Vrije Universiteit Brussel
Programming Technology Lab
Faculty of Science - Department of Computer Science
Academic year 1999 - 2000

A Reflective Forward-Chained Inference Engine to Reason about Object-Oriented Systems

Sofie Goderis

Promotor: Prof. Dr. Theo D'Hondt

Thesis submitted in partial fulfillment of the requirements for the degree of Licenciaat in de Informatica

Abstract

Recently, a lot of research concerning co-evolution has been done. Co-evolution is a brand new approach that tries to find a solution for the problem of the missing link between the design and the implementation of object-oriented software systems. To date this has been done by using a rule-based backward chained reasoning mechanism as a declarative meta layer on top of the implementation of an object-oriented system. However, one of the major drawbacks of this approach is the fact that a goal should be clearly specified. Such a (specification of the) goal is very often not available.
In this dissertation we will show that in some specific cases a rule-based backward chained reasoning mechanism is not sufficient. Therefore, as an alternative mechanism we suggest a rule-based forward chained reasoning mechanism to implement the declarative meta layer, in order to handle these specific cases. We will validate this thesis by means of some experiments done with a forward chained prototype we built. The prototype is based on the expert system tool KAN and implemented in Squeak. The conducted experiments confirm our thesis. Especially in cases where no specific goal can be specified and in cases where the reasoning state has to be preserved, our alternative approach proved to be preferable.

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation would never have been realised without the tremendous support which was given to me. Therefore, I wish to express my gratitude towards :

Prof. Dr. Theo D'Hondt for promoting this dissertation.

Wolfgang De Meuter who came up with the subject and who helped me through every stage of this work from preparation through implementation and writing to proofreading. His comments and professional advise were very supportive and constructive.

Bart Wouters and Maja D'Hondt for proofreading my work and giving valuable advice.

The researchers of the Programming Technology Lab for their listening and comments during the weekly thesis presentations.

Jim Walker for his linguistic evaluation of this dissertation.

My parents for the opportunity they gave me to study at a good university in the best possible circumstances, who believed in me and were my best supporters all the way long.

The Vrije Universiteit Brussel for the excellent education.

Chapter 1

Introduction

Over the last two decades, the object-oriented paradigm has become increasingly popular and widespread in the industry. Although this paradigm itself is not that recent anymore, and despite major research that has been carried out, object-oriented programming is still quite "new" to the larger public. Nevertheless, it keeps growing in popularity and is more and more used in the industry. In the meantime however, the researchers have not stopped their work.

In the same period of time, systems have grown bigger and more complex and the task to maintain them and to keep a global view on them, has become more difficult. Therefore, the need for more advanced tools arose and in order to assist with designing object-oriented systems, tools like UML ([Boo94, BRJ97]) came into existence. Furthermore, to allow the programmer to create better and more reusable code, techniques like frameworks ([CDSV97]), design patterns ([GHJV94]) and contracts ([HHG90]) have proved their usefulness and many of these techniques are now integrated into advanced system browsers. It is clear that these new techniques provide great help and tools for programming, however, a deficiency still exists and is experienced : that is, the link between design and implementation is missing.

During the development of a system, an analysis and a design are made prior to the implementation of the system, but after the implementation, the design still is useful to understand the implementation as it provides a model on the system. Models are crucial because they help the software engineer to evolve the software as it was intended ([DH98]), as they give a global overview of the system and information on the reasoning behind certain design decisions being made. Without the models, the software engineer can solve problems to the best of his abilities but this can result in code duplication, reinvention of designs and gaps in the software architecture.

Software will always evolve as a result of maintenance, bug-fixes, or updated versions, and due to the reuse of software for analogue systems (or when adapting it for new customers). Unfortunately, when software evolves, the design does not evolve with it and is therefore no longer up-to-date. The models provided by the methodologies of today are not resilient to change, resulting in inconsistencies between design and implementation. Thus, the synchronisation between design and implementation is lost because the link between both is absent. To get an idea of what the system looks like, a direct look at the implementation is required, but it is crucial to have a relevant design in order to understand the system, as we argued before. Ideally, the design must be connected to the implementation in a way that if either one

changes, then this affects the other, in this way design and implementation are synchronised.

The solution to the problem of de-synchronisation between implementation and design is called co-evolution ([DDVMW00]). Co-evolution performs synchronisation through logic meta programming which consists of a declarative layer (a meta level) on top of the object-oriented system (the base level). This declarative meta layer is used to detail and express design knowledge, while the implementation of the system resides in the object-oriented base level. Furthermore, logic meta programming implies that both the declarative and object-oriented layers are connected in a such a way that a change on one level will affect the other. The meta level (the declarative layer) can reason about the base level (the object-oriented layer); it can look up information and make modifications to the implementation in the base level. On the other hand, a base level modification can have an impact on the meta level; therefore, a change to either level will affect the other level. Furthermore, design knowledge described on meta level can be enforced upon the implementation of the base level, and the other way around. This means that design and implementation of the system are synchronised.

Co-evolution is a unique vision within the object-oriented world, and the work carried out to date, indicates that it is of benefit. At the VUB Programming Technology Lab, different systems have been developed, and until now these systems are based on Prolog as environment for the meta level. Prolog is one of the most widely known logic programming languages and is based on querying. In such systems, knowledge is described by using facts and rules, and a query allows the programmer to explicitly "ask" for information that is to be derived from these facts and rules. When starting a query, a hypothesis will be tried to be proved. This proving mechanism is called goal-driven reasoning since, when querying, the reasoning process will try to reach its goal. Goal-driven reasoning is also called backward chaining.
One of the disadvantages of querying is that, after the program has been written, the programmer must explicitly start these queries and to do this he must clearly specify what hypothesis, what goal, is to be proved. Nevertheless, when reasoning about object-oriented systems and when trying to synchronise design and implementation, the goal can be vague. When programming in an object-oriented system there is a need for 'something' that helps to take decisions or to "rap over the knuckles" when a rule is violated. To get to these kind of actions, a data-driven approach is more appropriate. This data-driven reasoning, also named forward chaining, starts from rules (and possibly a few facts), and new facts are derived until a goal is ("accidently") reached.

Thesis

Co-evolution, which aims to synchronise design and implementation of object-oriented systems, currently only uses backward chaining to steer the declarative knowledge of the design about the implementation. Our thesis is that a forward chaining technique is equally beneficial, because of the forward chainer's inherent data-driven nature. To validate this thesis some experiments have been carried out with a forward chainer built for this purpose. This forward chainer is based on an existing tool for building knowledge systems and will serve as a meta layer on top of an object-oriented (Smalltalk) base layer.

This dissertation proves this thesis. *Chapter 2* gives a general explanation of the concept of co-evolution and how it is supported to date by the use of goal-driven reasoning. This chapter also gives an overview based on the available literature for this research. *Chapter 3* explains knowledge systems since these are the context in which research on inferencing techniques, like data-driven and goal-driven inferencing, is carried out. This chapter also gives a comparison between these two inferencing techniques. *Chapter 4* describes the forward chainer which was implemented to support this dissertation and that is our version for co-evolution on Smalltalk code. *Chapter 5* validates this thesis by proving that certain aspects of co-evolution are possible by using forward chaining (data-driven reasoning), although not self-evident when using the goal-driven approach (i.e. backward chaining). Finally, *chapter 6* will give the conclusion and future work for this dissertation.

Chapter 2

Co-evolution : Reasoning about Object-Oriented Systems

This chapter sketches an idea of the evolution in the field of reasoning about object-oriented systems. First, the term co-evolution is clarified and a short explanation of logic programming and logic meta-programming is given, then some examples of the practical use of co-evolution are presented.

2.1 Introduction : Co-Evolution

In this section we will explain a current and major problem occurring when reasoning about object-oriented systems. The reasons why there is a problem are presented, together with a solution for this problem.

2.1.1 Software evolves

During years of evolution in computer technology, software maintenance and development have gained in importance. With his statement *"Programs, like people, get old. We can't prevent ageing, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable."* in [Par94], Parnas made us focus our attention to the fact that software will indeed need to evolve and that as designers, we should take this into account from the moment we start writing software. Furthermore, Lehman's first law of evolution *"A program that is used in a real world environment necessarily must change or become progressively less useful in that environment."* ([Leh97]) states that evolution is inherent to software development.
If a company wants to survive in today's world, it must keep up with the market and respond rapidly to new opportunities ([DH98]) and as a result the company's software must evolve in parallel with its changing objectives and activities, the software's functional requirements change. Futhermore the non-functional requirements of software, such as reusability, flexibility, adaptability, low complexity, maintainability, ... are also subject to change. [DH98] lists the situations in which software evolution is desired as :

- **New insights in the domain :** Moving general concepts of the software into the core benefits reusability.

- **Complexity of classes is too large :** Object-oriented software becomes more complex as it evolves.
- **New design insights :** Forgotten or neglected design issues in the initial design phase can turn into problems during later development.

Thus, these insights show that software evolution is indeed inherent to software development. In situations like these, evolving the software is necessary in order to keep the software endurable.

2.1.2 Software is complex

Lehman's second law of evolution *"As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure"* ([Leh97]) shows that controlling the complexity of a software system is of vital importance for the system's future. Also Davis confirms this by stating that every system undergoing changes continuously will grow in complexity and will become more and more disorganised ([Dav94]).
In the last few decades, more and more attention has been paid to techniques that will help achieve a better understanding of a software system, and thus a more controlled evolution; for instance, when it comes to object-oriented programming languages, different kinds of techniques that have been introduced are for example, frameworks ([CDSV97]), design patterns ([GHJV94]) and contracts ([HHG90]) became very popular. Refactoring is used to reorganise the software such that reuse can be improved and more recently components have been introduced with the intention to split the program into smaller components which are to cooperate together. However, as stated in [DDVMW00], how this cooperation happens is not any better understood than the software building strategies considered in the past 20 years. The partial failure of these techniques is attributed to the fluidness of software, where on one hand, there is a technology evolving at breakneck speed, and on other hand there are the requirements which must follow the economic vagaries of modern society ([DDVMW00]).
Thus, it is clear that software is complex and some techniques are needed in order to help the programmer when using and evolving these complex systems. Although these techniques have become very popular, there is still a need for more control over the evolution of software, thus, a need for managing this evolution.

2.1.3 Need for synchronisation

Currently, developing a software system happens in different phases. The different layers of the software development process usually are requirements capture, analysis, design, implementation and documentation. Each of these layers are coupled in the sense that, when developing one layer, it still is possible to fall back on the previous ones, however, it may be necessary that during the entire process all layers have to be updated. In practice, due to deadlines and timepressure, as development progresses, it is the implementation layer that gets all the attention and the remainders are neglected. This causes inconsistencies between implementation and design, although they should be synchronised. As a result, if a second evolution of the software is required, then there is no design to fall back on when problems arise and the developers need to get into the code in order to understand the system. A

good, up-to-date design will help subsequent evolutions. Therefore, design and implementation should be coupled and causally connected, as a software change will have an effect upon the design, and the other way around. The synchronisation between the different layers helps to manage software evolution. In current methodologies and/or development environments however, this synchronisation is absent.

2.1.4 Co-evolution

Declarative meta-programming

We thus claim (together with [DDVMW00]) that managing evolution requires the synchronisation between different layers in the software development process. In order to obtain this synchronisation the developer should be constrained to design decisions and implementation should become a strict superset of the design. This is what [DDVMW00] calls **co-evolution**. The aims of co-evolution are

- **effort reduction** : obtaining a faster code understanding, reuse of code, ...
- **effort estimation** : estimating the efforts that are needed to make a proposed software change (i.e. impact analysis)

To obtain these goals, co-evolution requires design information to be stated "over" the code. This design information is implicit available knowledge on the structure of the code that should be made explicit. This can easily be done by declaring facts and rules that then are used in a reasoning process. This indicates that a declarative approach is advisable. Furthermore, we want to reason about and act upon the implementation on a higher level, on a meta-level. Declarative meta-programming is the advisable approach for the actualisation of co-evolution. (More information on declarative meta-programming can be found in section 2.2.2.)

Reflection

In [Mae87] Maes gives the following definition of reflection :
*"A computational system is said to be **reflective** if it incorporates and also manipulates causally connected data representing (aspects of) itself."*
When applying this to co-evolution, we conclude that a co-evolution system is a reflective system as through the declarative meta-language it can reason upon aspects of itself, namely on the base-level (i.e. the level that is reasoned about). Moreover, these two levels are causally connected with one another.
A reflective system is a system that allows introspection and absorption of the base system. This is where introspection means that the meta system can access data-structures and information on the base system, and absorption is present when the meta system can also change the base system.

Co-evolution in practice

At time of writing this, co-evolution is still a pretty unique vision in the object-oriented society, but at the VUB Programming Technology Lab recent research on co-evolution already indicates that co-evolution is meaningful.

In this research, existing programming environments, suited for engineering large software systems, are being extended with a declarative meta layer. The design and implementation of this meta layer has been mainly based on Prolog as Prolog is a popular logic programming language with the power and capacity to support multi-way queries, an aspect found to be particulary attractive ([DDVMW00]). (A more deepened discussion and overview on Prolog can be found in section 2.2.1.) The research carried out so far is sketched in [DDVMW00] and further in this chapter some examples will be detailed.

2.2 Logic Meta Programming

The research on co-evolution, that has been carried out to date, uses a declarative meta layer on top of the object-oriented base layer. The implementation of this declarative layer is based on Prolog, a logic programming language. Therefore, this layer is called a logic meta programming language. In this section we will explain logic programming and logic meta programming.

2.2.1 Logic programming

Logic programming is a programming paradigm which was developed in the seventies. Rather than viewing a computer program as a step-by-step description of an algorithm (like traditional languages), the program is thought of as a logical theory and a procedure call is viewed as a theorem of which the truth needs to be established. Thus, the execution of a program comes down to searching for a proof. A logic program concentrates on a *declarative specification* of what the problem is, and not on a procedural specification of how the problem is to be solved. In order to perform this, the database of a logic program consists of facts and rules which are accessed by queries.

- *Facts* hold static information that is always true in the application domain.
- *Rules* derive new facts from existing ones. The conditional part of the rule should be true in order to conclude the premise of the rule.
- *Queries* are used to access the data in the database and finding an answer to such a query is carried out by matching it with facts (initial or derived).

In essence, matching is proving that a statement follows logically from some other statements. This reasoning process is also called *resolution*, which adds a procedural interpretation to logical formulas, besides their declarative interpretation. Because of this procedural interpretation, logic programming can be used as a programming language. Kowalski's equation *"algorithm = logic + control"* also denotes this. In this equation, logic refers to the declarative meaning of logical formulas, and control refers to the procedural meaning. However, in a purely declarative programming language it is not possible to express the procedural meaning.

Prolog is one of the most widely used logic programming languages, though not a purely declarative programming language for the procedural meaning of programs cannot be ignored. Prolog's inference engine uses backtracking to reconsider other possible solutions for its queries.

In Prolog, the format of a *rule* is : `head :- body`. The head and body are both predicates followed by some arguments enclosed in brackets. *Arguments* are terms and can either be a constant (denoted with a small letter), a variable (denoted with a capital letter), a compound term or a list. *Predicates* of a rule can be combined with a disjunction (denoted with a semi-colon) or a conjunction (denoted with a colon). *Facts* are represented by a single predicate. For a complete overview on Prolog's syntax, the reader is referred to [Fla94] which uses Prolog to explain the art of logic programming.

2.2.2 Logic Meta Programming

A *program* specifies the computational process which will manipulate a representation of entities and data. A *computational system* reasons about and acts upon some part of the world, called the *domain* of the system.
In a *meta-system* the computational system reasons about and acts upon another computational system, called the *base-system*, therefore, the meta-system has as domain the base-system, and the program of the meta-system is called the *meta-program* ([Mae87]). Thus, a meta-program is a program which reasons about another program (i.e. the base-program).

When using a logic programming language as such a meta-system we find *Logic Meta Programming*, a kind of multi-paradigm where base-languages are described by means of logic programs, where two paradigms, object-oriented programming and declarative programming, are joined together. This approach will allow synchronisation between design and implementation, called co-evolution, by expressing design information and architectural concerns as rules and constraints.

2.3 Logic Meta Programming with SOUL

The first example indicating that logic meta-programming languages are beneficial for co-evolution is SOUL which is a Prolog interpreter written in Smalltalk. SOUL creates a symbiosis between the declarative and object-oriented paradigm and is used to reason about object-oriented systems. In this section we firstly look at class management using logical queries, then we will describe LPS (Logic Programming in Smalltalk), the predecessors of SOUL. Finally SOUL (Smalltalk Open Unification Language) is discussed.

2.3.1 Class management using Logical Queries

As object-oriented systems grew and became more complex, new software engineering techniques were developed. These techniques shifted from single classes to more elaborate relations between classes (e.g. frameworks, contracts, design patterns). In [Wuy96] Wuyts states that class-based browsers fail to accommodate new insights and new techniques due to two problems :

- the lack of a sophisticated query system that enables queries ranging over the whole class-system, and
- the lack of customizability of the queries and the user-interface to present their result.

In [Wuy96] a tentative solution to this problem is elaborated. For this a logic programming language is proposed as a query-mechanism for questioning the class-system together with

```
Facts
Dictionary Object %classIncluded%
Collection Object %classIncluded%
OrderedCollection Collection %classIncluded%

Rules
isClass(?class) = ?class ?X %classIncludes%
isDirectedSubclass (?class ?super) = ?class ?super
%classIncluded%

inHierarchy(?root ?class) = isDirectSubclass(?root ?class)
  #or (isDirectSubclass(?root ?class-super)
  #and inHierarchy(?root ?class-super))

Queries
isClass(?X)
inHierarchy(?X, ?Y)
```

Fig. 2.1: Logic Programming Language : Syntax

custom user interface components for building the user interface. The benefit is that a logic programming language enables strong queries that can range over classes, but also over the full class-system.

Logic programming language

A small logic programming language was implemented in Smalltalk to be used as query mechanism. Like other logic programming languages, this query language uses facts, rules and queries. For every class to be taken into account, a fact is added such that the query language and the Smalltalk class system are completely separated from each other. The new query language has one extra feature, namely it allows to use Smalltalk blocks as predicates for rules and queries. This is the only place where Smalltalk can be used in the logic programming language. The syntax, as shown in figure 2.1 is very similar to Prolog syntax. Names starting with a question mark are variables.

Validation

The logic programming language and the custom user interface components were combined. This combination led to more powerful class browsers and it ensured support for different programming techniques. To validate these statements, browsers were build in Smalltalk to demonstrate the power and customizability on different domains. The first class browser was a simulation of the System Browser, the standard tool in VisualWorks Smalltalk environment that enables the programmer to have a look at all the classes available, their methods and definition. The second browser was a simple browser that enabled the programmer to walk through the class-system by applying queries and included a back-track facility. A browser for

```
QUERY findAll(?M,directVarAccess(?C,?I,?M),?Lst),
      findAll(?Mt,methodsInProtocol(?C,[#accessing],?Mt),?RMLst),
      difference(?RMLst,?Lst,?Result).

RULE directVarAcces(?C,?I,?M)
        IF instVarName(?C,?I),
           whichSelAccInstVar(?C,?M,?I).

DOMAIN <AllInstVarNames(aClass)> [aClass allInstVarNames].

VIRTUAL FACT instVarNAme(?C<Classes>,?I<AllInstVarNames(?C)>).

VIRTUAL RULE whichSelAccInstVar(?C<Classes>,?M<Methods(?C)>,
                                            ?I<InstVarNames(?C)>)
        IF [(C whichSelectorsAccess: I) includes: M].

VIRTUAL RULE methodsInProtocol(?C<Classes>,?P<Protocals(?C)>,
                                                 ?M<Methods(?C)>)
        IF [(C organization categoryOfElemet: M) == P].
```

Fig. 2.2: LPS : Indirect Variable Access Rule

the bridge-pattern was also built and showed that the combination of the logic programming language and the custom user interface components could be used to create browsers that support software techniques such as design patterns and thus add to the viability of the co-evolution concept.

2.3.2 LPS : Logic Programming in Smalltalk

LPS (Logic Programming in Smalltalk) is a logic programming language that can be used as a meta language to express structural information of software systems. This language was implemented in Smalltalk and based on Prolog, but with an extra extension that allowed virtual facts and rules to be "extracted" from Smalltalk, which made it possible to capture any Smalltalk language concept in logic rules without an explicit formulation.
In figure 2.2 the syntax of LPS is shown, and again this syntax is very similar to Prolog. Names beginning with a question mark are variables. The head and body of a rule are separated with `IF` and constant terms are expressions between square brackets. `DOMAIN` and `VIRTUAL FACT` are additional features, where a domain represents the name of an aspect of the base language we want to reason about and a virtual rule is a collection of rules which are obtained by restricting variables to defined domains. Both take a Smalltalk block (between square brackets) that can be evaluated within Smalltalk to get the value. A profound explanation of LPS is given in [Mic98].

In [Mic98] a declarative framework was built using LPS to express programming conventions. The core rules of this framework are dealing with the representation of elements

of the object-oriented base-language Smalltalk (such as classes and methods) in the logic meta-language. These rules make frequent use of Smalltalk expressions (through domains and virtual rules) to get to the base system and are bridging the two worlds.
On top of the core rules, the basic structural rules are defined to provide extra functionality and flexibility to deal with the basic structure of class-based object-oriented systems (such as inheritance and acquaintance relationships). The rules in this layer allow basic querying about the system. The query in figure 2.2 is constructed such that it tracks down methods that violate the Indirect Variable Access rule : *"always use accessor methods to access the instance variables"*. All methods that violate this rule will be listed in `?Result`. The rules, domains and facts used in the query are also shown in the example.

The contribution of Michiels shows that LPS is suitable to use as logic meta-programming for building sophisticated development tools. Although Michiels experienced the lack of parse-trees to be a limitation, it was clear that a declarative language could be very powerful and very meaningful when used to reason upon its base-level. The experiences gained with LPS and this framework, resulted in the development of SOUL.

2.3.3 SOUL : Smalltalk Open Unification Language

SOUL is the successor of LPS and is a logic meta-language implemented in tight symbiosis with VisualWorks Smalltalk and based on Prolog. In [Wuy98] it was used to construct a declarative framework that allowed to reason about the structure of Smalltalk Programs. This framework showed that the use of a logic meta-language to express and extract structural relationships in class-based object-oriented systems is meaningful. The incapability to express high-level structural information in a computable medium that is then used to extract implementation elements was and still is an important problem in object-oriented systems. The solution to this problem is a logic programming language as the meta-language to express and reason about the structural information of software system. [Wuy98] shows that there are two advantages to write down the structure in a meta-language :

- information is available in executable form in the programming language
- it is sufficient to specify the structural relationships in the meta-language. They do not need to be included in the code.

Wuyts implemented SOUL as an example of co-evolution. In the following paragraphs a short overview of SOUL is given, but the complete explanation can be found in [Wuy00].

The core of SOUL is a logic programming language with a resolution engine and is written in Smalltalk. The key components of SOUL are the parser, the parse tree elements, the repositories, the working memory used to bind variables during interpretation and the streams used to pass everything around during interpretation. The *parser* converts a string into a parse tree for representing a logic clause. The main syntax elements in SOUL, clauses and terms are *parse tree elements*. *Clauses* are the top-level parse elements and are constructed out of terms. The three basic clauses are facts, rules and queries like in Prolog. *Terms* are constants, variables, compound terms and lists. *Repositories* are the knowledge bases to be consulted during interpretation and they consist of a number of facts and rules. The *working memory* is used during the interpretation process to 'store' terms bound to variables. During

interpretation, all results are described by *streams*. SOUL is based on the software architecture of a rule-based system, but has been refined.

When developing SOUL the intention was to use SOUL as a logic meta-language for an object-oriented base language. Thus, somehow object-oriented systems should be represented as logic facts, so that logic programs could be written down using this representation. For this a logic representation of the parse tree of the system to be reasoned about, is used. Turning a logic programming language into a logic meta-programming language is performed by creating a knowledge base containing the logic representation of the (base) system to be reasoned about ([Wuy00]).
The base predicates of SOUL allow reasoning upon Smalltalk programs. These predicates are a minimal set sufficient to reason about the logic representation of classes, inheritance relationships, instance variables and methods. The repository containing the logic representation together with the base predicates forms a logic meta-programming language. An example of the Indirect Variable Access written in SOUL is given in figure 2.3.

Practically usable logic meta-programming language

The previous paragraphs explained how a logic programming language can be turned into a logic meta programming language by using a repository containing a logic representation of the system that is to be reasoned about. Thus, now it is possible to write regular logic programs that will manipulate the representation of the base program, so they are meta programs. The major drawbacks of this approach listed by [Wuy00], are:

- only the information in the database can be used by the logic meta-programming language
- the repository can become very large
- the actual source code is not linked with the logic representation of the system in the repository

Since SOUL is to be used practically, it had to be specialised into a true logic meta-programming language and such a language should directly reason upon programs expressed in the base language. To do so an extra mechanism was added. The extra mechanism used in SOUL is *reflection* (see section 2.1.4) which enables the logic meta-programming language to access its own implementation (on the base-level) and complete power of this base-level is available to the logic meta-programming language. As a result of this reflection, the logic meta-programming language gains more power. SOUL provides *smalltalk terms* as reflection operator to absorb Smalltalk objects as logic terms. Since reflection also implies causal connection, the smalltalk terms are causally connected with their Smalltalk counterparts ([Wuy00]). An example of this is shown in figure 2.3 with the implementation of the class predicate.

```
Query accessingViolator(?class, ?method, ?instvar)
```

Basic predicates

```
class(?class)
superclass(?super, ?sub)
instVar(?class, ?iv)
method(?class, ?m)
```

Rules

```
Rule acessingViolator(?c, ?m, ?iv) if
       class(?c),
       instVar(?c, ?iv),
       method(?c, ?m),
       not(accessor(?c, ?m, ?iv)),
       isSendTo(variable(?iv),
                ?violatingMessage,
                ?args).

Rule accessor(?class, ?method, ?varName) if
       instVar(?class, ?varName),
       classImplementsMethodNamed(?class, ?varName, ?method),
       accessorForm(?method).

Rule accessorForm(?method) if
       methodStatements(?method, <return(variable(?varName))>).
```

Implementation of class predicate

```
Rule class(?c) if
       generate(?class, [SOULExplicitMLI current allClasses]),
       equals(?c, ?class).
```

Fig. 2.3: SOUL : Indirect Variable Access Rule

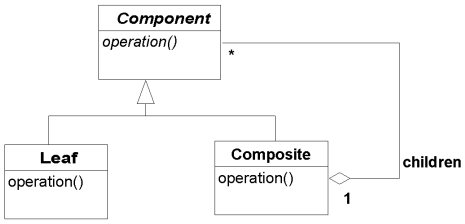

Fig. 2.4: Composite Design Pattern : class diagram

Using SOUL to express Design Patterns

Design Patterns ([GHJV94]) capture solutions to common problems, encountered while designing software. They record experience in designing object-oriented software such that people can use these records effectively. In [Wuy98] SOUL is used to "*express the structures that are described by design patterns in an open, formal, and non-constraining way*". Figure 2.5 shows the rules that describe the composite pattern whose class diagram is depicted in figure 2.3.3. The `composite` class is a subclass of the `component` class and has a one-to-many relationship with this component.

The rule `compositePattern` says that a composite pattern indicates a certain structural relationship between the component and the composite, and an aggregation relationship between the two. The `compositeStructure` rule defines that `?comp` is a class and `?composite` a subclass. The `compositeAggregation` rule says that the composite should override at least one method of the component, and in this overridden method it should do an enumeration over the instance variables.
When executing the query `Query compositePattern([VisualPart], ?comp,?sel)` there will be searched for all possible composite classes where the component is `VisualPart`. The result of this query is `VisualComposite`, which is indeed a composite class.
This example shows that with SOUL it is possible to reason about the object-oriented system. In this case, reasoning means searching for all composite patterns.

2.3.4 The contribution of SOUL in the context of co-evolution

SOUL is a logic meta-programming language based on Prolog and written in Smalltalk. The key concepts of SOUL are the parser, the parse tree elements, the repositories, the working memory used to bind variables during interpretation and the streams used to pass everything around during interpretation. SOUL's basic layer contains the core rules for bridging Smalltalk and SOUL, together with some pre-defined basic predicates. A logic representation of the parse tree of the system to be reasoned about is used to represent everything from the base-level as a meta-level entity. Together with the basic predicates, this representation results in a logic meta-programming language. Furthermore, SOUL is a reflective programming language since the meta-level can reason upon the base-level and the complete power of the base-level is available to the meta-level.
Experiments, carried out in and with SOUL (e.g. the composite pattern example of the previous section) , are a major contribution to research on reasoning about object-oriented systems. These experiments have shown that logic meta-programming language is of great benefit when

```
Rule
    head: compositePattern(?comp,?composite,?msg)
    body:
        compositeStructure(?comp,?composite),
        compositeAggregation(?comp,?composite,?msg).

Rule
    head: compositeStructure(?comp,?composite)
    body:
        class(?comp),
        hierarchy(?comp,?composite).

Rule
    head: compositeAggregation(?comp,?composite,?msg)
    body:
        commonMethods(?comp,?composite,?M,?compositeM),
        methodSelector(?compositeM,?msg),
        oneToManyStatement(?compositeM,?instVar,?enumStatement),
        containsSend(?enumStatement,?msg).

Rule
    head: oneToManyStatement(?method,?instVar)
    body:
        statements(?method,?stats),
        hasEnumerationStatement(?stats,?enumStatement),
        receiver(?instVar, ?enumStatement).

 Fact enumerationStatement([#do:]).
 Fact enumerationStatement([#collect:]).
 ....
```

Fig. 2.5: SOUL : Rules for the Composite Pattern

striving for co-evolution, for synchronisation between design and implementation.

2.4 Logic Meta Programming with TyRuBa

The example in section 2.3 showed how logic meta programming can be used to reason about Smalltalk systems. A further example of logic meta programming developed at the VUB Programming Technology Lab is TyRuBa. TyRuBa is a Prolog alike system and can generate code files. In this section we start with some words on TyRuBa's syntax, then, code generation with TyRuBa will be explained and finally the use of TyRuBa as an aspect-oriented logic meta programming language is dealt with.

2.4.1 TyRuBa : a Prolog alike programming language

TyRuBa is a concrete system providing a Logic Meta Programming language that allows code generation for Java. The system is built around a core language which is based on Prolog. TyRuBa's syntax is similar to Prolog, with a few differences.
TyRuBa has *directives* to instruct the system to perform a special action. TyRuBa *terms* are *variable* (identified with a leading '?'), *constants* (any other identifier), *compoundTerms* (enclosed by '<' '>'), *qoutedCode* (Java code delimited with curly braces) and *lists*. *QuotedCode* allows to include pieces of Java code as data in logic programs. It can contain references to logic variables or to other compound terms. Figure 2.6 shows a sample TyRuBa program. It represents a class *Set* in Java with a type parameter *?El*. The representation has a condition (on its generation) that specifies that the type parameter must implement the *Equality* interface. The example was taken from [Bri99].

2.4.2 Type-Oriented Logic Meta Programming on top of Java

The aim of TyRuBa is to achieve type-oriented logic meta-programming in order to generate code. Programs expressed in a type programming language, may actively use static type information and these type meta programs are run as a part of the compilation and type checking process, with respect to the base language programm ([DV98a]). The logic meta programming layer will contain type information about the base level programs.
TyRuBa was implemented in Java and also uses Java as a base language to reason about. The reasons why Java was chosen as base language are listed in [DV98b] and these reasons are :

- Java is popular
- Java is fairly simple and well designed
- Java has a static type system
- Java interfaces are natural candidates for describing object types

This last reason is important because TyRuBa programs talk mainly about interfaces and classes and how interfaces implement classes.

What is needed for logic meta-programming is a representation scheme of the mapping between the base-language program onto a set of logic propositions representing this program.

```
class_I(Set<?El>,{
    private List<?El> representation = null;

    public void insert(?El e) {
        if (!contains(e)) {
            representation = new List<?El>(E,representation);
        }
    }

    public boolean contains(?El e) {
        return listContains(representation, e);
    }

    private static boolean listContains(List<?El> l, ?El e) {
        return (l!=null && (
            l.first.equals(e) || listContains(l.rest,e)
        );
    }
}) :- implements(?el, Equality<El?>).
```

Fig. 2.6: TyRuBa : A sample program

Since TyRuBa tends to perform type-oriented logic meta programming, De Volder was interested in the types and dependencies between these types. The propositions focus on classes, interfaces and the relations between them. The class body is divided into pieces providing the implementations for interfaces.

2.4.3 Code generation with TyRuBa

The representational mapping (Java to TyRuBa) can be reversed by a code generator. If the code-generator and the representational mapping understand the same propositions in the same way, then the generator can reconstruct the base program out of set of propositions. TyRuBa regenerates the base programs based on the logic propositions and since TyRuBa is a logic programming language, writing queries and combining their results in a consistent way leads to the code generation.

Figure 2.7 is an example of code generation with TyRuBa ([DV98a]). This will generate the code for a class. The body of the class is looked up in the logic database. The extends and implements clauses are computed by two auxiliary predicates, `generate_extendsclause` and `generate_implementsclause`, which will look up all type names that are in an `extends` of `implements` relationship with the class being generated. From this an extends or implements clause is constructed. The complete example can be found in [DV98a].

A coarse-grained code generator has been hard coded in TyRuBa and this allows the user to specify a more fine-grained code generator. The code generator is driven by the *generate*

```
generate(?class,{
    class ?class
    ?extendsclause
    ?implementsclause
    { ?body }
})  :-  class_(?class,?body),
        generate_extendsclause(?class,?extendsclause),
        generate_implementsclause(?class,?implementsclause)).
```

Fig. 2.7: TyRuBa : Code generation

directive (in TyRuBa source files) which invokes the code generator and requests the generation of a class or an interface, corresponding to the name given by its argument.
Code generation is again a form of reasoning about object-oriented systems. The design of base level structures (e.g. classes) is described on meta level. By generating the code, the design is applied on the object-oriented base level. Thus, this kind of code generation is a powerful example of symbiosis between a declarative meta level and an object-oriented base level.

2.4.4 AOLMP : Aspect-Oriented Logic Meta Programming

Aspect Oriented Programming (AOP) addresses the problem of cross-cutting concerns in code that in general are not neatly packaged into a single unit of encapsulation ([DVD99]). These cross-cutting concerns are for example synchronisation, distribution, persistence, debugging, error handling, ... which have a wider impact. Aspect languages offer new language abstractions that allow cross-cutting aspects to be expressed separately from the base functionality and the aspect weaver generates the actual code by interweaving basic functionality code with aspect code.
All aspect languages have in common that they are declarative in nature and offer a set of declarations to direct code generation. [DVD99] states that "*the universal declarative nature of aspect languages begs for a single uniform declarative formalism to be used as a general uniform aspect language*". In that same paper a logic programming language is proposed for that purpose such that aspect declarations are expressed by logic assertions in a general logic language instead of being expressed in specially designed aspect languages. The weaver is implemented by a library of logic rules and the facts function as hooks into this library, and in this way AOLMP is a logic meta language to reason about aspects.
The logic language can be used as a simple general purpose declaration language, but it can also be used to express queries about aspect declarations or to declare rules which transform aspect declarations. When using logic facts to declare aspects, the aspect declarations can be accessed and declared by logic rules. This gives the ability to write logic programs which can reason about aspect declarations. Using this approach allows the user to extend or adapt the aspect language.
When we consider design knowledge as an aspect (see section 2.6), AOP is relevant for the purpose of co-evolution. Design knowledge is to be described on meta level and can be linked

with the implementation by interleaving the aspects and the base system.
In [DVD99] the aspect-oriented logic meta programming approach is proved with some experiments carried out in TyRuBa. In these experiments, both the usefulness of AOLMP from the user's point of view and from the aspect oriented programming implementor's point of view, are validated.

The contribution of TyRuBa in the context of co-evolution

TyRuBa is a symbiosis between declarative programming and object-oriented programming that is used for code-generation. Base programs can be generated based on logic propositions. TyRuBa has been found to be useful in aspect oriented logic meta programming, an alternative to specific aspect oriented programming languages and again this indicates that logic meta programming is a powerful system to reason about object-oriented systems.

2.5 Logic Meta Programming with QSOUL

Recent work on combining ideas of both SOUL and TyRuBa led to QSOUL. QSOUL is implemented in Squeak[1] and the intention is to reason about Smalltalk code in the original SOUL-style, as well as to generate Smalltalk code in the original TyRuBa-style.

2.6 Separating Algorithm and Knowledge

The final instance of co-evolution is due to [DD99] and [DDMW99].

Industrial applications suffer from difficulties with maintenance and reuse because of the hard coded presence of domain knowledge in the implementation. Domain knowledge is the collection of concepts, relations between them and constraints on these concepts within a domain. Almost all software is applied to some domain, but most of the time the domain knowledge is hard wined into the application code. The algorithm and the domain upon which the algorithm reasons, are separated. [DD99] lists major activities that lead to difficult maintenance and reuse of code :

- When existing domain knowledge is modified conceptually, the relevant knowledge has to be *localised* and adapted in the implementation which is a complex and error-prone task. It is almost impossible to reuse an application for a different domain since the domain knowledge is hard coded in the application.
- It is hard to reuse domain knowledge at the implementation level across a suite of applications. There is a lack of *synchronisation* between domain knowledge and each application it is used in.

The domain and the application can no longer evolve independently and [DD99] validates this with an example of a Geographic Information System (GIS). The example shows how constraints are translated into the application's algorithm and modifying the domain knowledge becomes hard. The proposed approach to solve this problem is based on the analogy

[1] An environment for Smalltalk. See also section 4.1

```
Fact city (Rijmenam)
Fact city (Boortmeerbeek)
...
Fact road (city (Rijmenam),city (Boortmeerbeek),[3 ])
Fact road (city (Keerbergen),city (Rijmenam),[4 ])
...
Fact prohibitedManoeuvre (city (Rijmenam),city (Bonheiden))
Rule roads (?current,?newResult)
  if findall (road (?current,?next,?distance),
  road (?current,?next,?distance),?result),
  privateRoads (?current,?result,?newResult).
Rule privateRoads (?current,?result,?newResult)
  if prohibitedManoeuvre (?current,?next),
  removeRoad (?result,road (?current,?next,?distance),?newResult)
Fact privateRoads (?current,?result,?result)
```

Fig. 2.8: Algorithm and Knowledge : The domain knowledge

with aspect oriented programming such that the component program, an algorithm, is cross-cut with the aspect domain knowledge. The benefits of considering domain knowledge as an aspect are :

- Domain knowledge can be dealt with separately which makes the program less complex.
- Applications are easier to understand and to maintain.
- The domain knowledge and the application evolve independently and both can be reused for other applications and on other domains.

Therefore, in order to illustrate the connection between this problem and co-evolution, the GIS example has been rewritten in SOUL by [DDMW99]. SOUL, as we explained in section 2.3, is a logic meta programming language with Smalltalk as base language and a Prolog-like language as the meta language. In this example the meta level is not used as a meta-layer. In this way it serves as an aspect language for describing domain knowledge at the same level as the base level. The domain knowledge is represented in SOUL (figure 2.8) and the application in Smalltalk (figure 2.9). The algorithm still uses nodes and edges, but this is obvious since the algorithm is graph-based and thus it requires the use of these concepts. The connection between the algorithm and the knowledge level lies in so called join points. In the example, the join points between the two are identified since *cities* map to `nodes` and *roads* map to `edges`. To fix the join points of the aspect-oriented program, linguistic symbiosis is used. This is a mechanism to manipulate Prolog objects in Smalltalk and vice versa.

In [DDMW99] it is also shown that sometimes an extra parameter might be needed to be passed around to perform the right computation. The authors state that "*the domain knowledge should not be burdened with this algorithmic information, but neither should the algorithm be tangled with an extra parameter that is needed to test a domain constraint concerning priority manoeuvres*". Furthermore, some extra kind of memory manager should run in the background to pass otherwise lost information to either the algorithm or domain layer.

```
branchAndBoundFrom: start to: stop
    |bound|
    bound :=999999999.
    self traverseBlock:[:node :sum|
        node free ifTrue:[sum < bound ifTrue: [node = stop
            ifTrue:     [bound := sum ]
            ifFalse:    [self branch: node sum: sum ]]]].
    self traverseBlock value: start value: 0.

bound branch: node sum: sum
    node free: false.
    node edges do: [:edge|
    self traverseBlock value: edge next value: sum + edge distance ].
    node free: true.
```

Fig. 2.9: Algorithm and Knowledge : The algorithm

SOUL, or a logic meta programming language, appeared to be useful for use as an aspect language, as representing domain knowledge as an aspect seems to have nothing but advantages. Some research still has to be carried out concerning this subject, but the idea is promising.
Co-evolution wants to synchronise design knowledge and implementation. This approach makes a distinction between the algorithm and the knowledge. Thus, this is again a form of co-evolution. the algorithm is the implementation, the knowledge is the design knowledge and both should be synchronised.

2.7 Summary

Software evolves and over the years techniques like design patterns, frameworks, contracts and components have tried to get more control over the evolution of the software. These techniques have only partially succeeded and today the need for managing software evolution is still present. Managing evolution requires the synchronisation between different layers and phases in the software development process among which are design and implementation. Co-evolution tries to achieve this synchronisation by using Declarative[2]Meta Programming. Declarative meta programming provides a declarative layer on top of and causally connected to the base system. Since both layers are causally connected, it is possible to write down knowledge about design with facts and rules that will be forced upon the implementation. Therefore, if the implementation changes, the design will change as well, and vice versa.
There have been extensions of logic programming languages with object-oriented aspects, but the idea of extending object-oriented languages with a declarative meta-layer is new and quite

[2]Until now research was done with Prolog-like meta programming languages, which was also called Logic Meta Programming. However, it is not essential to use a logic programming language as a basis, thus, any declarative programming language could be used. Therefore, from now on, the term **Declarative Meta Programming** will be used in stead of Logic Meta Programming.

unique. At PROG some major research in this field has been going on and so far this has resulted in some interesting experiments that already show the significance of co-evolution. SOUL and TyRuBa were developed. SOUL reasons about the base-system and TyRuBa generates code, but they have in common that they are both declarative meta programming languages that provide powerful means to reason upon object-oriented programming systems. Beside reasoning on code and code-generation, declarative meta programming has also proved to be beneficial as an aspect-oriented programming language.
The systems developed so far for the purpose of co-evolution were based on Prolog and all of them use a goal-driven inferencing process. This dissertation will show that a data-driven inference engine is another approach that has not been explored to date. A lot of aspects of software evolution have a goal that cannot be written down explicitly, and in these cases a goal-driven approach fails. Therefore, an important branch of research within the field of co-evolution, are systems that aid the programmer in a data-driven way during the evolution of software.

Chapter 3

Forward Chaining vs Backward Chaining

In the previous chapter co-evolution was explained together with an overview of the research that has been done to date in this field. All of the systems developed so far in the context of this research, are based on Prolog and use a goal-driven inferencing process.
In this dissertation, we will show that a data-driven inference engine is another approach that has not yet been explored, although it is an important alternative for the goal-driven approach in some specific cases.
In this chapter, a short introduction to knowledge systems and inferencing techniques is given. Following this, a detailed comparison between forward an backward chaining, together with a motivation for choosing forward chaining as inferencing technique. To conclude this chapter, the concepts of KAN, a tool for building knowledge systems that uses forward chained inferencing, will be explored.

3.1 Introduction : Shopping in Artificial Intelligence

A great deal of research in the world of Artificial Intelligence concerns knowledge systems and at the Artificial Intelligence Laboratory of the VUB, some major developments concerning knowledge systems have been performed. Prof. Dr. Luc Steels and his team introduced the "knowledge level".
The knowledge level describes the contents and the use of the knowledge by the system for purpose of reason. The focus on representing knowledge and its implementation, lies in the symbol level. An expert system (knowledge system) will be explored on the knowledge level before the system is implemented on symbol level. This new paradigm of modelling as a foundation for knowledge acquisition and design is an important evolution in knowledge systems. In this section, a short introduction to knowledge systems (using this paradigm) is given.

3.1.1 Knowledge systems

The content of this section is based on [Ste92] in which several examples are worked out, one of which is the example of the *finches*. In this example the knowledge system contains knowledge about different types of finches, i.e. what colour of beak, back and wings etc. The goal is to classify a finch into its type. In the figures (3.1 to 3.4) and in the examples in section

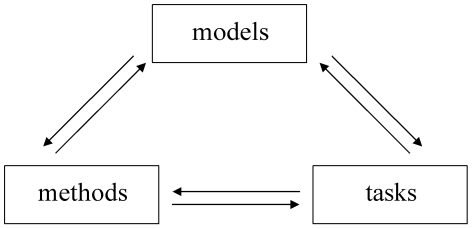

Fig. 3.1: Three perspectives of the knowledge level

3.3 on KAN, parts of this example are used and the complete example can be found in [Ste92].

Basically, knowledge systems are used to reason about knowledge that is stored in a knowledge base. The main characteristics of a this system are :

- they are capable of reasoning
- they provide a solution when an algorithmic solution is not feasible
- they are based upon the model of a human expert
- they have an explicit representation of their internal structure and functioning
- they have an immediately useable interface.

In order to create a knowledge system one has to go through a number of different phases. Knowledge has to be extracted from a human expert and a knowledge model constructed, then the symbol level (internal structure) has to be created and an inference engine (internal functioning) provided. The intention of this system is to solve problems through reasoning with the knowledge that is available, and for this the representations and mechanisms used in the problem solving process are grouped in one component, called the problem solver. In the next paragraphs the knowledge level and the symbol level are explained in more detail and inferencing techniques are discussed in section 3.1.2.

The ***knowledge*** level is not really about the representation of the knowledge, nor about the concepts of programming, but it denotes knowledge and its use. The intention of this level is to think about knowledge independently from the implementation. For this, in addition to the split between the symbol level and knowledge level, there is a split of the knowledge level into three perspectives: tasks, models and methods. The strong interaction between these three perspectives is depicted in figure 3.1.

The *task perspective* refers to the question "what needs to be done?" and this perspective describes tasks and task-structures. Tasks are the activities done by the problem solver. Mostly there is one main task to describe the application and this task is further split into smaller sub-tasks. The task-structure is illustrated in figure 3.2.

The *model perspective* refers to the question "what knowledge is available to execute the tasks?" and describes models and their mutual relationships. From this perspective, problem solving is seen as a modelling activity and the problem solver has various models of systems relevant to the application. There are three kinds of models, namely: domain models, case models and process models. *Domain models* contain the descriptions that are specific for a certain domain (e.g. for all finch types) and are plausible for all cases. *Case models* contain

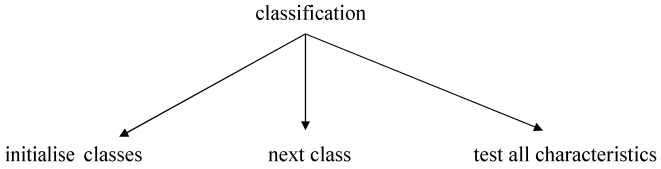

Fig. 3.2: Task-structure

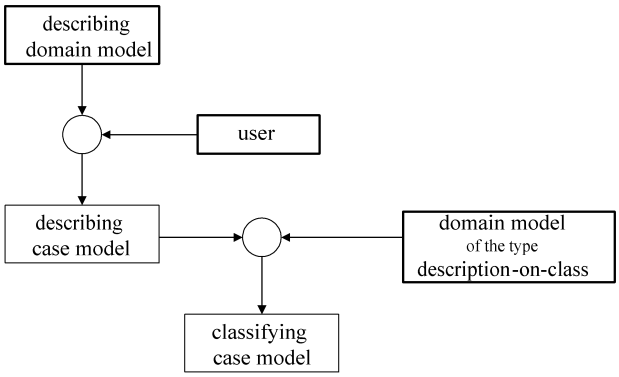

Fig. 3.3: Model diagram

descriptions specific to a certain case (e.g. for a specific finch we want to classify). *Process models* are modelling the problem solving process and are required when the problem solver has to determine explicitly what happens next. The relations between models are represented in a model diagram (figure 3.3).

The *method perspective* refers to the question "how and when is the knowledge used?". This identifies the methods and the control diagrams describing how methods work. Methods explain how the models are used in order to realise the tasks and control structures specify in what order tasks have to be executed and this information is represented in a control diagram (figure 3.4).

After an analysis of the knowledge level using tasks, models and methods, this knowledge has to be transferred to the ***symbol level***. To do so we need a way to represent models and methods. This is done through structured objects and procedures. A case *model* is represented by a structured object, and this object contains a set of facts to express the different descriptions that are part of a specific object. A domain model can have a declarative

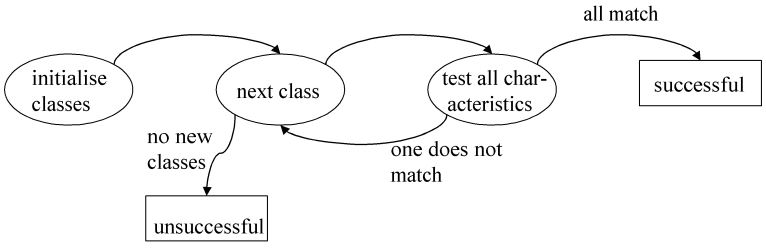

Fig. 3.4: Control Diagram

representation where domain model and solving methods are separated, or it can have a procedural representation where domain model and specific solving methods are merged. *Methods* are procedures and are formalised by a rule-based formalism. Each rule has a conditional and a consequential part being executed when the conditional part is satisfied. The basic unit on symbol level is the *problem solver* existing of three components

- a fact-base containing the models
- a rule-set containing the methods
- an inferencing mechanism that applies the rules on the fact-base.

The first component, the *fact-base* is the representation of the case model. Graphically this is done by a semantic network consisting of objects, roles, properties, attributes and relations. It contains the actual information of a model, but also accessibility information. How these networks are constructed goes beyond the context of this dissertation. Thereupon some kind of formalism is applied such that models (and their semantic networks) can be implemented and represented on the symbol level.
The formalism introduced by Steels is KAN. this formalism was implemented for educational purposes and it mirrors the structure of semantic networks. KAN is later explained in chapter 3.3. *Methods*, belonging to the second component of a problem solver, are procedures and can be expressed in any formalism. The KAN formalism is a rule-based formalism and its rules have a conditional and a consequential part. The conditional part describes a number of conditions, namely facts that should be present in one of the models. The consequential part is a set of actions that are executed when the conditional part was satisfied. Finally, a problem solver also has an *inferencing mechanism* to activate rules. This is explained in the next section 3.1.2.

3.1.2 Inferencing Techniques

Inferencing is a process used in knowledge systems for deriving new information from known (and assumed) information. The idea is to reason with knowledge, facts and problem solving strategies to draw conclusions. The two most important inference techniques are forward chaining (data-driven reasoning) and backward chaining (goal-driven reasoning).
In a *data-driven* reasoning process, the conditions of a rule are matched with the facts in the fact-base. If the conditions are satisfied, the actions are performed.
In a *goal-driven* reasoning process, the possible conclusions of a rule are matched since these are goals achieved by using the rule. The conditions are only tested if the rule can contribute to the achievement of the goal.
The skeleton of the algorithm to activate the rules is the same for forward and backward chaining and it contains three steps

- identification : what rules can possibly be fired
- selection : pick a rule
- execution : fire the rule.

Forward Chaining

Forward Chaining is a data-driven reasoning process that will try to generate possible solutions starting from initial facts. Rules are used for the derivation of new facts. The reasoning process continues until a goal state is reached or until no more new facts are deduced, but it may also go on forever. *Facts* are represented and stored in a working memory which is continuously updated. *Rules* represent the possible actions to be taken when specified conditions hold on items in the working memory. *Conditions* are patterns that must match items in the working memory. *Actions* involve adding or deleting items from the working memory. The reasoning algorithm is shown in figure 3.5.

The complete algorithm looks like this with `rule-set` the set of rules that still have to be looked at and `new-facts` an indicator to know if new facts were derived :

1. If the goal is reached, then the process stops. Otherwise the process continues at step 2.
2. `rule-set` is initialised with the set of all rules, `new-facts` is false.
3. If `rule-set` is empty and `new-facts` is false, then the process stops. If `rule-set` is empty, but `new-facts` is true, then the process continues at step 2.
4. The conditions of the first rule of `rule-set` are matched with facts from the model.
   - 4.1 If all conditions are matched, then the actions of the rule are executed. If new facts are derived, then `new-facts` is set to true and the process continues at step 5.
   - 4.2 If one of the conditions does not match, then `rule-set` becomes the rest of the `rule-set` without this rule and the process continues at step 3.
   - 4.3 If one of the conditions is not known, then :
     - 4.3.1 If it is allowed to gather information about this condition, then the information is collected and the matching restarts.
     - 4.3.2 If it not allowed to gather information, then `rule-set` becomes the rest of `rule-set` without this rule and the process continues at step 3.
5. If the goal is reached, then the process stops. Otherwise `rule-set` becomes the rest of `rule-set` without this rule and the process continues at step 3.

The interpreter is based on a cycle of activity, repeated until some specific goal is satisfied or until no more rules are fired (no new facts are derived). It can even cycle forever.

Backward Chaining

Backward Chaining is a goal-driven reasoning process. The process starts from a given goal state (a hypothesis) to try or to prove. Rules are used to derive new hypotheses. If the conclusions of a rule match the goal, the premises (conditions) of the rule become new hypotheses. These hypotheses are the new sub-goals to be reached. This process continues until the new derived hypotheses are known facts or when no new hypotheses can be found. The reasoning

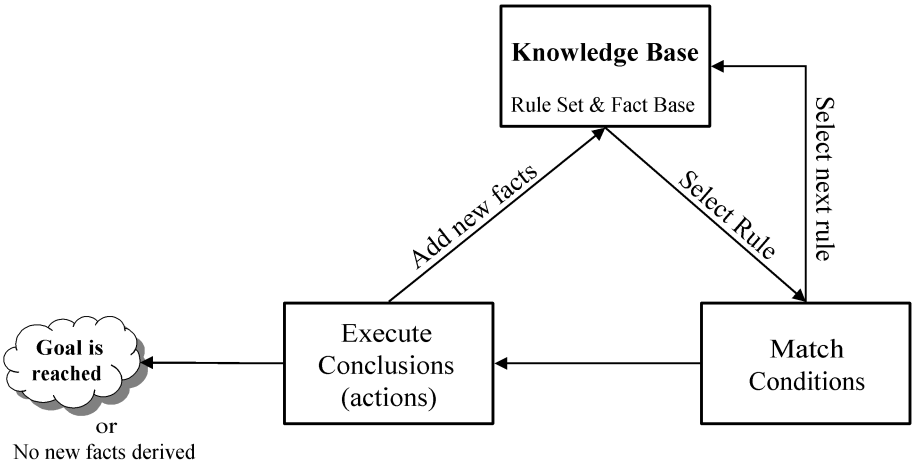

Fig. 3.5: Forward Chaining

algorithm is shown in figure 3.6.

The complete algorithm is described next. `Rule-set` represents the set of rules that still have to be looked at and `new-facts` stands for an indicator whether new facts are derived or not.

1. If the goal is reached, then the process stops. Otherwise the process continues at step 2.
2. `rule-set` is initialised with the set of all rules, `new-facts` with false.
3. If `rule-set` is empty and `new-facts` is false, then the process stops. If `rule-set` is empty, but `new-facts` is true, then the process continues at step 2.
4. If the conclusion (actions) of the first rule of `rule-set` contribute to reaching the goal, then try to match the conditions of this rule with facts from the model.
   - 4.1 If all conditions are matched, then the actions of the rule are executed. If new facts are derived, then `new-facts` is set to true and the process continues at step 5.
   - 4.2 If one of the conditions does not match, then `rule-set` becomes the rest of the rule-set without this rule and the process continues at step 3.
   - 4.3 If one of the conditions is not known, then :
     - 4.3.1 If it is allowed to gather information about this condition, then the information is collected and the matching restarts.
     - 4.3.2 If it not allowed to gather information, then a new problem solving process is started with as goal this condition.
       - 4.3.2.1 If the condition is still unknown after this process, then the rule-set becomes the rest of `rule-set` without this rule and the process continues at step 3.
       - 4.3.2.2 If the condition is known after this process, then the matching of conditions restarts.

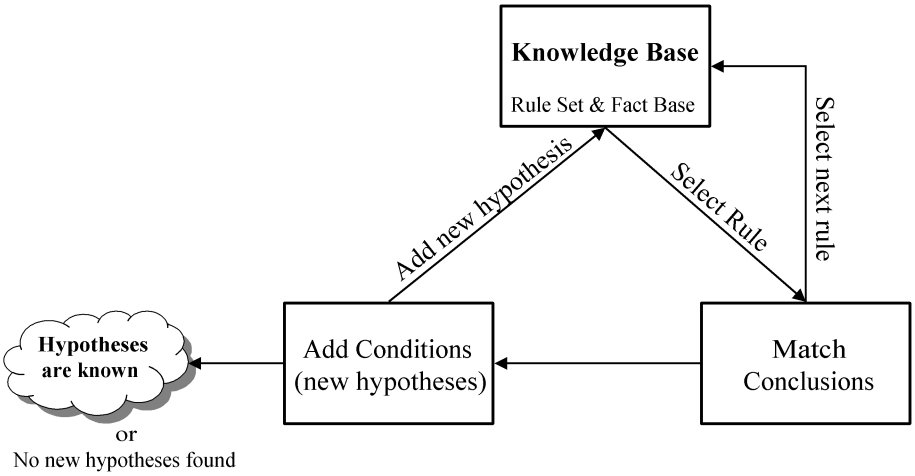

Fig. 3.6: Backward Chaining

5. If the goal is reached, then the process stops. Otherwise `rule-set` becomes the rest of `rule-set` without this rule and the process continues at step 3.

Summary

The two most important inference techniques are forward chaining and backward chaining. Forward chaining is a data-driving reasoning process. Starting from a goal, new facts are derived by using rules that fire (execute the conclusions) when their conditions hold. The process stops if the goal is reached or if no new facts are derived. The process may continue forever. Backward chaining is a goal-driven reasoning process. Starting from a goal that should be reached, new hypotheses (sub-goals) are derived by using rules that fire when their conclusions match a goal or a sub-goal. If there is a match, the conditions of the rule become new hypotheses. Goal-driven reasoning stops when the hypotheses are known facts or when no new hypotheses can be found. Section 3.2 compares both techniques.

3.2 Forward Chaining vs Backward Chaining

Two kind of inferencing techniques are forward chaining and backward chaining. In section 3.1.2 the inferencing algorithms of both techniques were explained. This section will give a comparison between forward and backward chaining and will make clear when to use one technique or the other.

3.2.1 Comparing Forward and Backward Chaining

Forward chaining is a data-driven reasoning process and the process starts from some initial facts and rules which are used to derive new facts. If the conditions of a rule match, the conclusions are executed and possibly new facts are added to the fact-base. The process finishes when the goal is reached or if no new facts are derived. In forward chainers, facts are represented and stored in a working memory which is continually updated. Forward chainers are also called *assertion-time inferencers*, because they reason and constantly conclude new

assertions (facts). Based on these new assertions, new inferencing is started by triggering rule-sets.
Backward chaining is a goal-driven reasoning process and starts from a given hypothesis (the goal). When the conclusions of a rule match, its conditions are added as new hypotheses. If all hypotheses are know facts or if no new hypotheses can be found, the process stops. In backward chainers facts reside both in the logic program (e.g. facts in a Prolog program) and in the environment system the chainer uses to reason. A backward chainer does not need to update a working memory. Instead it needs to keep track of what goals still need to be proved in order to reach the main goal. Backward chainers are also called *query-time inferencers* because their inference is triggered at the time a query is launched and not at the time an assertion is inferenced.

In *forward chaining* the inference process proceeds from the existing facts to a set of new facts, thus in a sense, all new facts are generated. As a consequence this results in the checking and firing of a substantial number of rules. These rule firings may do little to advance the system state towards a goal. Thus the use of many of the new facts generated in the process is somewhat questionable.
The *backward chaining* paradigm seeks only to prove the validity of a chosen fact expression. It is perhaps more computationally efficient than forward chaining, since it represents a goal-directed strategy that may eliminate checking of many superfluous paths. If more than one hypothesis is involved, backward chaining attempts to verify each one independently, and the number of computations may be far less if the problem were solved with some "memory" of truth of intermediate facts.

3.2.2 Selecting Forward Chaining or Backward Chaining

Whether one uses forward or backward chaining to solve a problem depends on the properties of the rule-set and initial facts.
Forward chaining is most useful when one knows all the initial facts, but does not have any idea what the conclusion might be. If it is known what the conclusion might be, or what specific hypothesis should be tested, forward chaining systems may be inefficient. One could keep going with the forward chainer until no more rules apply or until the hypothesis is added to the working memory. However, during the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to the working memory.
In summary, forward chaining may be better in the following circumstances

- there are a lot of things to prove
- there is a small set of initial facts
- a lot of different rules lead to the same conclusion
- it is difficult to form a goal or hypothesis to be verified
- most of the required facts are already in the initial database

According to [CM85] forward chaining should be used when the following criteria are met :

- backward chaining would be inefficient

- The formulas in question are likely to be queried about (such that the inferencing results will not go to waste).
- The forward chaining will terminate quickly.

Backward chaining is useful in situations where the quantity of data is potentially very large and where some specific characteristic of the system under consideration is of interest, i.e. various problems of diagnosis, such as medical diagnosis or fault-finding in electrical equipment.
In summary, backward chaining may be better in the following circumstances :

- trying to prove a single fact
- there is a large set of initial facts
- a lot of rules would be eligible to fire. (This potentially leads to the production of may extraneous facts.)
- relevant data should be acquired as a part of the inferencing process (e.g. by asking questions).

Therefore the decision between forward and backward chaining comes down to:

- a lot of data and few goals $\Rightarrow$ use backward chaining
- few data and many possible consequences $\Rightarrow$ use forward chaining
- no goal $\Rightarrow$ use forward chaining
- equal number of facts and goals $\Rightarrow$ use either

In favour of Forward Chaining A first property that pleads in favour of forward chaining, is described in [Sch90]. They "prove" that forward chaining somehow subsumes backward chaining. Of course, due to Turing completeness of both backward an forward chaining, the simulation of forward chaining by backward chaining is indeed also possible. Nevertheless, simulating forward chaining by backward chaining appears to be very clumsy. Simulating backward chaining by forward chaining on the other hand, is not so difficult. One needs two sets of rules. These two sets of rules are :

- a set of control rules that implement backward chaining. These are goal-splitting and goal-fusing rules.
- a set or rules for handling immediately solvable goals

The basis of their strategy is to split goals that are not immediately solvable into subgoals. Each of these subgoals then becomes a precondition of some rule, and once the conjunction of the subgoals is satisfied, "fusing" (the reverse of splitting) productions return to the parent goal. If all subgoals are solved, then their parent goal is also satisfied.

Secondly, and most importantly for our work, when using forward chaining, it is possible to "grab" state. When the problem solver has finished its reasoning, a set of facts is available,

so the state of the forward chaining problem solver is known. However, when a backward chaining problem solver has finished reasoning, we will know whether the goal was reached or not, but no state is preserved. Furthermore, with forward chaining, the problem solving process can be suspended, its state preserved, and at a later point in time the solver can be resumed with this state. We will call this property the persistence property of forward chaining.

Persistency Property of Forward Chaining

State can be preserved and retrieved between two problem solving sessions.

The reason why we stress this property so much is that we want to reason about the structures of code in a programming environment. A programming environment is per definition an interactive environment. When our forward chainer is reasoning we want to switch back to this environment from time to time. The reasoning stops for a while and will restart at some time in the future. During these two periods, the reasoning process is not active, but its state should be preserved. During these periods the reasoning process is inactive but its state (when the reasoning stopped) should be available when the process continues. For example we would like one problem solver to pass its state to another one before being "destroyed" such that when the new solver starts, at what moment soever, it use that state. Thanks to the persistency property it is possible for state to survive the problem solving process.

3.2.3 Conclusion

For the experiments in this thesis Forward Chaining was chosen. This choice is based on two good reasons.
First of all, the goal of a programmer is to write correct and clean code. This is the closest description to the goal we can think of. This means that the goal is to vague to offer to a backward chainer, therefore forward chaining is recommended.
Secondly, the context of this thesis is to reason about the structures of code in a programming environment. A programming environment is per definition an interactive environment and therefore we need the persistency property to preserve the state of the reasoning process during "non-active" periods.

3.3 KAN : Concepts

As previously mentioned, KAN was developed at the Artificial Intelligence Laboratory of the VUB. As Prof. Dr. Luc Steels and his team have performed major work on knowledge systems, have many years of experience in knowledge systems, and because they are our "own home-built experts", KAN is selected as a basis for the forward chainer that will be built in chapter 4.
KAN is a rule-based formalism using forward chaining for inferencing and it is used to implement the symbol level. The symbol level describes the data-structures and procedures necessary to run the problem solver. It also focuses on representing the knowledge and the implementation of it. The core of the symbol level is the problem solver which has a fact-base,

a rule-set and an inferencing mechanism. In KAN the fact-base and the rule-set have to be defined in advance in order to be able to run the problem solver.

3.3.1 Basic Entities

To indicate how KAN's fact-base and rule-set are created, this section describes the structure of KAN. For the syntax to define the KAN structures, we refer to [CSBG91] and [Ste92]. The structure of KAN's grammar was added in appendix A and shows how KAN's structures relate to each other.

KAN objects in general

The fundamental entities of KAN are KAN objects. These objects are described following a general pattern.
First of all, KAN objects are implemented by a *frame*. This is a data structure to associate information with the name of a KAN object. A frame has a set of slots. Each *slot* has a name and a value (a filler).
There are different types of KAN objects and each type has specific slots with some appropriate fillers. These different types are : vocabulary, object[1], object-type, rule-set, problem-solver, property, attribute, role, relation, fact and rule and they are divided in two major categories : global and local KAN objects.
A *global KAN object* is referred to by its unique name. A l*ocal KAN object* is always local to a global object and is referred to by its name and owner. The name of a local KAN object is unique within the scope of the global object.
Types of global KAN objects are : vocabulary, object-type, object, rule-set and problem-solver. Types of local KAN objects are : property, attribute, role, relation, fact and rule. Each of these objects have certain slots. Some of these are optional, others are mandatory and for those at least one value has to be specified. In table 3.1 all object types are listed together with their slots and possible fillers for these slots.

A problem solver in KAN

As mentioned before, a problem solver, the basic component on symbol level, has three components. These are a fact-base, a rule-set and an inferencing engine. Next we will explain how these components are created in KAN. Table 3.1 shows all objects with their possible slots and for each slot is indicated whether it is an optional slot or not and what its possible fillers are. Although the KAN syntax will not be explained, examples (parts of the finches example) using this syntax were added.

fact-base To get to a fact-base, first of all a vocabulary is required, and object-types and objects need to be defined. A vocabulary consists of descriptors and in order to come to facts we also need descriptions and object descriptions. These elements are described next.

A *vocabulary* groups a set of descriptors that can be used to describe the features of an object. A descriptor is a property, attribute, role or relation and is always local to a vocabulary. A vocabulary has a `documentation` slot that is optional and that indicates by default

[1]When talking about a KAN object of the type 'object', the name 'object' is used. Otherwise we use the term 'KAN object'.

global object	locals	slots	optional	possible fillers
vocabulary		documentation	yes	string
	property	to-ask	yes	string
		to-find-out	yes	*ask*, lookup
	attribute	to-ask	yes	string
		to-find-out	yes	*ask*, lookup
		possible-values	no	list of symbols
	role	to-ask	yes	string
		to-find-out	yes	ask, *lookup*, create, ask-user
		filler-type	no	object-type
	relation	to-ask	yes	string
		to-find-out	yes	*ask*, lookup
		argument-types	no	list of object-types
object-type		vocabulary	no	vocabulary
object	fact	truth-value	yes	*true*, false, unknown
		justification	yes	string
rule-set		object-type	no	object type
		goal	yes	descriptor
	rule	if	yes	conditions
		then	yes	actions
problem-solver		rule-set	no	rule set
		object	no	object
		goal	yes	descriptor

Table 3.1: KAN objects

that no documentation is available. e.g. (define vocabulary finch-vocabulary)

A *property* is a descriptor that is used to describe a boolean feature of an object. Its slots are to-find-out, to-ask and documentation and these are all optional. To-find-out can have the values lookup and ask and by default the value of this slot is ask. Lookup indicates that the value of the property only can be found by reasoning. Ask indicates that the user should be asked about this value. To-ask contains the string that should be used when the user is asked about the value of the property and by default the value of this slot is "Is property true for object?". Documentation slot is the same as for vocabulary. e.g.
```
(defīne (property finch-vocabulary) threatened
  (to-ask ‘‘Is the finch $o threatened with extinction?’’))
```

An *attribute* is used to associate a value with an indicator for an object. The indicator is the name of the attribute and the value is an element from the set of allowed values for this attribute. An attribute has the slots possible-values, to-find-out, to-ask and documentation. To-find-out, to-ask and documentation are analogue to the ones of property. The default filler of to-ask is "What is the value of attribute for object?". Possible-values is obligatory and lists symbols that possibly can be the value for an attribute. e.g. (defīne (attribute finch-vocabulary) species
```
  (to-find-out lookup)
  (possible-values r̄ed-eared-firetail b̄eautiful-firetail
  diamond-firetail p̄ainted-finch c̄rimson-finch
  star-finch r̄ed-browed-finch ḡouldian-finch ))
```

A *role* associates one object with another object (the filler of the role). The object-type of the filler is fixed and a role has the slots filler-type, to-find-out, to-ask and documentation. Filler-type specifies the object-type for a filler of the role and this slot is obligatory. To-find-out can have the values lookup, ask, create and ask-user. Lookup means the role only can be derived by internal reasoning. When the slot has the value ask, the user is asked about the role. Create means the system should automatically create a filler for the role when one is need. Ask-user is practically the same as create, but the user is asked for a name for the filler. To-ask is again optional with as default "what is the value of role for object?" and this question is used when the lookup value was ask. Documentation is the same as before. e.g. (defīne (role finch-vocabulary) food
```
  (filler-types food)
  (to-find-out create))
```
(with food a predefined object-type).

A *relation* expresses the relationship between an object and a number of other objects (the arguments of the relation). The number of arguments and their object-types are fixed. A relation has the slots argument-types, to-find-out, to-ask and documentation. Argument-types is obligatory and lists the object-types indicating what kinds of objects are possible and in what order. To-find-out, to-ask and documentation are analogue to the ones of property. The default filler of to-ask is "Select arguments for relation for object?". e.g. (defīne (relation finch-vocabulary) looks-like
```
  (argument-types finch finch)
  (documentation ‘‘whether two finches look like each other’’))
```

A *description* is not a global or local object, but is used to create facts and to formulate conditions in rules. There are simple descriptions and global descriptions. A *simple description* can be a basic description, a negated description or a unknown description. A *basic description* is directly based on a descriptor (property, attribute, relation or role) and its necessary complements. Because there are four possible descriptors for a vocabulary, there are four possible basic descriptions. The one based on a property only asks for the property and not for additional elements. `e.g. threatened`
The basic description based on an attribute asks for the attribute and a value. This value should be a possible value for the attribute. `e.g. (species diamond-firetail)`
Based on a relation, the description consists of the relation and its various elements grouped in a list. Each argument must be the name of an object or object-description (object-descriptions are explained later). `e.g.(looks-like (a finch))`
A basic description based on a role consists of the role and its filler. The filler is again the name of an object or object-description. `e.g. (beak (a beak))`
A *negated description* is the negation of a basic description and an *unknown description* indicates that a basic description is not known. `e.g. (not threatened)` and `(unknown (species diamond-firetail))`
All these descriptions are simple descriptions. *Composite descriptions* consist of such a simple description and an object-description. The simple description is applied to the object the object-description is referring to. `e.g. (== (>> beak) small)`

An *object-description* can be used in various places to refer to an object. There are relative and absolute object-descriptions. Object-descriptions contain a number of names of roles that make up a path that goes from one object to another object through the object hierarchy. In case of a relative object-description the path is followed starting from the current context. `e.g.(>> rol1 rol2 .... roln)`
With absolute object-descriptions the name of an object has to be provided as well and the path starts from this object. `e.g. (>> rol1 rol2 .... roln of myfinch)`

Object-types are global KAN objects and they are used to specify new types of objects. New objects (KAN objects of the type object) can be created by using this object-type. An object-type has the slots `vocabulary` and `documentation`. The `documentation` slot is again optional with a default filler indicating that there is no documentation available. The `vocabulary` slot contains the vocabulary that defines the descriptors upon which descriptions are based and these descriptions are used in facts associated with objects of this object-type. An object-type has no local objects. `e.g. (define object-type finch (vocabulary finch-vocabulary))`

Objects are global objects specified by an object-type. They have a fact-base which is a set of facts that are known about the object. An object has only one optional slot documentation indicating by default that no documentation is available. `e.g. (define finch myfinch)`

Facts are defined local to an object which results in adding the fact to the objects fact-base. A fact is identified by a description and has two slots `truth-value` and `justification`. `Truth-value` is an optional slot with possible values `true`, `false` or `unknown`. By default this value is `true`. `Justification` is optional and the value is by default set to "I was told". This slot is used to record how the fact was derived. `e.g. (define (fact finch-vocabulary)`

```
threatened
  (truth-value false))
```

rule-set The second component, the rule-set is created by the KAN objects rule-set and rules that are built on conditions and actions. A *rule-set* is a global KAN object denoting a collection of rules. Each rule-set assumes a particular object-type for all the rules in the set. When the rule is fired, this object will act as the context. The slots of the rule-set are `object-type`, `goal` and `documentation`. `Object-type` is obligatory and contains the object-type of possible contexts for this rule-set but it also implies a possible vocabulary that is compatible with the vocabulary assumed by the rules. `Goal` is optional with as default filler an indication that no goal was specified. Otherwise, the goal is a descriptor that is also a member of the vocabulary associated with the object-type. This descriptor will act as the default goal. `Documentation` is again a string with some extra information about the rule-set.

```
e.g.  (define rule-set finch-rules
  (object-type finch)
  (documentation ‘‘contains rules for recognizing finch species’’))
```

A *rule* is defined local to a rule-set and this corresponds to adding the rule to the rule-set. A rule has a set of conditions and a set of actions. While problem solving, the inference engine will trigger and/or fire the rule. Triggering means that the conditions of the rule are matched. If the result is true, the rule is fired : the actions are executed. A rule has the slots `if`, `then` and `documentation`. `Documentation` is the same as before. The `if` slot contains a condition. It is optional and when it is not specified by the user, the rule fires as soon it is triggered. The `then` slot contains a list of actions. This slot is also optional and when not specified nothing is done when the rule is fired.

```
e.g.  (define (rule finch-rules) diamond-firetail-rule
       (if
        (beak red)
        (lores black)
        (eyebrow insignificant-uniform)
        (ear-patch white)
        (crown grey))
       (then
        (conclude (species diamond-firetail))
        (communicate ‘‘the species is diamond firetail’’)))
```

A *condition* can be a condition based on a description (simple or composite condition) or a boolean condition. A *simple condition* corresponds to a simple description : basic, negated and unknown. `e.g.  (not (lores black))`. A *composite condition* corresponds to a composite description. `e.g.  (== (>> beak) small)`.
A *boolean condition* (conjunctive or disjunctive) combines a list of conditions. For conjunctive conditions all conditions in its list must be true in order to have this condition evaluated to true. It fails as soon as one of its elements is false or unknown. A disjunctive condition evaluates to true if one of its elements is true. This condition fails if all of its elements are false or unknown.

Actions occur in the `then` part of a rule. The possible actions are conclude, investigate, ask and communicate. *Conclude* is used to add a fact to the current object (the context) and it takes a `description` as argument. The fact based on this description is added to the fact-base. `e.g. (conclude (species diamond-firetail)).`
Investigate is used to start up a sub-problem solving process. It can take as arguments a `rule-set`, a `goal` and `object-description`. The `rule-set` is obligatory and refers to an existing rule-set. The `goal` is possibly optional and it is a `descriptor` or a combination of `descriptor` and `object-description`. When rule-set has a filler for its `goal` slot, that filler is the default value when no `goal` is specified. If there is no filler for the slot `goal` of the rule-set, then this slot is obligatory. `Object-description` is optional if the goal is specified. By default it denotes the current context. This argument will be the context for the new rule-set. When an investigate action is executed, the main task will be suspended and a new task is started until this sub-goal or the top-goal is reached. `e.g. (investigate food-rules food-class (>> looks-like)).`
Ask is used to force KAN to ask the user a question about a certain descriptor which can lead to adding new facts to the system. As arguments this action takes a descriptor and an object-description. The descriptor is obligatory and should be a descriptor in the vocabulary of the object denoted by object-description. Object-description is optional and will be the current context by default. `e.g. (ask lores (>> looks-like)).`
Communicate is used to communicate information to the user. Its argument are a string, a descriptor and an object-description. The string is obligatory and can contain special symbols which are not further explained here. The descriptor is optional and will be used to retrieve a fact of the fact-base of the current context. Object-description denotes an object which will act as the context to retrieve the fact. It is also optional and by default the current top context of the rule will be taken. `e.g. (communicate ‘‘the species is diamond firetail’’).`

KAN problem solver A problem solver at the symbol level has three components. Beside facts and rules, a problem solver has also an inference engine. The KAN problem solver is a KAN object that is constructed before the problem solving process is started and it is independent of the inference engine that is also part of the symbol problem solver. To start a problem solving process, KAN needs an object to reason about, a rule-set to run and a goal to end the reasoning process. A problem solver has the slots `object`, `rule-set` and `goal`. The `object` slot is obligatory and contains the top level object (the top of the model hierarchy) used in the problem solver. The `rule-set` slot is also obligatory and it contains the name of the rule-set the problem solving process will use to reason. `Goal` contains a descriptor that will indicate the goal of the problem solving process. When there is a fact in the object's fact-base which has the goal as descriptor, the problem solving stops. This slot is optional when a `goal` was specified in the `rule-set`. `e.g.`

```
(define problem-solver main-problem-solver
  (object the-finch)
  (rule-set finch-rules)
  (goal species))
```

3.3.2 Inference engine

Up to now facts, rules and problem-solver are defined in KAN. The third component in KAN is an object with the necessary elements for the problem solving process, but it does not

contain the inferencing technique to do the solving. Therefore, a fourth component is needed : the inference engine.
KAN uses forward chaining as an inference technique to reason upon its facts and rules. When a problem solver is started, the rule-set that is the filler of the corresponding slot in the problem-solver, is triggered. Triggering a rule-set means triggering each rule, starting with the first rule in the rule-base and subsequently making its way down the rules.
The triggering process is a cycle : when the last rule is triggered and new facts have been deduced because a rule fired, the engine will repeat the cycle, starting with the first rule. Triggering a rule is matching the conditional part of the rule and firing takes place when all conditions of the rule match. When a rule fires, the action part of the rule is executed. When all the actions of a rule are executed, the engine triggers the next rule. A new rule-set can be triggered with the `investigate action`. This action is finished when the engine exits the rule-set. Exiting a rule-set happens when no new facts are deduced during the last cycle or when the top goal or local goal is reached. When several consecutive calls to investigate are made, there will be a chain of goals and subgoals. The engine will only leave the rule-set if the goal of the rule-set (the one that is supplied in investigate) or the top level goal (the one of the problem solver) is reached.

The KAN-matcher is called to check whether the condition of the rule is true. It handles the if-part as a whole. The descriptions in the conditions are compared with the descriptions of the facts in the current context. The result is computed by taking into account the negated or unknown conditions and the truth-value of the facts. If no fact with a similar description is found and the descriptor (on which the description is based) has a to-find-out value ask, the matcher will transform the condition into a question to be asked to the user. The answer is stored as a new fact, which is then used to compute the result of the condition. An unknown condition will succeed if there is no fact in the fact-base and the to-find-out value is lookup or when the truth-value of the fact is unknown.

3.4 Summary

First an introduction to knowledge systems in general and overview of some inferencing techniques was given. Basically, a knowledge system is used to reason about the knowledge that is represented in a knowledge base. When such a system is created, knowledge is extracted from a human expert, a knowledge level is setup and finally the symbol level is created. This is the actual implementation of the system in which the main components on the knowledge level are objects, attributes, properties, roles and relations.
The knowledge level is then translated into a semantic network before this network is expressed on the symbol level. How a semantic network is to be implemented on symbolic level, is defined in a formalism. An example of such a formalism is KAN.
A knowledge system also needs an inference engine to do the actual reasoning. The two types of inferencing looked at in this thesis are forward chaining (data-driven reasoning) and backward chaining (goal-driven reasoning). KAN uses forward chaining.
Secondly, a more thorough comparison between forward and backward chaining was given in section 3.2 and we motivated the selection of a forward chainer for the purpose of this thesis. Chapter 4 explores a forward chainer based on KAN written in Squeak. Finally, the last part

of this chapter specified the concepts of the KAN formalism.

Chapter 4

A Forward Chainer on top of Smalltalk

In chapter 2 we explained co-evolution, together with an overview of the research that has been so far concerning co-evolution. All of the systems, developed in the context of this research, are based on Prolog and use backward chaining as an inferencing process. Nevertheless, forward chaining is an alternative approach with a lot of potential. In section 3.2 it was shown that this approach has two important advantages, which are the persistency property (implying preservation of state) and the allowed absence of a goal. To prove our thesis that a forward chainer is equally beneficial to a backward chainer, we will prove by means of some experiments that in these two cases a forward chainer does not fail, but first we will build a prototype forward chainer in order to do these experiments.

Since KAN (explained in section 3.3) was developed for educational purposes by people who know a lot about knowledge systems, this tool is considered to be rather simple but representative to build knowledge systems. When building a forward chainer, it was not our intention to do this research all over again and therefore, our forward chainer is based on KAN. The environment in which this was done is Squeak, an open source Smalltalk environment. In this chapter we introduce Squeak in more depth, together with the reason why it has been chosen as the programming environment. After this introduction to Squeak, the forward chainer SqueakKAN that was implemented, is explained in more detail. It was the intention of this forward chainer to show how it can be used to reason about object-oriented programming structures, therefore the last part of this chapter links SqueakKAN with Smalltalk in order to be able to reason upon the structures of this object-oriented programming environment.

4.1 Squeak : A squeak for Smalltalk

The forward chainer that will be built, was implemented Squeak. This section will first explain what Squeak is, how it came into existence and what it tries to achieve.

4.1.1 Open Source Software

Squeak is an environment for Smalltalk that was released on the Internet in September 1996. It was developed by Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. They wanted an environment “to build educational software that could be used – and even

programmed – by non-technical people, and by children" ([IKM+97]). The intention was to get a highly portable and small programming language based upon the principles of Open Source Software.
The idea of Open Source Software is to let the source code be freely available on the Internet and use contributed code (bug fixes, enhancements, redesigns) to develop the code ([Guz99]). Users are therefore able to examine source code for every part of the system, including the virtual machine itself.
In the squeak license agreement the right is explicitly granted to use Squeak in commercial applications royalty-free. The only requirement in return is that any ports of Squeak or changes to the base class library must be freely available on the Internet ([IKM+97]). This license agreement sustains the idea of Open Source Software and leads to a continuous development and sharing of Squeak by its users community and in this way, Squeak is very alive thanks to this still growing community.

4.1.2 An Implementation of Smalltalk

The developers of Squeak wanted to create a practical programming environment with platform independent support for colour, sound and image-processing. These three elements are very important in Squeak and the fact that the developers work at the Disney Studios has probably something to do with this. At first, Java was considered to create such an environment, but at that time Java was still too immature and unstable, and although existing Smalltalk-implementations met the technical requirements, there was a lack of control over graphics, and features for sound. Furthermore the Smalltalk engine itself was not open enough and could not be freely distributed over the Internet. Therefore the developers decided to build a new Smalltalk.
They started from an existing Apple Smalltalk-80 implementation and created a new image, a new interpreter and a translator to compile the virtual machine to C (written in Smalltalk). Their philosophy was to write everything in Smalltalk, so a Smalltalk virtual machine was written, then translated into C in order to compile the C code to a native machine executable. This executable was used to run the image and from then on, most of Squeak has been written in Squeak.

4.1.3 Why use Smalltalk?

Smalltalk is a pure object-oriented programming language and is consistent to the object-oriented paradigm. This paradigm is characterised by encapsulation, abstract data types, inheritance, polymorphism and a class plays the role of a template. In this way it contains all the information to create objects which are instances of a class. In Smalltalk everything is an object and all computing is carried out by sending a message to an object to have one of its methods invoked. Since everything happens in an object-oriented way of thinking, Smalltalk is the ideal language to start from when reasoning upon object-oriented programming structures.

4.1.4 Why Squeak as a programming environment for Smalltalk?

Unlike other Smalltalk environments, Squeak is quite young and it has a community that is very much alive. Squeak favours the idea of Open Source Software and allows the community to share their experiences, experiments and contributions to the environment. Furthermore, Squeak keeps developing, is freely available and it is ported to many platforms. Having an

active community also implies a lot of support for the novice and even experienced programmer. Because of its openness, Squeak is gaining popularity in academic societies. Currently, no complete and up-to-date information is available on Squeak, but this is being worked on and if anyone wants to get Squeak or more information on Squeak, it is found at [squ].

4.2 SqueakKAN : A Forward Chainer in Squeak and based on KAN

Now that Squeak and KAN are explained, SqueakKAN, a forward chainer written in Squeak and based on KAN, will be introduced. Squeak was chosen because it is a rather new environment with rising popularity. As explained earlier, KAN is considered to be decent for knowledge systems and forward chainers. The basic entities (explained in section 4.2.2) are used to construct a problem solver with facts, rules and a goal and when this solver is built, the inference engine can try to reach the goal by solving it.

4.2.1 Recapitulation of important topics

Before going into depth on the implementation of SqueakKAN, a small summary of the most important topics is given. A knowledge system has a knowledge level and a symbol level. SqueakKAN implements the symbol level. The basic component of this level is a problem solver consisting of

- a fact-base
- a rule-set
- an inference engine

To implement the fact-base one needs vocabulary, descriptors, descriptions, object-descriptions, object-types and objects. The rule-set needs rule-set, rules, conditions and actions. The inference engine is the algorithm to steer the problem solving.
SqueakKAN's basic entities and inference engine are combined such that each entity knows its function in the problem solving process. The basic entities are first explained without the problem solving aspect and then the problem solving itself is explained.
SqueakKAN is implemented directly into Squeak and when defining a knowledge system in SqueakKAN the entities have to be constructed explicitly since a parser was not implemented. In appendix B SqueakKAN's structure is described as a grammar which can easily be used as a basis when writing a parser.

4.2.2 Basic entities

The class diagrams of the basic entities of SqueakKAN can be found in figures 4.2 through 4.9. These diagrams give a better understanding of SqueakKAN. In the examples, used to explain the construction of the basic entities, the most specified constructor is used and all SqueakKAN objects are subclasses from `FCObject` (figure 4.1).

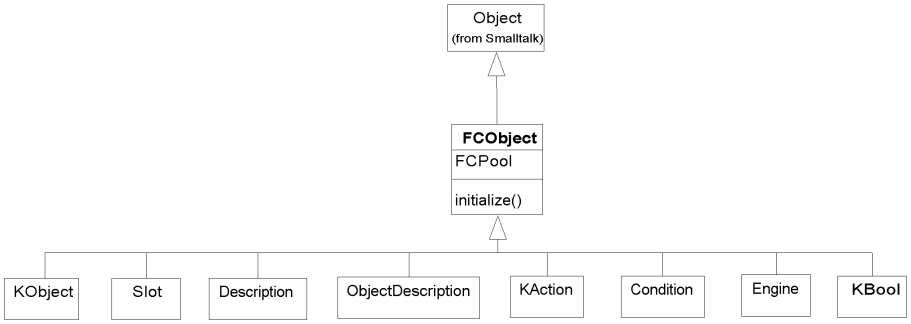

Fig. 4.1: SqueakKAN's top-level object : `FCObject`

Global and Local Objects

There are global and local objects with both a certain number of allowed slots. The `PermittedSlots` of an object indicate what slots are allowed for the object. A slot can be added with `addSlot:` and retrieved with `getSlot:`, however to add a slot, it has to be a member of `PermittedSlots`.
Global objects can have local objects, and `PermittedLocals` specifies the kind of locals that are possible for the global object. A local object can be added with `addLocal:`. When creating a *local object*, an owner has to be specified such that the local object can be added to the locals of this owner.
In table 4.1 an overview of the objects together with their possible local objects and their possible slots is given. The classes that were added to serve the purpose of reflection on Smalltalk, are explained in the section 4.3 and will be skipped for the time being.

fact-base

A fact-base consists of a vocabulary, descriptors, object-types, objects and facts. Also descriptions and object descriptions are explained in this section.
The class diagrams of global and local objects are shown in figure 4.2. Slots that are used with these objects are Documentation, ToFindOut, ToAsk, PossibleValues, FillerType, ArgumentTypes, VocabularySlot, ObjectTypeSlot, TruthValue and Justification. The class diagrams of these slots are shown in figure 4.3.

A *vocabulary* (class `Vocabulary`) groups a set of descriptors that can be used to describe features of objects. A vocabulary is a global object with a `Documentation` slot that can contain extra information on the vocabulary, this slot is optional and by default its filler indicates that no documentation is available. Descriptors are local objects with a vocabulary as owner. A vocabulary can be created with

```
Vocabulary withName: #exampleVocabulary
   andDocumentation: (Documentation withFiller: 'I am an example vocabulary.')
```

#exampleVocabulary is a Smalltalk symbol that, from now on, can be used to refer to the vocabulary. aDocumentation is a Documentation slot with a string as filler.

global object	locals	slots	opt.	possible fillers / *default filler*
Vocabulary	Property	ToAskProperty	yes	*'Is name true for $o?'*
		ToFindOut	yes	*ask*, lookup
	Attribute	ToAskAttribute	yes	*'What is the value of name for $o?'*
		ToFindOut	yes	*ask*, lookup
		PossibleValues	no	collection
	Role	ToAskRole	yes	string : *'What is the value of name for $o?'*
		ToFindOutRole	yes	ask, *lookup*
		FillerType	no	objectType
	Relation	ToAskRelation	yes	*'Select arguments for relation for $o ?'*
		ToFindOut	yes	*ask*, lookup
		ArgumentTypes	no	collection of object-types
	Smalltalk -Value	ToFindOutValue	yes	*lookup*, fromSmalltalk
ObjectType		VocabularySlot	no	vocabulary
Obj	Fact	TruthValue	yes	*KTrue*, KFalse, KUnknown
		Justification	yes	*'I was told.'*
SmtObj				
RuleSet		ObjectTypeSlot	no	objectType
		Goal	yes	descriptor
	Rule	If	yes	conditions
		Then	yes	actions
ProblemSolver		RuleSetSlot	no	ruleSet
		ObjectSlot	no	object
		Goal	yes	descriptor
All objects		Documentation	yes	*'No documentation available'*

Table 4.1: SqueakKAN objects

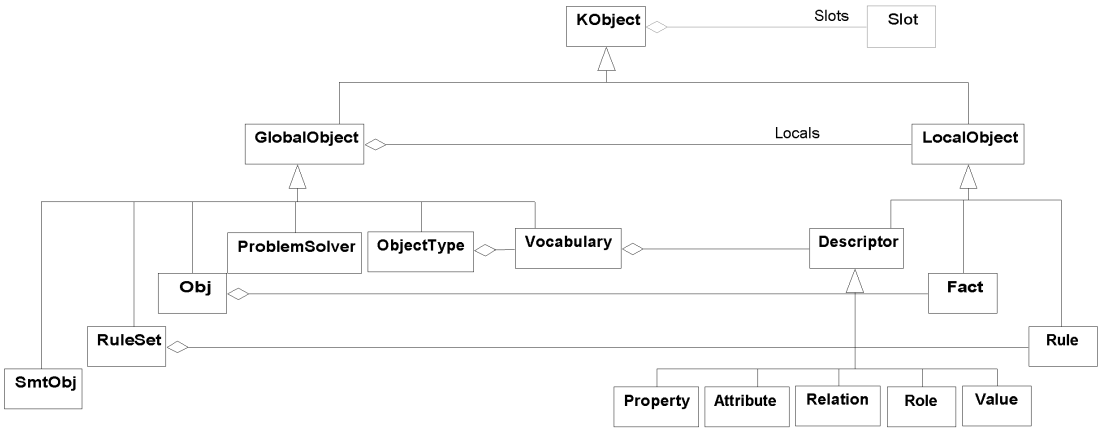

Fig. 4.2: Global and Local Objects

Descriptors are property, attribute, role, relation and value. Value was added for reflective reasons and is explained in section 4.3.
A *property* (class `Property`) is used to describe a boolean feature of an object. Permitted slots for a property are `ToFindOut`, `ToAsk` and `Documentation` and a property can be created with

```
Property withName: #exampleProperty
   andOwner: exampleVocabulary
   andDocumentation: (Documentation withFiller: 'I am an example property.')
   andToFindOut:(ToFindOut withFiller: #lookup)
   andToAsk: (ToAsk withFiller: 'Am I true?')
```

As said before, a descriptor has an owner vocabulary and `Documentation` is again a documentation slot. `ToFindOut` indicates how an unknown fact should be found. This can either be by asking the user `(#ask)`, or by internal reasoning `(#lookup)`. By default the value of this slot is `#ask`. `ToAsk` indicates what question should be asked when asking the user about the value of the property. When the default value for `ToAsk` is used, this slot is replaced with `ToAskProperty` with default value 'Is name true for $o?'. Name is replaced with the name of the property, and $o with the name of the object the property belongs to[1].

An *attribute* (class `Attribute`) allows to associate a value with an indicator for an object. The indicator is the name of the attribute and the value is an element from a predefined set of values that are possible for the attribute (possible values). Permitted slots are `PossibleValues`, `ToFindOut`, `ToAsk` and `Documentation` and an attribute can be created with

```
Attribute withName: #exampleAttribute
   andOwner: exampleVocabulary
```

[1]A property belongs to a vocabulary, an object is linked with the vocabulary by the object-type and facts that are added to this object are based on descriptors of that vocabulary.

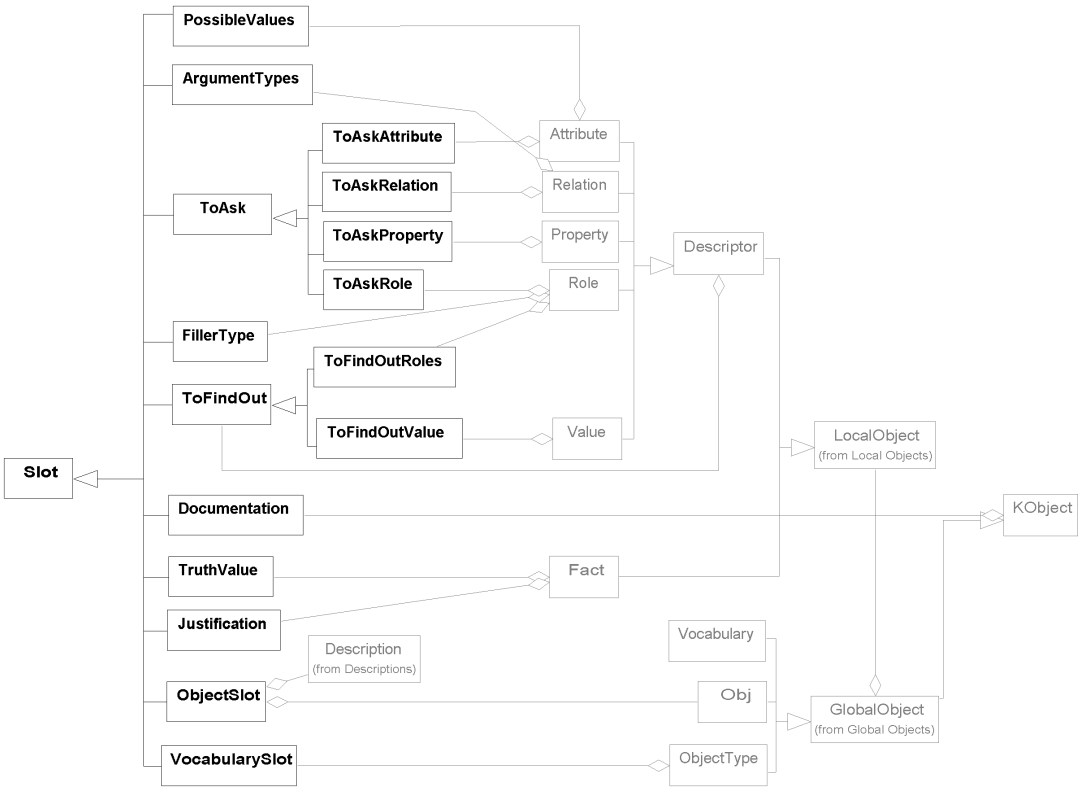

Fig. 4.3: Slots part 1 : used in objects of the fact-base.

```
    andDocumentation:(Documentation withFiller: 'I am an example attribute.')
    andToFindOut:(ToFindOut withFiller: #ask)
    andToAsk: (ToAsk withFiller: 'What is my value?')
    andPossibleValues: (PossibleValues withFiller:
                            (OrderedCollection  with: #value1
                                                with: #value2))
```

`Owner`, `Documentation`, `ToFindOut` and `ToAsk` are the same as in `Property`. `ToAskAttribute` has as default value 'What is the value of name for $o?'. Name is again replaced with the name of the attribute and $o with the name of the object the attribute belongs to. `PossibleValues` is a collection with the possible values of the attribute.

A *role* (class `Role`) is an association between two objects and corresponds to the name of the role. Permitted slots are `FillerType`, `ToFindOutRoles`, `ToAsk` and `Documentation` and this descriptor can be created with

```
Role withName: #exampleRole
    andOwner: exampleVocabulary
    andDocumentation:(Documentation withFiller: 'I am an example role.')
    andToFindOut:(ToFindOut withFiller: #lookup)
    andToAsk: (ToAsk withFiller: 'What is my value?')
    andFillerType: (FillerType withFiller: exampleObjectType)
```

`Owner`, `Documentation` and `ToAsk` are the same as in `Property`. When the `ToFindOut` slot is not specified, it becomes `ToFindOutRoles` slot which is analogue to `ToFindOut`, with possible values `#ask` and `#lookup`, but this time `#lookup` is the default value. `#ask` means the user will be asked about the role, `#lookup` indicates that the value of the role can to be found by reasoning only. The `FillerType` indicates of what object type the argument of the role is. This object type should already be defined, but object types are explained further on.

A *relation* (class `Relation`) expresses a particular relationship between an object and a number of other objects. Permitted slots are `ArgumentTypes`, `ToFindOut`, `ToAsk` and `Documentation`. The last three are the same as in `Property` and `ArgumentTypes` indicates what object type the arguments of the relation are. For example a relation can be created with

```
Relation withName: #exampleRelation
     andOwner: exampleVocabulary
     andDocumentation: (Documentation withFiller: 'I am an example relation.')
     andToFindOut:(ToFindOut withFiller: #lookup)
     andToAsk: (ToAsk withFiller: 'What is the value of my arguments?')
     andArgumentTypes: (ArgumentTypes withFiller:
                                 (OrderedCollection with: exampleObjectType))
```

Apart from descriptors, we also have *descriptions*. Descriptions are used to formulate conditions in rules and to create facts. The class diagram of descriptions can be found in figure 4.4. The three kinds of descriptions are composite descriptions, simple descriptions and send descriptions. The latter is used for the introspection of Smalltalk and is explained in the next section. A simple description can be a basic, negated or unknown description. For example a description can be created with

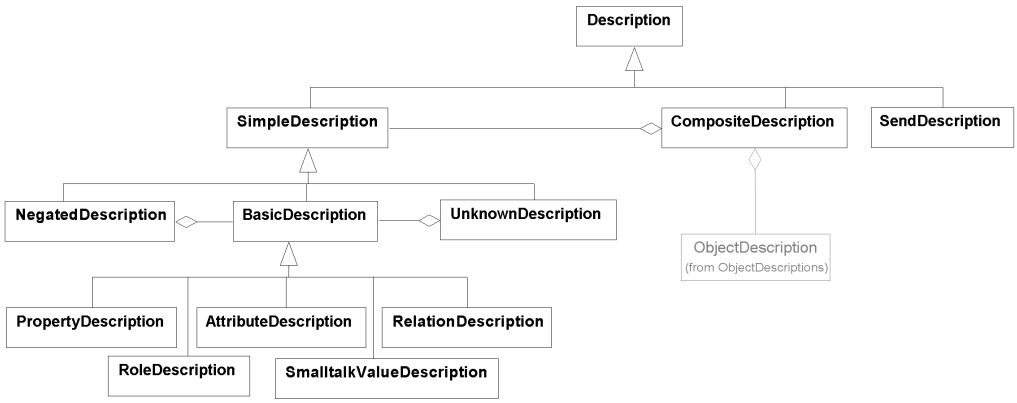

Fig. 4.4: Descriptions

```
CompositeDescription withDescriptor: (BasicDescription withDescriptor:
                                        exampleAttribute andValue: #value1)
   andObjectDescription: (RelativeObjectDescription withPath:
                                          (KCollection with: exampleRole)
```

A *basic description* (class `BasicDescription`) is based on a descriptor and thus it can be based on a property, an attribute with its values, a role with its filler and a relation with its arguments. The extra values provided when based on an attribute, role or relation are called the complements of the description. A *negated description* (class `NegatedDescription`) is the negation of a basic description and an *unknown description* (class `UnknownDescription`) indicates that the basic description's value is not known. A *composite description* (class `CompositeDescription`) consists of object description that refers to an object and a simple description that is applied to this object. Object descriptions will be explained soon.

An *object description* is used to refer to an object and this is carried out by specifying a path that leads from a certain context (object) through different roles to another object. There are relative and absolute object descriptions. A *relative object description* (class `RelativeObject- Description`) has a path that is followed starting from the current context. An absolute object description (class `AbsoluteObjectDescription`) follows the path starting from a specified object. When the path is empty, the result is the context itself. An object description can for example be created with

```
RelativeObjectDescription withPath: (KCollection with: exampleRole)
```

The class diagram of object descriptions is depicted in figure 4.5.

An *object type* (class `ObjectType`) is used to specify a new type of object and it has an associated vocabulary to describe the features of objects of this object type which is the value of the slot `VocabularySlot`, this is obligatory. The descriptors of this vocabulary can be used as basis for descriptions of the facts that will be associated with objects of this object type. An object type also has a `Documentation` slot which again contains some extra information about the object, the slot is optional and by default it indicates that no documentation is available. Creating an object type is fairly simple and is carried out like this :

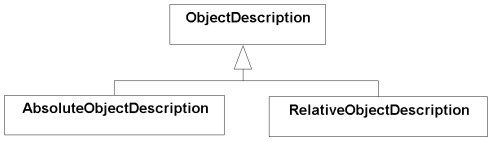

Fig. 4.5: Object descriptions

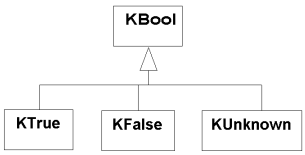

Fig. 4.6: KBool

```
ObjectType withName: #exampleObjectType
   andVocabularySlot: (VocabularySlot withFiller: exampleVocabulary)
   andDocumentation: (Documentation withFiller: 'I am an example object type.')
```

An *object* (class `Obj`) has an associated set of facts (called the fact-base), that are known about the object. Every object has a certain object type that should first be defined. This object type is then the link to a vocabulary that defines the descriptors that can be used for the descriptions of the facts that will be part of the object's fact-base. An object has a single slot `ObjectTypeSlot` that contains the object type of the object and can be created with

```
Object withName: #exampleObject
   andObjectTypeSlot: (ObjectTypeSlot withFiller: exampleObjectType)
```

exampleObjectType should already be defined.

Facts (class `Fact`) specify what is known about an object. They contain a description, a truth value and a justification. Often, a fact is created as result of the execution of a conclude action or ask action in a rule. A fact is local to an object and its permitted slots `TruthValue` and `Justification` are both optional. The `TruthValue` is a `KBool` with as default value `KTrue`. KBools will be explained shortly. `Justification` is a slot with information about how the fact was retrieved. By default the filler of this slot is 'I was told'. A fact can be created with

```
Fact withDescription: (BasicDescription withDescriptor: exampleAttribute
                                        andValue: #value2)
   andOwner: exampleObject
   andTruthValue: (TruthValue withFiller: KUnknown)
   andJustification: (Justification withFiller: 'I just made it up.')
```

Descriptions are used in facts to indicate what descriptor the fact knows about.

SqueakKAN uses *three-valued logic*. A boolean (a `KBool`) is either `KUnknown` (indicating the value is not known), a `KTrue` (the value is true) or `KFalse` (the value is false). The class diagram of KBools is depicted in figure 4.6. Table 4.2 and table 4.3 give the truth-tables of

value	NOT value	UNKNOWN value
KTrue	KFalse	KFalse
KFalse	KTrue	KFalse
KUnknown	KFalse	KTrue

Table 4.2: KBools : truth-table for NOT and UNKNOWN

v1	v2	v1 AND v2	v1 OR v2
KTrue	KTrue	KTrue	KTrue
KTrue	KFalse	KFalse	KTrue
KTrue	KUnknown	KFalse	KTrue
KFalse	KFalse	KFalse	KFalse
KFalse	KUnknown	KFalse	KFalse
KUnknown	KUnknown	KFalse	KFalse

Table 4.3: KBools : truth-table for AND and OR

these booleans for *not*, *unknown*, *and* and *or*.

Now the fact-base has been built, with vocabularies, descriptors (properties, attributes, roles and relations), descriptions, object descriptions, object types, objects and facts constructed as shown. The next step to build the rule-set.

rule set

A rule-set consists of a ruleSet and a number of rules, but also actions and conditions are introduced in this part. `RuleSet` is a global object and `Rule` is a local to a `RuleSet`. The class diagrams of these object are part of the diagram depicted in figure 4.2 and the slots that are part of these objects are shown in figure 4.7.

A *ruleSet* (class `RuleSet`) is a collection of rules with a slot `Goal`, `ObjectTypeSlot` and `Documentation`. `Documentation` is again a string giving some extra information about the ruleSet. `ObjectTypeSlot` contains the object type of the kind of objects the rules of the ruleSet will reason upon. `Goal` indicates the descriptor that is playing the role of goal. If this goal is reached, the problem solver will stop reasoning, however the slot `Goal` is optional, but if it is specified, thus it denotes a default goal for the problem solver. By default the filler is nil. A ruleSet can be created with

```
RuleSet withName: #exampleRuleSet
   andDocumentation: (Documentation withFiller: 'I am an example rule set.')
   andObjectTypeSlot: (ObjectTypeSlot withFiller: exampleObjectType)
   andGoal: (Goal withFiller: exampleProperty)
```

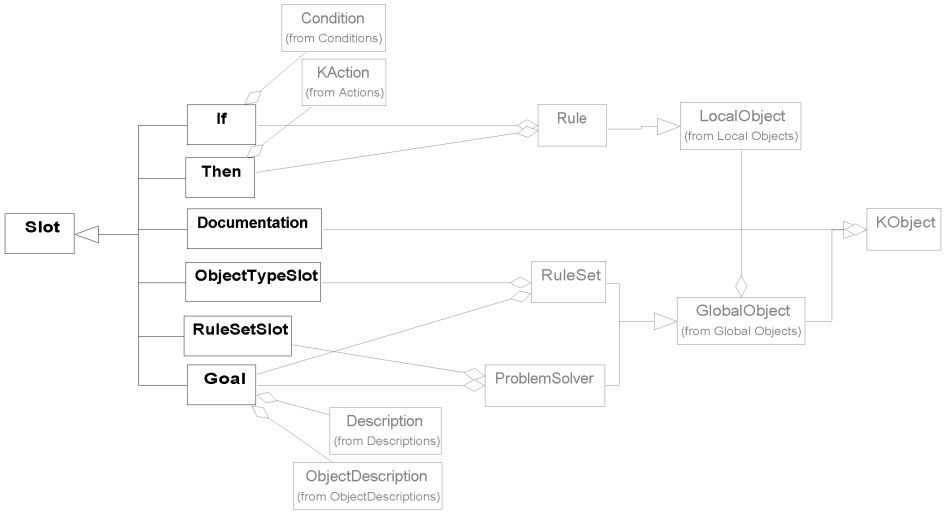

Fig. 4.7: Slots part 2 : used in objects of the rule-set.

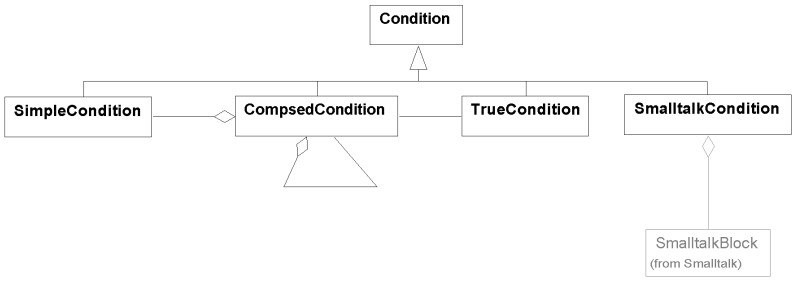

Fig. 4.8: Conditions

A *rule* (class `Rule`) is local to `RuleSet` and consist of an if-part (the conditions to fulfill) and a then-part (the actions to perform). During the solving process a rule is triggered which means that the conditions are checked and actions are possibly executed. (Conditions and actions will be explained shortly.) Permitted slots are `If`, `Then` and `Documentation` and a rule can be created with

```
Rule withName: #exampleRule
   andOwner: exampleRuleSet
   andIf: (ComposedCondition withConditions:
          (KCollection with: (SimpleCondition withDescription:
                              (BasicDescription withDescriptor:
                                      exampleProperty andValue: nil))
                      with: (SimpleCondition withDescription:
                              (NegatedDescription withDescription:
                                 (BasicDescription withDescriptor:
                                      exampleAttribute andValue: #value1))))
                      andOperator: #or)
   andThen: (KCollection new
          add: (Investigate withName: #exampleInvestigate
                             andRuleSet: exampleRuleSet
                             andGoal: (Goal withFiller: exampleProperty)
                             andObjectDesription: (RelativeObjectDescription
                             withPath: (KCollection with: exampleRole)));
          add: (Communicate withString: 'I am the communication string'.);
          add: (Conclude withDescription: (BasicDescription
                           withDescriptor: exampleAttribute andValue: #value2));
          add: (Ask withDescriptor: exampleProperty
                    andObjectDescription: (RelativeObjectDescription
                             withPath: (KCollection with: exampleRole));
          add: (Ask withDescriptor: exampleProperty
                    andObjectDescription: (AbsoluteObjectDescription
                    withPath: (KCollection with: exampleRole)
                                  ofObject: exampleObject)))
```

The `If` slot is optional and it contains a set of conditions, and by default this slot is empty, which means the rule will fire as soon as it is triggered. The `Then` slot contains the actions which is also optional and empty by default, which means that if the rule is fired, there are no consequences.

The class diagram of *conditions* (class `Condition`) are shown in figure 4.8. These conditions are used in the if-part of a rule and will evaluate to true or false. The three important kinds of conditions are simple condition, composed condition and Smalltalk condition. A *simple condition* (class `SimpleCondition`) corresponds to a simple description and it evaluates to true, false or unknown. A *composed condition* (class `ComposedCondition`) links simple or composite conditions with the boolean operators *and* and *or*. The sub-conditions are evaluated and combined using these boolean operators. A `TrueCondition` is an empty composed condition that will always evaluate to true. A *Smalltalk condition* (class `SmalltalkCondition`) allows to specify a block of Smalltalk code that should be evaluated in order to come to a

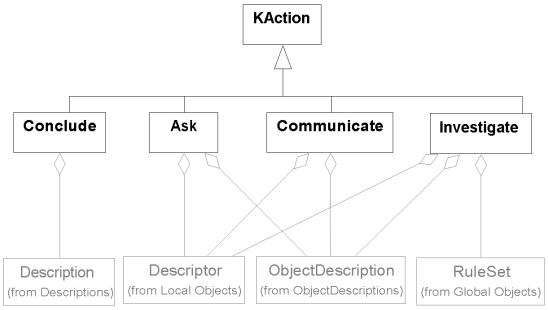

Fig. 4.9: Actions

boolean value.

Actions (class `Action`) occur in the then-part of a rule. Figure 4.9 illustrates the class diagram which contains the four possible types of actions are ask, communicate, investigate and conclude.
The action *ask* (class `Ask`) will ask the user about a certain descriptor which can lead to new facts being added to the system.
Communicate (class `Communicate`) shows some information to the user.
Conclude (class `Conclude`) is used to add a fact to the current object.
The action *investigate* (class `Investigate`) is used to start up an inferior problem solving process. For this sub-problem solver the same elements as for a problemSolver are needed, so this action takes as arguments a ruleSet, and optionally a goal and object description. The ruleSet is obligatory and refers to an existing ruleSet, whereas the goal is possibly optional and it is a descriptor or a combination of descriptor and object-description. When ruleSet has a filler for its `Goal` slot, then that filler is the default value when no goal is specified. If there is no filler for the `Goal` slot of the ruleSet, then it is obligatory to specify the `Goal` for this action. `ObjectDescription` (explained in the part on objects of the fact-base) is optional if the goal is specified. Its purpose is to indicate what object will serve as context for the new ruleSet. By default it denotes the current context.

Now that the rule-set has been created the two first components of a problem solver are complete. The next step is the inferencing engine, but as in KAN we first need an object problemSolver to group a ruleSet, context and goal.

problem-solver

The *problemSolver* (class `ProblemSolver`) is the final object that should be created before the engine can start the solving process. A problem solver contains the top level object, a rule set and the goal to be reached. These three necessary elements are contained in slots `RuleSetSlot`, `ObjectSlot` and `Goal`. The `RuleSetSlot` is obligatory and contains a ruleSet that was defined earlier. The `ObjectSlot` is also obligatory because it refers to the top-level object on which the problem solving will start. The `Goal` slot is optional when it was specified in the ruleSet of the problemSolver and if the slot is not specified, by default the goal of the ruleSet will be used. A problemSolver can for example be created like this :

```
ProblemSolver withName: #exampleProblemSolver
   andDocumentation: (Documentation withFiller: 'I am an example Problem Solver.')
   andRuleSetSlot: (RuleSetSlot withFiller: exampleRuleSet)
   andObjectSlot: (ObjectSlot withFiller: exampleObject)
   andGoal: (Goal withFiller: exampleProperty)
```

4.2.3 Inference Engine

Now that the basic structures are created and the problemSolver is defined, the *Inference Engine* is required to dictate the whole problem solving process. Starting from a given problem solver the engine will try to reach the goal and the process stops when the goal is reached, or when no new facts are derived during the previous reasoning step. The problem solving process (or engine) is started with `Engine solve: exampleProblemSolver`.

Engine : starting the problem solving process

A schematic representation of the way an Engine runs the problem solving process is given in figure 4.10.
The engine iterates through the rules of the ruleSet and tries to reach the goal. If this goal is not reached, but new facts were derived, then the iteration starts over again and new facts will be added to the fact-base of the current context. If no new facts were derived, then the goal cannot be reached, and in this case the user is informed and the problem solving process stops. However, if the goal is reached, then the process stops as well.
When iterating over rules, the rules are interpreted. This means that the if-part of a rule is checked by interpreting the conditions and if these evaluate to true, then the actions are executed. The evaluations of conditions and actions are explained in the next sections.

Interpreting conditions

A **condition** is evaluated by sending it the message `evaluateCondition:`*`aContext`*`inEngine:`*`anEngine`*. Figure 4.11 illustrates how conditions are evaluated. There are three types of conditions (Smalltalk, composite and simple conditions), and each condition has its own way of handling its problem solving.
A *Smalltalk condition* simply evaluates the Smalltalk code which should evaluate to a boolean.
A *composite condition* evaluates all it sub-conditions and combines them depending on the boolean operator and on the three-valued-logic explained in tables 4.2 and 4.3. The condition's operator can be either *and* or *or*. In the first case, the composite condition is true if all sub-conditions are true and none of them was unknown. If the boolean operator is *or*, the composite condition is true if at least one of the sub-conditions is true.
Evaluating a *simple condition* results in evaluating the description that is part of the condition. The result of that evaluation is a KBool and the value of this KBool is the result of the simple condition.
The next paragraph explains the problem solving of descriptions.

There are three kinds of **descriptions** (send description, composite description and simple description) and three kinds of simple descriptions (basic description, negated description and unknown description). An explanation of the *send description* will be given in section 4.3 on reflective capabilities of SqueakKAN. The other five descriptions are each evaluated in

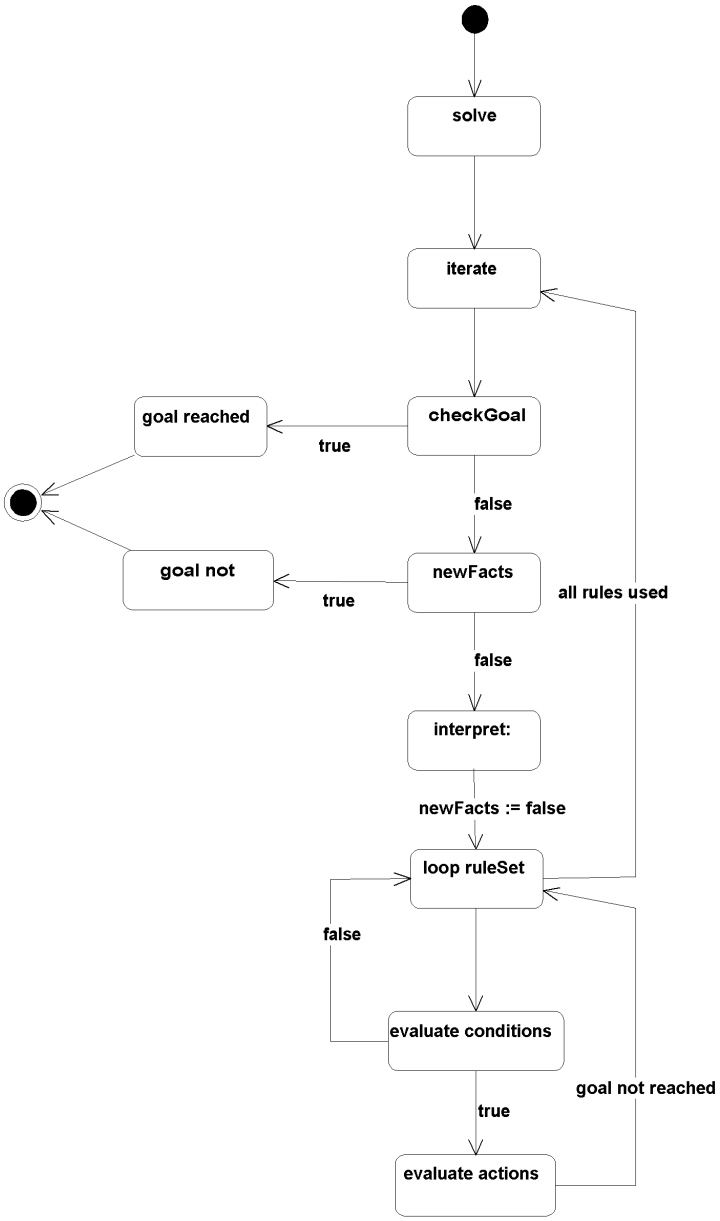

Fig. 4.10: Problem Solving Process in Engine

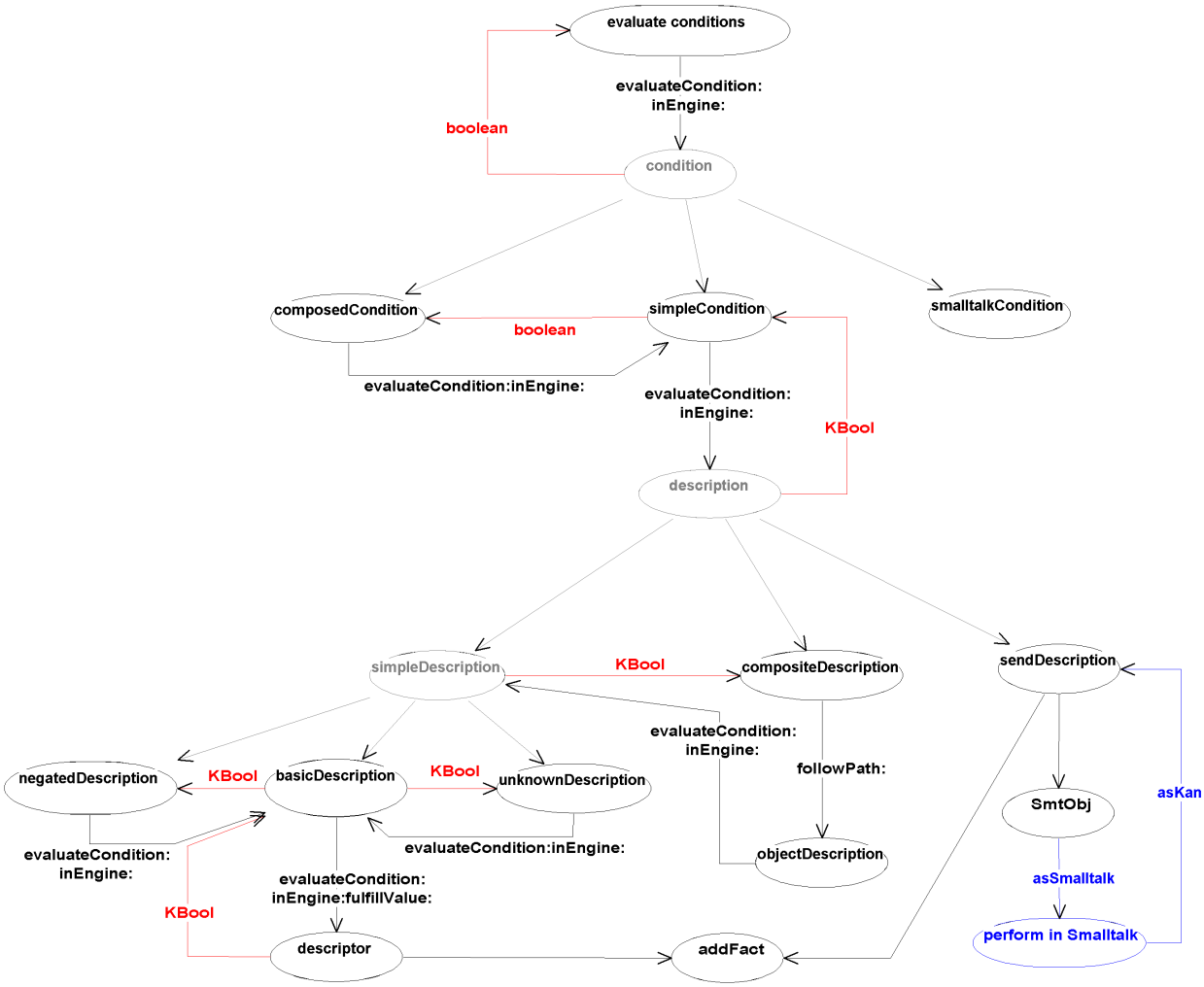

Fig. 4.11: Interpreting Conditions

their own particular way.
For simple descriptions, the evaluation of a *basic description* is carried out by sending the message `evaluateCondition:`*`aContext`* `inEngine:`*`anEngine`* `fulfillValue:`*`aDescriptor`* to the descriptor of the description. This evaluation returns a `KBool` and will be explained in the next paragaraph.
A *negated description* takes the negation of the evaluation of its basic description.
The *unknown description* also evaluates the basic description and transforms this result by sending the message `unknown` (see table 4.2).
And the *composite description* has a simple description and an object description. The path of the object description is followed and leads to another object. The simple description is then evaluated with this object as the context.

The evaluation of a **descriptor** results in a KBool. The message sent to a descriptor is `evaluateCondition:` *`aContext`* `inEngine:` *`anEngine`* `fulfillValue:` *`aCollection`*. `aContext` is the current context the solver is reasoning about and `anEngine` is the current engine that is being problem solved. This can be the engine reasoning on the main problem solver, but also an engine reasoning on a sub-problem solver that was instantiated because of an investigate action. `aCollection` contains the complements (values for the descriptor) of the basic description that sent this message. If the descriptor matches a fact in the fact-base of the context (i.e. a fact in the fact-base is based on the descriptor and its arguments are valid), and if the value belonging to this fact matches the complements of the description that sent this message, then the truth value of the fact is returned. If the complements did not match, then the result is `KFalse`. If the descriptor did not match a fact, then its value is searched for (depending on the value of the `ToFindOut` slot which can be `#ask`, `#lookup`,...) and the new fact is added to the fact-base of the context. Next, the value of the fact is again matched with the complements of the description.

Executing actions

If the conditions evaluate to true, then the actions are performed by sending the message `perform:inEngine`. This phase of the problem solving process is shown in figure 4.12 which illustrates the actions as ask, communicate, investigate and conclude.
The action *ask* asks the user for information about the descriptor and adds a fact to the fact-base of the context. The descriptor for this fact was provided at the creation of this action.
Communicate gives the user some information by using a simple pop-up window.
Investigate starts a sub-problem solver. The `path` of the object description of this action is followed to get to the context for this sub-problem solver. The investigate action also has a `RuleSetSlot` and a `Goal` and these three elements can make up a problem solver. A new engine is started to solve this sub-problem solver and the problem solving process stays the same. This sub-problem solver is quit when the sub-problem solving process exits.
Conclude will also result in adding a fact to the fact-base. The description to add this fact was supplied when the conclude action was created and is evaluated by sending it the message `evaluateConclude:inEngine:`. Each *basic description* (property, attribute, role, relation and smalltalkValue description) has its own evaluation for checking the possible values of the descriptor. *Negated*, *unknown* and *composite description* use this evaluation. *Send description* will be explained in section 4.3.

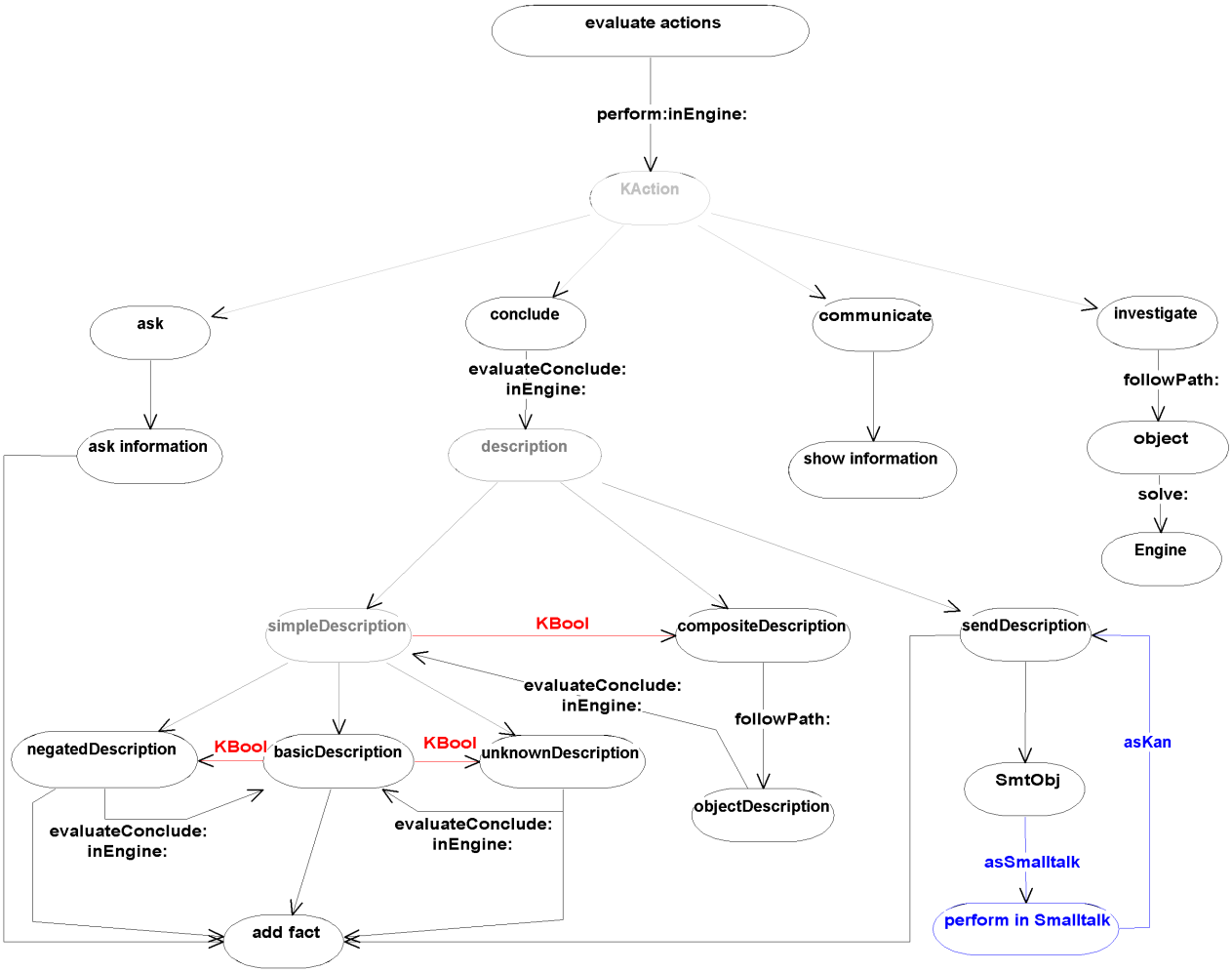

Fig. 4.12: Executing Actions

4.3 Reflective capabilities of SqueakKAN

SqueakKAN (as explained so far) is an implementation based on the KAN specification ([CSBG91]). At this moment it allows us to reason about knowledge provided in the format of facts and rules. Knowledge systems can be build in SqueakKAN in a similar wy to how they are built in the original KAN. However, the goal of this thesis was to reason about object-oriented systems.
Squeak is an environment for Smalltalk and Smalltalk is a pure object-oriented programming language in the sense that everything in Smalltalk is an object and it is completely built upon the principles of object-oriented programming. To reason upon object-oriented programming structures and Smalltalk, we should be able to represent "the underlying" Smalltalk in SqueakKAN using the basic entities of SqueakKAN. With this, we end up in the world of so called reflective systems, systems that reason about representations of their implementing language. In this section will explain how this representation is carried out. Firstly, we will repeat the idea of reflection, then we will show how this reflection is made available in SqueakKAN.

4.3.1 Reflection

One particular way to define (and construct) reflective system is by making use of a so-called linguistic symbiosis as introduced by Steyaert ([Ste94]). *Linguistic symbiosis* means that computations, specified in different formalisms, are mixed together in a transparent way. This allows us to specify a relationship between a high level language and its underlying implementation language, in a way that the programmer can profit from both worlds. It can therefore also specify the relation between the meta level language and the base level language of Logic Meta Programming.
Linguistic symbiosis between two systems enables introspection and absorption. *Introspection* means that a system can interrogate its implementation and thus this makes it possible to retrieve information and to look at the underlying language. From within the meta level we can access the base level, e.g. from within the declarative meta layer it is possible to retrieve information of classes, subclasses, methods, instance variables, ... of the object-oriented base level.
Absorption is used to indicate how the meta language can act upon the base language. The meta language can really change the underlying language, but by doing so, it can also change itself. For example, it is possible to adapt the object-oriented base level from within the declarative meta level by sending messages of which it is know that they change the base level. Thus, it also possible to send messages that change the classes implementing the meta level. When the declarative meta level sends messages that will affect the classes that implement this meta level, the declarative meta level is changing itself.
This means that meta level and base level are causally connected (changing one level affects the other). In this way, the system (the symbiosis) can incorporate and manipulate its own representation, which is causally connected with itself. This is Maes' definition of *reflection* ([Mae87]) and thus, a reflective system has both introspective and absorptive capabilities.

4.3.2 Reflection in SqueakKAN

We want to reason upon object-oriented systems, about the Smalltalk we are using, by reasoning on a meta level. To do so, we want to use forward chaining reasoning because, as validated in section 3.2.2, we need the persistency property of forward chainers : *" state can*

be preserved and retrieved between two problem solving sessions.". This persistency property is crucial since we want to interrupt the reasoning without losing previous results, therefore we use SqueakKAN.
However, besides a forward chainer to reason, we also need a way to introspect and absorb the underlying Smalltalk. Therefore, a knowledge base containing the complete Smalltalk image is necessary. However, in practice it is not possible to 'rewrite' the entire Smalltalk image conform with the representation of our meta level. Thus, the reification of Smalltalk in SqueakKAN is done virtually by providing a wrapping mechanism, which wraps Smalltalk entities when needed and provides an appropriate interface to cover up the differences with 'normal' SqueakKAN objects.

4.3.3 Providing a mechanism for Reflection

To allow introspection and absorption, and thereby allow reflection, it was necessary to add some structures to SqueakKAN and make some minor changes to Smalltalk as well. As already said, we used the linguistic symbiosis approach to construct reflection. The essence of linguistic symbiosis is that every Smalltalk object can be converted in an equivalent SqueakKAN object and vice versa. Therefore, each object (SqueakKAN or Smalltalk) has to understand the messages `asSmalltalk` and `asKan` so that there can be easily switched from meta level to base level and vice versa. To get a better understanding of how this works, an overview is given in table 4.4. In what follows, we further explain ourselves.

Additional components for SqueakKAN

Before explaining the newly added components to SqueakKAN, the concepts that these components represent ware presented.

Newly added concepts to SqueakKAN The additional features added to SqueakKAN have to provide a wrapping mechanism with the appropriate interfaces so that the SqueakKAN interpreter does not notice any differences between wrapped Smalltalk objects and 'normal' SqueakKAN objects. The new concepts are represented by adding :

- a descriptor `SmalltalkValue` (and a slot `ToFindOutValue`)
- a description `SendDescription`

Note that these components were already depicted in figures 4.2 through 4.12 and table 4.1.

The descriptor `SmalltalkValue` is used to retrieve Smalltalk classes from within SqueakKAN, which means that a Smalltalk class will be wrapped into an object with an appropriate SqueakKAN interface. This retrieval is necessary since we want to represent Smalltalk structures (objects) in SqueakKAN, however, this representation should be lazy, such that only the objects that are really needed, are reified.
Retrieving a Smalltalk class is done as follows : when creating a `SmalltalkValue` descriptor, the name of a Smalltalk class is specified, and when the value of this descriptor is asked for during reasoning, this class is looked up in Smalltalk and wrapped into a SqueakKAN object with an appropriate interface, namely with an interface that is similar to the one of `Obj`

(SqueakKAN Object).

The description `SendDescription` is used as a mechanism to send messages to Smalltalk classes from within SqueakKAN. SmalltalkValue allows to reify Smalltalk objects, but obviously we also need a mechanism to access these objects. In Smalltalk this accessing comes down to sending messages.
`SendDescription` takes as arguments a receiver, a selector and a collection of arguments. The receiver and all arguments are wrapped Smalltalk objects.
When problem solving this description, the receiver and all arguments are unwrapped, and the message (selector) is sent. The result of this Smalltalk action (sending the message to the receiver with the arguments) is again wrapped into a SqueakKAN object.

Thus, these two concepts allow to wrap Smalltalk objects and to send messages to objects. `SmalltalkValue` allows to reify the necessary Smalltalk objects and `SendDescription` provides a means to access them. Therefore, it is possible to introspect and absorb the Smalltalk level from within the SqueakKAN level.

Implementation of the new concepts The previous paragraph explained the concepts that were added to SqueakKAN in order to obtain a wrapping system such that Smalltalk can be represented and accessed from within SqueakKAN. This paragraph explains how these concepts are implemented. The components that were added to SqueakKAN are

- an object `SmtObj`
- a descriptor `SmalltalkValue` (and a slot `ToFindOutValue`)
- a description `SmalltalkValueDescription`
- a description `SendDescription`

A `SmtObj` is a wrapper around a Smalltalk object. It is created with the message `withFiller:` and this filler should be a Smalltalk object. This object can be retrieved by unwrapping the `SmtObj` with the message `asSmalltalk`.

The descriptor `SmalltalkValue` is, like other descriptors, local to a vocabulary and has a slot `ToFindOutValue`. This slot has as filler `#lookup` or `#fromSmalltalk` where `#lookup` is the default value and means that the value can be found by reasoning only. `#fromSmalltalk` indicates that the value should be retrieved from Smalltalk. When using this filler, a symbol will indicate what Smalltalk class is going to be retrieved.
`ToFindOutValue` can be created with the message `withFiller:` or with `withFiller: andName:`. The latter allows to specify the name (a symbol) of the Smalltalk class that will be retrieved from Smalltalk. If the filler of this slot is `#fromSmalltalk`, but the name was not specified, then the user will be asked to provide one.
When problem solving this descriptor (`evaluateCondition: inEngine:`), the class from Smalltalk will be retrieved and wrapped into a `SmtObj`.

`SmalltalkValueDescription` is a basic description based on the descriptor `SmalltalkValue`. Like other basic descriptions this descriptor associates the `SmalltalkValue` with its

Type of object	`asSmalltalk`	`asKan`
FCObject	SmalltalkKANObject	FCObject
KTrue	true	KTrue
KFalse	false	KFalse
KUnknown	false	KUnknown
Object	Object	SmtObj
true	true	KTrue
false	false	KFalse

Table 4.4: Results of `asSmalltalk` and `asKan`

value in the fact-base. A `SmalltalkValueDescription` is created with `withDescription:-andValue:` with as value a `SmtObj`, a wrapped Smalltalk class.
Problem solving this description is invoked by sending the message `evaluateCondition: inEngine:` and doing so, the descriptor `SmalltalkValue` will be problem solved.
`evaluateConclude:inEngine:` problem solves `SmalltalkValue` before a new fact, based on the description's descriptor and whose value is set to the description's value, is added.

`SendDescription` is a new kind of description which can be created with `withReceiver:` *`anObjectDescription`* `andSelector:` *`aSymbol`* `andArguments:` *`aCollection`*. The receiver and each argument of aCollection should be an object description with paths leading to a `SmtObj`.
When problem solving this description, the `SmtObj`'s are unwrapped and the message (selector) with the unwrapped arguments are sent to the unwrapped receiver. The result is again transformed to a valid SqueakKAN object (`SmtObj` or `KBool`).

Adaptations to SqueakKAN

In order to provide reflection, each SqueakKAN object should understand the message `asSmalltalk` and `asKan` (see table 4.4). This are the conversion methods that implement the linguistic symbiosis. They allow any Smalltalk object to be converted to a SqueakKAN object and vice versa. Therefore, we captured `FCObject` and adapted KBools.

Figure 4.1 illustrates that `FCObject` is the top object of all SqueakKAN structures. When sending `asKan` to this object, the object itself should be returned, and when sending `asSmalltalk`, a `SmalltalkKANObject` is created.
This `SmalltalkKANObject` is a wrapper surrounding a SqueakKAN entity. It is a Smalltalk object, but when it receives a message that is not understood, it will transform the selector and the arguments to SqueakKAN objects and send them through to its wrapped SqueakKAN object.
A `KBool` responds to the message `asSmalltalk` with its equivalent Smalltalk boolean.

Adaptations to Smalltalk

Smalltalk objects should also understand messages `asSmalltalk` and `asKan` (see table 4.4). Objects in Smalltalk will respond to the message `asSmalltalk` by returning themselves, and `asKAN` results in a `SmtObj` wrapping the original object. Smalltalk booleans, however, respond with their equivalent `KBool`. The reason for this "exception" is that the implementation of the problem solver needs special operations on SqueakKAN booleans (KBools), namely the three-valued logic operations. Therefore SqueakKAN booleans are not just wrapped Smalltalk booleans. Hence this exception.

Conclusion

Each Smalltalk object can be represented as a SqueakKAN object by wrapping it in a `SmtObj`. Conversely, SqueakKAN object can be represented in Smalltalk by wrapping it in a `SmalltalkKANObject`. Wrapping and unwrapping Smalltalk and SqueakKAN objects is carried out by the messages `asKan` and `asSmalltalk`. SqueakKAN also provides some additional objects so that it is possible to access Smalltalk from within SqueakKAN. Since everything in Smalltalk happens by sending messages (accessing, but also changing classes, methods, ...), then it is possible to introspect and absorb Smalltalk from within SqueakKAN. Hence, the system is reflective. The introspection can be used to reason about the underlying Smalltalk. Currently, the absorption was not really used, but in the future this can be used for code generation.

4.4 SqueakKAN : A Fictive Syntax

Although SqueakKAN is not provided with a parser, and thus the structures have to be constructed manually with the provided constructors, this section gives a fictive syntax. From now on, in the rest of this dissertation, examples of SqueakKAN structures will be noted in this syntax. This will increase the readability of the examples. (Note that this syntax, together with the grammar described in appendix B, can be used to write a parser for SqueakKAN.) Elements between square brackets are optional.

```
Problem Solver
 (problemSolver name rule-set object [goal])

Rule-set
 (ruleSet name object-type [goal])

Rules
 (rule name owner
    [(if condition-1 ... condition-n)]
    [(then action-1 ... action-n))]

Conditions
 (cond description)

Actions
 (askAct descriptor [object-description])
```

```
 (commAct a-string)
 (conclAct description)
 (invAct rule-set [goal] [object-description])

Descriptions
    basic-description
        (propDes property-name)
        (attrDes attribute-name value)
        (relaDes relation-name value-1 ... value-n)
        (roleDes role-name value)
        (smvalDes smalltalkValue-name value)
    negated-description
        (negDes basic-description)
    unknown-description
        (unkDes basic-decription)
    composite-description
        (compDes basic-description object-description)
    send-description
        (sendDes wso-rec selector [wso-arg-1 .... wso-arg-n])

Descriptors
    (owner has been omitted)
 (prop name [to-find-out] [to-ask])
 (attr name [to-find-out] [to-ask] (possible-value-1 ... possible-value-n))
 (rela name [to-find-out] [to-ask] (object-type-1 ... object-type-2))
 (role name [to-find-out] [to-ask] object-type)
 (smval name nameSmalltalkClass)

Wrapped Smalltalk Objects (wso)
 receiver
  (rec nameSmalltalkClass)
 argument
  (arg nameSmalltalkClass)
 wrapper
  (smtObj nameSmalltalkClass)

Object Descriptions
 (absObjDes rol-1 ... rol-2 object)
 (absObjDes rol-1 ... rol-2 wso-smtObj)
 (relObjDes rol-1 ... rol-2)

Objects
 (obj  object-type)

Object-type
 (objType vocabulary)
```

```
Vocabulary
 (voc name)

Goal
 (goal descriptor)

Fact
 (fact descriptor bool)

Bools
 true
 false
 unknown
```

4.5 Summary

This chapter concerned Squeak and the forward chainer SqueakKAN (built in Squeak based on KAN). SqueakKAN was built to demonstrate how forward reasoning can be used to reason about object-oriented programming structures which means SqueakKAN should be able to reason about Smalltalk. In order to do so, both Squeak (Smalltalk) and SqueakKAN were be linked together by representing Smalltalk in SqueakKAN. This was carried our by a representational structure that was transformed into practical use by adding some extra components to SqueakKAN. In the next chapter we will validate the claims previously made. We will setup two experiments to perform the validation of our thesis.

Chapter 5

Forward Chaining in the context of Co-evolution : Validation of the Approach

Section 3.2 gave a comparison between forward and backward chaining and concluded that there are two important reasons why the forward chaining approach should be considered, namely forward chaining does not fail when a goal is absent and forward chaining allows to preserve state (i.e. persistency property).
In this chapter our approach is validated with two experiments. These experiments are conducted with the prototype forward chainer SqueakKAN described in chapter 4.

5.1 Introduction

As explained in section 2.1, the aim of co-evolution is to achieve synchronisation between design and implementation. At the moment research in this field, is based on Declarative Meta Programming which means that a declarative meta layer is put on top of an object-oriented base layer. This research, carried out so far ([Wuy00], [DV98b]), has already shown that using declarative meta programming in order to achieve co-evolution is meaningful.
All research so far using declarative meta programming, has been based on Prolog, using goal-driven reasoning as an inference technique. This allows the user to start queries once a clear goal has been formulated. At the end of the problem solving process, the user will be informed as to whether or not the goal has been achieved. Unfortunately, it is not possible to interrupt the process, preserve its state and resume the process at some other time, as when reasoning about the structure of object-oriented systems, this is typically something we do want to do because of the interactivity of modern programming environments. The persistency property (section 3.2.2) shows that this is possible however when using forward chaining.
To validate the claim that forward chaining is as meaningful as backward chaining, two experiments have been conducted and they show that using forward chaining has great potential when longing for co-evolution. In this chapter, our experiments are explained. Section 5.4 discusses how the experiments validate our work.

5.2 A causally connected expert-system supporting co-evolution

Section 3.2 states that one of the distinctive properties of forward chaining is its usefulness in cases of goal-less reasoning. To prove the statement that reasoning in these cases is indeed useful, we conducted an experiment as validation. This experiment encloses identification and selection of reusable components. This cannot be expressed in a goal, since if one can formulate the description of such a reusable component, one already knows what component is needed, and thus there is no need at all for reasoning. In practice, a developer knows that there is a possible reusable component, but he does not know where to find it.
We created an expert system to help identify and select a reusable component and the experiment concerns Collections in Smalltalk. Up till now collections are one of the most successful reusable components, but besides the trivial ones (e.g. OrderedCollection and Dictionary), they are insufficiently used because the developer does not know exactly which collections are available.

5.2.1 From Smalltalk to SqueakKAN : `Reason`

By adding the method `reason` to `Object`, each Smalltalk object will understand this message. By default this gives an error message stating that there is no reasoning possible. However, by simply overriding this method in subclasses that can reason, and providing a problem solver, problem solving will be made possible.
In this way a switch can be made from within Smalltalk to SqueakKAN. This is one side of the causal connection between the meta and the base layer. The knowledge level (i.e. design) is causally connected to the coding level. The programmer just has "to push the right button" (in this case use `Class reason`) to trigger the reasoning process. One can imagine that (perhaps in the future) this triggering happens even more automatically. The text editors could for example see that, each time the name of one particular collection is used, a problem solver pops up to reason about whether indeed the right collection was used (a sort of wizard).

In our experiment, the problem solving starts when the message `reason` is sent to the class `Collection` and thus a problem solver on this collection is started. This solver is a predefined SqueakKAN problem solver. First, the problem solver will decide on the type of collection that is most suitable for the user's desires. For example, the rule named `bagRule` shown in figure 5.3[1] defines that when the collection is not ordered and contains multiple occurrences of elements, a `Bag` should be used.

Perspectives

In the future, it should also be possible to give extra arguments to the message `reason`. These arguments would then automatically be transformed to SqueakKAN entities to be used in the problem solving process. This however, is beyond the scope of this dissertation and was not further explored.

[1] The syntax used in this example is the fictive syntax we presented in section 4.4

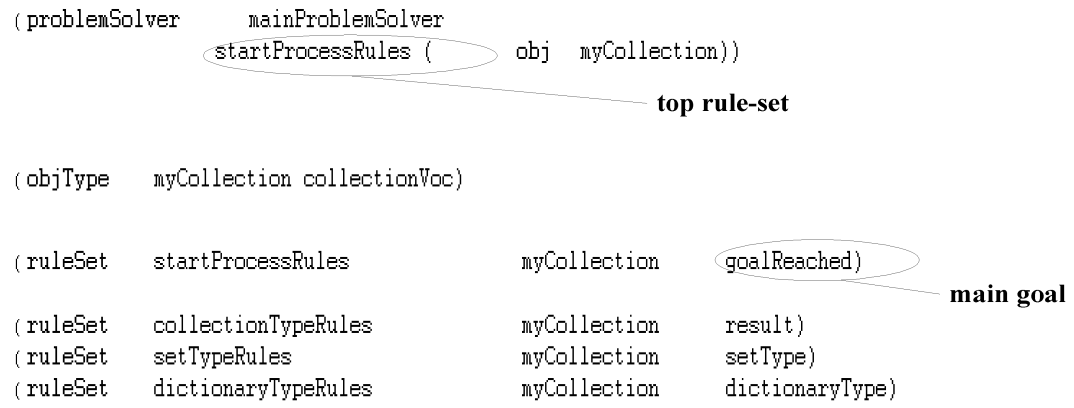

Fig. 5.1: Collection Problem Solver : Description of the ProblemSolver

5.2.2 From SqueakKAN to Smalltalk : `Send`

SqueakKAN provides a mechanism to access Smalltalk and to send messages to Smalltalk classes. This is carried out with the descriptor `SmalltalkValue` and the description `Send-Description` (see section 4.3.3).

Currently this mechanism is used in order to open a browser on the class of the decided collection type. This is carried out by concluding a `SendDescription` as is shown in figure 5.2. Furthermore, when this rule is triggered (i.e. the actions are executed), then the goal is reached and the problem solving process ends. Hence, this small experiment shows that the absorption mechanism can be used to invoke Smalltalk code from within the problem solver. Parts of the facts can thereby be transformed (as a parameter) to the Smalltalk level. This feature (although fully implemented in SqueakKAN) was not fully explored in the context of this dissertation. Further experimenting with the reflection is therefore on of our future work in chapter 6.

5.2.3 The Collection Problem Solver : an example

In this experiment we created a problem solver to assist in the identification and selection of a reusable component, namely of the component `Collection` in Smalltalk. The expert knowledge on collections has been written down in SqueakKAN in order to guide the programmer to the right collection type.
We will now give an idea of what this solver looks like by showing the rule-sets, descriptors and rules that are relevant to the reasoning process when reasoning towards a `Dictionary`. Note that one does not know in advance what the goal is, thus it is not known that a `Dictionary` will be the result. Nevertheless, we chose the example[1], as it is depicted in figures 5.1 to 5.6, relating to the reasoning process that would result in `Dictionary`.

Figure 5.1 shows the description of the problem solver. The main rule-set, which is used as the rule-set for the problem solver, is `startProcessRules`. `CollectionTypeRules`, `setTypeRules` and `dictionaryTypeRules` are rule-sets that will be used in `investigate` actions (i.e. actions that start up sub-problem solving processes).
Figure 5.6 lists the descriptors (attributes, properties, smalltalkValues) that are used in our

Fig. 5.2: Collection Problem Solver : Start Process Rules

example. By asking the user values for these properties or attributes (in case of a to-find-out slot with as value ask (see section 4.2)), new facts are added which will lead to firing certain rules.

The collection type `Dictionary` is an unordered collection with single occurrences (i.e. a set) and with associations between keys and values. To distinguish between a 'normal' dictionary and other types of dictionaries, the properties `weakly` and `extraProperties` should both be false or unknown in our example (since we are reasoning towards `Dictionary`).

The rule-set `startProcessRules` (figure 5.2) will be cycled trough first because this is the main rule-set.

The sub problem solving process `(invAct collectionTypeRules result)` will be launched in order to find the right collection type. Cycling through the rules of this set (figure 5.3) will result in using the rule-set `setTypeRules` (figure 5.4) because of the action `(invAct setTypeRules result)`.

On its turn, `setTypeRules` will again invoke another sub problem solving process in which the rule-set `dictionaryTypeRules` (figure 5.5) is used.

Finally, if all values of attributes and properties were as mentioned earlier, this will lead to concluding `(smvalDes result #Dictionary)`, i.e. the needed collection type is a `Dictionary`.

One by one, the sub problem solving processes will end, and the `goalReachedRule` in `startProcessRules` (figure 5.2) is triggered. If a collection type was found, the goal is reached and a browser is opened on the Smalltalk class of this type. Else, the goal is not reached and the problem solving process ends.

Constructing a problem solver like this is fairly simple, and moreover, results in an inter-

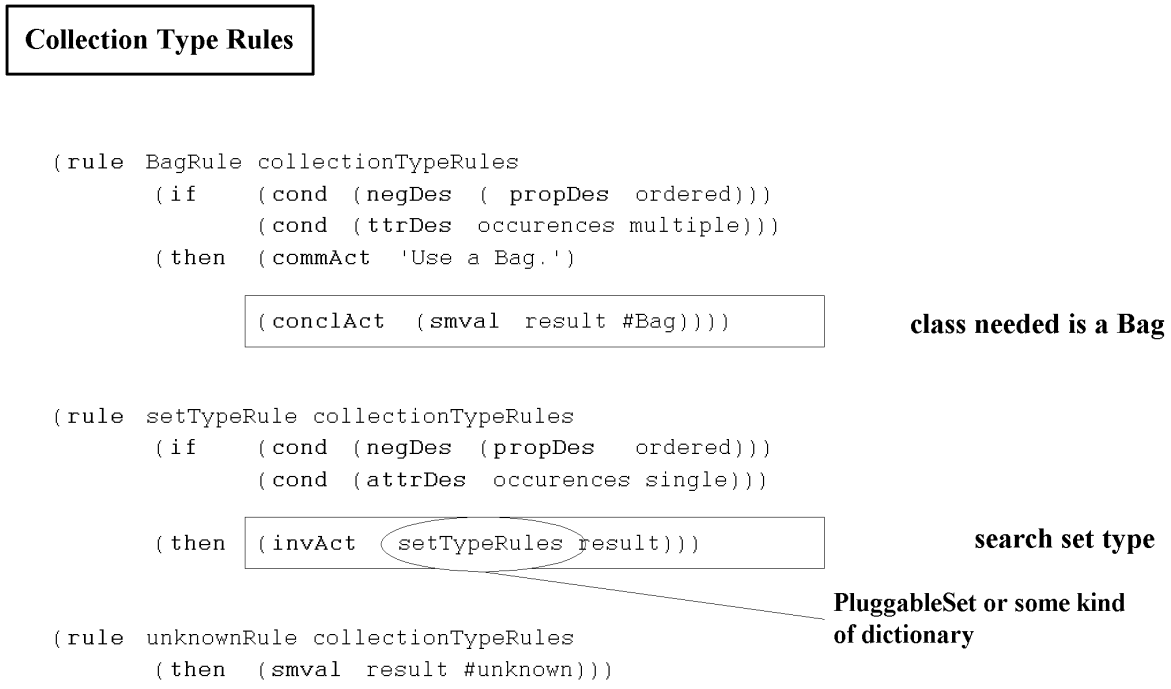

Fig. 5.3: Collection Problem Solver : Collection Type Rules

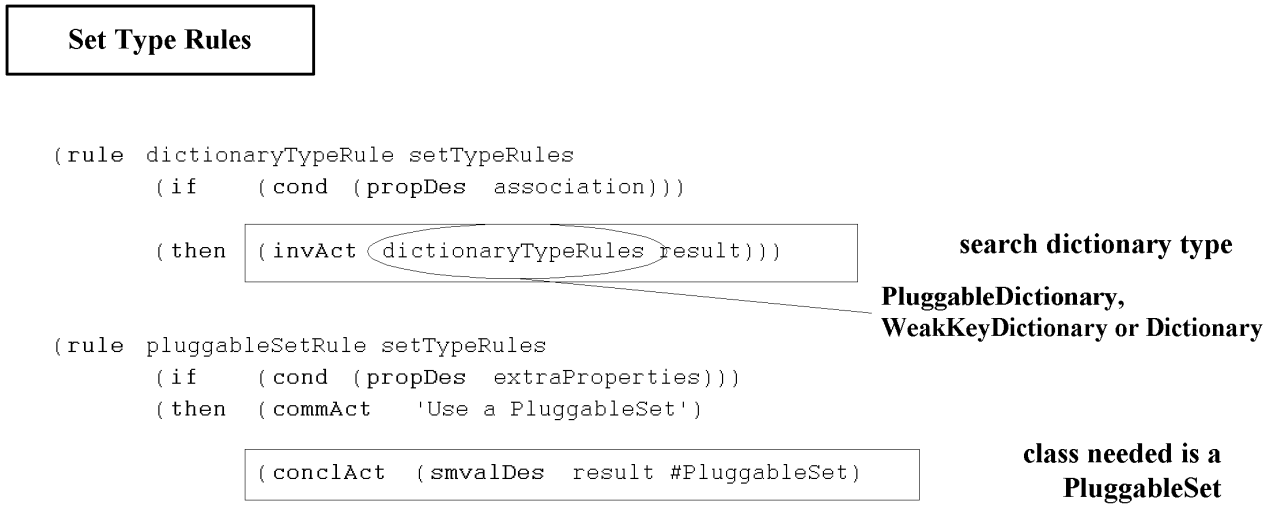

Fig. 5.4: Collection Problem Solver : Set Type Rules

Dictionary Type Rules

```
(rule pluggableDictionaryRule dictionaryTypeRules
      (if   (cond (propDes extraProperties)))
      (then (commAct  'Use a PluggableDictionary')
            (conclAct (smvalDes result #PluggableDictionary)))
```

class needed is a PluggableDictionary

```
(rule WeakKeyDictionaryRule dictionaryTypeRules
      (if   (cond (propDes weakly)))
      (then (commAct 'Use a WeakKeyDictionary')
            (conclAct (smvalDes result #WeakKeyDictionary)))
```

class needed is a WeakKeyDictionary

```
(rule DictionaryRule dictionaryTypeRules
      (then (commAct  'Use a Dictionary')
            (conclAct (smvalDes result #Dictionary))))
```

class needed is a Dictionary

Fig. 5.5: Collection Problem Solver : Dictionary Type Rules

```
(smtVal result lookup)
(prop goalReached lookup)

(attr  collectionType lookup   (#Array #Array2D #ByteArray
#FloatArray #IntegerArray #String #Symbol #Text #WordArray #RunArray
#Heap #Interval #LinkedList #MappedCollection #OrderedCollection #Bag
#Set #Dictionary #PluggableDictionary #WeakValueDictionary
#WeakKeyDictionary #PluggableSet #CharacterSet #WeakRegistry)).

(prop ordered)
(attr  occurences (single multiple)
(prop association  'Do you want to associate values with keys?').
(prop extraProperties 'Does the user known extra properties of the
                    objects that will be stored in the collection?')
(prop weakly 'Is the collection weak?')
```

Fig. 5.6: Collection Problem Solver : Descriptors

esting tool for, through asking the right questions, the programmer is helped in his decision on what component to reuse, i.e. what type of collection is needed.

5.3 A cookbook for using the LAN Framework

The first example in section 5.2 showed how a forward chained inference engine guides the programmer in taking certain decisions, especially when a goal is absent. Our second experiment goes a little further by allowing the problem solver to stop reasoning temporarily such that the programmer is again guided, but is also allowed to undertake certain actions in between. In this experiment we will prove the crucialness of the persistency property (see section 3.2.2) by reusing a framework. The problem solver will give different reuse possibilities and indicates what classes, subclasses, methods,.... have to be created. Due to the persistence of the problem solver these reuse possibilities can interactively and step by step, be performed by the user.

5.3.1 A need for active cookbooks

One of the ways framework reuse is supported, is by means of so-called cookbooks ([KP88, Joh92]). The idea of a cookbook is to provide standard recipes that explain step by step how the framework can be reused concretely. Documenting a framework with a cookbook with reuse-recipes, is very useful. However, a problem with these cookbooks arises when one does not know what recipe to use, or when one does not even realise that he is executing such a recipe. If the cookbook would trigger itself actively, these problems would vanish. As the programmer starts reusing, the active cookbook pops-up and will then guide the programmer through the right recipe. Moreover, it will allow the programmer to carry out the right actions along with the cookbooks' instructions. In the future, parts of the "reuse-code" could even be generated immediately, but this goes beyond the scope of the dissertation.
In the second experiment we will take a particular framework and write down the "reuse knowledge" as a KAN problem solver. The problem solver thus functions as an active cookbook. At certain times during the reuse process, the problem solver will stop and give control back to the programmer. When the programmer has executed some of the reuse steps of a particular recipe, the problem solver will continue its reasoning in order to continue the reuse recipe.

5.3.2 The LAN framework

The framework we will use is the so called LAN framework ([Luc97]). This framework simulates a simple and circular LAN network. The main elements of the network are node, packet, workstation and output-server and their class diagram is shown in figure 5.7. The framework was developed for educational purposes, and these main elements constitute the basis of the framework. However, this framework is extended and adapted in order to explain better programming and reuse skills.

Each *node* in a LAN is connected with one other node and its responsibility is to send and receive packets of information.
A *packet* is an object holding information that is sent from one node to another node. Its responsibility is to allow the user to define the originator of the packet, the address of the

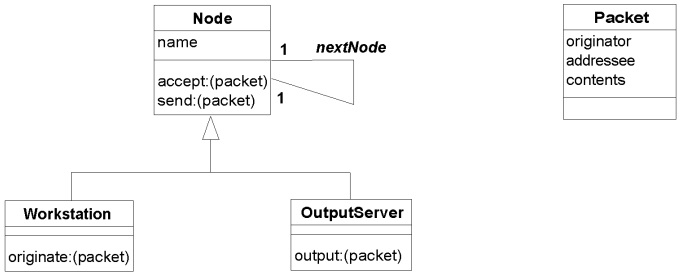

Fig. 5.7: Elements of the LAN Framework

receiver and the contents.
A *workstation* is the entry point for new packets onto the LAN network. It is a kind of network node but provides additional behaviour.
An *output-server* is another kind of node that allows to output the contents of the packet it received.
The LAN consists of a circular ring connection of nodes which can receive packets from one neighbour and send packets to its next neighbour. Workstations are special in a sense that they can put new packets onto the network, and they can perform some extra actions upon the received packet. An output-server will print the packet, store it or carry out some other output action.

In the example problem solver we built, a more specified LAN was used, namely a LAN framework that was adapted in order to obtain better reuse. This implies that some extra methods were added, code-duplication was removed, new classes handle new behaviour, etc. The classes and methods that are relevant for the experiment are shown in figure 5.8.

5.3.3 Switching between SqueakKAN and Smalltalk

With this experiment we will prove the usefulness of the persistency property of section 3.2.2. Thus, the problem solving process will temporarily be suspended and its state will preserved. However, to do so, an extra "mechanism" was needed (on top of the reflection mechanism) to improve the smoothness of communication between SqueakKAN and Smalltalk. This ad-hoc mechanism added to our Squeak environment, is used as a kind of bridge between Smalltalk and SqueakKAN. This will help to link Smalltalk and SqueakKAN when suspending the problem solving. The basic goal of the mechanism is to regulate the flow of control and information between Smalltalk (the programmer's side) and SqueakKAN (the designer's side).

The communication between SqueakKAN and Squeak is illustrated in figure 5.9. The add-hoc mechanism to improve this communication is implemented with an extra class `TaskPool`. The general idea of this pool is that it contains a number of "reuse problems" to be solved by the reuser. It is the job of the rules in the active cookbook to post new problems in the task pool (e.g. " a subclass of C has to be made") and it is the task of the Smalltalk programming environment to remove problems from the pool (e.g. "a subclass of C has just been made"). Together with each problem, the problem solver creates a flag to indicate whether it has to be restarted because the problem solver will stop everytime a conclusion was added to the

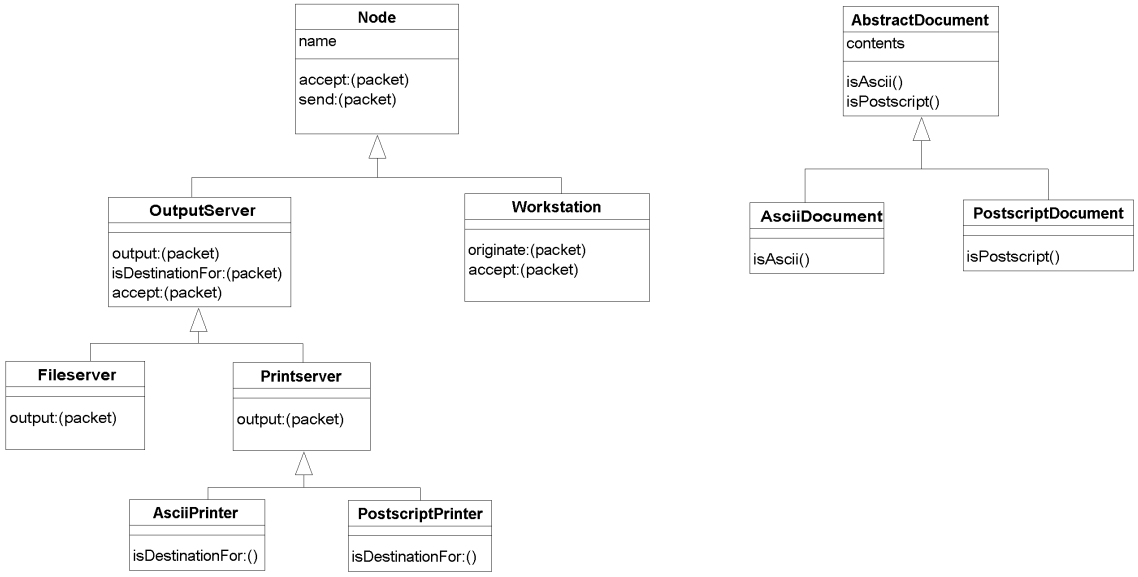

Fig. 5.8: The LAN Framework extended

task pool.

SqueakKAN can access the `TaskPool` by using the messages `classNeeded:andSolve:` and `methodNeeded:forClass:andSolve:`. These messages will add class- and method-names of classes and methods that should be added to the pool.
Sending these messages to the `TaskPool` is done with the mechanism provided by `SendDescriptions`. This mechanism (as explained in section 4.3.3) provides a means to send Smalltalk messages to Smalltalk objects from within SqueakKAN. The description unwraps its receiver and arguments, sends the messages and then the result is wrapped into a SqueakKAN object. For example, informing the TaskPool that a class C should be added, is expressed in SqueakKAN with `(sendDes (rec TaskPool) #classNeeded:andSolve: (arg C) (arg true))`. Unwrapping the receiver, results in the Smalltalk class TaskPool which will understand the selector #classNeeded:andSolve:.

The Smalltalk programming environment can access the pool with the messages `removeClass:` and `removeMethod:forClass:` which are invoked when accepting a class or method. In our experiment (in the Squeak 2.7 environment) we intervene in the Smalltalk programming environment by adapting the method `defineClass:notifying:` on the class `Browser` and the method `defineMessage:notifying:` on the class `Browser`. By adapting the first method, the accept of a new class is intercepted and the second method intercepts when accepting a new method. With these interceptions the `TaskPool` will be notified of an added class or method when new classes and methods are accepted. Notifying the TaskPool is done by sending the messages `removeClass:` and `removeMethod:forClass:` which will remove the class- or method-name from the pool, if they were present in the pool.

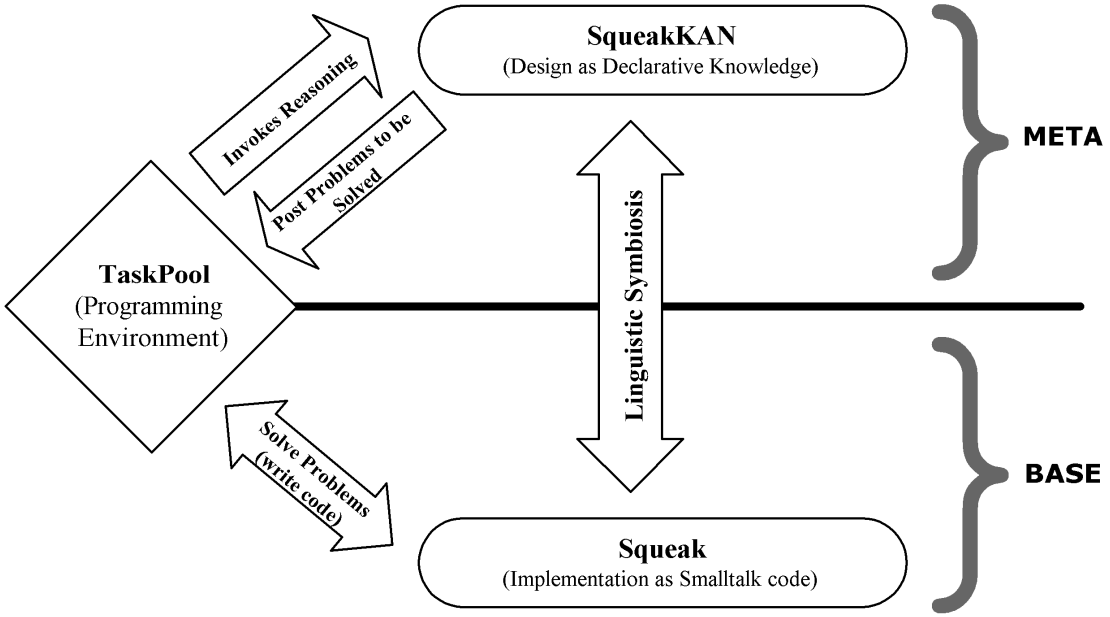

Fig. 5.9: Communication between SqueakKAN and Squeak

In the future other problems, beside adding a class or a method, should be solved as well. Therefore, the places of interception in the Smalltalk programming environment have to be enlarged. Table 5.1 shows the possible methods where this interception can be done (in Squeak version 2.7) for the most common actions a programmer would perform.

In this experiment it is the user's responsibility to indicate when the problem solver is suspended and resumed. Suspending a problem solver is a new kind of action which invokes a simple error handler that stores the problem solver's state and stops the process. The user can use these suspend-actions in SqueakKAN rules.
Resuming a problem solver can possibly be done after a removal from the `TaskPool`. This resuming also depends on the user's input since, when the messages `classNeeded:andSolve:` and `methodNeeded:forClass:andSolve:` were sent, a boolean was passed along (`andSolve:`) to indicate whether the problem solver needs to be resumed or not.
This user responsibility to suspend and resume the problem solver might be a limitation, but for the purpose of proving the persistence property, this mechanism is sufficient.

5.3.4 Reusing the LAN framework

The connection between SqueakKAN and Smalltalk was used to implement active cookbooks with reuse scenario's. If we think of the LAN framework as a LAN application family, there are different kinds of reuse we could think of; the existing family can be extended with new kinds of nodes, output-servers, packets, packet delivery systems (e.g. broadcasting), address-

Action	Class and Method of interception
add class adapt class remove class	Browser defineClass:notifying: Browser defineClass:notifying: Class removeFromSystem
add method adapt method remove method	Browser defineMessageFrom:notifying: Browser defineMessageFrom:notifying: Behaviour removeSelector:
add category remove category	ClassOrganizer addCategory:before: ClassOrganizer removeCategory:
add protocol remove protocol	ClassOrganizer addCategory:before: ClassOrganizer removeCategory:
add instance variable remove instance variable	Browser defineClass:notifying: Browser defineClass:notifying:
add class variable remove class variable	Browser defineClass:notifying: Browser defineClass:notifying:
add instance variable for class remove instance variable for class	Browser defineClass:notifying: Browser defineClass:notifying:
add pool remove pool	Browser defineClass:notifying: Browser defineClass:notifying:
add comment	ClassDescription comment:stamp:

Table 5.1: Interceptions in the Squeak 2.7

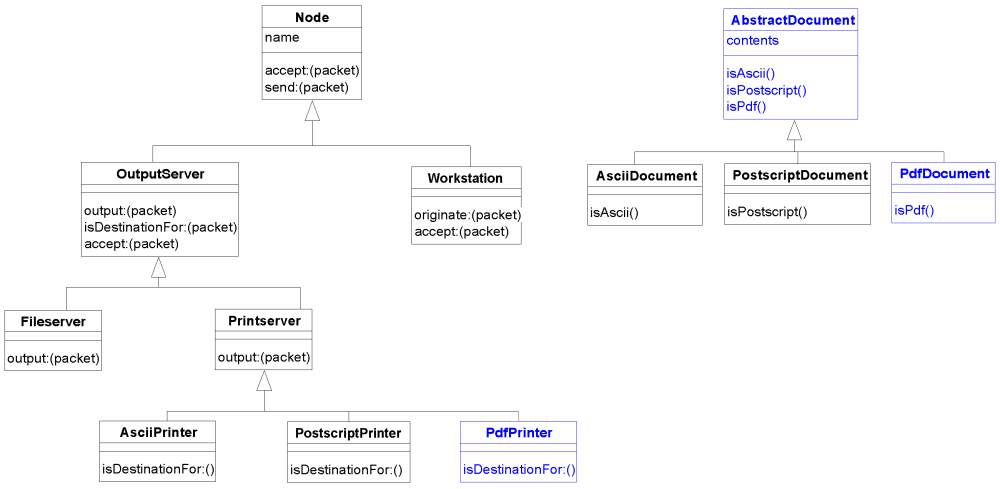

Fig. 5.10: The LAN Framework extended using a Problem Solver

ing schemes,... or a new kind of family could be created, for example a loggingLAN. However, in order to do these extensions right, the reuser has to follow explicit reuse scenarios that follow the design of the framework. The reuse scenarios are encoded in the active cookbook we implemented.

In our experiment we will help the user to create a new kind of printer, e.g. a `PdfPrinter` for the existing LAN application family. To create a new kind of printer, the user needs to create a new subclass of `Printserver` with a method `isDestinationFor:` and a new subclass of `AbstractDocument` with extra methods added to the new class and to the superclass.

To help the programmer to reuse the LAN application when he wants to construct a new kind of printer, i.e. `PdfPrinter`, the problem solver is continuously suspended and restarted. The solver tells the user gradually which class or method to create by concluding this in the task pool and the process stops to allow the user to perform these actions. Of course, at this point the user has full control. He can program whatever he wants, but as soon as these actions are done, the problem solver is restarted. Figure 5.11 shows how suspending and restarting the problem solver switches between Smalltalk and SqueakKAN. Figure 5.10 shows the class diagram of the newly added classes and methods after the process is executed completely.
The problem solver is resumed, but its previous state is preserved. For instance, once the `PdfPrinter` class is created, certain methods need to be added. The rule that indicates this, can only be triggered if that precondition is satisfied.

The rule in figure 5.12 is fired if the user was not told before what printer class he should add. If the rule is fired, `concludePrinterClassDone` adds a fact to the factbase indicating that the printer class was decided. Furthermore, a problem is posted to the TaskPool indicating that a class needs to be added and next, the problem solver is suspended.

Figure 5.13 is the rule that is triggered when the problem solver is resumed. Because the

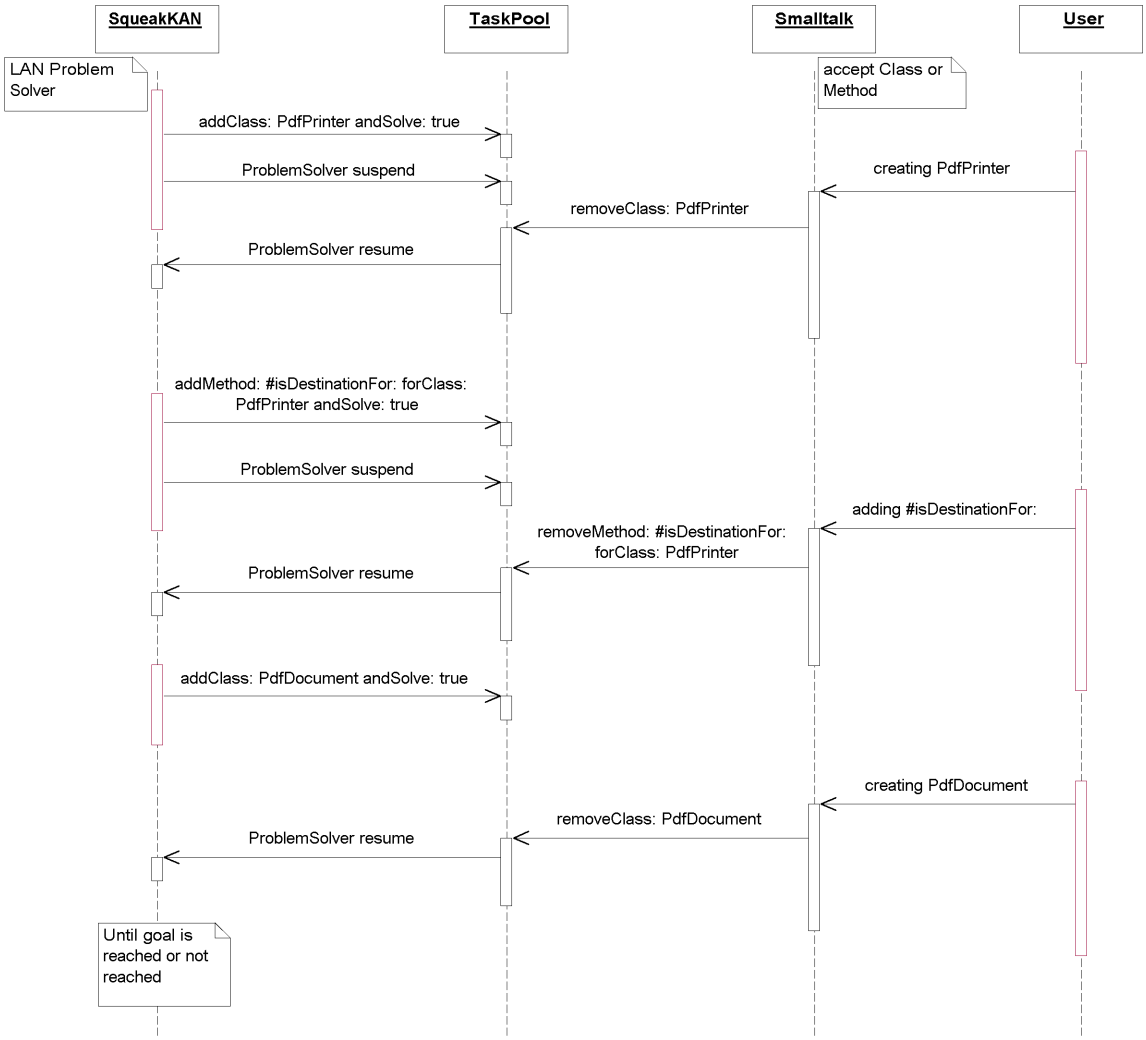

Fig. 5.11: Interaction between Suspending and Resuming the Problem Solver

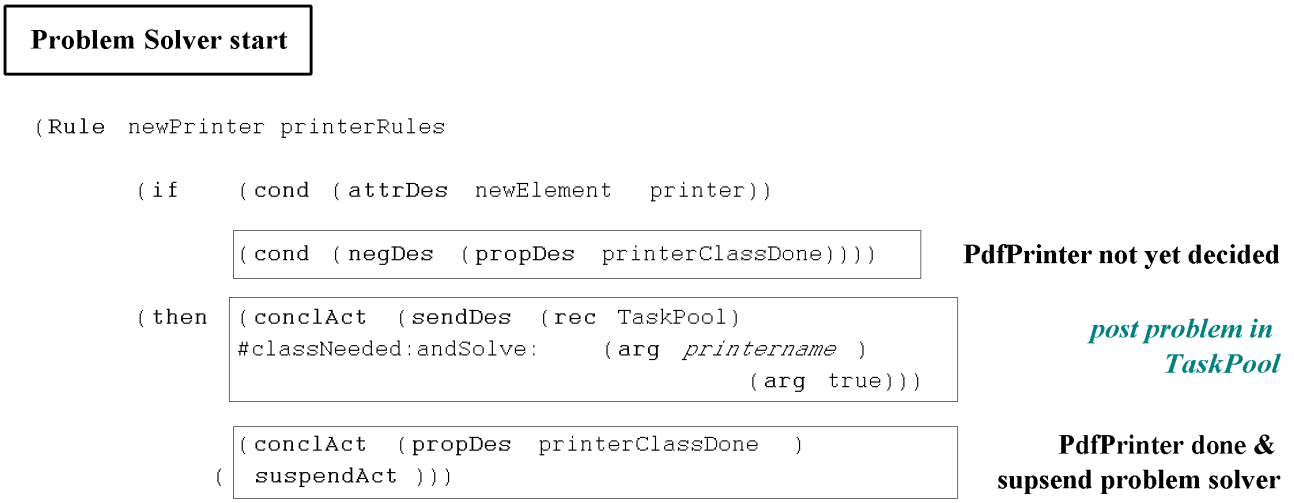

Fig. 5.12: LAN Problem Solver : Rule fired in first cycle

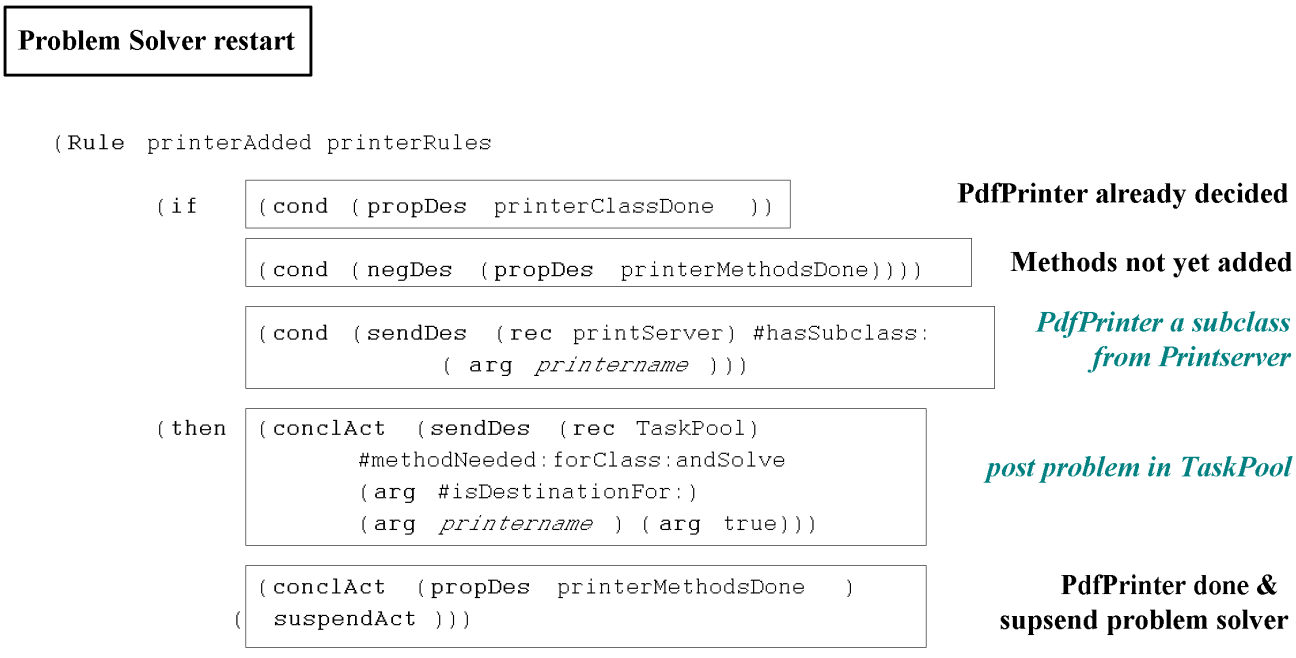

Fig. 5.13: LAN Problem Solver : Rule fired in second cycle

Fig. 5.14: LAN Problem Solver : Rule fired in third cycle

problem solver's state was preserved, the property `printerClassDone` is still known. Thus, the first rule will no longer fire but this second rule will, since one of the preconditions of this rule is `printerClassDone`. The TaskPool is notified that a method should be added. The rule triggered in the third cycle of the problem solving process is depicted in figure 5.14.

5.4 Validation

Section 3.2.2 made a comparison between forward chaining and backward chaining. Two important cases in which backward chaining fails, are when the goal is vague or absent and when state needs to be preserved. However, forward chaining performs excellent in these cases.

The first experiment (the collection experiment) showed that forward chaining can be used when a goal is absent. The problem solver wants to help in deciding what collection the programmer needs. Although it is known that some kind of collection is needed, obviously it is not known if this will be a dictionary, a set or a string. Thus, the goal is not known and certainly not expressible as a logic query for a backward chainer. As shown in the experiment, it *is* possible to reason and reach very attractive results with a forward chainer. We consider this as a proof-of-concept for one particular subproblem of co-evolution : the identification of reusable components.

The second experiment (the LAN framework experiment) shows the importance of the persistency property of forward chaining, i.e. the preservation of state. Thus, the problem solver is suspended, its state is preserved and the solver will be restarted some other time. Because of the stored state, the process will continue where it stopped earlier. This is crucial in establishing the causal connection between he knowledge level and the code level without giving the programmer the feeling that he is out of control.

5.5 Summary

In this dissertation it was the intention to show that forward chaining is equally beneficial as backward chaining when it is used for the purpose of co-evolution and for the purpose of reasoning about object-oriented systems. Backward chaining fails if there is no goal available or if state should be stored when the problem solver is temporarily suspended. Nevertheless, the experiments explained in this chapter show that forward chaining does not fail in these cases and thus forward chaining is indeed equally beneficial for co-evolution.

Chapter 6

Conclusion and Future Work

6.1 Summary

As object-oriented systems became bigger and more complex, maintaining them and keeping a global view on them became more difficult. Advanced tools like UML, frameworks, design patterns and contracts were developed in order to help the software engineer in maintaining the systems. These advanced tools are helpful for developing models reflecting the design of the systems. Since software evolves, due to maintenance, bug-fixes or reuse, these models are very helpful in keeping a global overview on a system, and they also provide decisions made through the developing. When a software engineer has good models at his disposal, he can eliminate the time-consuming exploration of the system's code. Without models it is very likely to have code (and behaviour) duplication, reinvention of designs and gaps of the software architecture; obviously, models are crucial.
Although these advanced tools provide great help in programming, the models they provide are not resilient to changes resulting in inconsistencies between design and implementation. In contrast to what is currently the case, the design should be connected to the implementation such that if either one changes this affects the other. Thus, design and implementation are synchronised and will remain synchronised.

Co-evolution provides a solution such that this synchronisation can be technically realised. This solution is based on a symbiosis between the object-oriented and declarative paradigm, also named declarative meta programming. The declarative paradigm will serve as a meta level on top of a object-oriented base level. Design issues will then be described on the meta level and reason upon the implementation in the base level and due to the causal connection between the two levels, changes on one level will have an impact on the other.
Co-evolution, or the symbiosis between object-oriented and declarative programming, is a causally connected system meaning that changes on one level do affect the other level. The system is also reflective, such that the logic meta programming allows to introspect (retrieve information of) the object-oriented system, but it also allows to absorb (make adaptations to) this system. The other way around, since the logic meta programming language (the meta level) will be implemented in the object-oriented base level, this base level is also able to adapt the meta level. This again reflects the causal connection between the two levels.

Within the object-oriented community, co-evolution is still a new and unique vision relating to reasoning about object-oriented systems. At the VUB Programming Technology Lab,

some prototypes (e.g. SOUL and TyRuBa) of co-evolutionary systems already show that this co-evolution indeed is beneficial. These systems were thoroughly described in chapter 2. Up till now these systems are Prolog-based implying they use a querying system to reason about the object-oriented base system. Querying systems are typically goal-driven (i.e. backward chained) reasoning.
Although extremely useful ([Wuy00], [DV98a]), systems using backward chained reasoning also have some drawbacks. For instance, the goal should be clearly specified because a vague goal cannot be used in backward chained reasoning. Also, it is difficult to preserve state during different phases of the problem solving process. If this process is temporarily interrupted during reasoning, its state will be lost. Thus, in these cases another approach might be more appropriate. In this dissertation we have shown that a data-driven (i.e. forward chained) reasoning approach does allow to preserve state and use vague goals. This was fully explained in section 3.2.

To validate the choice of forward chaining, a prototype was built. This prototype was written in Squeak and based on KAN, an educational tool for developing knowledge systems. In chapter 5 some experiments were carried out with this prototype. These experiments proved that forward chained reasoning can be used if no goal is present and when interaction between the problem solving and the programming environment is needed, thus when the solver needs to be suspended and resumed. These two cases are exactly the two properties (see section 3.2) that distinct a forward chainer from a backward chainer. After all, in these two cases backward chaining will fail. Our forward chaining approach has proven to be valuable and is definitely equally beneficial to the backward chaining approach.

6.2 Future work

It is clear that using a reflective forward chained inference engine to reason upon object-oriented systems has a lot of perspectives. Nevertheless, a lot of research still remains to be done, including a lot of additional experimenting with SqueakKAN. For example, in this dissertation we did not explore the power of SqueakKAN's absorption mechanism and this is certainly a point for continuation of our research.

Improvements and extensions to the prototype

Some improvements to be considered are a more powerful and user-friendly prototype.

- A *parser* would increase the user-friendliness because building a problem solver with the current available constructors is very cumbersome.
- The tool could be extended with *improved data-structures*. Some of the current SqueakKAN structures are deep and complex (e.g. objects with slots) and they could be adapted such that SqueakKAN system becomes simpler.
- The inference engine should be extended with a clever *resolution strategy*. The resolution strategy decides what rule is most the suitable to be fired. However, for the purpose of this dissertation, a simple strategy (first found, first triggered) was sufficient.

The built system is a reflective system making it possible to access Smalltalk from within SqueakKAN. It would be interesting to provide some *core rules* bridging SqueakKAN and Smalltalk. These core rules would create a link between the prototype and the base level. Then, other rules could use these core rules without knowing how to access the Smalltalk base level, and in this case they would serve as a sort of library for the Squeak programmer.

The experiments demonstrated that SqueakKAN can help in taking decisions during implementation and it can guide the software engineer through the creation of new applications (e.g. the LAN framework). These kind of *cookbooks* should be provided for other constructions as well. For instance, how to create a user-interface or how is error handling carried out, and what about a problem solver assuring that no programming conventions are violated. There are a lot of such applications one can think of and they make fully use of the problem solver's capabilities.

Future research

The work currently carried out using backward chaining (e.g. reasoning upon code ([Wuy00]) and code generation ([DV98b])), can also be executed with forward chained reasoning. For instance, *code generation* can also be performed with forward chained reasoning.
The SqueakKAN system is reflective and supports both introspection and absorption. Although we only focussed on introspection during the experiments, absorption is not that difficult to achieve. Especially not in Smalltalk because all computation in Smalltalk is done by sending messages between objects and the underlying Smalltalk layer can be adapted by sending the right messages (from the SqueakKAN meta layer using the SqueakKAN's messages sending mechanism). Code generation (creating new classes, adding methods, changing classes, adapting methods, instantiating new variables,.....) is achieved by sending the right message to the right object, e.g. sending subclass: to a class. Although this might sound fairly simple, the whole process of code generation needs to be given more attention such that possible pitfalls are avoided.

This dissertation showed that forward chaining is comparatively beneficial with backward chaining, but nevertheless, using a *combination* of both would be more realistic. The two approaches should complement each other such that one can choose the best approach for each particular situation.
Firstly, both backward chaining and forward chaining should be compared more thoroughly within the scope of declarative meta programming. A clear view of what approach to use in a particular situation is an important prerequisite. Secondly, a new framework combining the two approaches is to be developed and validated. Eventually, this future research should demonstrate the advantages and disadvantages of this methodology.

Co-evolution starts with declarative meta programming. To achieve this, goal-driven and data-driven rule-based systems were investigated. A combination of the two should be considered, but also other alternatives may be investigated. Other expert systems that are not rule-based may be interesting as well. Furthermore, other algorithms may be used besides goal-driven and data-driven reasoning, such as constraint-driven reasoning.

Although still a lot of research remains to be done, this dissertation, together with previous

research done on co-evolution, proved that there are a lot of perspectives for using co-evolution as a new approach on reasoning about object-oriented systems and it cannot be denied that co-evolution will gain importance. Therefore, research in this field definitely has to be continued.

Appendix A

KAN's grammar

```
vocabulary                      : descriptor*

descriptor                      :  property
                                |  attribute
                                |  role
                                |  relation

description                     : simple-description
                                | composite-description

simple-description              : basic-description
                                | negated-description
                                | unknown-description

basic-description               :  property
                                |  attribute value
                                |  role object
                                |  role object-description
                                |  relation object*
                                |  relation object-description*

negated-description             : no basic-description
                                | not  basic-description

unknown-description             : unknown basic-description

composite-description           : == object-description simple-description

object-description              : relative-object-description
                                | absolute-object-description

relative-object-description     : >> role*

absolute-object-description     : >> role* of object(-name)
```

```
object-type                 : vocabulary

object                      : object-type fact*

rule-set                    : object-type rule*
                            | object-type goal rule*

rule                        : if then

if                          : condition*

then                        : action*

condition                   : simple-condition
                            | composite-condition
                            | boolean-condition

simple-condition            : basic-condition
                            | negated-condition
                            | unknown-condition

composite-condition         : object-description simple-condition

basic-condition             : basic-description

negated-condition           : negated-description

unknown-condition           : unknown-description

boolean-condition           : conjunctive-condition
                            | disjunctive-condition

conjunctive-condition       : and condition*

disjunctive-condition       : or condition*

action                      : ask
                            | communicate
                            | conclude
                            | investigate

ask                         : descriptor
                            | descriptor object-description

communicate                 : string
                            | string descriptor
```

```
                                | string descriptor object-description

conclude                        : description

investigate                     : rule-set
                                | rule-set goal
                                | rule-set goal object-description

goal                            : descriptor

problem-solver                  : object-slot rule-set
                                | object-slot rule-set goal

object-slot                     : object
                                | object-description
```

Appendix B

SqueakKAN's grammar

The elements in the *slanted* typeface are entities that were added to SqueakKAN to support its reflectiveness. The entities in *italic* are a local object to the entity there part of in the grammar.

```
Vocabulary                  : Descriptor*

Descriptor                  : Property
                            | Attribute
                            | Role
                            | Relation
                            |  SmalltalkValue

Description                 : SimpleDescription
                            | ComposedDescription
                            |  SendDescription

SimpleDescription           : BasicDescription
                            | NegatedDescription
                            | UnknownDescription

BasicDescription            : PropertyDescription
                            | AttributeDescription
                            | RoleDescription
                            | RelationDescription
                            |  SmalltalkValueDescription

PropertyDescription         : Property

AttributeDescription        : Attribute value

RoleDescription             : Role Obj
                            | Role ObjectDescription
```

```
RelationDescription              : Relation Obj
                                 | Relation ObjectDescription

SmalltalkValueDescription        : SmalltalkValue SmtObj

NegatedDescription               : BasicDescription

UnknownDescription               : BasicDescription

ComposedDescription              : ObjectDescription SimpleDescription

SendDescription                  : ObjectDescription symbol ObjectDescription*

ObjectDescription                : RelativeObjectDescription
                                 | AbsoluteObjectDescription

RelativeObjectDescription        : path

AbsoluteObjectDescription        : path Obj

Obj                              : ObjectType Fact*

SmtObj                           : wrapValue

ObjectType                       : VocabularySlot

VocabularySlot                   : Vocabulary

RuleSet                          : ObjectTypeSlot Rule*
                                 | ObjectTypeSlot Goal Rule*

ObjectTypeSlot                   : ObjectType

Rule                             : If Then

If                               : Condition*

Then                             : Action*

Condition                        : SimpleCondition
                                 | CompositeCondition
                                 | SmalltalkCondition

SimpleCondition                  : Description

ComposedCondition                : Condition*
```

```
SmalltalkCondition          : smalltalkBlock

Action                      : Ask
                            | Communicate
                            | Conclude
                            | Investigate

Ask                         : Descriptor
                            | Descriptor ObjectDescription

Communicate                 : string

Conclude                    : Description

Investigate                 : RuleSet
                            | RuleSet Goal
                            | RuleSet Goal ObjectDescription

Goal                        : Descriptor

ProblemSolver               : ObjectSlot RuleSetSlot
                            | ObjectSlot RuleSetSlot Goal

ObjectSlot                  : Object

RuleSetSlot                 : RuleSet
```

Appendix C

Class diagrams of SqueakKAN

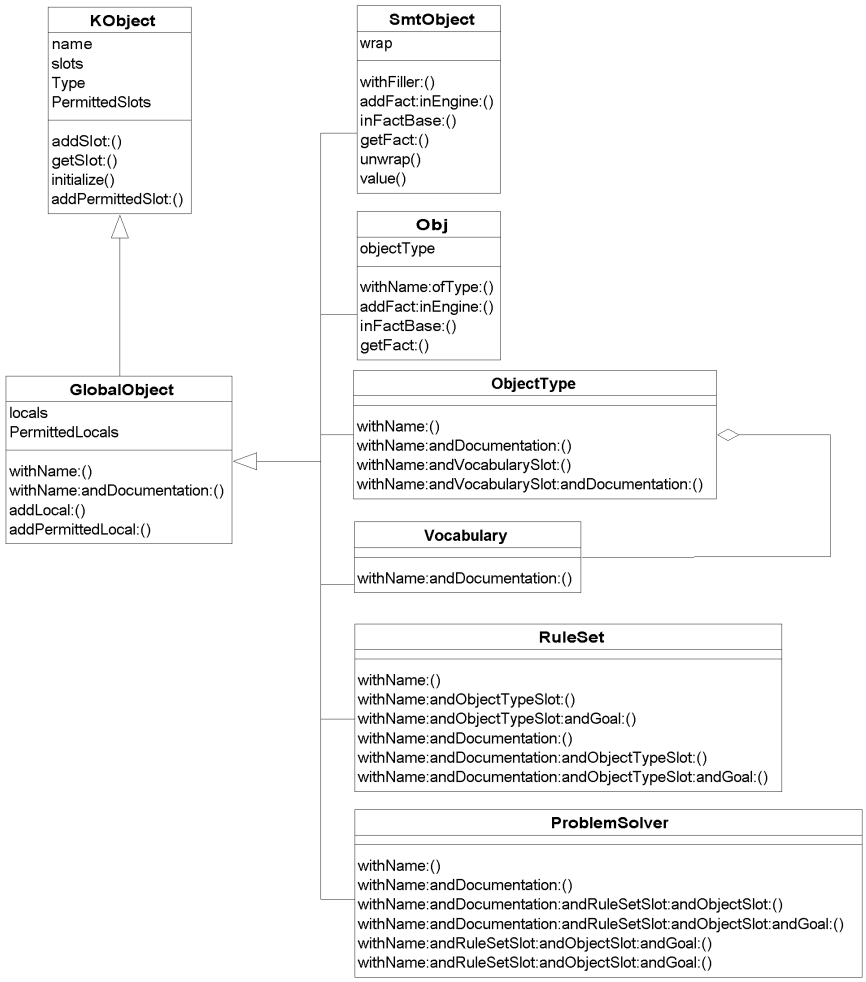

Diagram C.1: Global Objects

Diagram C.2: Local Objects

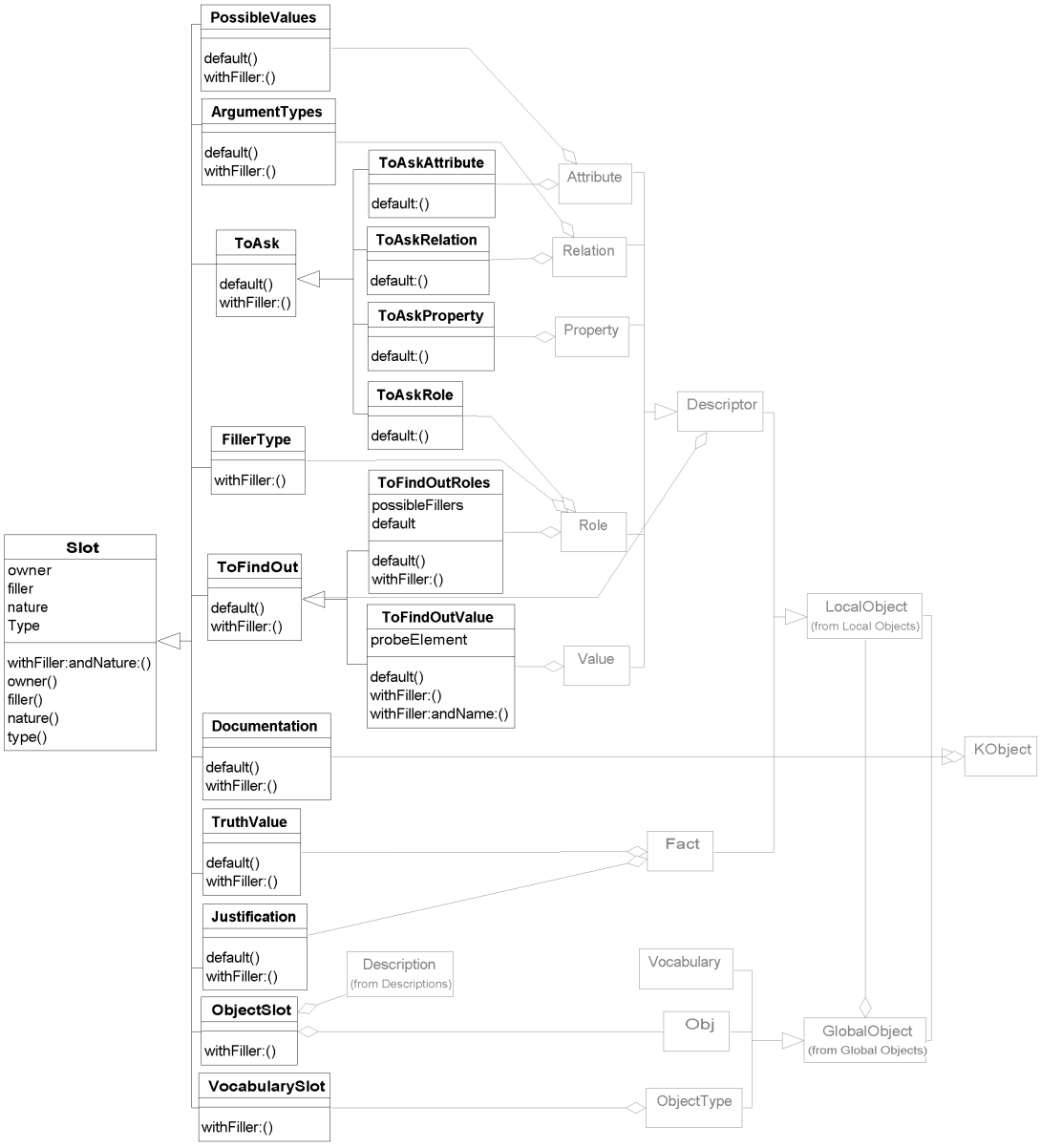

Diagram C.3: Slots part 1 : used in objects of the fact-base.

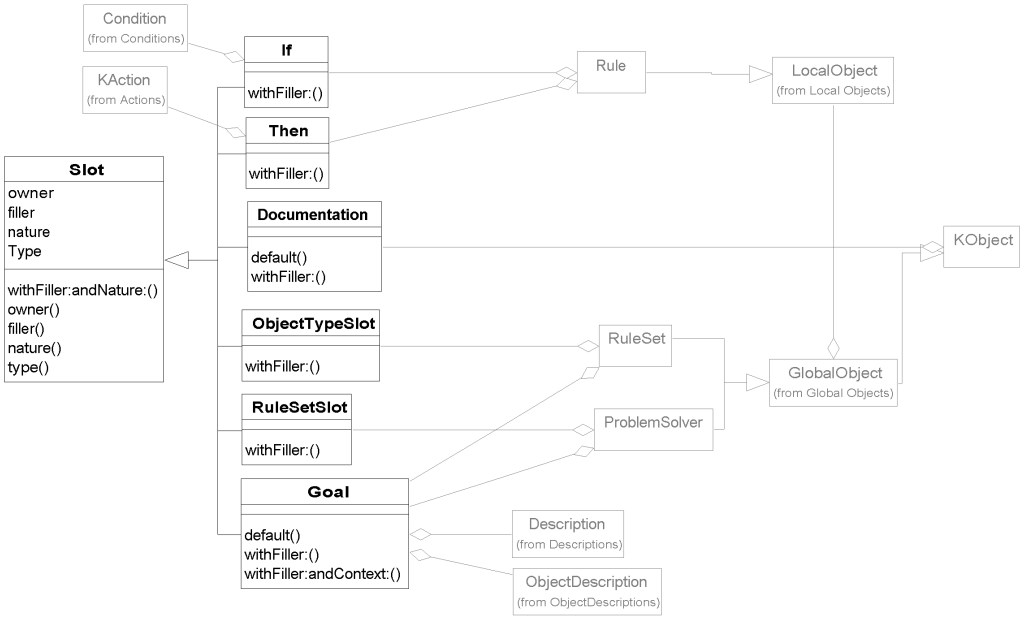

Diagram C.4: Slots part 2 : used in objects of the rule-set.

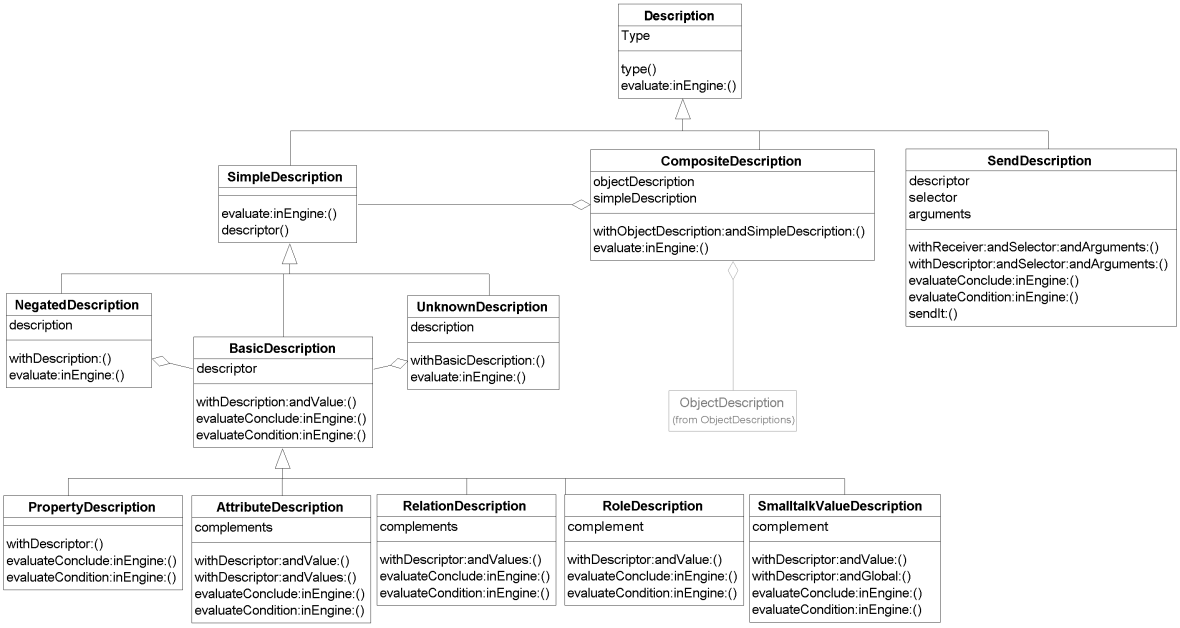

Diagram C.5: Descriptions

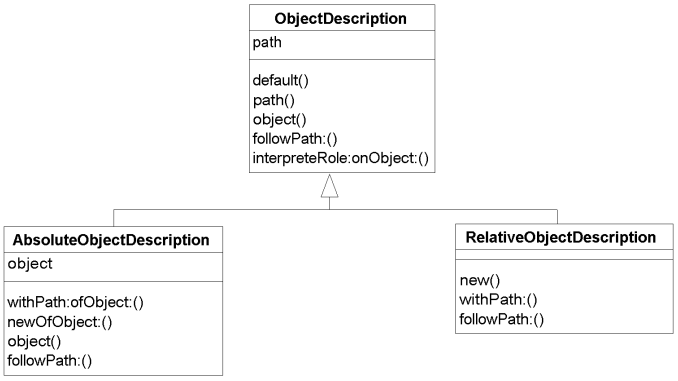

Diagram C.6: Object descriptions

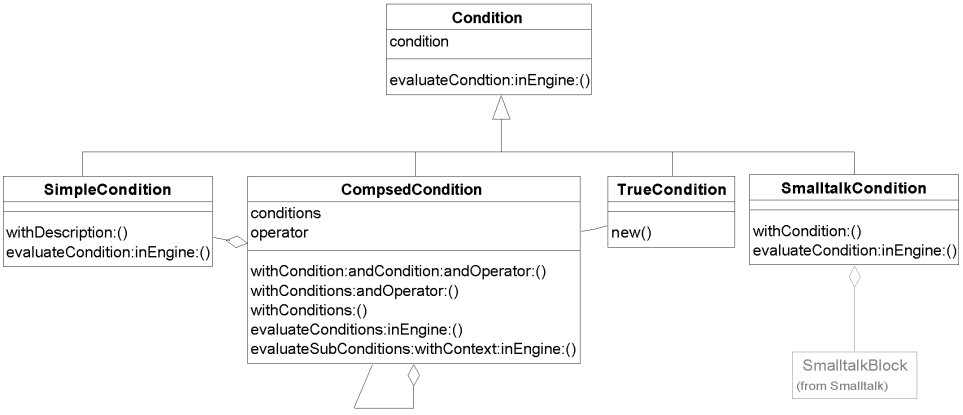

Diagram C.7: Conditions

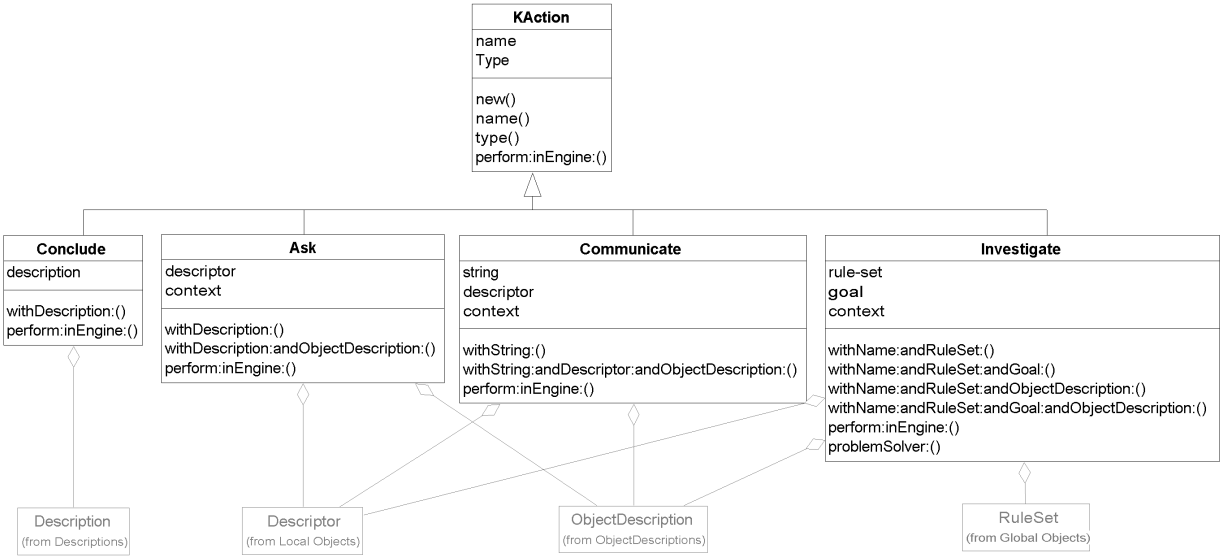

Diagram C.8: Actions

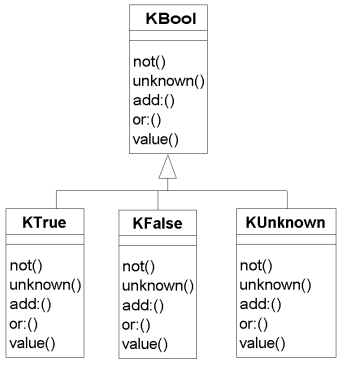

Diagram C.9: KBool

Appendix D

Example of a Problem Solver in SqueakKan

Defining Vocabularies

```
Vocabulary withName: #exampleVocabulary
   andDocumentation: (Documentation withFiller: 'I am an example vocabulary.')
```

Defining ObjectTypes

```
ObjectType withName: #exampleObjectType
   andVocabularySlot: (VocabularySlot withFiller: exampleVocabulary)
   andDocumentation: (Documentation withFiller: 'I am an example object type.')
```

Defining Descriptors

```
Property withName: #exampleProperty
   andOwner: exampleVocabulary
   andDocumentation: (Documentation withFiller: 'I am an example property.')
   andToFindOut:(ToFindOut withFiller: #lookup)
   andToAsk: (ToAsk withFiller: 'Am I true?')

Attribute withName: #exampleAttribute
   andOwner: exampleVocabulary
   andDocumentation:(Documentation withFiller: 'I am an example attribute.')
   andToFindOut:(ToFindOut withFiller: #ask)
   andToAsk: (ToAsk withFiller: 'What is my value?')
   andPossibleValues: (PossibleValues withFiller: (OrderedCollection with: #value1
                                                                     with: #value2))

Role withName: #exampleRole
```

```
    andOwner: exampleVocabulary
    andDocumentation:(Documentation withFiller: 'I am an example role.')
    andToFindOut:(ToFindOut withFiller: #lookup)
    andToAsk: (ToAsk withFiller: 'What is my value?')
    andFillerType: (FillerType withFiller: exampleObjectType)


 Relation withName: #exampleRelation
     andOwner: exampleVocabulary
     andDocumentation: (Documentation withFiller: 'I am an example relation.')
     andToFindOut:(ToFindOut withFiller: #lookup)
     andToAsk: (ToAsk withFiller: 'What is the value of my arguments?')
     andArgumentTypes: (ArgumentTypes withFiller:
                                (OrderedCollection with: exampleObjectType))
```

Defining Objects

```
 Object withName: #exampleObject
    andObjectTypeSlot: (ObjectTypeSlot withFiller: exampleObjectType)
```

Defining Facts

```
 Fact withDescription: (BasicDescription withDescriptor: exampleAttribute
                                          andValue: #value2)
    andOwner: exampleObject
    andTruthValue: (TruthValue withFiller: KUnknown)
    andJustification: (Justification withFiller: 'I just made it up.')

Fact withDescription: ( CompositeDescription withDescriptor:
                                    (BasicDescription withDescriptor:
                                             exampleAttribute andValue: #value1)
     andObjectDescription: (RelativeObjectDescription withPath:
                                           (KCollection with: exampleRole)
     andOwner: exampleObject
     andTruthValue: (TruthValue withFiller: KFalse)
     andJustification: (Justificatin withFiller: 'I was told.')
```

Defining RuleSets

```
 RuleSet withName: #exampleRuleSet
    andDocumentation: (Documentation withFiller: 'I am an example rule set.')
    andObjectTypeSlot: (ObjectTypeSlot withFiller: exampleObjectType)
    andGoal: (Goal withFiller: exampleProperty)
```

Defining Rules

```
Rule withName: #exampleRule
   andOwner: exampleRuleSet
   andIf: (ComposedCondition withConditions:
          (KCollection with: (SimpleCondition withDescription:
                              (BasicDescription withDescriptor:
                                      exampleProperty andValue: nil))
                      with: (SimpleCondition withDescription:
                              (NegatedDescription withDescription:
                                 (BasicDescription withDescriptor:
                                      exampleAttribute andValue: #value1))))
                      andOperator: #or)
   andThen: (KCollection new
               add: (Investigate withName: #exampleInvestigate
                                 andRuleSet: exampleRuleSet
                                 andGoal: (Goal withFiller: exampleProperty)
                                 andObjectDesription: (RelativeObjectDescription
                                    withPath: (KCollection with: exampleRole)));
               add: (Communicate withString: 'I am the communication string'.);
               add: (Conclude withDescription: (BasicDescription
                                 withDescriptor: exampleAttribute andValue: #value2));
               add: (Ask withDescriptor: exampleProperty
                         andObjectDescription: (RelativeObjectDescription
                               withPath: (KCollection with: exampleRole));
               add: (Ask withDescriptor: exampleProperty
                         andObjectDescription: (AbsoluteObjectDescription
                                   withPath: (KCollection with: exampleRole)
                                   ofObject: exampleObject)))
```

Defining ProblemSolvers

```
ProblemSolver withName: #exampleProblemSolver
   andDocumentation: (Documentation withFiller: 'I am an example Problem Solver.')
   andRuleSetSlot: (RuleSetSlot withFiller: exampleRuleSet)
   andObjectSlot: (ObjectSlot withFiller: exampleObject)
   andGoal: (Goal withFiller: exampleProperty)
```

Bibliography

[Boo94] Grady Booch, *Object-oriented analysis and design with applications*, 2nd ed., Benjamin Cummings, 1994.

[Bri99] Johan Brichau, *Syntactische abstracties voor logisch meta programmeren*, Bachelors thesis, Vrije Universiteit Brussel, 1999.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified method language 1.0*, Technical report, Rational, 1997.

[CDSV97] W. Codenie, K. D'Hondt, P. Steyaert, and A. Vercammen, *From custom applications to domain-specific frameworks*, ch. 40(10), pp. 71–77, Communications of the ACM, October 1997.

[CM85] Eugene Charmiak and Drew McDermott, *Introduction to artificial intelligence*, Addison-Wesley Publishing Company, 1985.

[CSBG91] Sven Van Caekenberghe, Luc Steels, Dany Bogemans, and Filip Gilbert, *The kan reference manual*, Reference manual, Artificial Intelligence Laboratory, VUB, Pleinlaan 2, 1050 Brussels, Belgium, 1991.

[Dav94] Alan M. Davis, *Fifteen principles of software engineering*, pp. 94–101, McGraw-Hill, November 1994.

[DD99] Maja D'Hondt and Theo D'Hondt, *Is domain knowledge an aspect?*, Proceedings of the ECOOP99 Aspect-Oriented Programming Workshop, 1999.

[DDMW99] Maja D'Hondt, Wolfgang De Meuter, and Roel Wuyts, *Using reflective programming to describe domain knowledge as an aspect*, Proceedings of GCSE '99, 1999.

[DDVMW00] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts, *Co-evolution of object-oriented software design and implementation*, TACT Symposium Proceedings, Kluwer Academic Publishers, 2000.

[DH98] Koen De Hondt, *A novel approach to architectural recovering in evolving object-oriented systems*, Phd hesis, Programming Technology Lab, Vrije Universiteit Brussel, December 1998.

[DV98a] Kris De Volder, *Type oriented logic meta programming*, Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1998.

[DV98b] Kris De Volder, *Type oriented logic meta programming for java*, Technical report, 1998.

[DVD99] Kris De Volder and Theo D'Hondt, *Aspect-oriented logic meta programming*, Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99, Springer-Verlag, 1999, pp. 250–272.

[Fla94] Peter Flach, *Simply logical*, John Wiley and sons, 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*, Addison-Wesley, 1994.

[Guz99] Mark Guzdial, *Squeak : Object-oriented design with multimedia applications*, In preparation, December 1999.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay, *Contracts: Specifying behavioural composition*, Object-Oriented Systems, Proceedings of the OOPSLA-ECOOP'90 Conference, ACM PRess, 1990, pp. 169–180.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay, *Back to the future: The story of squeak, a practical smalltalk written in itself*, OOPSLA'97 Conference Proceedings (1997), 318–326.

[Joh92] Ralph E. Johnson, *Documenting frameworks using patterns*, Oopsla'92, 1992.

[KP88] G.E. Krasner and S.T. Pope, *A cookbook for using the model-view-controller user interface paradigm in smalltalk-80*, ch. 31(3), Journal of Object-Oriented Programming, March 1988.

[Leh97] M.M Lehman, *Laws of software evolution revisited*, Tech. report, Department of Computing, Imperial College, London, United Kingdom, 1997.

[Luc97] Carine Lucas, *Documenting reuse and evolution with reuse contracts*, Phd hesis, Programming Technology Lab, Vrije Universiteit Brussel, 1997.

[Mae87] Pattie Maes, *Computational reflection*, Phd thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[Mic98] Isabel Michiels, *Using logic meta-programming for building sophisticated development tools*, Bachelors thesis, Vrije Universiteit Brussel, May 1998.

[Par94] David L. Parnas, *Software aging*, Proceeings of the 16th International Conference on Software Engineering (Soronto, Italy, May 16-21, IEEE Press, 1994, pp. 279–287.

[Sch90] Robert Schalkoff, *Artificial intelligence an engineer approach*, McGraw-Hill, 1990.

[squ] *www.squeak.org*, This page, conform the Open Source Model, constantly contains the late breaking news on Squeak.

[Ste92] Luc Steels, *Kennissystemen*, Addison-Wesley, 1992.

[Ste94] Patrick Steyaert, *Open design of object-oriented languages, a foundation for specialisable reflective language frameworks*, Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1994.

[Wuy96] Roel Wuyts, *Class-management using logical queries, application of a reflective user interface builder*, pp. 61–67, University of Groningen, 1996.

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

[Wuy00] Roel Wuyts, *Synchronizing implementation and design using logic meta programming*, Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, 2000, In preparation.