have only been able to brush the surface of possible applications and many open questions remain.

A common problem in approximate reasoning is the interpretation of the obtained truth values. In the software re-engineering setting, these might be used by tools to indicate

for instance the particular “badness” of a bad smell through different colours in the source code editor.

We believe that software patterns in which many unknowns can be determined up to a degree of certainty using different declaratively codified heuristics are particularly interesting application domains. Examples comprise approximate type derivations in dynamically typed languages.

Whether employing approximate reasoning techniques to common problems such as overly idealised rules and vague concepts won’t significantly increase the amount of false positives up to a point where analysis rules lose their relevance, is another important question that needs to be resolved.
Another interesting problem comprises finding a suitable similarity relation for method statements: either a purely syntactical one or a semantical one where the semantics of methods could be compared using dynamic analysis on run-time events obtained in absence of execution scenarios through the aforementioned abstract interpretation technique.

Appendix A

Extracting Run-time Events

In order to collect the events generated during a program's execution, we will have to alter the source code or byte code of the methods we are interested in. This process is called *instrumentation*. In contrast to uninstrumented methods, instrumented methods will trigger events during their execution. In the case of a post-mortem analysis, these events are collected and transformed to SOUL facts when the program finishes while in the case of an ad-hoc analysis, individual events are requested one-by-one as the analysis of the program progresses. The transformation of the method's source code is however the same in both variants of dynamic analysis. This chapter discusses the various techniques we considered for implementing instrumentation.

A.1 Method Wrappers

The Method Wrapper technique [BFJR98] allows introducing new behaviour (e.g. generating run-time events) that is executed around or in the place of an existing method. Smalltalk developers traditionally change the lookup process to implement such functionality while this approach modifies the objects returned by the lookup process instead.

Methods are represented in Smalltalk as instances of the *CompiledMethod* class, which contains a pointer to the position of the method's source code in the Smalltalk image, an integer representing the compiled byte codes and an instance variable representing the class that compiled this method. The methods belonging to a class are kept together in a *MethodDictionary* filled with *CompiledMethod* instances.

Method Wrappers could be implemented by modifying the method's source code to include the code that needs to be executed before and after the method call, but this would imply that the method has to be recompiled when the wrapper is installed and this may take too long for large programs. Instead, the method wrapper technique replaces the *CompiledMethod* instance in a class' *MethodDictionary* by a *MethodWrapper* class which holds a reference to the original *CompiledMethod* object. When the *MethodWrapper* instance is invoked, it will simply forward the call to its *CompiledMethod* instance and execute its before and after code. This setup is shown in Figure A.1.
The major advantage of this technique is that it is reasonably fast at run-time and doesn't require an instrumented method to be recompiled. Since it doesn't alter the

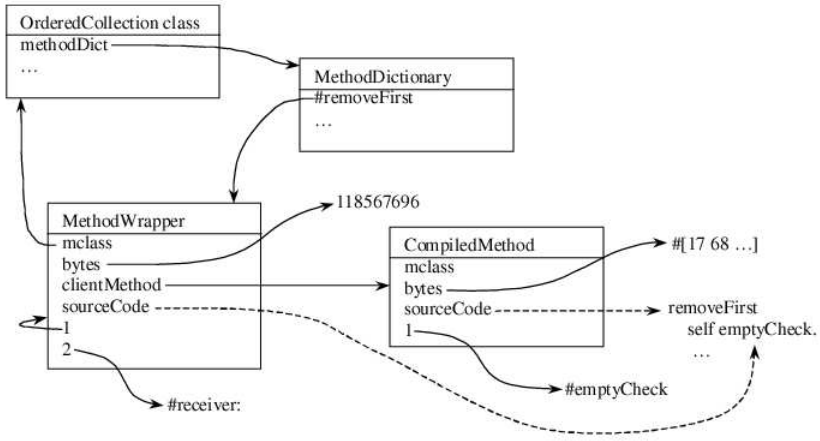

Figure A.1: A method wrapper installed on the `OrderedCollection>>removeFirst` method [BFJR98].

source code of the wrapped method itself, the Smalltalk class browser shows the original source code and thus prevents users getting confused by automatically inserted before and after statements woven through their own source code.

Richner's tool for automated software re-engineering described in section 2.2.2 deploys method wrappers in its instrumentation process.

However, with this technique it isn't possible to trace anything except method invocations so we abandoned it as soon as we identified the need for variable assignment tracking.

A.2 Aspect-Oriented Programming

Aspect-Oriented Programming is *an approach for realizing separation of concerns and allows different concerns to be weaved into existing applications* [HU]. Concerns usually cross-cut the entire architecture which can clutter an application's implementation. Examples of concerns are for example the correct opening and closing of network connections throughout a program. Aspect-oriented languages allow these concerns to be expressed in a clean way.

We could use aspects to support tracing by essentially weaving the required functionality through an existing program. We have not pursued this path further because not many Smalltalk implementations exist and the higher-level aspect abstraction functionality of aspects (such as manipulation and instantiation) are a bit overkill for simple method instrumentation.

A.3 Parse Tree Rewriting

A.3.1 Description

Finally, we have opted for an approach in which we recompile the classes we are interested in with a modified compiler that rewrites method parse trees in order to insert instrumentation code which allows tracking the variable assignments too. We do not change the actual source code of the method, but only output byte codes different from a normal compiler. Apart from a special icon which indicates that a class is being traced, the user's view of a method's source code in the class browser is left intact.
The source code below is an extract from the *DDTestInvoker* class:

```
bar: aNumber
  | foo |
  Transcript show: 'foobar'.
  foo := 2 + aNumber.
  ^foo
```

When we compile this method with our `TraceCompiler`, its parse tree is rewritten at compile time before we handle it over to the regular Smalltalk compiler. We can ask Smalltalk to decompile the resulting byte codes into regular source code and obtain the following result for the `DDTestInvoker>>bar:` method:

```
bar: t1
  | t2 |
  Tracing.Trace
        method: #bar:
        sender: thisContext sender
        receiver: self
        arguments: ((OrderedCollection new) add: t1; yourself).

  Transcript show: 'foobar'.

  t2 := Tracing.Trace
          variable: 'foo'
          object: self
          method: #bah:
          value: 2 + t1.

  ^Tracing.Trace return: t2
```

This example shows how the `Tracing.Trace` class is notified of each of the three types of execution events:

Method invocations The central `Trace` class is notified of method invocations by a `method:sender:receiver:arguments` call which is placed at the beginning of each traced method.

The first argument contains the selector of the invoked method. The sender of the message is obtained at run-time by the `thisContext sender` statement. The receiver of the message call is recorded in the `receiver:` argument while the arguments to the call are collected in the `arguments:` argument.

Variable assignments The right hand-side of a variable assignment is replaced by a call `variable:object:method:value` to the `Trace` object which returns the value of the original right-hand side expression.

The arguments of the call are comprised of (from left to right) the name of the assigned variable, the object and method in which the assignment happens and, to conclude, the assigned variable.

Method returns Return statements are replaced by a call `return:` to the Trace object with the returned value as argument.

Due to the experimental nature of our implementation, the above transformation scheme ignores method exits that were caused by a raised exception. The implementation can however be changed in a straightforward manner to correctly handle this kind of exit by wrapping method bodies in a block to which an `ensure:` message is sent with code to notify the `Trace` class of the returned value as its argument.

A.3.2 Selecting the Appropriate Compiler

For the actual implementation of this scheme, we must first of all be able to change the compiler a class is normally compiled with. We accomplish this by means of Smalltalk's reflective capabilities described in the outline by Foote and Johnson [FJ89].

To summarise, when a new method must be compiled for a certain class, the class is asked what compiler should be used through the `Behavior>>compilerClass` method. The `Behavior` class defines the minimal behaviour of all Smalltalk classes.

If we were to change the default compiler in this method, it would affect all classes in the class hierarchy beneath the class whose behaviour we would like to monitor. As this does not allow a selective instrumentation on a class-per-class basis, we have to raise this method one level to the `Metaclass` [1] class instead:

```
Behavior>>compilerClass
  ^self class traceCompiler
```

In the above code, we ask a class for its class and receive a `Metaclass` instance on which we invoke the message `traceCompiler`. This method is implemented in the `MetaClass` class and its subclass `TraceMetaclass`:

```
Metaclass>>traceCompiler
  ^Compiler

TraceMetaclass>>traceCompiler
  ^TraceCompiler
```

If we want to instrument a class by compiling it with the `TraceCompiler`, we simply have to change its meta-class to the `TraceMetaClass`. Users can do this by sending the `swapTracing` message to the class whose execution they want to trace.

[1] In the Smalltalk Meta Object Protocol, classes are instances of class `Metaclass`

```
Metaclass>>swapTracing
        self changeClassToThatOf: TraceMetaclass new

TraceMetaclass>>swapTracing
        self changeClassToThatOf: Metaclass new
```

In addition, we have also defined a little icon in the `TraceMetaClass` that is shown in the Smalltalk class browser alongside traced classes.

A.3.3 Rewriting the Parse Trees

The `TraceCompiler` class is a subclass of the default Smalltalk `Compiler` class. It overwrites the `compile:in:notifying:ifFail` method used to compile methods.

The compilation begins by constructing a parse tree of the given source code using the `RBParser` class defined in the Refactory Browser package. It then lets a `TraceInstrumentationVisitor` do the actual parse tree rewriting on the obtained tree. The result is finally given to the `compile:in:notifying:ifFail` method defined in its superclass to continue the normal compilation process on the transformed tree.

```
TraceCompiler>>compile: textOrStream
               in: aClass
               notifying: aRequestor
               ifFail: failBlock
        |tree|
        tree := Refactory.Browser.RBParser
                  parseMethod: textOrStream.
        TraceInstrumentationVisitor new
                  instrument: tree.
        ^super compile: (tree formattedCode)
               in: aClass
               notifying: aRequestor
               ifFail: failBlock
```

The following source code extract illustrates how the `TraceInstrumentationVisitor` replaces each `RBReturnNode` instance in the parse tree by a`RBMessageNode` instance which results in the parse tree transformation shown in the beginning of appendix A.3.1.

```
acceptReturnNode: aReturnNode
  super acceptReturnNode: aReturnNode.
  aReturnNode replaceWith:
    (Refactory.Browser.RBReturnNode
       value: (Refactory.Browser.RBMessageNode
                 receiver: (Refactory.Browser.RBVariableNode
                              named: 'Tracing.Trace')
                              selector: #return:
                              arguments: (Array with: aReturnNode value)))
```

Appendix B

Implementation of Declarative Object State Tracking

Keeping track of the different states an object goes through during a program's execution is a challenging problem. We could use a database filled with serialised versions of each object at a particular moment, but such a database would be massive and serialisation of random objects isn't easy nor efficient since we will have to make deep copies of all of the objects involved.

Instead, we chose to log all variable assignments and declaratively calculate the values of a variable at each moment in time. To understand the definition of the rules below, we recall that a variable assignment event happens whenever a variable is assigned within a method. The recorded name is thus either a temporary variable in the invoked method or an instance variable of the object to which the message was sent (method arguments can be excluded). It is impossible for a symbol to designate an instance variable and a method temporary variable at the same time. We only have to ensure that the lifetime of a temporary variable ends with the method return while an instance variable is permanent during the object's existence.

For this reason, the `?variable in: ?instance at: ?methodInvocationNumber value: ?value` predicate, which states that a `?variable` related to a particular `?instance` contained a `?value` at the end of the method invocation identified by a `?methodInvocationNumber`, is split in two:

```
?variable in: ?instance at: ?methodInvocationNumber value: ?value  if
  not(?variable isInstVarOf: ?instance),
  sequenceTotal(?max),
  constrain(?methodInvocationNumber,[1 to: ?max]),
  ?a assignmentDuring: ?methodInvocationNumber ,
  not(? overridesInternalAssignment: ?a with: ?),
  equals(?a,assignment(?,?methodInvocationNumber,?instance,?variable,?value))

?variable in: ?instance at: ?methodInvocationNumber value: ?value  if
  ?variable isInstVarOf: ?instance,
  sequenceTotal(?max),
  constrain(?methodInvocationNumber,[1 to: ?max]),
```

```
  ?a assignmentPrecedes: ?methodInvocationNumber,
  not(? overridesInternalAssignment: ?a with: ?),
  not(? overridesExternalAssignment: ?a with: ? limit: ?methodInvocationNumber),
  equals(?a,assignment(?,?,?instance,?variable,?value))
```

In order for the lower rule to hold, the `?methodInvocationNumber` must lie between 1 and the program's end, and a variable assignment must have occurred before the chosen `?methodInvocationNumber`. This assignment may however not be overridden by another assignment in the same method invocation event nor be superseded by any later assignment to the variable before the chosen `?methodInvocationNumber`. If the assignment is in fact the last assignment in or before the method invocation identified by the chosen `?methodInvocationNumber`, the assigned value is the value of the variable after the method invocation.

The sub-goal `overridesExternalAssignment` is defined as follows:

```
?x overridesExternalAssignment: ?a with: ?v limit: ?endMN  if
  equals(?a,assignment(?sn,?mn,?instance,?variable,?)),
  equals(?x,assignment(?sn2,?mn2,?instance,?variable,?v)),
  not(equals(?mn2,?mn)),
  greater(?sn2,?sn),
  smallerOrEqual(?mn2,?endMN)
```

while the sub-goal `overridesInternalAssignment` is defined similarly.

Appendix C

Example Execution Trace

The following is an example trace of the Visitor design pattern discussed in section 5.1.2 on the example object structure shown in figure 5.2.

```
:- discontiguous bexit/2.
:- discontiguous bassignment/5.
:- discontiguous bdirect/5.
methodEntry(1, 0, 1, '#new', [] ).
  methodEntry(2, 1, 2, '#initialize', [] ).
    assignment(3, 2, 2, treeRoot, 0 ).
    methodEntry(4, 2, 3, '#new', [] ).
      methodEntry(5, 3, 4, '#initialize', [] ).
        methodEntry(6, 4, 5, '#new', [] ).
          methodEntry(7, 5, 6, '#initialize', [] ).
            assignment(8, 7, 6, test3, 7 ).
          methodExit(10, 7, 6 ).
        methodExit(11, 6, 6 ).
        assignment(12, 5, 4, test2, 6 ).
      methodExit(13, 5, 4 ).
    methodExit(14, 4, 4 ).
    assignment(15, 2, 2, treeRoot, 4 ).
  methodExit(16, 2, 2 ).
methodExit(17, 1, 2 ).
methodEntry(18, 0, 2, '#start', [] ).
  assignment(19, 18, 2, visitor, 9 ).
  methodEntry(20, 2, 4, '#accept:', [9] ).
    methodEntry(21, 4, 6, '#accept:', [9] ).
      methodEntry(22, 6, 7, '#accept:', [9] ).
        methodEntry(23, 7, 9, '#visitParagraph:', [7] ).
        methodExit(24, 23, 10 ).
      methodExit(25, 22, 10 ).
    methodEntry(26, 6, 9, '#visitSection:', [6] ).
    methodExit(27, 26, 10 ).
    methodExit(28, 21, 10 ).
    methodEntry(29, 4, 9, '#visitChapter:', [4] ).
    methodExit(30, 29, 10 ).
```

```
  methodExit(31, 20, 10 ).
methodExit(32, 18, 2 ).
sequenceTotal(32).
```

In the trace, the following integer-to-object mappings exist:

0	`nil`
1	`ChapterVisitorInvoker`
2	`a ChapterVisitorInvoker`
3	`Chapter`
4	`a Chapter`
5	`Section`
6	`a Section`
7	`a Paragraph`
8	`a ByteString`
9	`a ChapterVisitor`
10	`a True`

Bibliography

[AA01] Hervé Albin-Amiot. JavaXL, a Java source code transformation engine. Technical Report 2001-INFO, École des Mines de Nantes, 2001.

[AACGJ01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the 16^{th} conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

[AF99] F. Arcelli and F. Formato. Likelog: a logic programming language for flexible data retrieval. *ACM Symposium on Applied Computing*, pages 260–267, 1999.

[AJ74] Wagner R. A. and Fischer M. J. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.

[Als01] Teresa Alsinet. *Logic Programming with Fuzzy Unification and Imprecise Constants: Possibilistic Semantics and Automated Deduction*. PhD thesis, Universitat Politécnica De Catalunya, May 2001.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. *Lecture Notes in Computer Science*, 1445:396–??, 1998.

[Bra03] Francisca Munoz Bravo. A logic meta-programming framework or supporting the refactoring process. Master's thesis, Vrije Universiteit Brussel - Belgium in collaboration with Ecole des Mines de Nantes - France, 2003.

[Cin01] Petr Cintula. About axiomatic systems of product fuzzy logic. *Soft Computing*, 5:243–244, 2001.

[CL96] Yves Caseau and François Laburthe. Claire: Combining objects and rules for problem solving. In Yike Guo, Jose Meseguer, Tetsuo Ida, and Joxan Jaffar, editors, *proceedings of the JICSLP workshop on Multi-Paradigm Logic Programming*, pages 105–114. Technischen Universität Berlin, September 1996. Technical report 96-28.

[DDVMW00] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the international symposium on Software Architectures and Component Technology 2000.*, 2000.

[DMS01] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[DP80] Didier Dubois and Henri Prade. *Fuzzy Sets & Systems: Theory and Applications*, volume V.144, 393 p. Academic Press, New York, mathematics in science and engineering series edition, 1980.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In Norman Meyrowitz, editor, *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 327–335. ACM Press, 1989.

[Fla94] P. A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

[Fow97] Martin Fowler. *UML Distilled*. Addison Wesley, 1997.

[FSCdS94] Daniela V. Carbogim Flávio S. Correa da Silva. A system for reasoning with fuzzy predicates, 1994.

[GA98] And L. Godo and T. Alsinet. Fuzzy unification degree. August 11 1998.

[GAA01] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the 17th conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GJ01] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *proceedings of the 1st IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[Goe33] K. Goedel. Zum intuitionistischen aussagenkalkul. *Ergebnisse eines mathematischen Kolloquiums*, 4:34–38, 1933.

[Gog67] J. Goguen. L-fuzzy sets. *Journal of Mathematical Analysis and Applications*, 18:145–174, 1967.

[GS00] David Gilbert and Michael Schroeder. FURY: Fuzzy unification and resolution based on edit distance. In *IEEE International Conference on Bioinformatics and Biomedical Egineering*, pages 330–336, 2000.

[Gué02] Yann-Gaël Guéhéneuc. Three musketeers to the rescue – Metamodelling, logic programming, and explanation-based constraint programmingtest for pattern description and detection. In Kris De Volder, Kim Mens, Tom Mens, and Roel Wuyts, editors, *proceedings of the 1st ASE workshop on Declarative Meta-Prorgramming*. Computer Science Department, University of British Columbia, September 2002.

[Gué03] Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. PhD thesis, École des Mines de Nantes, juin 2003.

[H́98] Petr Hájek. Basic fuzzy logic and bl-algebras. *Soft Computing*, 2:189–212, 1998.

[HG] Petr Hájek and Lluis Godo. Deductive systems of fuzzy logic. To appear in Tatra Mountains Mathematical Publications.

[HGE96] Petr Hájek, Lluis Godo, and Francesc Esteva. A complete many-valued logic with product-conjunction, 1996.

[HU] Stefan Hanenberg and Rainer Unland. Grouping objects using aspect-oriented adapters.

[Hyd02] Dominic Hyde. Sorites paradox. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2002.

[IK85] Mitsuru Ishizuka and Naoki Kanai. Prolog-ELF incorporating fuzzy logic. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 701–703, Los Angeles, CA, August 1985. Morgan Kaufmann.

[JB00] Narendra Jussien and Vincent Barichard. The PaLM system: Explanation-based constraint programming. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[Joh92a] Ralph Johnson. HotDraw (abstract): A structured drawing editor framework for Smalltalk. In *Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Lanuages, and Applications*, page 232, 1992.

[Joh92b] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[KC99] E. E. Kerre and M. De Cock. Linguistic modifiers: An overview. *In G. Chen, M. Ying, and K.-Y. Cai, editors, Fuzzy Logic and Soft Computing*, pages 69–85, 1999.

[KK94] Frank Klawonn and Rudolf Kruse. A łuckasiewicz logic based prolog. *Mathware and Soft Computing*, 1:5–29, 1994.

[Kle82] E.P Klement. Construction of fuzzy σ-algebras using triangular norms. *Journal of Mathematical Analysis and Applications*, 85:543–565, 1982.

[Lee72] Richard C. T. Lee. Fuzzy logic and the resolution principle. *Journal of the ACM*, 19:109–119, 1972.

[Lin65] C.H. Ling. Representation of associative functions. *Publicationes Mathematicae Debrecen*, pages 189–212, 1965.

[LL89] D. Liu and D. Li. A new fuzzy inference language f-prolog. *Computer Engineering*, 1:23–27, 1989.

[LL90] Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

[MDC01] E. E. Kerre M. De Cock, Zabokrstky. Modelling linguistic hedges by l-fuzzy modifiers. *Proceedings of CIMCA'2001 (International Conference on Computational Intelligence for Modelling Control and Automation)*, pages 64–72, 2001.

[MMW01] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[MY60] R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, March 1960.

[NP00] V. Novák and I. Perfilieva. Some consequences of herbrand and mcnaughton theorems in fuzzy logic. *Discovering the World with Fuzzy Logic*, pages 271–295, 2000.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, 1999.

[RD01] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. Technical Report IAM-01-007, 2001.

[RDW98] Tamar Richner, Stéphane Ducasse, and Roel Wuyts. Understanding object-oriented programs with declarative event analysis. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP '98 Workshop Reader)*, LNCS 1543. Springer-Verlag, July 1998.

[Ric02] Tamar Richner. *Recovering Behavioral Design Views: a Query Based Approach*. PhD thesis, Universität Bern, Philosophisch-naturwissenschaftlichen Fakultät, Bern, Swiss, May 2002.

[Ses02] Maria I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275:389–426, 2002.

[SS63] B. Schweizer and A. Sklar. Associative functions and abstract semigroups. *Publicationes Mathematicae Debrecen*, pages 69–81, 1963.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.

[TL98] T.Alsinet and L.Godo. Fuzzy unification degree. *Logic Programming and Soft Computing - Theory and Applications, A Post-conference Workshop of JICSLP'98*, 1998.

[VGoH02] C. Vaucheret, S. Guadarrama, and S. Mu oz Hernández. Fuzzy prolog: A simple general implementation using $clp(r)$. *Proceedings of LPAR 2002. Lecture Notes of Artificial Intelligence 2514: Logic for Programming, Artificial Intelligence, and Reasoning.*, pages 451–463, 2002.

[WD01] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

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

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

[Zad65] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad75a] L. Zadeh. The concept of a linguistic variable and its application to linguistic reasoning. *Information Sciences*, pages 8: 199–249, 301–357, 9: 43–80, 1975.

[Zad75b] Lotfi Zadeh. Fuzzy logic and approximate reasoning. *Synthèse*, 30:407–428, 1975.