Vrije Universiteit Brussel
Programming Technology Laboratory
Faculteit Wetenschappen - Departement Informatica

A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation

Roel Wuyts

Advisor: Prof. Dr. Theo D'Hondt

January 2001

Proefschrift ingediend met het oog op het behalen van de graad van
Doctor in de Wetenschappen

Body and mind are two phenomena , observed under different conditions, but of one and the same ultimate reality. Body and mind are aspects of the living being. They operate within a peculiar principle of synchronicity wherein things happen together and behave as if they are the same . . . yet can be conceived of as separate.

–Staff Medical Manual, Ginaz School.
Brian Herbert & Kevin J. Anderson - Prelude to Dune: House Harkonnen

Contents

Acknowledgments

I would like to take this opportunity to thank my adviser, my fellow researchers and everybody who helped make this possible.

First of all I would like to thank my adviser, prof. dr Theo D'Hondt, for his never-ending patience and support. I would like to thank him for acting on gis gut-feeling. He is, after all, the spiritual father of the logic meta-programming approach adopted by several members of the Programming Technology Lab (PROG).

Besides Theo, I received lots of help from all the other fellow researchers at PROG during my five years stay at the lab. More specifically I would like to thank Kim Mens, Tom Mens, Kris De Volder and Wolfgang De Meuter for helping me through the difficult first stages when I started writing this dissertation. They were also the first people to proofread the initial drafts of the texts. Thank also to my proofreaders for their critical comments. In arbitrary order: Bart Wouters, Dirk Deridder, Maja D'Hondt, Johan Brichau, Tom Tourwé, Isabelle Michiels and Johan Fabry. Special thanks goes to Bart Wouters for his help with some of LaTeX 'features'. In general, all my thanks to everybody at PROG (members old and new) for creating a great atmosphere for doing research, for helping out on several occasions, for giving fruitful discussions and for permitting heroic battles in favour or against some particular research aspect.

My explicit thanks to some of the former members of PROG that guided me in the beginning of my research: Patrick Steyaert, Serge Demeyer, Carine Lucas and Koen De Hondt.

I also want to thank the courageous developers from MediaGeniX (and more specifically Patrick Steyaert, Koen De Hondt and Gerrit Cornelis) for providing me with the opportunity to perform experiments on their software, and to proofread the resulting chapter. Special thanks also go to the OOPSLA'99 Doctoral Symposium Team (Satoshi Matsuoka, Oscar Nierstrasz, Gregor Kiczales and Craig Chambers). I recommend anyone to submit a paper there before starting writing the dissertation, and hope they get as much feedback and help as I got. Finally I would like to thank Stéphane Ducasse for his continuing support and feedback from the day I got to know him.

Although it has become a cliché, thank goes out to a number of people for the little things, things that inadvertently pushed me further and shaped me: most of all to my parents for raising me to have an open view and a general interest in everything happening in the world; to Reinhilde for her belief in me, her devotion to mathematics and her commitments; to Luc and Nicole for their (mainly) moral support; and finally to Inge, my friend and wife-to-be, for putting up with me through the ups and downs of the last year writing my thesis. I'll do anything in my power to pay back this huge debt.

Chapter 1

Introduction

This dissertation provides a framework to synchronise design and implementation by expressing design as a logic meta program over implementation. In this first chapter we introduce the basic terminology, problems and solutions offered by this dissertation.

1.1 Design as abstraction of implementation

We first introduce the definition of *design* that is used in this dissertation. Our definition of *design* is rather broad. To illustrate this, let us take a look at the software development life cycle. In its most common form, it is divided into three distinct phases [RWL96, Som96, Pfl98, RJB99]:

1. the *analysis phase* identifies and models the problem that needs to be solved;
2. the *design phase* describes how the system should be structured in order to solve the problem described in the analysis phase. This phase is commonly divided into two smaller steps, the *architectural design* and the *detailed design*;
3. the *implementation phase* translates the design into a working solution in a particular programming language.

The primary concern in the first phase is the *problem*, while the design and implementation phases are concerned with the *solution*. So the input for the design phase is the problem and its output are blueprints that can be used in the implementation phase. In this view, we can see design as an abstract solution to the problem that is afterwards codified in the implementation phase. This results in the following definition of (software) design[1]:

Definition
Software design is an abstraction of implementation.

This definition is consistent with descriptions of design that can be found in common software engineering literature [Bud94, GR95, Som96, Pfl98]. However, it is more general because it also characterises other things as being design, such as programming conventions (for example idioms [Cop98] or best practice patterns [Bec97]). The important aspect of this definition of design is that we require it to be *explicitly related to implementation*. We consider *any* notation that can be regarded as an explicit abstraction of implementation to be design.

[1] Whenever in this dissertation we use the term *design*, we mean *software design*.

Our definition thus yields the view that design is not a stand-alone entity, but that it defines a complete range of different abstractions over implementation. This spectrum of abstractions ranges from more local and detailed design (such as programming styles [Jon87, LH89, Bec97]) to high-level abstractions that provide global views of the implementation (as in high-level design or software architectures [PW92, GS93, BJ94, BMR+96, SG96]).

1.2 The gap between design and implementation

Having clarified what we mean by design, we can now look at the relation between design and implementation. In traditional software engineering literature, implementation is typically viewed as a concretization of design [Bud94, GR95, Som96]. This implies a very general, unspecified and unidirectional relation from design to implementation. In fact, general forward engineering techniques do not bother with making this relation between design and implementation explicit. This implicitness leads to serious problems when developing object-oriented systems, as shown by the following well-known problems that are the result of the link between design and implementation being implicit:

drift and erosion *Drift* is the problem where implementation and design evolve in different directions because they are not explicitly related. *Erosion* is the process where the initial design is breached more and more in the implementation, because the design ages quickly as the implementation changes to accommodate new requirements [PW92];

documentation problems Severe problems occur when one documents a system and has to keep this documentation up-to-date. This problem is clearly visible in *object-oriented frameworks*. An *object-oriented framework* is defined as a set of classes which embody an abstract design for solutions to a family of related problems [JF88]. It can be seen as a skeleton that implements an abstract application for some specific domain. In order to get a working application, this skeleton then has to be fitted with the specific outward appearance. This is called *instantiating the framework*.

Instantiating a framework is a very conscientious and difficult process, since the correct methods and classes need to be implemented in order for the framework to be fully instantiated and usable. Depending on the application that is needed, it can be sufficient to just fill in basic information by overriding the abstract methods from the framework. However, it might also be necessary to override more methods in order to change the behaviour implemented by the framework in some points to accommodate for a rare feature of the application that was not yet foreseen by the framework developer. Because frameworks define a whole spectrum of applications that have to be realised through adapting the source code, there is a large need of support for documenting frameworks and the decisions made when implementing them. This documentation is an obvious weak point of frameworks: it should be sufficiently elaborate to allow simple instantiations of the framework, and should also provide sufficient information to allow customisations of (parts of) the framework that change the general behaviour. The problem is that the relations between all the implementation parts that make up a framework have to be documented.

The problem of supporting framework documentation in order to allow any user to customise it to a certain level is also present in other development activities, such as maintenance, reverse engineering, porting or simply extending any piece of software. For reuse to succeed, one needs to understand the system completely, and grasp its overall structure, before making changes to it.

This not only leads to problems when maintaining the software, but also when new requirements need to be included, or when novice developers join the team and need to be productive as quickly as possible;

supporting iterative development is also very hard. To make this clear, let's first go back in time to have a look at the software engineering process induced by the *waterfall model*. The waterfall model is a forward engineering, top-down approach: from analysis to design to implementation to maintenance. When an error is encountered in some phase, a complete rollback to a previous phase is necessary. When developing an application in a fairly new application domain, the design is typically not perfect from the first time on, and most errors will only become clear in the implementation. When following the waterfall mode, a lot of backtracking would occur here. *Iterative development* is targeted more towards the development of a system built for new domains and with changing user requirements. The strong point of iterative development is that it integrates top-down development (typically done in the design phase) with bottom-up development (typically encountered when implementing the design in some programming language). In each pass, the implementation learns from the design, and the design learns from information gathered in the implementation phase. This integration of top-down and bottom-up development makes iterative development much more reactive towards changing requirements and reuse. However, this flexibility comes at a cost: *synchronisation*. Properly supporting iterative development is impossible if the design phase and the implementation phase (through which is continuously cycled) have to be synchronised manually. This is not a shortcoming of incremental development alone; it just shows how crucial it is to be able to synchronise design and implementation.

The fundamental problem underlying the problems sketched above is that there is *no explicit relation* between the design and the implementation. Because design and implementation are unrelated, they can be modified independently of each other, and a modification of either one does not leave any trace in the other. As a result, synchronising such two loosely coupled entities is at best difficult and ad-hoc, and most of the time impossible. This discrepancy results in a practical development process where analysis and design are used for the initial implementation, but evolution is applied to the implementation alone [DDVMW00].

1.3 Thesis

The general context of this dissertation is the support of a *co-evolution* software development approach: both design and implementation are subject to evolution, and they influence each other continuously[2]. In the long run this should result in a development environment where all development artefacts are related to each other, such that the evolution of one artefact influences the evolution of other artefacts. This dissertation is a first step towards such a development environment, and hands over a conceptual and technological framework that forms the foundation of such an environment. Since the focus of co-evolution is on *changes* of artefacts, and how these changes impact other artefacts, the core technological component that is needed to support co-evolution is a means of *synchronizing changes* of artefacts. In this dissertation we study the characteristics of such synchronization mechanisms, and implement a *synchronization framework* to build tools that need synchronization between design and implementation. The cornerstone of this framework is a logic meta-programming

[2]The term *co-evolution* actually comes from a field in biology that studies interacting species, and their influence on each other.

language that is integrated in the object-oriented development environment. This allows to make the relation between design and implementation explicit by expressing design as a logic meta program over implementation. Moreover, since the design is a *logic program*, it can be used to generate, reason about and constrain the implementation, and vice versa. We have formulated our solution in the thesis we defend throughout the remainder of this dissertation:

> **Thesis**
> *A framework for co-evolution of design and implementation, where design and implementation are related in such a way that the one can check, generate or constrain the other, can be achieved in a logic meta-programming language integrated with a software development environment.*

Note that we limit the research to *existing* object-oriented programming languages and designs, without adapting them to fit into our approach. Instead, we want our approach to be general such that it can be adapted to different design and programming languages.

1.4 Approach and contributions

To prove our claim, we feel that it is necessary to first *study the synchronization* of design and implementation, and then to actually *implement an artefact* and perform *experiments* with it. The artefact consists of a *synchronization framework*, that is based on a logic meta-programming language integrated with the development environment. The reason we feel that the building of an artefact is so important is because it is not trivial to integrate a logic meta-programming language and an object-oriented programming language. So, to get hands-on and complete experience with the necessary technology, it is necessary to build a working prototype. This *proof by construction* approach is not new, and was also employed in a number of other cases. For example, to experiment with an object-oriented environment for building simulations using constraints, the *ThingLab* system was implemented [Bor79]. Another example is the *Refactoring Browser* tool, that was implemented in the context of the PhD research of Don Roberts [Rob99].

The proof of our claim thus consists of three parts:

1. a study of synchronization of design and implementation. It consists of a study of related work and a number of characterizations of synchronization mechanisms;

2. a logic meta-programming language and synchronization framework: the cornerstone of the artefact is a logic meta-programming language called SOUL, the *Smalltalk Open Unification Language*, in the object-oriented programming language *Smalltalk*. SOUL's technical contribution is its symbiosis with Smalltalk, meaning that Smalltalk objects and expressions can be used from within SOUL. SOUL is used as the implementation language for the *declarative framework*, a layered library of logic rules that allows us to reason about Smalltalk code. The design layer expresses several design notations (*programming conventions*, *design pattern* structures and basic *UML class diagrams*). We then conceive the *synchronization tool framework*, a framework to integrate the declarative framework in the Smalltalk development environment, so that it can react on notifications of changes to design and implementation. The *synchronization framework* is the combination of the *declarative framework* and the *synchronization tool framework*;

3. experiments to show that the synchronization framework lives up to our claim. The first case study uses the *HotDraw* framework to show how we support different kinds of synchronisation. The second case study is performed in industry on a real-world application, and shows the practical application and scalability of our approach.

To summarise, we now list the contributions made by this dissertation:

1. the first contribution is the study and characterization of synchronization mechanisms. These characterizations are used as the key variation points of our synchronization framework;

2. the second contribution is the design of the logic meta-programming language, and more specifically its symbiosis with the underlying implementation language. This symbiosis allows the logic meta-programming language to wrap or evaluate expressions in the implementation language during the logic interpretation process;

3. the third contribution is the *synchronization framework* that is used to build tools that need synchronization of design and implementation.

1.5 Research context

This dissertation should be seen as a step in a more general effort conducted at the Programming Technology Lab that focuses on how the emerging technique of *declarative meta programming* (DMP) can be used to build state-of-the-art software development support tools. DMP is an instance of hybrid language symbiosis, merging a declarative meta-level language with a standard object-oriented base language. As described in [DDVMW00], DMP emerged as a unifying approach that combined the research of several members of the lab. Before, several members were using their specific declarative languages in order to reason about or manipulate an underlying object-oriented programming language:

- Tom Mens used graph rewriting techniques to formalise the *reuse contract model* previously introduced in [SLMD96, Luc97]. A prototype tool was implemented in a logic programming language [Men99];

- Koen De Hondt's PhD research focussed on supporting reverse engineering. It introduces *software classifications* as general medium of storing and relating all kinds of software artefacts [DH98];

- Kris De Volder described *Tyruba*, a precompiler that generates Java-code from logic meta programs [DV98];

- Kim Mens described software architectures as logic meta programs to check them against the implementation [MWD99, Men00];

- Tom Tourwé uses a functional language with logic extensions to write declarative code transformations that can replace framework and design pattern structures by optimised implementations [TDM99, Tou00];

- Maja D'Hondt expresses domain knowledge as a separate aspect that can be factored out from the base program in a logic meta-programming language [DDMW99, DD99];

- [Wuy96, Wuy98] reasoned about the structure of object-oriented systems using SOUL (the Smalltalk Open Unification Language), resulting in this dissertation;

Recently we ported and extended SOUL (the logic meta-programming language that we introduce and use throughout this dissertation) to the Squeak Smalltalk environment. This new language, QSoul, is used as the common platform to develop more declarative meta programming applications. The first experiments that use QSoul are described in a number of workshop position papers [MMW00, Bri00a, Bri00b, DW00, WDVP00, DVFW00].

1.6 Dissertation overview

Chapter 2 gives a general overview of co-evolution, and discusses the characterization of synchronisation of design and implementation and related work. Then it introduces in more detail the approach we propose in this dissertation.

Chapter 3 introduces the implementation and usage of the logic meta-programming language *SOUL* we built to perform the experiments needed in the dissertation. We introduce the language, the development tools and the incremental solver.

Chapter 4 is devoted entirely to the *declarative framework*, a layered set of rules that allow to reason on a high level of abstraction about the implementation. The top level of the framework expresses three design notations (*programming conventions*, *design pattern structures* and *UML class diagrams*) as logic meta programs of implementation.

Chapter 5 describes the *synchronization framework*, a combination of the *declarative framework* and the *synchronization tool framework*.

Chapter 6 uses the well-known HotDraw framework for drawing editors to validate that our approach supports different kinds of synchronisation. We show different ways of synchronising design and implementation.

Chapter 7 describes the experiments we performed on a real-world Smalltalk application to demonstrate that our approach works in practice and is indeed scalable.

Finally, chapter 8 concludes the dissertation. It shows that the thesis has been proved, enumerates the contributions of this dissertation, and discusses future work.

Chapter 2

Co-evolution

In this chapter we give more information about co-evolution, and about synchronization. We start by defining the major concepts we need. We then propose characterizations of synchronization, and use them to discuss related work. Finally we introduce our synchronization framework from a high-level perspective. The rest of the dissertation is concerned with discussing the building blocks of the synchronization framework in more detail, and validate it.

2.1 Introduction

When a company starts developing a new product, it typically uses a clean forward engineering scheme and goes (iteratively or not) through requirements analysis, high-level design, design and implementation phases. This development process changes when a first implementation is finished. From then on, the implementation receives more and more attention at the cost of the maintaining the higher-level artefacts (such as design, analysis and documentation). This has been observed for different phases, and was coined 'architectural drift and erosion' in [PW92] for the high-level design phase. We feel this term describes the problem very well: over time the artefacts from the original phases erode more and more under the constant pressure of the ever changing implementation.

One of the reasons of this erosion is (tool) support. First of all, artefacts from the higher-level development phases merely serve as documentation and roadmap in the implementation phase. Secondly, these artefacts need to be kept in sync manually when changes are made to the implementation. These problems complement and reinforce each other, and typically result in a downward spiral where only implementation evolves, and the artefacts from other development phases stand still. Of course, this results in only the implementation being up-to-date, and an ever more difficult job to update the rest.

The development process we envision is one of *co-evolution*, where all possible artefacts evolve, separately or together, toward a solution. Evolution is then performed on all possible artefacts, with support to synchronise changes between all artefacts. To support co-evolution in practice, we need a mechanism to *synchronise* changes between design and implementation. It is clear that the scope of co-evolution is a very large one that encompasses the complete development life cycle. In this dissertation we therefore focus only on the *design* and *implementation* phases, and we build a framework that allows us to synchronise design (documentation) with implementation. However, before discussing our solution we first describe the biological foundations of co-evolution, classify different kinds of synchronisation and discuss some related work.

class	subclass	species1	species2
co-operation	commensalism	+	0
	mutualism	+	+
antagonism	allelopathy	-	0
	exploiter-victim	+	-
	competition	-	-

Table 2.1: Main classes of interactions between species, and their subclasses. + means positive feedback, - means negative feedback, 0 *means no feedback.*

2.2 Co-evolution in biology

The term *co-evolution* comes from biology, where it is the name of a research field that studies interacting species which influence each other's evolution. Enumerating all the biological definitions of co-evolution is beyond the scope of this dissertation (see [Def99] for several definitions). We only give the definition by J. Thompson [Tho94] since it is compatible with most other definitions:

Definition
Co-evolution is reciprocal change in interacting species.

The two major concepts in this definition are *species* and *interaction*. In biological terms, *species* is mostly defined in terms of *reproductively isolated*, which means that two individuals belong to the same species when they can produce fertile offspring. *Interaction* between species is much more complicated, since there are different forms of interactions with different names throughout the literature. A common taxonomy is based on the reward a species expects from an interaction with another species, and is given in table 2.1. This allows us to define two classes of interaction, each with some subclasses:

1. *co-operative interactions* are interactions where none of the species participating in the interaction is harmed. Depending on whether just one, or both species benefit from the interaction, this class is split in *commensalism* and *mutualism* cases.

2. *antagonistic interactions* are interactions where at least one of the participating species is harmed. When one species benefits from this interaction, we call this an *exploiter-victim* interaction. When both species get negative feedback from the interaction, we have a *competitive* interaction. In the case where one species gets negative feedback while the other has no feedback, we talk about *allelopathy*.

The goal of co-evolution in biology is to study the evolution of a species with respect to other species and its environment. This can then lead to information why some species get extinct, or if and how complex symbiosis between two species occurs. Hence the goal is actually to find a *model* to *simulate continuous interaction* between species in order to study the *long term effects*. While this would be very interesting to apply to software engineering, it falls outside the scope of this dissertation and is discussed in the future work in section 8.3.4. Speaking in biological terms, this dissertation is interested in the *short term effects* only, such as detecting differences between species at a certain point of time (*conformance checking*) or seeing the impact of a single evolution step in one species on the other (*impact analysis*). We are also interested in the generation of a species by another species and by the constraints imposed by one species on another species. Especially for these last two situations

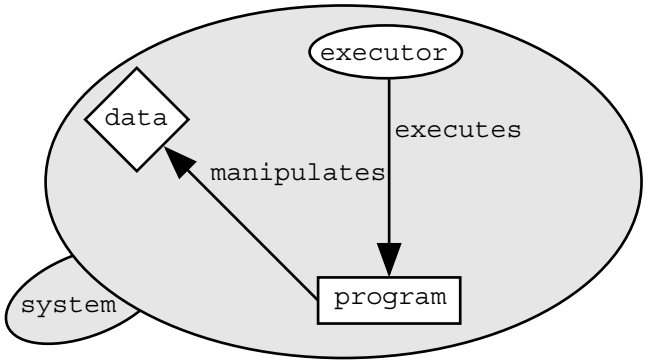

Figure 2.1: A computational system

it is hard to find matching biological situations. Hence we only use the general ideas and terminology of biological co-evolution in this dissertation.

2.3 Definitions

Before we take a closer look at the synchronization of design and implementation, and propose the *synchronization framework*, we first want to define some of the key concepts that are used throughout this dissertation: *logic meta-programming* and *reflection*.

2.3.1 Logic meta-programming

One of the cornerstones of our approach is that design is expressed as a logic meta program over implementation. However, we have not yet defined what logic meta-programming is. Before doing so we first introduce the necessary terminology, starting from the definition of a *computational system*.

A *computational system* is a system that reasons about and acts upon some part of the world, called the *domain* of that system. The main idea is that a computational system consists of *data*, a *program* and an *executor*. This is depicted in figure 2.1. The *data represents* the domain of the system. The *executor* runs the program. The *program* manipulates the data and, by doing so, conveys new information about the domain or acts upon the domain [Mae87]. We are interested in the program, since it specifies (or describes) the computational system [Ste94].

Definition
A program *is a formal, executable specification of a computational system [DV98].*

The program is expressed in a formalism that can be interpreted automatically by the executor in order to obtain the computational system it specifies. We call this formalism a 'programming language'.

Definition
A programming language *is a formalism that can be interpreted in an automatic manner in order to obtain the computational system specified by the program written in it [DV98].*

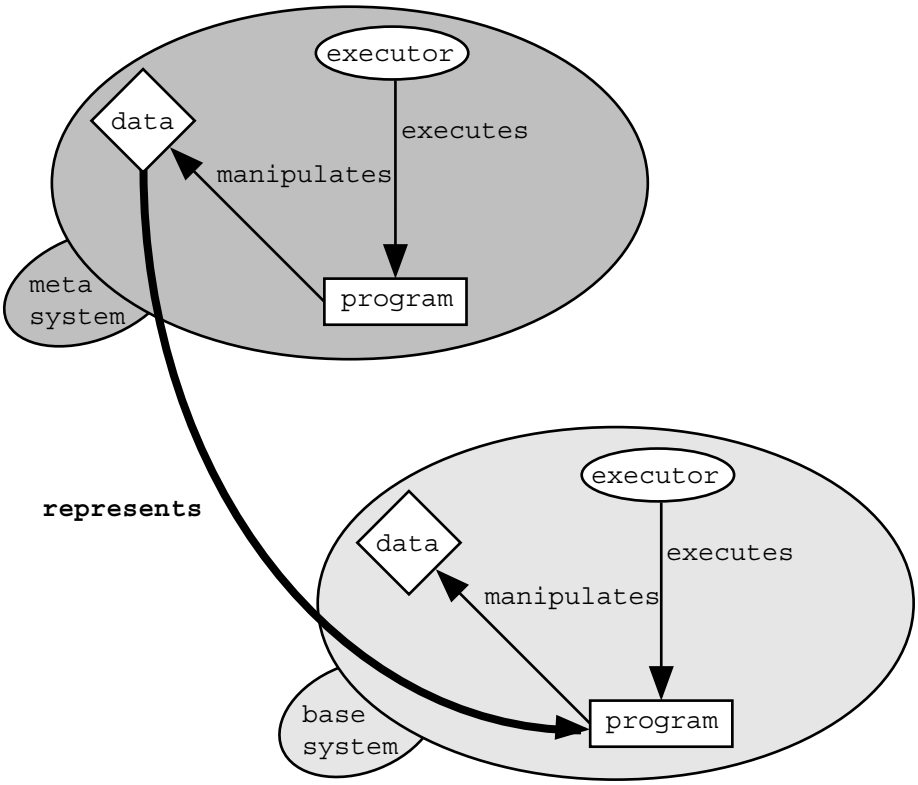

Figure 2.2: A meta system

The data of a computational system is used to represent the domain of the computational system, so that it can be acted upon by the program. It is important to stress that this is actually a mapping: some properties and relations of the domain are described *explicitly* in the data. Some other relations are *implicit* in the internal relations between the data elements, the process that interprets the representation and the domain in which the system is embedded.

Definition
Every aspect of the internal workings of a computational system that has an explicit representation in the data of that system is said to be reified.

Definition
Every aspect of the internal workings of a computational system that has no, or an implicit, representation in the data of that system is said to be absorbed.

Computational systems can be constructed for almost any domain. One only needs to represent the domain in data, and write a program to specify the system. Hence, computational systems can have other computational systems as their domain, as showed in figure 2.2.

Definition
A meta system *is a system that has as its domain another computational system, called its* base-system *[Mae87].*

Definition
A meta program *is the program specifying the* meta system *of a computational system.*

We emphasise that the concepts of *meta system* and *meta program* are *relative* concepts. This means that one and the same system (program) can be meta system (or meta program) and base system (base program) at the same time, depending on the context. We would also like to stress that the meta system does not directly manipulate its base-system; the meta system manipulates *programs* of the base-system [Ste94].

Since the only thing 'special' about a meta program is the data it manipulates (representations of base programs), meta programs can be written in general-purpose programming languages. However, to facilitate the writing of meta programs, there are domain-specific programming languages that have specific data structures and routines to represent and reason about base programs. We call such languages *meta-programming languages* (or *meta languages* in short). The language the base program is implemented in is called the *base language*.

Definition
A meta language *is a programming language specifically tuned for specifying meta programs [DV98].*

Definition
The base language *for a given meta language is the programming language for which the meta language is specifically tuned [DV98].*

We already mentioned that we express design as logic meta program over implementation. Now we have enough terminology to give a precise definition of a logic meta-programming language.

Definition
A logic meta-programming language *is a logic programming language that is used as meta language.*

2.3.2 Introspection and reflection

In this dissertation we extensively use meta programming, but we also skirt the borders of reflection. More specifically, the logic meta-programming language we have implemented is introspective, and has some reflective capabilities. Therefore we give the basic definitions here. More detailed information and examples of reflection can be found in the dissertations of Maes [Mae87] and Steyaert [Ste94], and in [DM98].

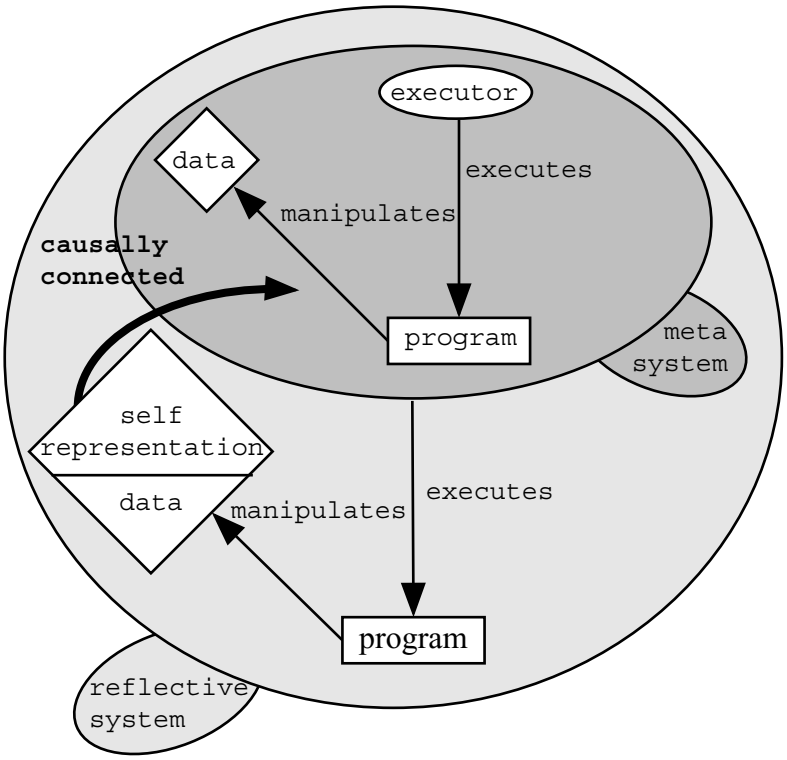

Figure 2.3: A reflective interpreter

We start by defining *causally connected*, that actually means that two artefacts are related in such a way that if one of the two changes, this leads to an effect on the other.

Definition
A computational system is causally connected *to its domain if the computational system is linked with its domain in such way that, if one of the two changes, this leads to an effect on the other.*

A classic example is a robot arm, where the domain is a set of numbers indicating the position of the robot arm. Updating these coordinates results in the robot arm to move. Vice versa, moving the robot arm updates the coordinates to reflect the new position of the arm. Now that we have defined *causally connected*, we can define reflection.

Definition
A reflective system *is a causally connected meta system that has as base system itself [Mae87].*

As is shown in figure 2.3 of a reflective interpreter, the data of a reflective system thus contains a causally connected representation of itself, called the *self representation*. Not only can the reflective system see this representation, it can also alter it. Because the representation is causally connected to the meta system, this means that the meta system is altered. Besides this 'full reflection', we can define a weaker form that allows a system to only inspect its own representation, but not alter it:

introspection.

Definition
An introspective system *is a meta system that has as base-system itself.*

There are philosophical discussions about what is exactly meant with self-representation, or otherwise said: what exactly an introspective or a reflective system can reason about. One could assume that the 'self representation' is a representation of its own program. However, this is an arbitrary line, since it is not clear where this ends. For example, are the libraries used by the program part of the self-representation or not ? If so, are libraries used by these libraries also taken into account ? In this dissertation we assume a broad view on 'self-representation'. We take this view because most practical reflective or introspective systems support this view: not only do they reason about themselves, they also reason about the rest of the system executing them.

2.4 Co-evolution and synchronization

In chapter 1 we have already explained that the general context of this dissertation is to support co-evolution. In the long run this should result in a development environment where all development artefacts are related to each other, such that the evolution of one artefact induces evolution on other artefacts. This dissertation is a first step towards such a development environment, and hands over conceptual and technological frameworks that form the foundation of such an environment. Since the focus of co-evolution is on *changes* of artefacts, and how these changes impact other artefacts, the core component that is needed to support co-evolution is a means of *synchronizing changes* of artefacts. Therefore, this dissertation focuses on a *synchronization framework* such that design and implementation can be checked, enforced or generated from the other, at a user-definable time. The next sections take a closer look at the synchronization of design and implementation, and at the *synchronization framework* we propose.

2.5 Synchronizing design and implementation

In order to give support for co-evolution, we first of all need a framework to synchronize design and implementation. In this section we first take a closer look at the characteristics of synchronization, and we then discuss some related work.

2.5.1 Characterizing synchronisation

In this section we classify different kinds of *synchronisation*. Strictly speaking, *synchronisation* means *to occur at the same time, to move or operate in unison* [Web96]. So, when we synchronise two participants this means that when one of the two changes, we see the effect on the other. After the synchronisation, both participants are *in sync*, meaning that there are no inconsistencies between them. This 'general' definition implies a direct connection between the two participants: as soon as one of the two changes, we see this effect on the other. In practice, the meaning of synchronisation is broader since the timing constraint can be more relaxed. The effect on the other does not necessarily take place immediately. However, the result of the synchronisation is still that both participants are *in sync*. This more relaxed definition opens up a spectrum of synchronisation. At the one end there is the use of synchronisation according to the 'traditional' definition that implies that directly after a change both participants are synchronized. At the other end of the spectrum we find the case where

the synchronisation process is initiated manually, possibly some time after a number of changes to one or both participants have occurred. When closely investigating the synchronization of design and implementation, we found the combinations of the following characterizations helpful to distinguish several kinds:

direction of synchronisation Although there are two partners to be synchronized (design and implementation), the process does not necessarily works in both directions. When only one partner can be derived from the other, we have a *unidirectional* synchronization. With a *bidirectional* approach, design can be derived from implementation and vice versa. This classification has a strong impact on the results that can be expected from the synchronisation: a unidirectional system can only be used to generate one of the two participants from the other, or to do a limited conformance check. A bidirectional system can be used both for conformance checking and for generating one participant from the other and vice versa;

action to be taken Different actions can be taken when the synchronization detects contradicting or missing items. This can result in a *report* so the user can choose what to do, or in an attempt to *resolve* the situation automatically;

notification time When missing or contradictory items are found, the user has to be notified that this change resulted in a loss of synchronisation. An important question is when this user notification should occur: *before*, *during* or *after* the change has been applied to the participant. We call these respectively *proactive*, *reactive* or *retroactive* notifications;

trigger time The synchronisation can be triggered *directly* after every single change, or *delayed*, after several changes were made;

scope A synchronization process is governed by rules that determine whether two items are conflicting or not. These rules can have different scopes: *global* or *local*. Global means that they are applicable to all items. *Local scope* means that they are only usable for the particular part of implementation or design where they are defined.

implementation granularity Reasoning over the implementation in order to synchronize it with design can be done on different levels of granularity. For example, only specific information of the implementation (such as call-graph information) might be used. On the other, complete parse trees or objects might be used. This characterization takes this granularity into account;

static/dynamic Up until now we did not specify whether the synchronization process used static information, dynamic information or both. Static information means that the source code is used, while dynamic information is gathered at runtime. Both could be combined in order to get a better view of the implementation.

We use these characterizations to discuss the most important related work that tries to synchronise design and implementation in some way or another, and as the design space for the synchronization framework.

2.5.2 Related work

In this dissertation we make the relation between design and implementation explicit in order to synchronise design and implementation. The goal is to build *one* framework that can be used to support *different kinds* of synchronisation. In this section we discuss the most important related techniques

	Eiffel	**CCEL**	**CS**	**AstLog**	**LGA**	**Lint**	**SRM**	**FM**	**SF**
direction	one	one	one	one	one	one	one	both	both
action	report	report	report	report	action	report	report	action	both
not. time	reac	retro	retro	retro	pro	retro	retro	retro	all
triggering	del.	del.	del.	del.	del.	del.	del.	del.	both
scope	local	local	local	global	global	global	global	global	global
granularity	comp.	partial	comp.	comp.	partial	comp.	partial	partial	comp.
static-dyn.	dyn.	static	static	static	dyn.	static	static	static	static

Table 2.2: The related work that is of interests to us, classified using the characterizations from section 2.5.1. CS *stands for CoffeeStrainer,* LGA *for Law Governed Architectures,* SRM *for Software Reflexion Models and* FM *for the Fragment Model. The last entry (SF) is an abbreviation for the* synchronization framework *that will be introduced in section 2.6.*

```
make(ce: POINT; ra: REAL) is
        –Set circle to have center ce and radius ra.
    require
        point_exists:ce |= void;
        positive_radius: ra >0.0
    ...Rest of routine declaration omitted ...
```

Figure 2.4: Example of an Eiffel precondition that states that for a call of make *to be correct the first argument must be non-void and the second argument must be positive.*

that support forms of synchronisation that were of interest to our work. Because of the large diversity we divided them into groups, where we list the key techniques. Table 2.2 compares the most important techniques using the characterization from previous section. Note that the last entry in this table (*SF*) is the entry for the *synchronization framework*, the software artefact constructed for this dissertation that we introduce further on in this chapter. While we do not discuss our entry in this section, we added it here for completeness.

Constraining the implementation

Some related work is concerned with letting the user put constraints on design or implementation. These constraints make some design explicit in the implementation, such as a certain assumption about the state of an object or a programming convention.

Eiffel The example that springs to mind is the explicit *assertions* (pre- and postconditions) construction in the object-oriented programming language *Eiffel* [Mey88, Mey00]. With assertions, developers can describe specifications of software components by specifying invariants. An assertion takes the form of a boolean Eiffel expression, and can be used as pre- and postconditions of routines, as invariants of a class and in loops. Hence the scope is local, and is determined by the position in the source code. The assertions can be checked at runtime, to help with debugging. For example, figure 2.4 gives an example of an assertion that expresses that, when creating a circle, the centerpoint that is

```
PointersAndAssignment {
        // If a class contains a pointer member, it must declare an assignment operator:
        AssignmentMustBeDeclaredCond1 (
                Class C;
                DataMember C::cmv | cmv.is_pointer();
                Assert(MemberFunction C::cmf; | cmf.name() == "operator=");
        );
        // If a class inherits from a class containing a pointer member, the
        // derived class must declare an assignment operator:
        AssignmentMustBeDeclaredCond2 (
                Class B;
                Class D | D.is_descendant(B);
                DataMember B::bmv | bmv.is_pointer();
                Assert(MemberFunction D::dmf; | dmf.name() == "operator=");
        );
};
```

Figure 2.5: Example of a CCEL constraint class, with two constraints that express that whenever a C++ class contains a pointer member, or inherits from a class containing a pointer member, it must declare an assignment operator.

```
sametree(node)
<-      op(nodeop),
        with(node, op(nodeop)),
        not(and(    with(node, kid(n, nkid)),
                    kid(n, not(sametree(nkid)))));
```

Figure 2.6: An Astlog predicate sametree *used to compare two parse trees (the current node that is implicit, and the passed argument,* node*). The predicate holds if root nodes have the same opcode and all corresponding children have the same structure.*

```
public abstract class MediaStream {
    public void initialize() {
    /*-
        private AMethod initializeMethod() {
            return Naming.getInstanceMethod(thisClass,
                "initialize", new AType[0]);
        }
        private boolean overrides(AMethod m1, AMethod m2) {
            if (m1 == null) return false;
            if (m1.getOverriddenMethod() == m2) return true;
            else return overrides(m1.getOverriddenMethod(), m2);
        }
        private AStatement getFirstStatement(ConcreteMethod m) {
            AStatementList s1 = m.getBody().getStatements();
            return s1.size() >0 ? s1.get(0) : null;
        }
        private boolean callsInitialize(AStatement s) {
            if(!(s instanceOf ExpressionStatement)) return false;
            AExpression e=((ExpressionStatement) s).getExpression();
            if(!(e instanceOf InstanceMethodCall)) return false;
            AMethod called=(InstanceMethodCall e).getCalledMethod();
            return called == initializeMethod()
                || overrides(called, initializeMethod);
        }
        public boolean checkConcreteMethod(ConcreteMethod m) {
            rationale = "when overriding initialize, " +
                "super.initialize() must be the first statement";
            return implies(overrides(m, initializeMethod()),
                callsInitialize(getFirstStatement(m)));
        }
    -*/
```

Figure 2.7: Example of a CoffeeStrainer constraint that specifies that subclasses that override a method initialize *should first call* super.initialize() *before doing anything else.*

passed has to exist, and the radius has to be positive. Of course, this means that the information that is used is dynamic. Note that assertions can only be used to check specifications, and for example not to query the software system (for example, using the example from above, to find all parts of the implementation that check whether the radius is positive), or to generate code.

CCEL CCEL [MDR93] allows us to express and enforce constraints on C++ code, such as *The member function M in class C must be redefined in all classes derived from C*, *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor*, or *All class names must begin with an upper case letter*. The constraints are included in the source files in specially formatted comments. Syntactically, CCEL constraints resemble expressions in first-order predicate calculus, allowing programmers to make assertions involving existentially or universally quantified CCEL variables. Constraints can be grouped in constraint classes, but there are no provisions for composing such classes, or calling constraints from one class in another class. An example of a class grouping two constraints is given in figure 2.5. Note that CCEL constraints only have access to the top-level declarations (class declarations, signatures of methods and field declarations), and not to the complete parse tree.

Astlog Another constraint system that works on C++ code is *Astlog* [Cre97]. Astlog is a logic programming language with two specific additions that facilitate the reasoning over parse trees. First it avoids the overhead of translating the source code into the form of a Prolog database by allowing predicates to work directly on C++ code. Second, terms are matched against an implicit current object, rather than simply proven against a database of facts, leading to a distinct "inside-out functional" programming style. Using Astlog we can perform logic queries on C++ code, for example, as shown in figure 2.6 to compare the structure of two parse trees. Hence we can use it to check whether the C++ implementation conforms to the structure we describe in the query (but not to generate code, for example). The result of the query is the report that indicates whether, and where, the implementation conforms to the design.

CoffeeStrainer *CoffeeStrainer* [Bok99] is a system that allows us to statically check structural constraints on Java programs. The constraints are written in stylised Java, and have access to the complete Java parse tree. An example of a constraint is given in figure 2.7. The scope of the constraint is determined by its position in the source code. For example, a constraint that appears in a class or interface applies to that class or interface and its subtypes. Subclasses can strengthen constraints, but never weaken them.

LGA Last but not least we mention the work on *Law Governed Architecture (LGA)* [Min96, MP97], which expresses global constraints (called *laws*) over the interactions between the modules of a system. The laws allow to regulate interactions between *objects*. An *object* is a triple containing an *exterior*, an *interior* and an *agent*. The exterior and interior of an object are bags of Prolog terms describing attributes. The semantics of these attributes are given by the law that uses them. The *agents* are treated as black boxes that generate messages. The laws are Prolog programs that prescribe the result of an object sending a message. While the laws thus regulate run-time interactions, static interactions can be regulated when a configuration of objects is created[1]. Also noteworthy is that not only the object-oriented programming language Eiffel is supported, but also Prolog, even though Prolog

[1]However, it is still information about interactions that is regulated. Hence, in table 2.2 the entry that says that only dynamic information is used.

has no notion of objects or messages. The trick is to partition the clauses in subspaces that can be viewed as objects by the laws[2].

Generation and weaving of code

Recently a number of techniques were introduced that allow the separation of a base program from other, specific concerns [KLM+97, HO93, CE00]. The core idea is to write a *base program* in some programming language, and to write (non-functional) aspects in dedicated *aspect languages*. These aspects are typically cross-cutting concerns that have to be merged with the complete base program. Typical examples are security or persistence. The code in the aspects and the code of the base languages are then merged by a so-called *weaver*. An approach we find particularly interesting is *aspect-oriented logic meta programming* [DV98, DVD99]. Here the aspects are all expressed as logic programs, and the weaving is done by a logic meta-programming that generates source code. While originally developed for Java, this work is now also continued in Smalltalk [Bri00a, BDMDV00, Bri00b, DW00, WDVP00].

Conformance checkers

Several tools allow us to check whether an implementation conforms to some given design. The basic example is that of *Lint* [Joh77], originally a tool to check C code for common programming mistakes. Lint is built around a regular expression search engine that allows us to express fairly sophisticated string patterns. An interesting port of the original Lint is *SmalltalkLint* [RBJO96], that allows regular expressions searches on Smalltalk parse trees. *Lint* and its derivatives are a great example of lightweight approaches to express simple, string-based programming conventions. Note that sacrifices were made in the expressivity of patterns to gain better performance, most notably regarding abstraction facilities and recursion.

Software reflexion models [MNS95, MN95, Mur96] show where an engineer's high-level model of the software does and does not agree with a source model, based on a declarative mapping between the two models. The idea is that an engineer defines a *high-level model* of the software, then extracts a *source model* (such as a call-graph or an inheritance hierarchy) from the source code, and then defines a declarative mapping between the two models. The mapping uses regular expressions to relate entries from the high-level model with the source model. Then a software reflexion model is computed that shows where the high model agrees with and where it differs from the source model. This information is then used to update the high-level model, the mapping or the source code, and to compute a new reflexion model. This can of course be repeated iteratively. Applications include re-engineering, design conformance checking and system understanding, which was confirmed on several case studies. Generation of code using the models and the mapping is however not supported. Note that the mapping is expressed using a medium (regular expressions) that is not very expressive or powerful, but can be checked very fast. An approach that might be seen as starting from the opposite direction is proposed by Kim Mens [MWD99, MMW00, Men00]. The idea is to use a very expressive but much slower logic programming language. It allows us to do a conformance check between a software architecture and an implementation. The software architecture is expressed as a logic meta program, and the actual conformance check is done by a logic programming language.

[2]Note that, although Prolog is supported, the laws only regulate interaction between objects, as noted in table 2.2.

Fragment Model

Last but not least we want to discuss the Fragment Model, and the tool support for object-oriented patterns implemented in Smalltalk using this model [Mei96, FMvW97]. The goal of the tools is to make the use of patterns easier in software development, more specifically to provide support to bind program elements to roles in a pattern, to check whether patterns still meet the invariants and to generate program elements. The idea is to capture every component that is relevant to a design pattern in a pattern *fragment*. Fragments are defined as a combination of *structural elements* (class-roles, method-roles that must be fulfilled, inheritance relationships, etc.) and *constraints* that restrict the reorganizations that can take place on the design level. The constraints are pieces of Smalltalk code that implement boolean checks and that can use a number of predefined *inquiry operators* to get to the properties of the fragment they are working on. The constraints are validated whenever an editing operation (as provided by the fragment) has modified the fragment, or whenever validation is triggered by another fragment. When inconsistencies are detected, *exceptions* are raised. The system includes several possible exception handlers (of which only one can be active at any given moment). The exception handler is responsible to implement the action that needs to be taken. Several types are implemented that allow the developer to *ignore*, *discard*, *warn*, *repair* or *choose between different options* when differences between the fragment and the implementation are encountered.

Once a fragment is defined, it can be bound manually to source code (mapping the design elements represented by the fragment to the implementation). When the fragment is bound, the constraint can check whether the implementation conforms to it. The fragment can also be used to generate template code. However, when generating code, only the structural information of the fragment is used. The semantics of the fragment, that are implemented by the constraint, are not taken into account. In practice this means that only class hierarchies and methods without an implementation can be generated. However, the interesting aspect is that this approach is clearly bi-directional: the fragments can be used for extracting design information from the code and to generate source code. This is quite different from all the other approaches, that are unidirectional.

2.6 A framework to synchronize design and implementation

As said before, the goal of this dissertation is to provide a framework to synchronise design and implementation so that the one can check, generate or constrain the other. The cornerstone of our solution is that design is expressed as a logic meta program over implementation. Now that we have seen the necessary terminology, we can describe our approach to the synchronization framework from a high-level perspective.

To synchronise design and implementation we propose a setup as depicted in figure 2.8. As can be seen, there are three participants: a *design repository* containing the design information, an *implementation repository* with the implementation, and the actual *synchronization framework*, that consists of the *declarative framework* and a mechanism to trigger design and implementation changes and binding actions to these changes. The *declarative framework* is actually a mapping between design and implementation, and consists of logic meta programs that express design as an abstraction of implementation. The fact that design is expressed as a logic meta program has several advantages:

1. the relation between design and implementation is made explicit, since the design is expressed in terms of the implementation;

2. we use the open character of a logic programming language, which allows us to build a system where rules can easily be added to implement specific behaviour, and where logic repositories

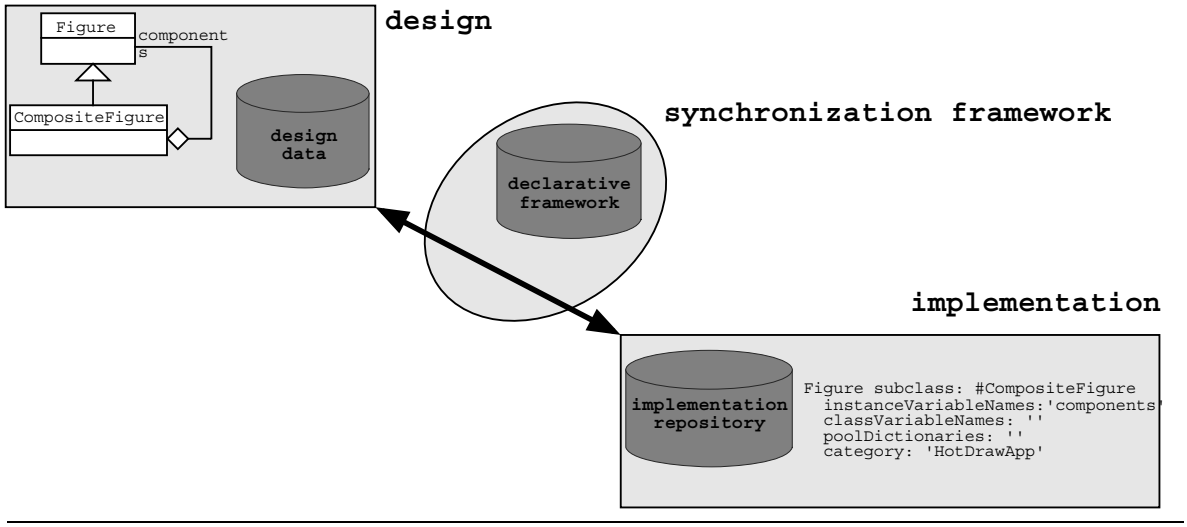

Figure 2.8: General setup of our framework to synchronize design and implementation

are used to group and nest rules;

3. the inherent declarative nature of logic programming is very well suited to express design notations, since these are typically also declarative in nature;

4. since all design notations we support are expressed in the same medium (as logic meta programs), they can be expressed in terms of each other. For example, the structure of design patterns [GHJV94] can be described using UML class diagrams [RJB99], taking *best practice patterns* [Bec97] or other programming conventions into account;

5. logic programs express relations between their variables, in a mathematical sense. This property is also referred to as the *multi-way property* of logic programming languages. Concretely this means that the same logic program can be used in many different ways depending on the information that is passed to it.

For example, consider the following composite pattern relationship between two classes [GHJV94]. It relates two logic variables *?component* and *?composite* using the relation described by the *compositePattern* rule[3].

```
compositePattern(?component, ?composite)
```

[3]This example uses the logic meta-programming language we have implemented (see chapter 3), where variables are denoted by question marks. The *compositePattern* rule is one of the rules expressing design patterns (discussed in chapter 4)

We can use this relation in 4 different ways (all possible combinations of the two arguments):

- when we pass two actual classes, the relationship returns wether it holds for these two classes. For example,

  compositePattern([VisualPart], [CompositePart])

  can be used to check whether *VisualPart* and *CompositePart* are in a composite pattern relationship;

- when we pass only the component class, then we can infer the classes that play the role of *composite class*:

  compositePattern([VisualPart], ?composite)

- We can also pass the composite class, and infer all the component classes for that composite class:

  compositePattern(?component, [CompositePart])

- when we pass no information (but only variables), all possible combinations of components and composite classes as described by the relation are found:

  compositePattern(?component, ?composite)

The actual synchronisation is done by a logic meta-programming language that uses the logic meta programs to compare information about design and implementation. The results of the synchronisation (that indicate possible discrepancies between design and implementation) can then be reported, or used by other tools to take appropriate action. One possibility is to use the results in other logic programs, for example to generate parts of the implementation. Of course the synchronization engine needs to be integrated in the development environment, so that it can receive notifications of changes in implementation or design. As the logic meta-programming language is integrated in the development environment, results from the synchronisation can also be used directly in the development process, for example to constrain the development or to generate code.

Last but not least we want to apply the characterizations discussed in section 2.5.1 to our approach. An overview can be found in table 2.2, where our approach is shown as the last entry, labelled *SF* (short for synchronization framework). First of all the *direction* characterization is *both*, since the logic meta-programming language allows us to see the impact of changes to design on the implementation, and vice versa. Because of the integration of the logic meta-programming language with the development environment, we cannot only *report*, but also *act* when discrepancies between design and implementation are detected. For example, we show how the design documentation can be updated automatically as the implementation changes. A second result of the integration is that the *notification* of discrepancies can occur at any time: before the change is applied, immediately after it is applied or even later on. A third result of the integration is that the *trigger time* can be direct or

delayed. The scope of the rules governing the synchronization process is global in our approach. The reason is that the rules mapping design to implementation reside in a logic repository that has nothing to do with either design or implementation. This is analogous to the other approaches that use a logic programming language as synchronization engine. Another approach is to integrate these rules with implementation or design, as is done in the constraint languages we saw in section 2.5.2. The granularity of our approach is completely user-defined, but ultimately supports objects or parse trees. As we will see later on, the reason is the symbiosis of our logic meta-programming language with Smalltalk that allows us to wrap and use any Smalltalk object in our logic meta-programming language. In this dissertation we only use static reasoning over the implementation. While we did experiments with reasoning over dynamic information collected at runtime [RDW98], we currently have no support for good runtime integration of our logic meta-programming language. This is discussed in the future work in section 8.3.2.

2.7 Validation and roadmap

The previous paragraph describes the foundations of the framework we propose to synchronise design and implementation so that the one can check, generate or constrain the other. The validation of this claim is *proved by construction*, in different steps:

1. first we introduce a logic meta-programming language called SOUL (an acronym for *Smalltalk Open Unification Language*) in the object-oriented programming language Smalltalk [GM90, Lew95]. This language is integrated in the Smalltalk development environment and allows us to express logic meta programs, and perform logic queries that use these programs to reason about the Smalltalk source code;

2. then we describe the *declarative framework*, a layered structure of rules to reason about the base language. Closest to the implementation we find the *representational layer* that contains the rules that reify the core concepts of the base language. In the case of SOUL reasoning over Smalltalk code, these are *classes*, *methods*, *instance variables* and *inheritance*. Other layers then build on this one, raising the level of abstraction of the predicates reasoning about the implementation.

3. in the top layer in the *declarative framework* we express design notations as logic meta programs over implementation. As examples we use *programming conventions*, *design patterns* and *UML class diagrams*;

4. next we introduce the *synchronization tool framework*, a Smalltalk framework to integrate our synchronization engine in the development environment. We combine this with the *declarative framework* to get the actual *synchronization framework*;

5. then we use the *synchronization framework* on two case studies to prove that it can indeed be used to synchronise design and implementation. First we use the *HotDraw* framework (a framework for structured drawing editors [Bra92, Joh92, BJ94, Cha94]) to show how the checking, generation and enforcement of design on implementation, and vice versa, is done. Then we experiment on a real world Smalltalk application to demonstrate that the approach is scalable.

2.8 Summary

In this section we introduced the necessary background regarding co-evolution and synchronisation. We started by discussing the biological view of co-evolution as a mutual interaction between species. Then we introduced the necessary definitions, most notably the notions of *meta-programming*, *introspection* and *reflection*. Then we introduced our view on co-evolution as a development approach where both design and implementation are subject to evolution, and where they influence each other. To allow tools that support co-evolution we therefore need to synchronize changes between design and implementation. Therefore we first of all studied the characteristics of the synchronization of design and implementation. The resulting characterizations are a conceptual contribution of this dissertation. They are then used to discuss the related work. While several approaches look promising, none currently exists that encompasses the complete spectrum of synchronization we described by the characterizations. Hence, in order to prove our claim that we can build such a framework, the following section describes from a high-level perspective our overview of such a framework . In the remainder of this dissertation we will build the synchronization framework, and validate its support for co-evolution, hence proving our claim.

Chapter 3

SOUL and the incremental solver

In this chapter we introduce the logic meta-programming language we use to synchronise design and implementation. We describe the basic syntax and usage, the development tools and the incremental solver. In the next chapter we use this language to implement the *declarative framework*, a layered set of rules to facilitate reasoning over the implementation.

3.1 Introduction

For the validation of our claim, we need a logic meta-programming language to express design as a logic meta program over implementation and to synchronise design and implementation. The research language we conceived is called the *Smalltalk Open Unification Language* (*SOUL*). SOUL is a logic programming language (analogous to Prolog [CM81, SS88]) that is implemented in, and lives in symbiosis with, the object-oriented programming language *Smalltalk*. SOUL allows users to perform logic queries over Smalltalk source code, without the need of representing this source code explicitly in the logic repository. This is done with the `smalltalk term`, a special construct that allows us to invoke Smalltalk code during the logic interpretation process. Using the `smalltalk term`, concepts from the base language can easily be reified in SOUL.

Once we have a logic meta-programming language we can express design as a logic meta program over implementation. This is done in a structured manner, resulting in a *declarative framework*. The idea is to layer the rules expressing design in function of the implementation, where rules in one layer are expressed in terms of the lower level layers. The layer that reifies the base-language's concepts (and that all other layers depend on) is the *representational layer*. Other layers then add lots of predicates that are expressed in terms of the predicates from previous layers and that facilitate reasoning about implementation. This structure makes it easy to swap one layer for another should this be required.

Note that this chapter does not go into the details of SOUL's implementation (the manual [Wuy00] includes an overview of the implementation). Instead we focus our attention on the specific language features and rules that allow SOUL to reason over Smalltalk. These rules are used in subsequent chapters, when we express design as a logic meta program over implementation and when we perform experiments.

3.2 SOUL as a logic meta-programming language

In chapter 2 we defined a meta language as a programming language with specific data structures and routines to represent and reason about base programs. For SOUL we chose to make a symbiosis with Smalltalk such that we could directly reason on Smalltalk base programs. Therefore the first part of this section discusses this symbiosis in detail, and more specifically the language constructs that allow the symbiosis: the `smalltalk term`, the *generate predicate*. Afterwards we discuss the additions to the Smalltalk term that make SOUL a reflective logic meta-programming language. Then we describe the *incremental solver* that uses local propagation techniques to solve networks of logic relations expressed in SOUL. We end this overview with the development tools that are available to SOUL developers to interact with SOUL, and the tools for the Smalltalk developers that we built using SOUL. However, we start this section by giving SOUL's syntax.

3.2.1 Syntax of SOUL

Before we give the first SOUL code examples, figure 3.1 gives the syntax of SOUL: the starting production is underlined, square brackets (*[ ]*) delimit optional constructs; braces ({ }) indicate zero or more repetitions of the enclosed construct; parentheses (*( )*) indicate simple grouping of constructs; a vertical bar (|) indicates choice of one from many; literal text in definitions is denoted using bold typewriter font.

3.2.2 Symbiosis with Smalltalk

In this section we first of all motivate why we need symbiosis between the logic meta-programming language and the base language for the purpose of synchronization. Then we discuss the *up-down mechanism* uses to obtain the symbiosis. Finally we discuss some of the implementation details of the implementation of the symbiosis mechanism in SOUL.

Symbiosis for synchronization

For a logic meta-programming language to reason about a certain base program, one possibility could be to build a large logic repository containing the complete source code of that base program in logic format. This indeed allows to write logic programs that reason about the source code in the logic repository. However, this also means that every program is represented two times: once as regular source code used in Smalltalk, and once in a logic format to reason about in SOUL. This poses two problems. The first is where to draw the boundary when making a logic representation of a program. Is it enough to represent only the classes of the program, or do we also need the code of the libraries that are used by the program ? And the libraries these libraries use ? Making this decision is very difficult to make in general. The second problem has to do with the synchronization. Since we use the logic meta-programming language to synchronize changes between design and implementation, the fact that there are two representations for one and the same base program means that we need an extra synchronization step. This is necessary to make sure that the logic representation used in the logic meta-programming language is kept in sync with the source code.

To solve these problems, we chose to make a *symbiosis* between Smalltalk and SOUL. This symbiosis makes any Smalltalk object directly usable as a logic term in SOUL, and even allows to write Smalltalk expressions (that can be parametrized by logic variables). Both features are enabled by the `smalltalk term`, a logic construct containing Smalltalk code that can be executed during logic interpretation. This way Smalltalk expressions can be wrapped and used as logic constructs. Before

clauseSequence := clause { **.** clause }
clause := *fact* | *rule* | *query*
fact := **Fact** *term*
rule := **Rule** *compoundTerm* **if** *terms*
query := **Query** *terms*

term := *simpleTerm* | *compoundTerm* | *specialTerm*

simpleTerm := *constantTerm* | *variableTerm* | *booleanTerm*
constantTerm := word
variableTerm := *normalVariableTerm* | *specialVariableTerm* | *underscoreTerm*
normalVariableTerm := **?**word
specialVariableTerm := **@**word
underscoreTerm := **_**
booleanTerm := **true** | **false** | **fail**

compoundTerm := *regularCompoundTerm* | *listTerm*
regularCompoundTerm := *simpleTerm* **(** *possiblyEmptyTerms* **)**

specialTerm := *smalltalkTerm* | *quotedString* | *cutTerm*
smalltalkTerm := **[** `extended smalltalk code` **]**
quotedString := { `string` }
cutTerm := **!**

possiblyEmptyTerms := *terms* | *empty*
terms := *term* { **,** *term* }
empty :=

listTerm := *emptyList* | *nonEmptyListTerm*
emptyList := <>
nonEmptyList := <*terms* | (*terms* | (*variableTerm* | *nonEmptyList*)) >

Figure 3.1: SOUL Syntax

we continue, we have a note on behalf of Prolog users: despite its name, a `smalltalk term` can be used both as term and as clause.

The first, and simplest usage of a `smalltalk term` is to wrap Smalltalk objects and use them in SOUL as constants. For example, the `smalltalk term` *[Array]* is a SOUL constant that wraps the Smalltalk class *Array* in order to use it in SOUL. For example, if we want to ask SOUL to test whether *Array* is a class, we could evaluate the following query [1]:

Query class([Array])

Since *Array* is indeed a class in the Smalltalk system, this query succeeds. We like to stress that the `smalltalk term` really passes the Smalltalk class *Array*, and not just the name or another representation of the class. For example, would we like to pass the name of a class, we have to use a `smalltalk term` with a string containing the name of the class. For example, *['Array']* is a `smalltalk term` representing the Smalltalk string 'Array'.

The `smalltalk term` is not only used to wrap objects and use them in SOUL. It can also contain Smalltalk expressions that are evaluated during the interpretation. Moreover, these Smalltalk expressions can be parametrized with logic variables. When the `smalltalk term` is evaluated, these logic variables are substituted by their current binding. If they are unbound, the interpretation of the `smalltalk term` fails. As an example, we define a rule that gives all selectors (the names of methods) of a class. In Smalltalk you can get the selectors of a class by sending that class the message *selectors*. The result is a Smalltalk collection with all the selectors of that class. Since the Smalltalk system can tell us what the selectors are for a class, we use a `smalltalk term` to ask the selectors of a class. This is expressed in the following rule:

Rule simpleSelectors(?c, ?selectors) **if**
    class(?c),
    equals(?selectors, [?c selectors]).

This rule uses a `smalltalk term` to ask the logic variable *?c* for its selectors. Intuitively, the meaning is clear: the class that is bound to the variable *?c* when the `smalltalk term` is evaluated, is asked for its selectors by sending it the *selectors* message. If *?c* is not bound when the `smalltalk term` is evaluated, the `smalltalk term` fails. So, using the *simpleSelectors* rule we can get the selectors of the class *Array* in SOUL by evaluating the following query:

Query simpleSelectors([Array], ?sels)

When this query is evaluated, it uses the *simpleSelectors* rule, binding the variable *?c* to the `smalltalk term` *[Array]*. Then, when the `smalltalk term` is evaluated the class *Array* is asked for its selectors, and the result is bound to the variable *?selectors*. The result of the query is a solution for the variable *?sels*, binding it to a `smalltalk term` that wraps the Smalltalk collection with the selectors of Array:

[1] Of course, a predicate *class* should be defined. We implement this predicate in section 4.3. For now, just suppose it is implemented and succeeds if the argument passed is a Smalltalk class, and fails if it is not.

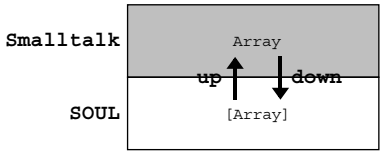

Figure 3.2: The up-down mechanism to let Smalltalk objects travel between Smalltalk and SOUL.

```
?sels      ->  [IdentitySet(#identityIndexOf:replaceWith:startingAt:stoppingAt:
                  #decodeAsLiteralArray #refersToLiteral: #storeOn: #printOn: #multiBecome:
                  #startingAt:replaceElementsIn:from:to: #literalArrayEncoding #isLiteral
                  #equalsLiteral: #multiAllInstances #identityIndexOf:from:to:ifAbsent:
                  #replaceFrom:to:with:startingAt: #multiAllLiveInstances #emWriteLiteralOn:)]
```

The up/down mechanism

In the previous section we saw that a `smalltalk term` contains a Smalltalk expression. This Smalltalk expression can be very simple, and consist of nothing but a class, but it can also be parametrized by logic variables. In this section we want to explain how a Smalltalk term is interpreted in SOUL, and hence how the symbiosis works. The up/down mechanism we use was introduced in the PhD dissertation of Steyaert as the core implementation mechanism for a framework for open designed object-oriented programming language [Ste94]. The implementations of the object-based object-oriented programming language *Agora* uses the *up/down* mechanism to get reflection with their object-oriented implementation language (Smalltalk, C++ and Java) [DM98]. Here we use it as the cornerstone to get reflection between two languages from different paradigms. To explain how this works, we first of all have to consider *two* levels in the semantics of SOUL:

1. the *up* level is the level of SOUL's implementation language, Smalltalk;
2. the *down* level is the SOUL level being evaluated by the *up* (Smalltalk) level.

The basis for the symbiosis is that Smalltalk objects can cross this boundary. Hence, any Smalltalk object can be used in Smalltalk, but it can also be used as a logic term in SOUL. It is this transition (shown in figure 3.2) between these two levels that allows for the symbiosis between Smalltalk and SOUL. Hence we can *down* a Smalltalk object *Array* (which is the class *Array*) to get the logic term *[Array]*. We can also *up* the logic term *[Array]* and get the Smalltalk object *Array*. The conversion is done implicitly during the evaluation of the `smalltalk term`. We will explain this in two steps, first for a `smalltalk term` that is not parametrized with logic variables, and then for a `smalltalk term` containing references.

Interpreting a `smalltalk term` with an expression that contains no logic variables is straightforward, as is illustrated with the interpretation of the following query:

Query [Array selectors isEmpty]

To interprete the `smalltalk term` in SOUL, we evaluate the Smalltalk expression it contains. The result of this evaluation in Smalltalk is a Smalltalk instance of class Boolean instance (*false*, since

the class *Array* contains methods). This Smalltalk object is then downed, and is then used in the rest of the interpretation of the query (which will fail).

Interpreting a `smalltalk term` that is parametrized by logic variables, is analogous. However, it is more complex because the bindings of the logic variables have to be passed to the Smalltalk expression. The problem is that these bindings are used in SOUL (and thus consist of logic terms), but that they are here used in a Smalltalk expression to be evaluated in Smalltalk. Therefore they have to be *upped* to get their Smalltalk representation. The interpretation of the following query illustrates this:

Query equals(?c, [Array]),
[?c selectors isEmpty]

The interpretation of the *equals* predicate unifies the logic variable *?c* with the `smalltalk term` *[Array]*. This adds a binding in the logic environment, indicating that the logic variable *?c* is bound to the logic term *[Array]*. Then the `smalltalk term` *[?c selectors isEmpty]* has to be interpreted. As before, this means we have to evaluate the Smalltalk expression it contains, and down the resulting object. However, to evaluate the Smalltalk code in the `smalltalk term` we have to lookup the binding for the logic variable *?c* in the logic environment. This gives us the logic term *[Array]*, a logic construct. Therefore we *up* it, and get its Smalltalk representation, the class *Array*. Then we can evaluate the Smalltalk expression *Array selectors isEmpty*, which yields the Smalltalk result *false*. Since the result of the SOUL evaluation of a Smalltalk term has to be a SOUL term, the Smalltalk result *false* is downed to get *[false]*.

So, in general this means that to interprete a `smalltalk term` *t* in a logic environment, we need to fetch the logic term for any logic variable used in the `smalltalk term`, and then *up* this logic term to get its Smalltalk representation. Then we have a complete Smalltalk expression, that we evaluate in Smalltalk. The result is a Smalltalk object, that is then *downed* to get a logic term. This logic term is the result of the evaluation of *t*.

Implementation details

The previous section explains in general how a `smalltalk term` is interpreted. In this section we want to say something more about the implementation of this mechanism in SOUL. Basically, a `smalltalk term` is converted internally into a Smalltalk blockclosure, that can be evaluated to get a result. Of course, since a `smalltalk term` has access to logic variables, an environment has to be passed to this block that contains the values to use. Therefore, when a `smalltalk term` is parsed, a *block* and an *environment* are constructed. The environment is simply an array that will be used to hold the values for the variables referenced in the smalltalk term. The block contains the Smalltalk code, but replaces every occurrence of a logic variable to a lookup in the environment. This value is *upped* to Smalltalk level. For example, for the `smalltalk term` that was used in the *simpleSelectors* rule, the following block is created:

[:env | ((env at: 1) soulUp) selectors]

Of course, when a variable is used multiple times within the block, it uses the same index in the environment. When the `smalltalk term` is interpreted, the environment containing the values for the referenced variables is passed. When a variable remains unbound, an error is shown and the

interpretation stops. For example, the following query will give a SOUL runtime error because the referenced variable *?c* is not bound when the `smalltalk term` is evaluated:

Query [?c selectors]

Note that because the `smalltalk term` contains regular Smalltalk code (extended with access to logic variables), this also means that Smalltalk runtime errors can occur while interpreting the `smalltalk term`. We just let these errors end the logic interpretation process, and a standard Smalltalk exception is thrown and results in a dialog box. Alternatively we also did experiments where the Smalltalk exception handler is used to catch the Smalltalk runtime errors, and just fails the predicate. However, in practice the latter poses problems for the SOUL developer because it makes it very hard to debug Smalltalk errors in `smalltalk terms`.

3.2.3 The generate predicate

The `smalltalk term` wraps Smalltalk expressions, and allows us to evaluate Smalltalk expressions during logic interpretation, wrapping the resulting object. This wrapping was illustrated in the previous section, where we get one result when we ask the selectors of the class *Array*. This one result is a wrapped Smalltalk collection containing the selectors of class *Array*. However, sometimes we want to get *separate* logic results for each selector. Such functionality is offered by the *generate predicate*, which generates a set of solutions (described by a `smalltalk term`) for a variable. The first argument of the generate predicate specifies a *logic variable* to bind the results to. The second argument is a smalltalk term that describes a *stream of solutions*. Each of these solutions is bound, one by one, to the first argument. As an example, we revise the *simpleSelectors* predicate to use the generate predicate, and turn it into a *simpleSelector* predicate. Therefore we replace the call to the *equals* predicate by a call to the *generate* predicate. We also change the `smalltalk term` to produce a stream with the selectors of the class:

Rule simpleSelector(?c, ?selector) **if**
    class(?c),
    generate(?selector, [?c selectors asStream]).

When the *generate* predicate is evaluated, it results in X solutions for the variable *?selectors*, where X is the number of elements in the stream. For example, the query to ask the selectors of class *Array* now yields 15 results. Each result is a binding for the variable *selector*, containing the name of a selector of *Array*. We only show the first four of these results:

?selector -> [#identityIndexOf:replaceWith:startingAt:stoppingAt:]
?selector -> [#decodeAsLiteralArray]
?selector -> [#refersToLiteral:]
?selector -> [#storeOn:]
...

3.2.4 The quoted string

The `smalltalk term` is used to wrap and use Smalltalk code (that can reference logic variables) in SOUL. When a `smalltalk term` is interpreted, its smalltalk source code is invoked (after all its logic variables where substituted). We would also like to be able to represent Smalltalk code as is, without it being evaluated. To support this functionality SOUL provides the *quoted string* language construct. A quoted string is used to denote strings that can contain logic variables. Note that these strings do not necessarily have to represent syntactically correct Smalltalk expressions.

For example, the following rule describes a simple HTML description of a class. Two quoted strings are used. The first one is in the head of the rule, and states that the html file for a certain class *?class* (the first argument) is a html file with some heading and followed by a list. The second quoted string is used to construct the list items containing the names of the methods of the class. The *findall* predicate accumulates these items as in Prolog. It takes three arguments (a term, a goal and a list) and finds the list of all the instances of the term for which the given goal is true. Note that the *list2String* predicate is responsible for collapsing the list with strings describing the items into one single string:

```
Rule classHtml(?class, { <html><body>
                         <h1>Methods of ?className</h1>
                         <ul>
                         ?selectorNameStrings
                         </ul>
                         </body></html>
                         }) if
     className(?class, ?className),
     findall(    {<li>?sel</li>},
                 classImplements(?class, ?sel),
                 ?ms),
     list2String(?ms, ?selectorNameStrings).
```

When extended, rules like *classHtml* could form the foundation for a documentation extracting system. The idea is to query the implementation for information, and export the results in html. When combined with the rules defined in the *declarative framework* in chapter 4, a powerful documentation system could be constructed. By using the *synchronization framework* the extracted documentation could even be kept in sync with the implementation.

3.2.5 Introspection and reflection in SOUL

Up until now we viewed SOUL as nothing more than a logic meta-programming language that has some extensions that allow it to reason about Smalltalk. However, since SOUL is implemented in Smalltalk, and since the `smalltalk term` can be used to reason about any Smalltalk base program, it can reason about the SOUL implementation itself. Hence, SOUL is *introspective* as defined in section 2.3.2. For example, the following query gets the names of the methods implemented by the class *SOULRule*, the SOUL class representing rules:

```
Query simpleSelector([SOULRule], ?selector)
```

However, beside introspection, SOUL also uses lightweight forms of reflection. The goal of this reflection was to be able to write second-order logic predicates from within SOUL. Therefore we

reify two concepts that are important during the evaluation of a logic term: the logic repository and the logic environment that holds on to the bindings. We chose to make these two concepts available in the `smalltalk term`, under the form of two hardcoded variables: *?repository* and *?bindings*. The *?repository* variable references the logic repository used when interpreting the `smalltalk term`. The *?bindings* variable holds the current set of bindings. This simple addition makes it possible for a `smalltalk term` to inspect and influence its interpretation. As an example we give the implementation of three widely used logic predicates: *assert*, *one* and *call*. The *assert* predicate adds a new logic clause to the current repository. The *one* predicate finds only the first solution of the term passed as argument. If this first solution is found, the bindings are updated and the predicate succeeds, otherwise the predicate fails. The *call* predicate is analogous to the *one* predicate, but does not keep the results. Hence it just needs to succeed when the argument term has at least one solution:

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

Rule one(?term) if
        [       | solution |
                solution := ( ?term resultStream: ?repository ) next.
                solution isNil
                        ifTrue: [false]
                        ifFalse: [ ?bindings addAll: solution. true ]
        ].

Rule call(?term) if
        [(?term resultStream: ?repository) next isNil not]
```

Speaking in reflection terminology, the two hardcoded variables *?repository* and *bindings* are a causally connected self-representation. Therefore the `smalltalk term` (and hence SOUL) can reason about and even alter a part of its implementation. Note however that the introspective and reflective capabilities of SOUL add nothing to the meta-programming abilities. As explained in section 3.2.2 a construction such as the `smalltalk term` is enough to get a logic meta-programming language. The additions explained in this section only allow us to implement some higher-order functionality from within SOUL itself. While this adds to the usability and expressive power of SOUL (particularly while experimenting), it is no conceptual addition to help reasoning about the base language.

3.2.6 The development tools

To complete the description of SOUL we want to show the tools that make up the SOUL development environment. They mainly allow launching queries, edit and compose logic repositories and view results. The basic tool for a SOUL developer is the *Repository Inspector* tool, shown in figure 3.3. This tool has different panes that allow us to launch queries (the *Queries* pane, which is not shown in the screenshot), to compose repositories (the *Configuration* pane) and to view and edit clauses (the *Clauses* pane). The *ClauseCopier* shown in figure 3.4 allows to move and copy clauses between different repositories. Last but not least, figure 3.5 shows an inspector on the result of a query. This inspector displays the number of results in its title bar, and can show different aspects of the results by selecting the appropriate entry in the list on the left (such as the query that was launched, the time

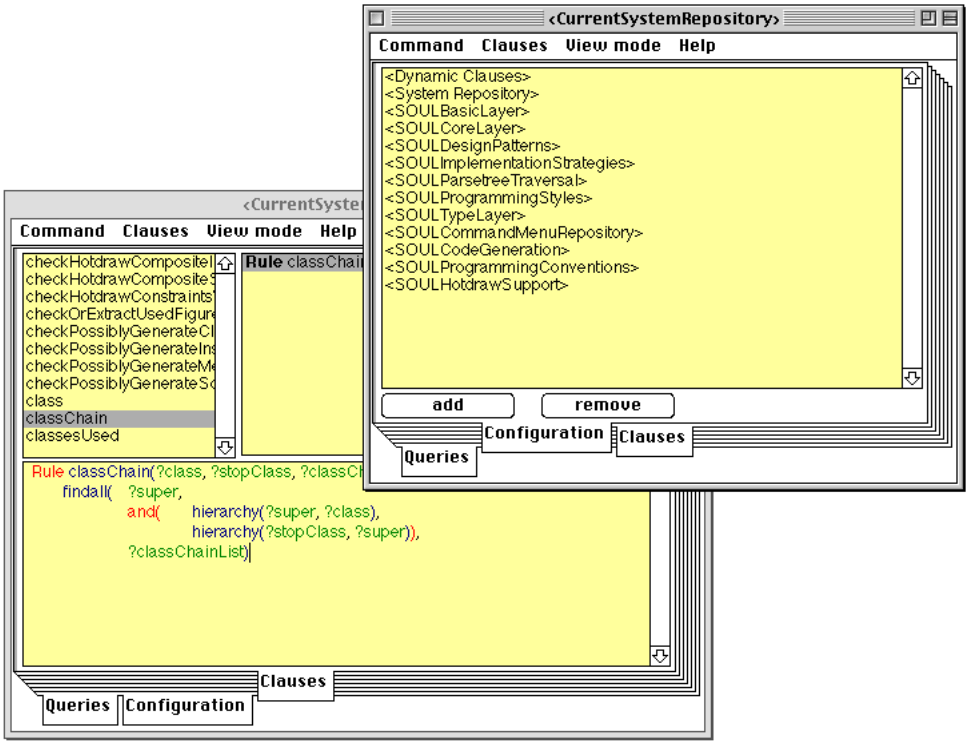

Figure 3.3: The repository inspectors

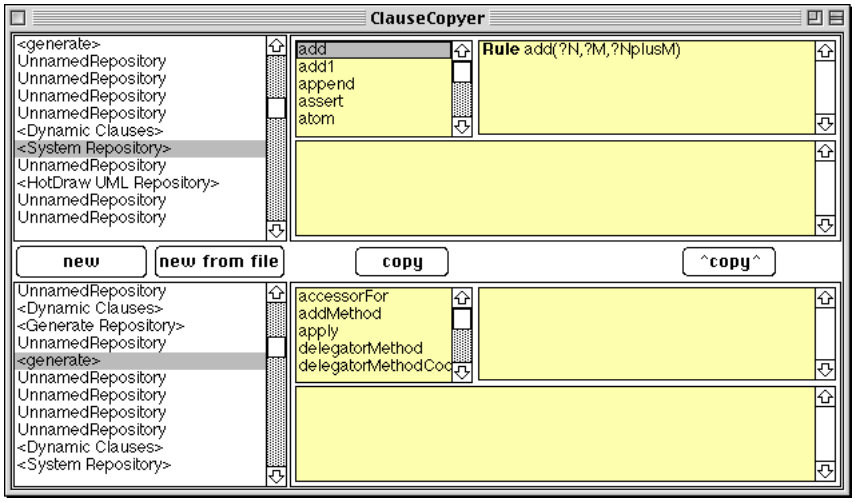

Figure 3.4: The clause copier

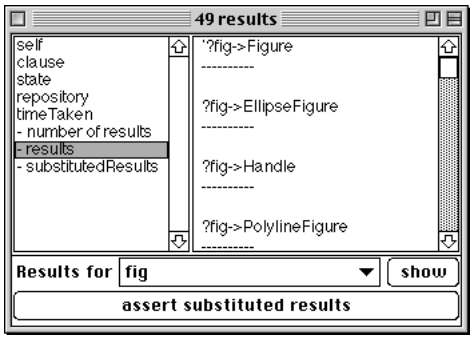

Figure 3.5: An inspector on the results of a query

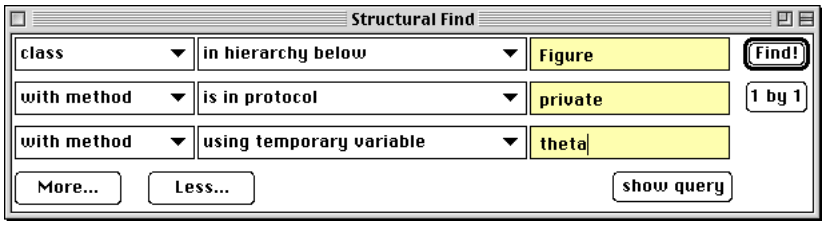

Figure 3.6: Structural Find Application

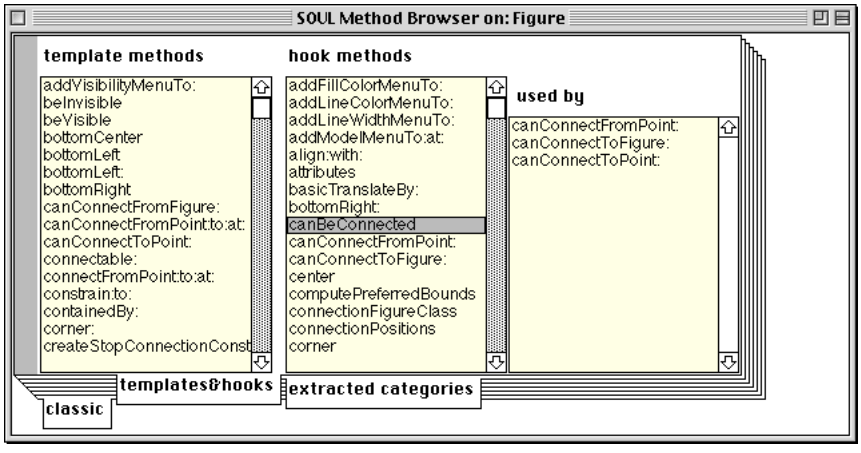

Figure 3.7: The XRay Browser showing the template and hook methods of class Figure

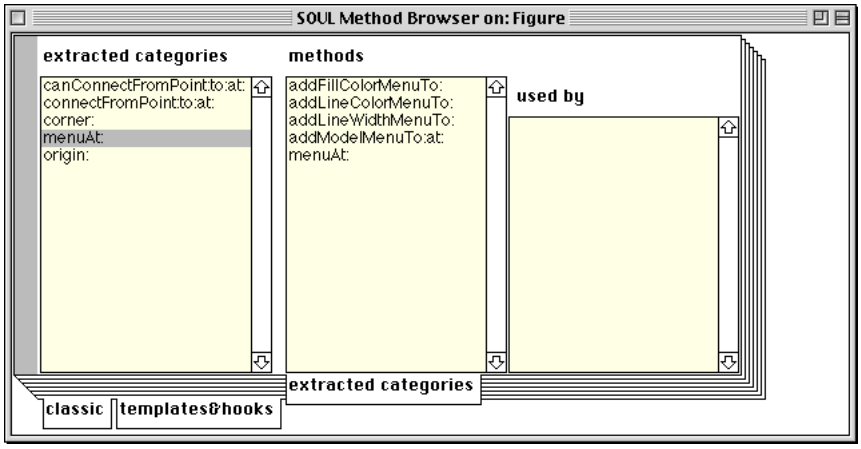

Figure 3.8: The XRay Browser showing the extracted categories of class Figure

it took to solve the query, the results itself, and the results substituted in the original query. By double clicking the entries, more detailed information can be obtained about each of these items.

We also show the tools for Smalltalk developers that use SOUL as a reasoning engine. These tools are actually very simple to build, since they are basically GUI shells that only have to display the information from queries. The first example is the *Structural Find* tool, shown in screenshot 3.6. It allows developers to find classes, methods or instance variables according to certain criteria that can be selected from drop-down boxes. Another example is the *X-Ray Browser*, that allows us to display the methods of a Smalltalk class according to three different categorizations: *classic*, *templates and hooks* and *extracted categories*. The *classic* classification is standard Smalltalk, where methods are grouped by hand in named groups (called *protocols*). The *templates and hooks* classification uses the messages to *self* (the receiver itself) to differentiate *template methods* (methods calling other methods through self-sends) from *hook methods* (methods that are being called in other methods of the class). Screenshot 3.7 shows the X-Ray browser opened on the HotDraw class *Figure*, showing the template and hook methods. The *extracted categories* classification goes a bit further. Each method that only has *one* sender in the class is assigned to a classification around that one method that calls it. Screenshot 3.8 shows the XRay-browser on the extracted categories of class *Figure*.

3.3 The incremental solver

During our experiments we show how SOUL is used to support different kinds of synchronisation. As seen in section 2.5.1, one possible classification of synchronisation differentiates between the time the synchronisation is triggered: *delayed* or *direct*. When the synchronisation occurs delayed, we use SOUL to launch a query to find the differences between design and implementation. However, this process is too costly to support direct synchronisation, since it starts from scratch for each change to design and implementation. Therefore SOUL includes an *incremental solver* that retains the results of previous queries and can directly propagate small changes to these results. In this section we describe this incremental solver. We start by discussing the local propagation techniques used in numeric constraint solvers, and then we see how we used this idea in our symbolic context.

3.3.1 Local propagation in numeric constraint solvers

The SOUL incremental solver uses techniques that are borrowed from *constraint programming* [JL87, Coh90], and more specifically from the *incremental constraint solving* area, where constraints are used in interactive graphical user interface building and where responsiveness and efficiency are primary concerns [FB89, FBMB90, BAFB96, San94, SMFBB93, BFB95, BB98]. Therefore, we first introduce some common constraint programming terminology that helps explain our approach. The classic meaning of a *constraint* is a relation between variables (the *constraint variables*) that should be maintained at all times [FBMB90, BAFB96, SMFBB93, BFB95, BB98]. Constraints are grouped in *constraint networks*, where every constraint variable has a *domain* associated with it. The domain contains the possible values for the constraint variable in the constraint network. A value is possible if it makes the constraint hold in the given constraint network. When a constraint network is constructed, it can be *solved*.

Solving a constraint network means finding the domain for each of the constraint variables, or failing if this is not possible. The core idea in *incremental* constraint solving is first to build a network (finding the domains for each constraint variable) and then keep on updating this network whenever a value in the domain of a variable changes. Updating of the network starts with the variable whose domain was affected, and proceeds recursively by updating the domains of variables that have a direct relation with the initial variable. As a result, a change in one domain is propagated to all other domains that need to be updated (and not more).

3.3.2 Local propagation in SOUL

Incremental constraint solvers seemed very good candidates to use as a foundation to support direct synchronization of design and implementation. However, using existing incremental constraint solvers proved impossible because the domains of these algorithms were almost always numerical. In our case we want the values in the domains to be *symbolic*, since we want to express relations between Smalltalk objects. Therefore we chose to build a simple incremental symbolic solver, using local propagation techniques analogous to those found in the incremental, numeric constraint solving community. In our approach, constraints are expressed as logic terms using SOUL. Hence we use SOUL as the language for describing the relations between variables. The domains of constraints are calculated using SOUL, meaning that the approach is in essence multi-way (whenever a multi-way predicate is used). However, because of limitations of our current implementation we only allow at most two variables at this moment. Supporting more than two variables was not necessary for this proof of concept, so we omitted it. However, we think it should not prove to be too difficult to implement when the need arises.

For example, suppose we want to find all subclasses of a class *Figure* that implement a method *initialize*. We can express this in the following query, that returns all the initialize methods in variable *?m*:

Query hierarchy([Figure], ?c)
classImplementsMethodNamed(?c, [#initialize], ?m)

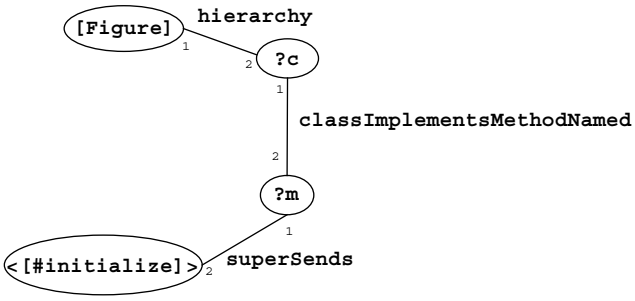

Figure 3.9: Incremental solving example

If we then want to find all of the above *initialize* methods that do a super send (to make sure that all the initialisation behaviour implemented in the superclasses is not forgotten), we have to run the following query:

Query hierarchy([Figure], ?c)
classImplementsMethodNamed(?c, [#initialize], ?m)
superSends(?m,<[#initialize]>)

This second query, however, first has to find all the *initialize* methods again, and then selects the ones doing the *initialize* supersend. In the incremental solver three stages will be used, where the results from the previous stage are used in subsequent stage. The network (after the three constraints have been added and solved) is depicted in figure 3.9. Ellipses represent constants and variables, and are linked by lines representing the relations between these variables. The lines are labelled with the predicate describing the relation, and the numbers that are used as roles near the end points give the index of the argument in that relation. The network is built by creating a new incremental solver, and adding the relations to it one by one. The solving process is explained in the following section.

```
SOULIncrementalSolver new
        name: 'Thesis Example';
        add: 'hierarchy([Figure], ?c)';
        add: 'classImplementsMethodNamed(?c, [#initialize], ?m)';
        add: 'superSends(?m, <[#initialize]>)'
```

3.3.3 The incremental solving process

In this section we explain the basic workings of our local propagation incremental solver. We use the (simple) network from the previous section as running example. This network consists of three separate relations that are added one after the other. Note that the order in which the relations are added makes no difference regarding the results of the network, although it may have an impact on the efficiency. For example, if we start this constraint network the other way round (starting with the *superSends* relation), the initial domain of the variable *?m* contains all the methods in Smalltalk that do a super send of *initialize*. The subsequent constraints then limit this domain to only *initialize* methods below class *Figure*. In the order we use here, we first limit the scope and then start finding the methods in this scope, which is more efficient.

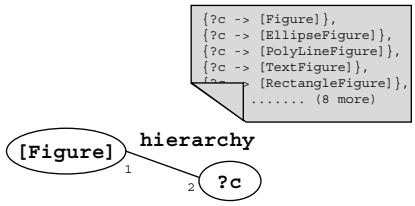

Figure 3.10: Solving the network, state 1

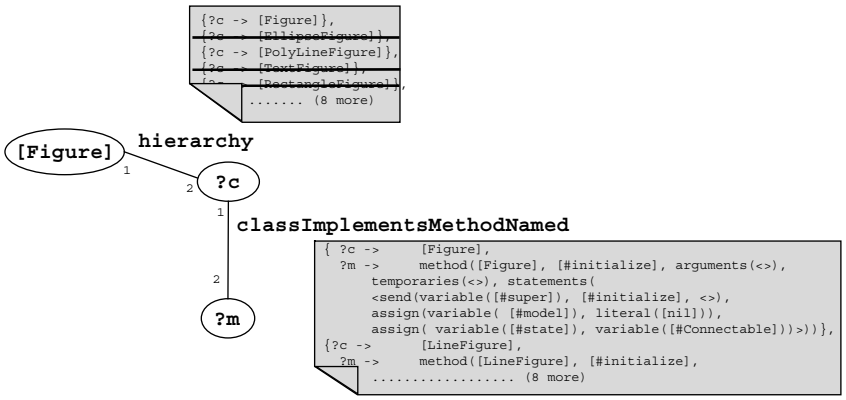

Figure 3.11: Solving the network, state 2

In the example we start with an empty network where we add the first relation that binds the variable *?c* to be a subclass of a class *Figure* using the *hierarchy* predicate. Since this is the first relation, there is no existing domain for the variable *?c*. Hence the incremental solver just uses SOUL to answer the query described by the constraint:

Query hierarchy([Figure], ?c)

The solver holds the results of the query together with the *hierarchy* relation. The state of the network is depicted in figure 3.10. Note that the variable itself does not hold on to its domain, but that the results are kept in the relation instead. When we ask for the domain of the variable, it is constructed by filtering the results from one of its relations.

We are then ready to add the second relation to our network. This is actually a relation between two variables: *?c* and *?m*. When we add this second relation, we already have the domain for *?c*, but *?m* is a new variable. So we have to evaluate the query constructed from the relation in the scope of the network, taking the domain of *?c* into account. Therefore we enumerate the values in the domain of *?c*, constructing a query where *?c* is replaced by one of its values and *?m* is calculated. The result of this process is a collection of bindings giving couples of values for *?c* and *?m* that are solutions to the second relation. In our example, we get 10 results: all the subclasses of *Figure* that implement an *initialize* method. This also means that we now have to update the domain of *?c* in all the other relations except the one we just added, since there is a chance that the domain of *?c* changed (as is the case in this example). Since we only have one other constraint that uses variable *?c* (our first relation), we then update its solutions to only have the 10 classes we found as result in the second constraint. If there would be other relations that use *?c*, or if the removal of some solutions of *?c* in another relation

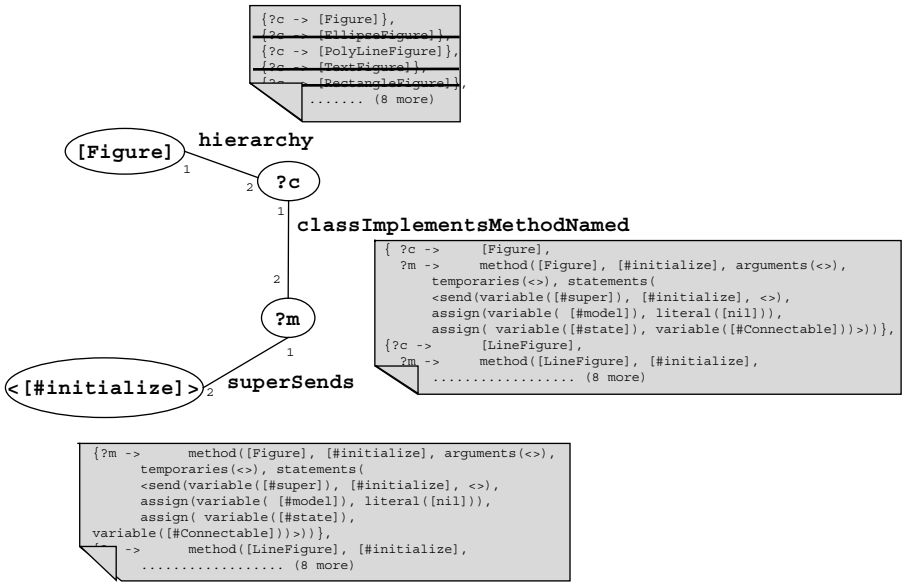

Figure 3.12: Solving the network, state 3

would change the domain of yet another constraint, then these changes would also propagate. This process stops when there are no more relations that have domains that change. Figure 3.11 depicts the state of the network after the two first relations were added.

Then we add the third relation that states that the *initialize* method has to use a *super send*. This relation introduces no new variables, and thus serves only as an extra constraint that the methods in the domain of *?m* have to satisfy. We thus evaluate the query constructed from the relation for each solution of *?m*. Since this succeeds for every method (they all do super sends), there is no change of the domain, and there is no need to propagate any changes. The state of the network after the third relation has been added is depicted in picture 3.12.

The algorithm we implemented handles cycles in the network. When traversing the graph defined by the constraint network, a stack is kept so that we can determine whether we have already visited some node or not. If we reach a node that was already updated, we check to see if it is consistent with the removals we need to do. If it is, then we proceed with the rest of the network (if this is necessary) or we stop successfully. If there is a conflict, then the constraint network is inconsistent due to the addition of the latest constraint. We then stop with an error, and perform a rollback that brings the network in the state it was in before we added the last constraint.

3.3.4 Limitations of the current implementation

The current implementation of the incremental solver has some limitations. First of all, the relations can only have at most two logic variables. While the solver itself is multi-way, the limitation lies in the way we have implemented the detection of relation violations. Second, we have not yet added support to remove relations from a network. Since individual solutions can be removed from constraints (and are then propagated), there is no problem to add such a functionality. Thirdly, there is no rollback of the side-effects done by constraints. For example, if a constraint is triggered, and as a result removes a method, then this method is not restored when the change is rolled back in the constraint network. The net effect of this is that care has to be taken with automatic changes to the base implementation

by the constraints themselves. These side-effects should be done after change propagation, as part of an action phase where the user takes some action indicated by the constraints. The action could be automatic (such as the generation or removal of code) or manual (the user using the development tools to make changes to the implementation).

3.4 Conclusion

In this section we introduced the logic meta-programming language SOUL, which allows us to reason about Smalltalk code. We discussed the main features that make SOUL a logic meta-programming language: the `smalltalk term`, the *generate predicate* and the *quoted string*. Then we introduced the introspective and reflective capabilities of SOUL, obtained by adding two special hardcoded variables to the `smalltalk term` (*?repository* and *?bindings*). Then we showed the development tools that are available to SOUL developers to interact with SOUL, and the tools for the Smalltalk developers that we built using SOUL. Last but not least we discussed the *incremental solver* that was built on top of SOUL.

The symbiosis between SOUL and Smalltalk is one of the technical contributions of this dissertation. SOUL introduces the *up-down* mechanism between two languages of a different paradigm. This allows Smalltalk objects to be used as logic terms, and eases the implementation of the reflection. From the user's point of view, this makes it very easy to use SOUL to reason about the base language and to implement extensions of SOUL.

Using local propagation techniques to implement a symbolic incremental solver is another technical contribution of this dissertation. The SOUL incremental solver allows to build a network of logic relations that share logic variables. The solver is then capable of keeping the results to these relations consistent when solutions to these relations change (because relations are changed or under external influences).

In the next chapter we use SOUL to implement the *declarative framework*, a layered library of predicates to reason about Smalltalk.

Chapter 4

The declarative framework

This chapter introduces the *declarative framework*, a layered set of rules to facilitate reasoning over an implementation. We introduce the four layers that currently make up this framework: the *logic layer*, the *representational layer*, the *basic layer* and the *design layer*. In particular the latter is of interest for this dissertation. We express three design notations in this layer, namely *programming conventions*, *design pattern structures* and *UML class diagrams*. These three design notations are thus expressed in terms of the implementation, and are used in subsequent chapters where we validate of our claim.

4.1 Introduction

The previous chapter introduced the basic language features and tools of our logic meta-programming language. In this chapter we use these language features to implement predicates that reason about the implementation of a base system. These predicates are structured in what we call the *declarative framework*, depicted in figure 4.1. The declarative framework is a layered rulebase, where predicates in one layer have access to all the predicates from lower layers. Each layer contains groups of rules with similar or related functionality:

- the *logic layer*: this layer contains the predicates that add core logic-programming functionality, such as *list handling*, *arithmetic*, *program control*, *repository handling*, ... It is at the top since it used by all other layers;
- the *representational layer*: this layer reifies the base-language's concepts, such as *classes*, *methods*, *instance variables* and *inheritance*;
- the *basic layer*: this layer adds a lot of auxiliary predicates that facilitate reasoning about implementation. Since the *representational layer* only provides the most primitive information, this layer is absolutely necessary to interact on a reasonable level of abstraction with the logic meta-programming language;
- the *design layer*: this layer groups all predicates that express particular design notations. In the next chapter we describe some design notations that we have expressed to experiment with, namely the *programming conventions*, *design patterns* and *UML class diagram*.

Since every layer contains many predicates, we have logically divided them into groups. Table 4.1 lists these groups within the layers, and gives the most important predicates for each group. We will now study each of the layers in more detail, as this gives more information about the structure and contents of the declarative framework.

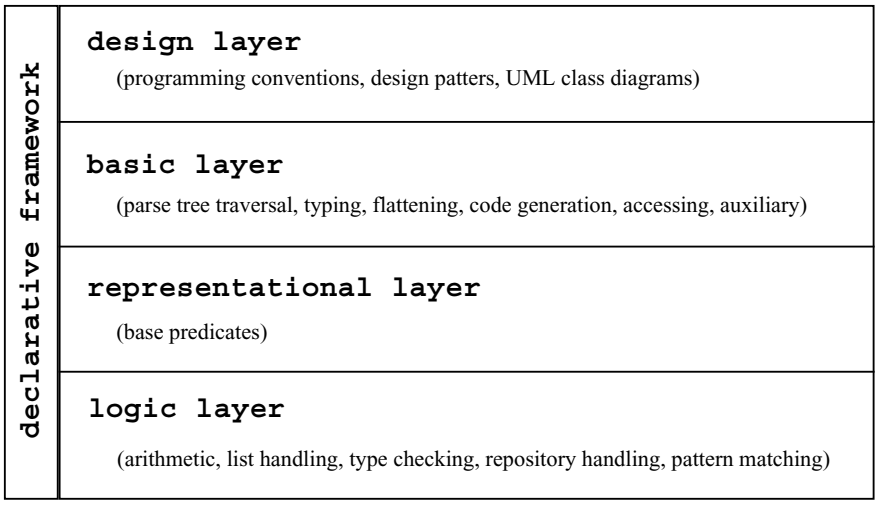

Figure 4.1: The declarative framework

layer	**group**	**major predicates**
logic layer	arithmetic	*add*, *sub*, *greaterThan*, *smallerThan*
	list handling	*append*, *length*, *member*, *flatten*, *head*, *tail*
	type checking	*var*, *atom*, *ground*
	repository handling	*assert*, *retract*
	pattern matching	*patternMatch*, *stringSplit*
representational layer	base predicates	*class*, *method*, *instVar*, *superclass*
basic layer	parse tree traversal	*isSendTo*, *assignmentStatements*, *classesUsed*, *globalsUsed*, *returnStatements*
	typing	*instVarTypes*, *collectionElementType*
	flattening	*classChain*, *implementationChain*, *flattenedMethod*
	code generation	*generateClass*, *removeClass*, *generateMethod*, *removeMethod*
	accessing	*methodName*, *methodClass*, *methodStatements*
	auxiliary	*rootClass*, *hierarchy*, *understands*, *abstractClass*
design layer	programming conventions	*accessor*, *mutator*, *badSupersend*
	design patters	*compositePattern*, *visitor*, *abstractFactory*, *factoryMethod*, *singleton*, *bridge*
	UML class diagrams	*umlDiagram*, *umlClassifier*, *umlGeneralization*, *umlAssociation*

Table 4.1: Overview of the different groups of predicates in each layer, and the most important predicates for each group

4.2 The logic layer

The first layer we encounter is the *logic layer*. As can be expected from the name, this layer contains predicates that implement basic logic functionality. Table 4.1 lists the most important predicates in this layer. Note that the layer is subdivided into several groups of predicates with related functionality. In section 3.2.5 we already saw some examples of some of these logic predicates that use the reflective capabilities of SOUL. In this section we therefore give some other examples, starting with two well-known *list handling* predicates (*head* is used to unify the first argument with the head of the list, while *intersection* defines the intersection of two lists) and a *type check* predicate (*ground*, that only succeeds if the argument does not contain variables):

```
Fact head(?first,<?first | ?rest>).

Rule intersection(?list1, ?list2, ?intersection) if
        findall(    ?common,
                    and( member(?common, ?list1),
                         member(?common, ?list2)),
                    ?intersection).

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

Before proceeding with the other layers, we want to show the implementation of the *length* predicate, that is used to relate the length of a logic list and a logic list. So, when the list is known, it is sufficient to count the elements. This is implemented by using a *generate predicate* to determine the length of a given (bound) list. However, if the length is known but the list is unbound, then we have to construct a list containing logic variables of the specified length. This is again done using a generate predicate, but this time it constructs a SOUL list of the known length, containing logic variables called *var* followed by their index. These two rules make sure the *length* predicate can be used multi-way: regardless the input (known values or variables for the *?list* or *?length* arguments), a useful result or check will be given:

```
Rule length(?list, ?length) if
        nonvar(?list),
        generate(   ?length,
                    [?list soulSize asStream]).

Rule length(?list, ?length) if
        var(?list),
        atom(?length),
        generate(   ?list,
                    [((1 to: ?length) collect: [:idx |
                         SOULVariableTerm name: ('var', idx printString)]) asSoulList asStream])
```

predicate	reifier
class(?class)	classes
superclass(?super,?sub)	inheritance relationship
instVar(?class,?iv)	instance variables in a class
method (?class,?m)	methods of a class

Table 4.2: SOUL Representational Predicates

statement	syntax example	translation
literal	1	literal([1])
variable	t	variable([#t])
assignment	x := y	assign(x,y)
return	↑x	return(x)
message send	x msgpart1: arg1 msgpart2: arg2	send(x, msgpart1:msgpart2:, <arg1,arg2>)
cascaded message	x msg1: arg1; msg2: arg2	send(x,msg1:,<arg1>), send(x,msg2:,<arg2>)
block	[:arg1 :arg2 | |t1 t2| ...]	block(arguments(<arg1,arg2>), temporaries(<t1,t2>), statements(<...>))

Table 4.3: SOUL Logic Representation of Smalltalk Statements

4.3 The representational layer

The representational layer reifies concepts of the base language in order to reason about them. Since SOUL is a logic programming language, we represent the reified Smalltalk concepts in a logic format. This section describes the mapping that we use to map Smalltalk parse trees to a logic format, and then discuss the predicates implementing this mapping (table 4.2 shows these predicates). We will not discuss the *instVar* and *superclass* predicates because their implementation are analogous to the implementation of the *class* predicate.

4.3.1 Representing base programs

The mapping we use to represent Smalltalk programs in the declarative framework is fairly straightforward, and follows the Smalltalk parse tree structure. Actually, there is no parse tree for classes (classes just contain methods). We represent a Smalltalk method as a functor with five arguments: the class, its name, the names of the arguments, the names of the temporary variables and the statements. The mapping of Smalltalk statements is given in table 4.3. As a concrete example figure 4.2 shows the code of the *printOn:* method on class *SOULList* and its SOUL. We use this method as an example since its parse tree is not trivial, yet neither is it overly complicated.

4.3.2 The *class* predicate

We start with the general implementation of the *class* predicate. This predicate allows us to check if the passed argument is a class, or to generate all classes. Note that the smalltalk predicate uses the

```
SOULList>>#printOn: aStream
        "add a textual representation of the receiver to the stream"

        self isEmptyList
                        ifTrue: [aStream nextPutAll: '<>']
                        ifFalse: [aStream
                                nextPut: $<;
                                print: self term;
                                nextPut: $>]


method( [SOULList],
        [#printOn:],
        arguments(<[#aStream] >),
        temporaries(<>),
        statements(<    send(  send(variable(    [#self]),
                                                 [#isEmptyList],
                                                 <>),
                                        [#ifTrue:ifFalse:],
                                        <    block(arguments(<>),
                                                temporaries(<>),
                                                statements(<    send(variable([#aStream]),
                                                                        [#nextPutAll:],
                                                                        <literal('<>')>)>)),
                                             block(arguments(<>),
                                                temporaries(<>),
                                                statements(<    send(variable([#aStream]),
                                                                        [#nextPut:],
                                                                        <literal([$<])>),
                                                                send(variable([#aStream]),
                                                                        [#print:],
                                                                        <send( variable([#self]),
                                                                                [#term],
                                                                                <>)>),
                                                                send(variable([#aStream]),
                                                                        [#nextPut:],
                                                                        <literal([$>])>)>)>)>)>))
```

Figure 4.2: The Smalltalk implementation of printOn: *for class* SOULList*, and its logic representation in SOUL*

class *SOULExplicitMLI* as a facade to facilitate and centralise the calls to Smalltalk[1].

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

This predicate uses the *generate* predicate to generate all classes in the system. One by one it binds these values as result for the *?c* variable. This definition ensures that the predicate can be used regardless whether a constant or a variable is passed. For example we can then perform a query asking for all the classes in the system:

```
Query class(?c)
```

However, using the multi-way directionality of logic programming, we can also solve the following query asking whether the argument class *Array* is indeed a class:

```
Query class([Array])
```

Note that we actually pass an *Array* class here, using the Smalltalk term and the fact that classes are first-class objects in Smalltalk.

4.3.3 The *method* predicate

Analogous to the *class* predicate, we define a general *method* predicate that relates classes and methods in the base program. The class *SOULExplicitMLI* is again used as a facade to allow easy access to the Smalltalk meta system, and is now asked to return all the methods:

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

Once this predicate is added (together with the *class* predicate defined above), we can reason about classes and methods in the object-oriented system. This allows us to perform different queries, such as asking whether a specific method is indeed a method of some class, finding all the methods of a class, and so on.

[1] The implementation of the facade *SOULExplicitMLI* actually uses a singleton design pattern [GHJV94], which explains the *current* message that is sent to the *SOULExplicitMLI* class to retrieve the actual facade instance used. This instance is then asked for all the classes in Smalltalk using the *allClasses* message.

4.3.4 Optimising the representational predicates

The current implementation of the *class* and *method* predicates are very declarative and simple. While this is clean, it also penalises the performance when used in some circumstances. Suppose for example that the logic representation of a method is given, and that we want to know the class this method belongs to. Since the *method* predicate relates classes and methods, we can use it to find this out. The following query is used to retrieve the class of a particular method (the logic representation of method *isConstant* of class *SOULAbstractTerm*):

```
Query method( ?c,
              method( [SOULAbstractTerm],
                      [#isConstant],
                      arguments(<>),
                      temporaries(<>),
                      statements(<return(literal([false]))>))
```

However, to solve this query we run into performance problems because of the purely declarative and simplistic implementation of the *method* predicate introduced before. This implementation first generates a list of all classes. For each of these classes, every method is generated. All these methods are then traversed and matched one by one to the method we gave as an argument. It is clear that this is overkill, especially since our method representation includes the class the method belongs to as first argument! Because of these performance reasons, the representational predicates are therefore written less declaratively but more efficiently. Note that we take care not to compromise the multi-way aspect of these predicates, so that from the outside it is transparent whether they are written in purely declarative or in optimised form.

Optimizing the class predicate

As a first example, let us revisit the implementation of the *class* predicate. We write the predicate using two rules instead of one. The first rule is used when the argument passed is a variable. The second rule is used when the argument given is a constant. The real performance gain, compared to the previous implementation, lies in this second predicate. If a constant is given, we do not need to get all classes and then find one that matches. Instead we use a `smalltalk term` to find out whether this constant is a class. This results in the following two rules for the *class* predicate, one for each case (the predicates *var* and *atom* are used to check whether the passed argument is a variable or a constant, respectively). This definition of the *class* predicate is the actual working implementation that is used in the SOUL system:

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

Rule class(?class) if
        atom(?class),
        [SOULExplicitMLI current isClass: ?class].
```

Optimizing the method predicate

The optimisation of the method predicate is a different story. Where the representation of a class is an atom (a `smalltalk term` representing the class), the representation of a method is a composite structure (a logic functor with 5 arguments, as seen in section 4.3.1). As we saw in the query in the beginning of this section, the user can choose for each of the elements in the structure to use a variable or a constant. The variables need to be bound to possible values, while the constants need to be checked to see if they are valid. For performance issues we want to take the given information into account when finding out correct values for the variables. Another example of a very inefficient usage of the *method* predicate is the following query that finds all methods named *isConstant*, without arguments or temporaries. Note that no information about the statements is specified (a variable is passed):

```
Query method( ?c,
              method( ?c,
                      [#isConstant],
                      arguments(<>),
                      temporaries(<>),
                      statements(?statements ))
```

When run, the query yields the following result:

```
?c          ->  [SOULAbstractTerm]
?statements ->  <return(literal([false]))>
```

This query shows that the obvious approach to optimising the *method* predicate (which is to check whether the passed method is a variable or not, as we did in the *class* predicate), would not work. Since the method passed can be a compound structure, possibly containing variables, such a check is not enough. The trick is to use the multi-way property of logic programming to extract as much information as possible from the functor containing the method description. This information is then used to parse and compare the required methods only. The result is a more refined implementation than the crude *method* predicate given above, but one that is still declarative:

```
Rule method(?class, ?method) if
01      methodClass(?method, ?class),
02      class(?class),
03      methodName(?method, ?name),
04      classImplements(?class, ?name),
05      calculateParseTree(?class, ?name, ?generatedMethod),
06      equals(?generatedMethod, ?method).
```

As can be seen, line `01` first calls the *methodClass* predicate to relate the *?method* variable with its *?class*. Depending on what was passed as *?method* this can have different consequences:

- if *?method* is a variable, then it is bound to an empty method template:

  ```
  method(?class,?sel,arguments(?args),temporaries(?ts),statements(?statements)).
  ```

  Note that the first argument in this template (the *?class* variable) is the same as the *?class* variable in the template;

- if *?method* is a complete method template, then *?class* is bound to the constant giving its class;

- if *?method* is a partially filled in method template, then the variable *?class* is bound to the variable or constant in the first position of that template;

- if *?method* is anything else, the *methodClass*, and hence the *method* predicate, fails.

The important thing to notice is that after the call to the predicate *methodClass*, the *?method* variable is certainly bound to a method template, and the *?class* variable points to the first element in that template. Line `02` then states that *?class* should be a class. This means that if *?class* was bound to a constant, it is checked to make sure that it is a class. However, if *?class* was still a variable, it is bound to all possible classes in the system, and also the method templates are filled in accordingly. Lines `03` and `04` do the same trick, further filling in or checking the method template with the names of the method (the selector). Note that the *classImplements* predicate used in line `04` relates classes and names of methods. When line `05` is reached, we have information regarding the classes and the methods, and we have sufficient information to calculate the parse tree. This is stored in the variable *?generatedMethod*. The *equals* predicate is then used in line `06` to unify the variable *?method* (containing a possible partially filled in method template or variable) with the parse tree in the *?generatedMethod* variable. This last step fills in the remaining variables from *?method*.

This optimised implementation of the *method* predicate always uses as much information as possible to minimise the parsing of methods. For example, to find the class of a given method (the first example at the beginning of this method), only one method in one class is parsed, whereas the previous implementation parsed every method of every class in the system.

4.4 The basic layer

The *logic layer* and the *representational layer* provide all the basic mechanisms to use SOUL to reason about Smalltalk code. However, the level of abstraction is not very high, and for almost every query to reason about the implementation we should have to write lots of logic code. Therefore we factored out a lot of functionality and created the *basic layer*. This layer adds a lot of auxiliary predicates that facilitate reasoning about implementation, and raises the level of abstraction significantly. Describing the implementation of all these predicates falls outside the scope of the dissertation (there are currently over 140 predicates in this layer). Therefore we describe the predicates in groups, and then give some examples on how to use them:

- *parse tree traversal*: a lot of predicates have to traverse the parse tree of a method in search for certain variables or message sends. Therefore we have implemented the *traverseMethodParseTree* predicate to travel over the logic parse tree of a method. Using this predicate we have implemented some traversals that are commonly used (such as *isSendTo*, *assignmentStatements*, *classesUsed*, *globalsUsed* and *returnStatements*);

element	predicate
class	generateClass(?className, ?superclass) removeClass(?class)
method	generateMethodInProtocol(?parseTree, ?protocol) generateMethodInProtocol(?quotedTerm, ?class, ?protocol) generateMethod(?parseTree) generateMethod(?quotedTerm, ?class) cpgMethodInProtocol(?parseTree, ?protocol) cpgMethodInProtocol(?quotedTerm, ?class, ?protocol) cpgMethod(?parseTree) cpgMethod(?quotedTerm, ?class) removeMethod(?class, ?selector) removeMethod(?parseTree)
instance variable	generateInstVar(?instVarName, ?class) removeInstVar(?instVarName, ?class)

Table 4.4: The code generation predicates

- *typing*: Smalltalk is a dynamically typed object-oriented programming language. Therefore we added some predicates that analyse the source code in order to find possible types for variables. We use a lightweight type inferencing scheme that basically tries to detect all messages that are sent to a certain variable (the interface), and then looks for all the classes that understand the complete interface. This gives an indication of the type of the variable. Of course specific programming conventions or refined type inferencing rules can complement these rules. This is accomplished by adding rules to the framework as we describe in the next chapter when we discuss *programming conventions*;

- *flattening*: the basic rules representing classes and methods are *incremental*, meaning that the information about a class is only what that class implements. Information from the hierarchy is not taken into account. For example, when we use the *method* predicate to get the methods of a class we only get the methods that are really implemented by that class (and not all methods it understands). The predicates in this group allow us to reason about classes in their flattened versions, taking inheritance into account;

- *code generation*: since smalltalk predicates can contain any Smalltalk code, this code can use the standard Smalltalk meta facilities to generate or remove code. The rules in this group use this to offer predicates that generate code from logic descriptions of methods or from quoted strings;

- *auxiliary*: there is also a number of auxiliary methods (such as *rootClass*, *hierarchy*, *understands*, *abstractClass*) that implement various useful predicates that are frequently used.

4.4.1 The code generation predicates

An important category of predicates are the *generation predicates*, listed in table 4.4. In this section we discuss the predicates for generating methods. The implementation of these predicates relies on the combination of the smalltalk predicate on one hand, and on the other hand on the fact that Smalltalk itself is reflective [FJ89]. Note that we chose not to discuss the implementation of the predicates

to generate classes and instance variables, as these implementations are similar to those of the class generating predicates.

The basic predicate is *generateMethodInProtocol*. It allows us to generate a method in a certain protocol[2]. The implementation of the method to generate can come in two forms: a logic description or a quoted string. Note that in the case of a quoted string, a class needs to be supplied as well (the logic representation contains the class of the method as first argument in its representation). When a logic description is supplied, the predicate checks to see if it does not contain any logic variables. If it does not contain any variables, the *methodSource* predicate is used to convert the logic description into a source string. If it contains unbound variables, the predicate fails and nothing is generated. When a quoted string is supplied, it is directly compiled and stored. Note that, when the source string contains invalid Smalltalk code, the Smalltalk compiler produces no code, and *nil* is returned. This causes the predicate to fail, as expected:

```
Rule generateMethodInProtocol(?methodParseTree, ?protocol) if
        atom(?protocol),
        methodClass(?methodParseTree, ?class),
        existingClass(?class),
        methodSource(?methodParseTree, ?source),
        generateMethodInProtocol(?source, ?class, ?protocol).

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

In most cases when we want to generate a method, we do not explicitly want to specify the protocol to be used. Therefore we implemented a *generateMethod* predicate, that is implemented in terms of *generateMethodInProtocol*. It simply uses an auxiliary predicate *protocolForSelector* that looks to see if the class where the method is generated has a superclass that already implements it. If so, the method is generated in the same protocol. If the method is new, then a default protocol is used.

The *generateMethod* and *generateMethodInProtocol* predicates always generates a method, regardless whether it already exists or not. This is not always convenient. For example, we might only want to generate a method if it does not yet exist. If it already exists, we do not want to change it[3]. Since this is frequently used in practice, we offer a set of predicates that are prefixed with *cpg* (which is short for *checkPossiblyGenerate*). The *cpg*-versions of predicate only generate source code if there is no source code artefact present yet.

4.4.2 Examples using the basic layer predicates

Having enumerated the main groups of predicates of the basic layer, this section gives two examples that use some of the basic layer predicates. The first example expresses the programming convention that an *initialize* method should always do a super send first, and then finds violations against this

[2]In Smalltalk every method belongs to exactly one protocol. Therefore, when generating, the protocol that is used for a method has to be supplied. We see a little further that we provide predicates where this protocol is extracted from the implementation.

[3]Note that other schemes are possible, in different gradations. For example, when a method already exist, we could make sure that its implementation is the same as the one we want to generate.

convention. The second example expresses what an interface of a class is, and then uses this to find classes that conform to a certain interface. Then we show how this can be used to generate template methods that are missing, given a class and an interface.

Initialize method should do a super send

First of all we express the programming convention that was also expressed in CoffeeStrainer (in the related work in section 2.5.2). This convention expresses that whenever a class overrides a method *initialize* it should first call *super initialize* before doing anything else. As an example we write a query that checks this convention for every subclass of a class *Figure*. We can get all the subclasses from *Figure* using the *hierarchy* predicate. Then we use the *overrides* predicate to check which of those classes override a method called *initialize*. For each of these classes we then get the implementation of the overridden *initialize* method using the *classImplementsMethodNamed* predicate. We then have the parse tree of the method for which we have to check that its first statement is *super initialize*. We check this using the *methodStatements* predicate, immediately indicating that the first statement has to be *send(variable[#super]), [#initialize], <>)*. The tail of the list (after the vertical bar) can be anything, so we use an underscore variable:

Query
```
hierarchy([Figure], ?c),
overrides(?c, [#initialize]),
classImplementsMethodNamed(?c, [#initialize], ?m),
methodStatements(?m, <send(variable([#super]), [#initialize], <>) | _>)
```

The following example writes a predicate that expresses the interface (messages send to) an instance variable of a class. It uses a *findall* predicate to collect all sends to the variable in a list. The sends to the variable are found by the *isSendTo* predicate, one of the parse tree traversal predicates. The interface is this set of methods, but without any duplicates:

Rule
```
varInterface(?class, ?var, ?interface) if
    instvar(?class, ?var),
    findall(        ?varSend,
                    isSendTo(?class, _, ?var, ?varSend),
                    ?varSendsList),
    noDups(?varSendsList, ?interface).
```

Using the *varInterface* rule we can then write a rule to find all sends to an instance variable that are not understood by some other class. This allows us to check whether some class we see as a possible type for the instance variable could indeed be used as such.

The implementation simply gets the interface of the instance variable, and extracts all selectors from it that are not understood by the type (that has to be a class):

```
Rule interfaceDifferences(?class, ?var, ?varType, ?missingSelectors) if
        class(?varType),
        varInterface(?class, ?var, ?interface),
        findall(        ?missingSelector,
                        and(  member(?newSelector, ?interface),
                              not(understands(?varType, ?missingSelector))),
                        ?missingSelectors).
```

For example, using this predicate we can check whether *Number* is a possible type for the instance variable *x* of class *Point*:

```
Query       interfaceDifferences([Point], [#x], [Number], ?missing)
```

As could be expected, the query succeeds and returns as only possible value for the *?missing* variable the empty list. This means that *Number* is a possible type for the instance variable *x*, at least by looking at the messages sent to *x*. If *Number* would not have been correct, then *?missing* would contain a list of selectors that *Number* should understood in order to be usable as type. While it is practical to know which selectors sent to our instance variable are not understood by the class we passed as possible type, we can use this information for more. For example, we can use the code generation predicates to generate template methods on the class we pass as type. Therefore we simply enumerate all the selectors from the list we get from the *interfaceDifferences* predicate using the *forall* predicate. For each selector we generate a method with template code (the *notYetImplementedSource* gives a quoted string with Smalltalk source code for a given selector):

```
Rule adjustClass(?class, ?var, ?type) if
        interfaceDifferences(?class, ?var, ?type, ?missingSelectors),
        forall(member(?sel, ?missingSelectors),
              and(  notYetImplementedSource(?sel, ?code),
                    generateMethod(?type, ?code))).
```

4.5 The design layer

The last layer we discuss is the *design layer*. In this layer we have grouped the support for three design notations we support: *programming conventions*, *design pattern structures* and *UML class diagrams*. Because the predicates expressing these design notations only use the other layers (and each other), design is expressed as a logic meta program over implementation. Indeed, all the predicates are, sooner or later, expressed in function of the *representation layer*.

4.5.1 Programming conventions

First of all we are interested in expressing *programming conventions*. We use the term *programming convention* as a common term to represent all kinds of conventions and styles, such as *idioms* [Cop98],

```
var
    "direct accessor for an instance variable var"

    ↑var

var
    "most common lazy initialized accessor for an instance variable var"

    ↑var isNil
        ifTrue: [var := 0]
        ifFalse: [var]

var
    "other lazy initialized accessor form for an instance variable var"

    var isNil ifTrue: [var := 0].
    ↑var
```

Figure 4.3: Common implementations for accessor methods for an instance variable var

best practice patterns [Bec97], *naming conventions*, ... In this section we express the structure of *accessing methods*, give some examples, and extend the typing predicates to use this information.

Accessing methods

One of the ways to make data and operations more transparent is by hiding every access to data by a message send. This is the motivation behind the concept and usage of *accessing methods*. An *accessing method* is a method that is responsible for getting or setting the value of an instance variable. By consequently using accessing methods (and never accessing instance variables directly), tThe caller side never knows whether it is calling an accessing method (and thus manipulating data) or just performing a message send that calculates something. This system therefore makes it easy for subclasses to override these accessing methods to modify all kinds of data definitions. Note that, for this system to work, *all* accessing of instance variables should be done using the accessing methods. Even one violation can result in a bug when an internal representation is changed.

There are two kinds of accessing methods:

- *accessor methods* are unary methods that are used to get the value of an instance variable;
- *mutator methods* are methods that take one argument, and set the value of an instance variable to the value of that argument.

In these examples we only discuss the support we have implemented for *accessor methods*, since the support for *mutator methods* is analogous.

Accessor methods can be implemented in numerous ways. Three common implementations are given in figure 4.3. The first one is the straightforward implementation that just returns the instance variable. The two other ones use *lazy initialization*. The rationale behind this is that an instance variable does not need a value unless it is actually used. Therefore lazy initialization is built into

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

Rule accessorForm(?method, ?var, [#lazyClassic]) if
        methodStatements( ?method,
                          <return(send(      ?nilCheck,
                                             [#ifTrue:ifFalse: ],
                                             <?trueBlock,?falseBlock>))>),
        nilCheckStatement(?nilCheck,?var),
        blockStatements(?trueBlock,<assign(?var,?varinit)>),
        blockStatements(?falseBlock,<?var>).

Rule accessorForm(?method, ?var, [#lazyAlternative]) if
        methodStatements( ?method,
                          <send(       ?nilCheck,
                                       [#ifTrue:],
                                       <?trueBlock>),
                          return(?var)>),
        nilCheckStatement(?nilCheck,?var),
        blockStatements(?trueBlock,<assign(?var,?varinit) >).

Rule accessorForm(?method, ?var) if
        accessorForm(?method, ?var, _).

Rule accessorForm(?method) if
        accessorForm(?method, _).
```

Figure 4.4: The accessorform rules, making the implementations of the accessor methods shown in figure 4.3 explicit.

the accessing method. This form of accessing first checks to see whether the variable was already initialized or not (by checking whether the value of the instance variable is *nil*). If the variable was not yet initialized (its value is nil), then it is initialized and returned. If it was already initialized (its value is not nil), then the value is returned. Two possible implementations are given for this scheme in the implementation.

The three forms we show in figure 4.3 are made explicit in three rules shown in figure 4.4 (complemented by two rules that can be used when not all the arguments are known or needed). The first rule expresses the simplest form of an accessor, describing it as a method with just one return statement that returns an instance variable. The two following rules make the other implementations explicit. Now we can easily write a predicate to relate a class, an instance variable and an accessor method to each other:

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

This *accessor* predicate states that *class* has an instance variable with name *varName*, and a method with the same name as *varName* and implementation conforming to *accessorForm*. This means that we have codified a Smalltalk naming convention that states that accessing methods typically have the name of the instance variable they are accessing. Of course other naming conventions could be used, for example the more C++ or Java-like one to prefix the name of the method with *get*.

Now that we have described what an accessing method looks like in a logic meta program, we can write queries that check the source code for violations of this rule. Methods that violate the encapsulation imposed by the accessor methods programming convention are methods that directly send messages to instance variables (of course accessor methods themselves are excluded, because they are the only ones allowed to do this). We can write a rule for such violations, that checks for every method implemented in a class whether that method sends messages that have as receiver an instance variable:

```
Rule accessingViolator(?c, ?m, ?iv) if
        class(?c),
        instvar(?c, ?iv),
        method(?c, ?m),
        not(accessor(?c, ?m, ?iv)),
        isSendTo(  variable(?iv),
                   ?violatingMessage,
                   ?args)
```

We can then invoke a query to find violations:

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

Another violation is to assign values directly to instance variables (which should be done by calling the mutator methods). To find such methods, we ask every non-mutator method for its assignment statements, and then we check whether it includes an assignment statement with an instance variable as left-hand side. This means that there are direct assignments to this instance variable.

```
Rule accessingViolator(?c, ?m, ?iv) if
        class(?c),
        instvar(?c, ?iv),
        method(?c, ?m),
        not(mutator(?c, ?m, ?iv)),
        assignmentStatements(?m, ?assignmentsList),
        member(assign(?iv, ?violatingAssignment), ?assignmentsList)
```

Complementing the typing rules

In the *typing predicates* we saw in section 4.4, typing instance variable was done by looking at the sends to the instance variable, and using this information to determine the possible types.

Using the *instVarTypes* predicate we can then directly get the possible types for an instance variable, for example the variable *x* on class *Point*:

Query instVarTypes([Point], [#x], ?possibleTypeList)

Of course, when all accesses to instance variables are done through *accessing methods*, then this mechanism does not work anymore. The solution is rather simple: we have to take the sends to accessor methods into account when determining the messages sent to an instance variable. So, in order to complement the typing rules, we add a rule *instVarTypes* that expresses this information. When the design layer is then used, there are two rules that can give solutions when we have a query asking for types of instance variables: one using direct sends to instance variables, the other one using the sends to accessors.

Generating accessor methods

Previously we looked for accessor methods in the implementation, and for violations against the *always use accessor methods* programming convention. However, sometimes we also want to generate *accessor* methods for an instance variable in a class. Combining the *accessorForm* predicate describing the *simple* accessor form in combination with the *generate predicates* from the basic layer makes this easy to do:

```
Rule generateAccessor(?class, ?instvar, ?accessorMethod) if
    instvar(?class, ?instvar),
    accessorForm(?accessorMethod, ?instvar, [#simple]),
    methodClass(?accessorMethod, ?class),
    cpgMethod(?accessorMethod).
```

We want to notice that with the current form of the *accessorForm* predicates, it is not possible to generate any other form then the *simple* accessor method. The problem is that the source code of the methods as given by the *accessorForm* predicates is not complete. It is actually only a partial description that is matched against the source code, where some parts remain unspecified. For example, when we look at the *lazyClassic* accessor form given in figure 4.4, the code that gets assigned to the instance variable is left unspecified. Hence we cannot use it to generate fully functional code, but have to limit ourselves to generate template code where pieces have to be filled in manually by the developer later on. We could also make the *accessorForm* predicates more specific (for example by passing the initialization code as an optional argument):

```
Rule accessorForm(?method, ?var, [#lazyAlternative], ?varinit) if
    methodStatements( ?method,
                      <send(    ?nilCheck,
                                [#ifTrue:],
                                <?trueBlock>),
                      return(?var)>),
    nilCheckStatement(?nilCheck,?var),
    blockStatements(?trueBlock,<assign(?var,?varinit) >).
```

That way they can be used both for searching and for generating.

name	predicate
composite	compositePattern(?component, ?composite, ?method)
visitor	visitor(?visitor, ?element, ?accept, ?visitSelector)
abstract factory	abstractFactory(?class, ?element)
factory method	factoryMethod(?class, ?method, ?element)
singleton	singleton(?class)
bridge	bridge(?left, ?right)

Table 4.5: The design pattern predicates in the design layer

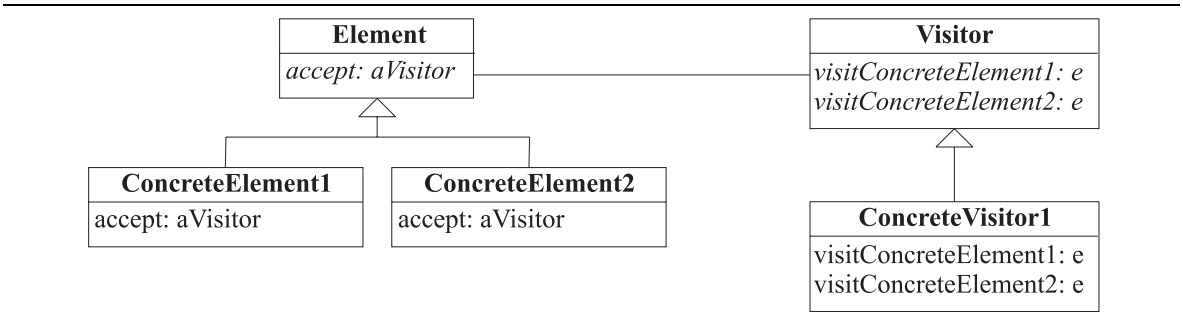

Figure 4.5: Visitor Design Pattern Structure

4.5.2 Design pattern structures

The second design notation we support is structures as described by design patterns [GHJV94]. We want to stress at this point that the intent of these rules is *not* to capture the *complete design patterns*, but only the part expressing their structure. A design pattern contains far more information than only structure that is not captured by these predicates (such as a *motivation*, *intent*, *applicability*, and relationships with other design patterns). Therefore we always refer to these rules as being *design pattern structure* rules.

In general, a design pattern is detectable if its template solution is both distinctive and unambiguous [Bro96]. We express the structural information in the template solutions by writing logic meta programs. This is possible since these logic meta programs have access to the full parse trees of the object-oriented system they are reasoning about. Table 4.5 lists the design pattern structures we expressed. In this section we give the implementation of the *visitor* design pattern.

The general idea of the Visitor design pattern is to separate the *structure* of elements from the *operations* that can be applied on these elements. This separation makes it easier and cost-effective to add new operations, because the classes of the object structure do not have to be changed. As depicted in figure 4.5 there is a hierarchy describing the elements, and there is a separate hierarchy implementing the operations. Call *Element* the root class of a hierarchy on which the class *Visitor* and its subclasses define operations. Every *Element* class defines a method *accept*, that takes a *Visitor* as argument and calls this visitor using an operation that indicates its type. For example, the implementation of *accept* on class *ConcreteElement1* will send the message *visitConcreteElement1* to the visitor. The *Visitor* hierarchy consist of the classes that define operations on the *Element* classes. They just need to implement the calls made by the element classes. The typical example of the visitor design pattern is to separate parse trees from the operations that are typically performed on these parse trees (such as generating code, pretty printing or optimizations).

The rule describing the structure of the visitor design pattern is fairly straightforward. It expresses first of all that the *visitor* is a class, and that it implements the *visit* methods (that have a name *visitSelector*). In the same way, *element* is a class too, and implements methods called *accept* with a body *acceptBody*. The arguments passed to this method are given by *acceptArgs*. The body is responsible for calling the passed visitor *v* with the actual visit operation *visitSelector* and passing along the arguments *visitArgs*. One of the arguments has to be the receiver (denoted by *self* in Smalltalk), and the passed visitor *v* actually has to be an argument of the accept method:

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

For example, since SOUL is implemented in Smalltalk (and uses a visitor pattern to enumerate its parse tree) we can use the *visitor* predicate to find all the non-abstract parse tree elements of the SOUL parse tree that do not comply to the visitor pattern. To do so, we select all subclasses of class *SOULParseTreeElement* that are not abstract, and for each of those we find the ones that do not comply to the *visitor* rule:

```
Rule soulParsetreeVisitor(?node) if
        hierarchy([SOULParseTreeElement], ?node),
        not(abstractClass(?node)),
        not(visitor(?visitor, ?node, [#doNode:], ?callbackMsg))
```

The last line in this rule gives the name of the visit-method used by the visitor to visit the nodes. It is a Smalltalk Symbol with the name of the method, *doNode:*[4]. The results of this query contain the methods that do not comply to the SOUL visitor design pattern, and that might need to be changed. If the query fails, then all the classes and methods comply to the visitor design pattern.

4.5.3 UML class diagrams

The *UML class diagram* predicates express the basic concepts of UML class diagrams [BRJ97, RJB99]: *classifiers* (with *operations* and *attributes*) and the *generalization* and *association* relationships. Table 4.6 lists the logic representations we use for the different UML concepts we support. We then have to write predicates to express these concepts in terms of the implementation. Table 4.7 lists those predicates. As with the other layers we again describe one of the predicates in detail.

In this case we take the most complicated one, namely *mapUMLAssociation*. This predicate is used to map UML associations against the source code. Because Smalltalk is dynamically typed,

[4]When we do not know this, we could have supplied a variable. The system would then have deduced the name used in this specific visitor pattern instance. However, in this example we wanted to express the current situation and see if the implementation conformed to this structure.

UML concept	logic representation
classifier	classifier([#class], ?name, ?attributeNames, ?operationNames)
generalization	relation([#generalization], ?classifierName1, ?classifierName2)
association	relation([#association], ?classifierName1, ?roles1, ?classifierName2, ?roles2)
role	role(?name, multiplicity(?mul), type(?type))

Table 4.6: The logic representation of UML concepts. Note that the attributeNames*,* operationNames *and* roles *variables are lists.*

UML concept	predicate
classifier	mapUMLClass(?classifier, ?class)
generalization	mapUMLGeneralization(?generalizationRelation, ?superClass, ?subClass)
association	mapUMLAssociation(?associationRelation, ?leftClass, ?rightClass)

Table 4.7: The predicates mapping the logic representations of UML concepts shown in table 4.6 to the implementation.

extracting and checking collaborations between classes is hard. However, we can use the *typing predicates* to extract possible associations. The core of mapping the UML association relation to the implementation is the *associationRelation* predicate, that types the instance variables of the left class (using the *instvarTypes* and *stripHierarchyClasses* predicates) and uses that information to see if there is an association with the right class. This is done by taking the instance variables of the left class and, for each of them, determining their type. If the type is not a Smalltalk collection, then the multiplicity is set to 1. If the type is found to be some Smalltalk collection class, then the multiplicity is set to *many*, and the type of the elements contained in the collection is determined (with the *collectionElementType* predicate). For each possible type we then construct a *role* functor with the extracted information (type and multiplicity). Since the *?allRoles* then contains possible nested lists, we flatten the results before returning them:

```
Rule associationRelation(?leftClass, ?instvar, ?leftRoles) if
        instVarTypes(?leftClass, ?instvar, ?typeList),
        stripHierarchyClasses(?typeList, ?possibleTypes),
        findall( ?roles,
                and(member(?possibleType, ?possibleTypes),
                    or( and(containerType(?possibleType),
                            collectionElementType(?leftClass, ?instvar, ?types),
                            stripHierarchyClasses(?types, ?strippedTypes),
                            findall( role(?instvar, multiplicity([#many]), type(?possibleType, ?type)),
                                    member(?type, ?strippedTypes),
                                    ?roles)),
                        and(not(containerType(?possibleType)),
                            equals(?roles, <role(?instvar, multiplicity([1]), type(?possibleType))>)))),
                ?allRoles),
        flatten(?allRoles, ?leftRoles).
```

4.6 Instantiating and reusing the framework

Now that we have discussed the *declarative framework* a natural question that arises is *how it can be instantiated and reused*. Before we answer this question, we want to note that the lookup of predicates in a logic programming language is *flat* (where it is *hierarchical* in object-oriented systems, taking inheritance into account). This means that, despite the layering in the *storage* of the predicates, they are seen as one flat pool of predicates at runtime (during the logic interpretation). In order to discuss the reuse of the framework, let's see how we can *extend*, *refine* and *remove* predicates.

Extension of the framework is easy: any layer can add predicates to be used at runtime. We gave an example of this with the *instVarTypes* predicate: it is defined in the *basic layer*, but we complemented it in the *design layer* in section 4.5.1. Hence, when both layers are used at runtime, all the rules defined in the predicates are taken into account. This can even be done at runtime using the *assert* predicate.

Refinement of rules (in a object-oriented sense) is currently very hard. What we would actually like is a mechanism to implement a rule in one layer, and *refine* it in another (which is done by *overriding* methods in a object-oriented system). The important aspect is that the original implementation does not need to be changed, and that the new implementation can still reference the old one (doing a *super* send in object-oriented programming languages). In SOUL, refining a rule can be done in two ways. The first way to do this is by changing the original implementation, or by removing it from one layer and reimplementing it in the other. The second way is by *swapping repositories*. Since in SOUL the logic repositories can be nested, we can swap the repository containing the original implementation of the predicate with a repository containing the refined implementation. While this is clean and supported by the tools (see the *Configuration* pane in the *Repository Inspector* in the SOUL development tools from section 3.2.6), this means that not one, but a number of predicates are swapped. Hence this is more appropriate to accommodate large changes in functionality. None of these ways allow us to invoke the previous implementation (other then copying its implementation). To recapitulate, *refinement* is currently not very well supported in SOUL[5].

Removal of rules is not very hard. Individual rules can easily be removed (even at runtime using the *retract* predicate). Also, complete logic repositories can be removed using the SOUL development tools.

So, in short, the *declarative framework* allows us to easily *extend* and *remove* individual predicates, and lets you *refine* predicates (but in a harder way).

4.7 Lessons learned

Constructing the *declarative framework* gave us insight in how to write and structure logic meta programs expressing information about the implementation. While this complete chapter tries to convey the feeling for the framework, and especially its expressivity, we now want to explicitly enumerate some of the key lessons we learned in general before concluding this chapter.

4.7.1 Guidelines for writing logic meta programs

First of all we collected a number of general guidelines that are useful when expressing design as a logic meta programs:

[5] In the future work in section 8.3.2 we discuss a mechanism that we are implementing that allows *delegation* in repositories, making *refinement* of rules easy.

1. the logic meta programs are first of all *source code*. This means that they have to be written in a disciplined and clean fashion, avoiding code duplication, just like any other implementation;
2. take care that every predicate is multi-way usable. When writing purely declarative, this is normally the case (see for example the first example of the *class* predicate in section 4.3.2). However, the common way of optimizing a logic meta program is typically by only writing it for an argument of a certain type (a constant, a variable, or even a specific class or a method). This means that it is not multi-way anymore, since the the caller has to know and anticipate this. The lesson to learn is clear: when optimizing a predicate, also implement the other cases as was done with the optimized *class* predicate in section 4.3.4;
3. group related predicates in individual repositories. Since repositories can easily be manipulated (added, swapped and removed), this makes it easy to change the structure of the framework;
4. SOUL only allows us to express predicates reasoning about the structure (parse tree) of a program. This means that behavioural information is hard or impossible to express directly. However, typically this can be extracted from the programming conventions used. For example, it is very hard to write a rule to find all methods that have to do with *printing textual representations*. However, in Smalltalk such methods typically belong to a *protocol* with as name *printing*. Another example was given when we expressed *accessor methods* that typically have the name of the instance variable they are accessing. Using such programming conventions allows us to -indirectly- support some behavioural information. While this chapter did not give much information about this, the experiments and validation will give more explicit examples;

4.7.2 Causal connection

Second we want to comment on the fact that classes in SOUL are causally connected to classes in Smalltalk, while this is not the case for methods. The representational layer introduces a *class* and a *method* predicates. While at first hand they do not seem very different, there is a difference regarding the causal connection. The results of the *class* predicate are `smalltalk terms` wrapping the actual Smalltalk classes (that are thus causally connected, since it are the classes themselves). The results of the *method predicate* are logic representations that are decoupled from the actual smalltalk compiled methods. For example, the following query gets the source code of the method *size* of class *Set* (in logic form), then removes the method in Smalltalk, and then asks the name of the method:

Query classImplementsMethodNamed([Set], [#size], ?m),
removeMethod([Set], [#size]),
methodName(?m, ?name).

This is only possible because the logic representation that is kept in variable *?m* is decoupled from the Smalltalk method. Note that this poses no problems for our experiments, as a logic programming language is used functionally, in a *read/apply* fashion. First a method body is asked, then it is manipulated and then - if necessary - it is written to the implementation again using the *code generation* predicates. In the middle of the manipulation, the logic representation of the method can be syntactically incorrect for Smalltalk. However, this is no problem because we only want it to be syntactically correct when it is generated.

However, the different treatment of classes and methods regarding their causal connection with the source code could be worth investigating. It could be one of the results of the further reflection between SOUL and Smalltalk that we discuss in the future work in section 8.3.2.

4.8 Conclusion

This chapter discusses the *declarative framework*, a layered set of rules to express design as a logic meta program over implementation. Closest to the implementation we find the *representational layer* that reifies the basic concepts of the base language we want to make explicit in the logic meta-programming language. The other layers build on this layer to implement ever higher abstractions of the implementation. We discussed the *basic layer* that -amongst other- has predicates for generating source code, and the design layer. The latter implements *programming conventions*, *design pattern structures* and *UML class diagrams*. Because the *declarative framework* is a framework, we also discussed how it can be instantiated to be used in specific circumstances.

Throughout the chapter we have included the implementations of predicates to give a concrete feeling of the expressivity of using a logic meta-programming language to express design. We also gave several examples of how the predicates can be used to reason about the implementation on a high-level of abstraction. For example, we extracted all classes in the system, looked for methods called *initialize* that forgot to do a super send, checked whether two classes are substitutable for each other, generated accessor methods, and looked for participants of a visitor design pattern.

In the next chapters we discuss how the declarative framework is integrated in the development environment using the *synchronization tool framework*. The combination of both frameworks is our synchronization framework that we use for validation.

Chapter 5

The synchronization framework

In this dissertation we want a framework to synchronize changes between design and implementation. Conceptually, our solution rests on three cornerstones:

1. express design as a logic meta program of implementation, and hence provide a mapping from design to implementation;
2. use the logic meta-programming language as the synchronization engine to detect differences between design and implementation;
3. integrate in the development environment so that changes to design or implementation can be intercepted, and acted upon;

We have already discussed two of these cornerstones: the logic meta-programming language and the *declarative framework*. Now we want to discuss the *synchronization tool framework*, the framework that is responsible for integrating synchronization tools in a development environment. We then call the *synchronization framework* the combination of the *declarative framework* and the *synchronization tool framework*, and show how it can be instantiated for every characterization of synchronization as discussed in section 2.5.1. The following chapters then perform more practical experiments to show the usability and scalability.

5.1 The synchronization tool framework

The *synchronization tool framework* is an application framework to build tools that need synchronization of design and implementation. It consists of the following parts, that are depicted in figure 5.1

1. design repository: this is a logic repository that sends *changed messages* whenever a clause is added, removed or changed. These changed messages are intercepted by the *design change monitor*;
2. application model: in the Smalltalk environment we use (VisualWorks Smalltalk), the class *ApplicationModel* is the general root class that is subclassed to build applications. We have created a subclass (*SOULToolApplicationModel*) that provides the core implementation for applications that need synchronization. All the SOUL applications are subclasses from this class. For users of the framework, this is the class they will certainly use to build their applications. It can be configured for several kinds of synchronization;

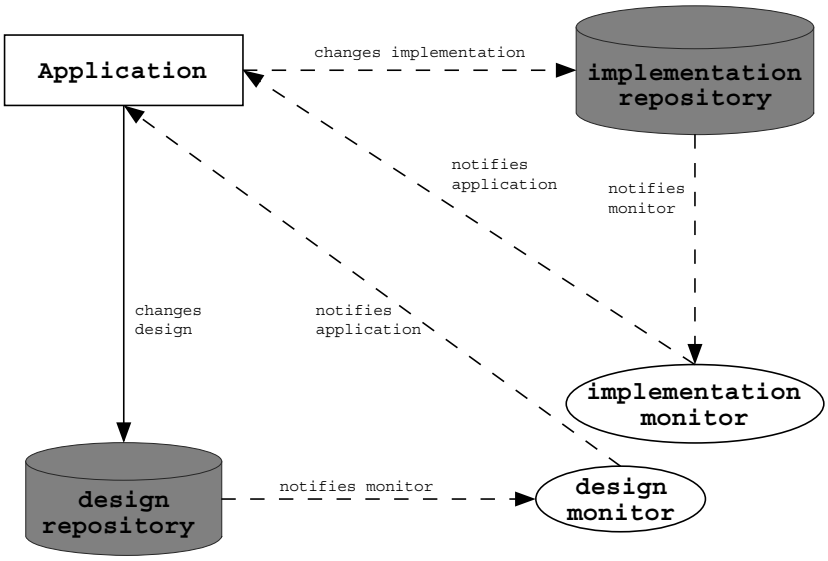

Figure 5.1: The major elements of the synchronization tool framework, and their dependencies. Dashed arrows indicate indirect dependencies using a registration mechanism and event system. Plain arrows indicate direct references.

3. implementation monitor: other classes can register to this monitor to receive notifications of changes to the implementation (after the changes were performed). It is very important to note that the monitor is implemented at the Smalltalk meta level, which means that *any* change, regardless of the tool, is captured. Hence all possible tools are supported, existing ones as well as new ones, and they do not need to be modified. Note also that, in the current version, we do not support proactive notifications. The implementation is analogous to the implementation of the retroactive notifications, so we foresee no trouble implementing this;

4. design monitor: other classes can register to this monitor to receive notifications of changes to the design (before or after the changes have been performed). The monitors receive their notifications from the design repositories.

Note that the application has a direct reference to the design repository is uses, but that all other dependencies are indirect, and work using the Smalltalk *dependency mechanism*. This allows any objects to register themselves to receive notifications from *models* (objects that send change messages to their dependants). Important of this kind of reference is that it is a dynamic, runtime model that uses a well-defined API.

For the user of the framework, the instantiation of the framework almost always includes creating a subclass of *SOULToolApplicationModel*. By default this class instantiates a new empty design repository and registers itself with both monitors. It thus receives notifications for both design and implementation changes. Every change results in a method being called, that is by default implemented to do nothing. Table 5.1 lists what messages are called as a result from what change in design or implementation. By overriding these methods, actions on changes can immediately be taken, as is explained in the next section. *SOULToolsApplicationModel* also implements methods to begin and stop receiving notifications from design or implementation.

kind	change	message
design	adding a clause	clauseAdded:
	removing a clause	clauseRemoved:
	changing a clause	clauseChanged:
implementation	adding a class	classAdded:
	removing a class	classRemoved:
	changing a class	classChanged:
	adding a method	selectorAdded:class:
	removing a method	selectorRemoved:class:
	changing a method	selectorChanged:class:

Table 5.1: The notifications that can be sent by the design and the implementation monitor, and the corresponding method called in SOULToolApplicationModel.

Note that the synchronization tool framework does not necessarily has to be used with the declarative framework. It merely allows tools to be notified of implementation or design changes. For example, it could also be used with *Smalltalk Lint* [RBJO96] (which we discussed in related work in section 2.5.2) or other tools. However, we chose to combine it with the declarative framework since we were interested in a very expressive and powerful reasoning mechanism.

5.2 The synchronization framework

The synchronization framework, shown in figure 5.2, is the combination of the declarative framework and the synchronization tool framework. It allows to build tools that support co-evolution, and that can be customized towards particular forms of synchronization as described by the characterizations of implementation. The general setup is depicted in figure. This shows an application that needs synchronization of design and implementation. Using the declarative framework to pose logic queries, the application can extract design information from the implementation, generate parts of the implementation or do conformance checks between design and implementation. Whenever this triggers a change in design or implementation, any application can receive notifications of these changes. If it chooses to do so, the application can then act on these changes (again by using queries).

In the following sections we discuss two concrete tools that are built using the synchronization framework: a *style checker* tool that constantly reports violations against programming conventions, and a *UML* editor that is kept synchronized with the implementation. Then we discuss how users of the synchronization framework can instantiate the framework to obtain specific forms of synchronization. This also forms the conceptual proof that the framework indeed supports all characterizations of synchronization discussed in section 2.5.1.

5.2.1 Style checker

One tool that we have implemented (shown in figure 5.3) monitors the quality of methods in the system. Therefore it fires user-definable queries whenever a method is changed in the system. These queries express criteria methods should comply to. Failures to adhere to these criteria results in a warning being written to the log application. The entries in the log application can then be double-clicked to open a browser on the method causing the warning. Entries in the to-do log are overridden when the same method is changed (and thus contain the results of the latest version of the method),

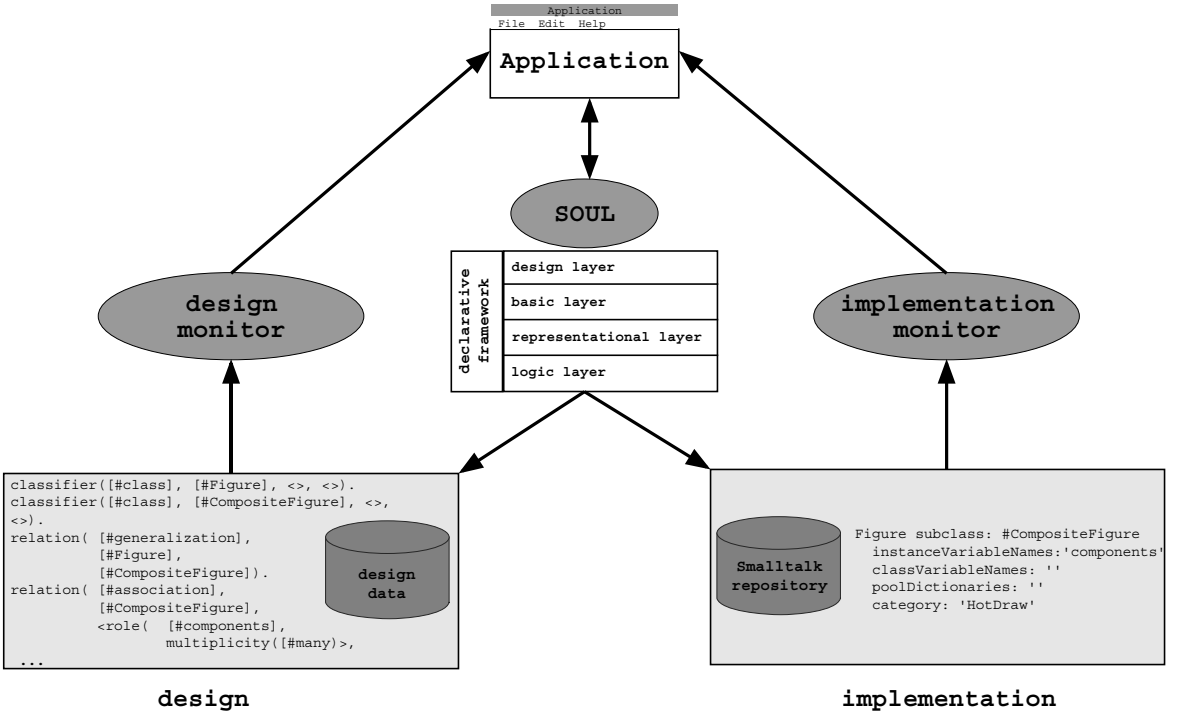

Figure 5.2: The general setup of the synchronization framework*, showing the two main constituents: the* declarative framework *and the* synchronization tool framework*.*

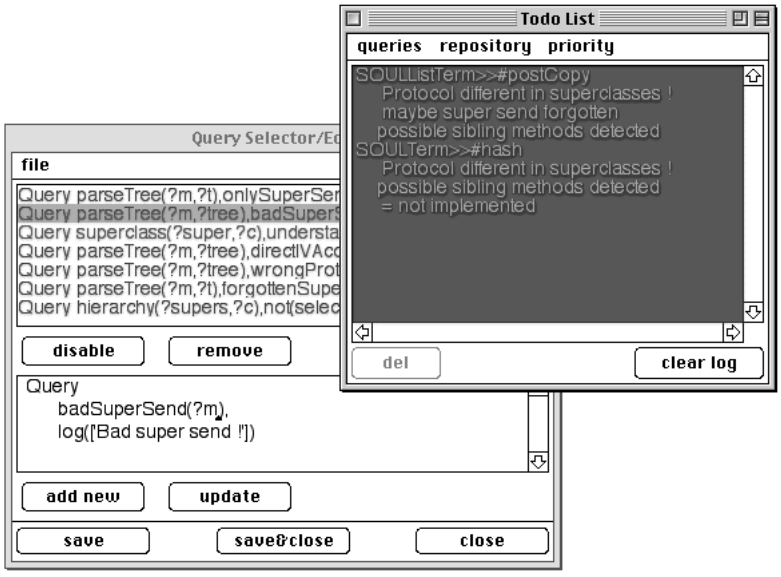

Figure 5.3: The Style checker application in action. The screenshot shows the to-do log containing some warnings the developer needs to look at, and the application where checks can be enabled, disabled, added, removed and saved.

or can be removed manually. The developer is thus not hindered: violations only result in logs to be reviewed.

To use the synchronization framework to build this tool, we have to instantiate it. This comes down to instantiating both frameworks: the declarative framework and the synchronization tool framework. To instantiate the declarative framework, we merely have to select which parts of the declarative framework we use, and add extra predicates that might be needed. This depends of course on the criteria we want to check. For example, in figure 5.3 we have highlighted a check that looks for *bad super sends* in methods. Therefore it uses a predicate *badSuperSend*, and logs a violation with a descriptive string if this predicate succeeds. The second step is the instantiation of the synchronization tool framework. For this we subclass *SOULApplicationModel*, and provide two user interfaces: one for the log where the violations are shown, and one to view and edit the queries that are checked on every method. Then we override the methods *methodAdded:* and *methodChanged:* to invoke the method *checkMethodCriteria:* whenever they are executed. The method *checkMethodCriteria* invokes the queries wanted by the user, and updates the log showing the violations (if there are any). Since this application is not interested in receiving notifications of changes in the design, we also override the *initialize* method to indicate this.

When the application is active, it receives notifications of any change in methods in the application, and performs checks on these methods. The basic form of this application as described above can be extended in several different ways. For example, by using dialog boxes instead of a logging strategy we can make it harder for users to violate the criteria. When we have an appropriate monitor (that sends changes before the changes were actually applied to the implementation[1]) we could even forbid to make changes that do not follow the conventions. Another application might offer rewrite abilities for certain violations. For example, when we want to enforce that people use *accessor methods* (see also section 4.5.1), and a direct send to an instance variable is detected, this can automatically be rewritten to use the accessor.

5.2.2 UML tool

UML is a general-purpose visual modelling language that is used to specify, visualize, construct, and document the artefacts of a software system [RJB99]. One of the diagrams offered is the *class diagram*, a static view that is used to model concepts in the application domain as well as concepts local to the implementation of an application. It mainly consists of classes (with attributes and operations) and their relationships. Several tools exist to draw UML class diagrams. Most of these tools allow us to generate template code from a UML diagram (generally by using a customizable scripting system). Some of them also allow us to extract UML diagrams from the source code (although typically not customizable).

When instantiating the synchronization framework to build a UML class diagram tool, most work is done on the user interface side. The subclass of *SOULToolApplicationModel* first of all has to implement the drawing editor to view and edit UML class diagrams. Extracting or generating a UML class diagram from the implementation is done by evaluating a query as is shown in section 6.3.1 in the experiments. Both of these processes are fully customizable, since the 'scripting language' used is our full-fledged logic meta-programming language.

While this functionality is nice, we can go a step further and use our notification system. For example, we can make a UML tool that enforces that every change in a design diagram should comply to the implementation. For example, when an operation is added to a classifier, the implementation

[1] Our current implementation monitor ensures that the changes have been applied to the code before wend the change message. This could easily be changed to send notifications before the change is actually applied.

can be checked to make sure that the class corresponding to the classifier indeed implements such method. If not, this can be logged or template code might be generated. Implementing this scheme merely comes down to overriding the *clauseAdded:* and *clauseChanged:* methods.

5.2.3 Conceptual validation

In this section we evaluate the synchronization framework regarding the characteristics of synchronization as discussed in section 2.5.1. For every characterization we show how the synchronization framework can be instantiated to accommodate it:

direction of synchronization : we express design as a logic meta program of implementation. As explained in section 2.6, the logic meta program can then be used for checking (when both design and implementation exist) and to generate the one from the other (if only one of them is specified). In order to do this, no specific instantiation is necessary, since it is a direct result from using a logic meta-programming language to make the design explicit;

action to be taken : when elements are found that are out of sync, there are two possibilities: the differences can be reported (allowing the user to take action later), or they can be fixed automatically. SOUL supports both the reporting and the action. The reporting is a direct result of using a logic meta-programming language, since the report is actually formed as the results of the query one formulates to do the synchronization. Actions have to be implemented by instantiating the *declarative framework*, and adding rules that describe the actions. Then the *synchronization tool framework* has to be instantiated to invoke the actions at certain moments. Note that a number of rules exist with built-in actions, namely the *checkPossiblyGenerate* rules we saw in sections 4.4.1;

notification time : what notification time is possible depends entirely on the instantiation of the synchronization tool framework. More specifically, tools can subscribe themselves to receive notifications of changes to design and implementation. The synchronization tool framework has to send all these events, so the tools can choose whatever they need. Note however that, in the current implementation of the *synchronization tool framework*, we support no proactive notifications. However, as we explained in section 5.1, this would not be very hard to add;

trigger time : whether direct or delayed triggering is used, depends on the instantiation of the *synchronization tool framework*. By default, direct synchronization is supported because the tools using the synchronization tool framework receive notifications for any change to design and implementation. However, the actions that need to be taken when a change is detected have to be implemented. Note that typically, for performance reasons, the SOUL incremental solver will be used in combination with direct synchronization. The tools can also be set to receive no notifications, and then the queries have to be launched manually to initiate synchronization. Because the event mechanism is a very general and fine grained one (every change is received), even combinations are possible;

scope : the scope of rules is global, since they are contained in a global repository and not bound to certain pieces of the implementation. Local scope therefore needs to be specified in the rules. For example, queries will typically restrict their search to certain hierarchies of classes;

implementation granularity : by default, the representational predicates use complete parse trees since we wanted to use fine-grained information. However, the declarative framework can be instantiated to use other representations that are partial;

static/dynamic : the synchronization framework currently uses only static information. We have two important remarks to make here. First of all, we have experimented with static reasoning on top of information that was collected dynamically [RDW98]. Of course, this runtime information can be combined with the static information, since they complement each other. Second, because of SOUL's symbiosis with Smalltalk, Smalltalk objects can be used directly in SOUL. When both are integrated even further, reasoning about dynamic information will be very easy. We discuss this in some more detail in section 8.3.2.

By discussing how the synchronization framework can be instantiated to support the different characterizations, we have also shown that, in general, it supports all the possibilities of synchronization. There is one exception: the reasoning is essentially static. Of course, this general discussion still has to be backed with a concrete validation. This is done in the next two chapters, were we described our experiments.

5.3 Conclusion

In this chapter we combine the *declarative framework* introduced in the previous chapter with the *synchronization tool framework*. The *synchronization tool framework* is a Smalltalk application framework that is used to build applications that need synchronization of design and implementation. Therefore it implements a mechanism that tools can use to receive notifications of changes to design and implementation. The combination of the *declarative framework* and the *synchronization tool framework* results in the *synchronization framework*. We discuss two applications that use the synchronization framework to support co-evolution: one application that logs violations against programming conventions, and a UML tool that is synchronized with the implementation. Then we evaluate the synchronization framework and show that it conceptually indeed supports all the characterizations of synchronization (except the support for dynamic information). In the two following chapters we perform practical experiments with the synchronization framework. A first series of experiments is performed on the HotDraw framework and experimentally shows the complete spectrum of synchronization supported by the synchronization framework. It also assesses the practical usability of our approach. The second series of experiments shows the usability and scalability of the synchronization by performing several experiments on a large industrial framework.

Chapter 6

Supporting co-evolution

The previous chapter discussed the synchronization framework, and showed conceptually that it indeed supports the complete spectrum of synchronization described by the characterizations. However, we are also interested to assess the practical usability and scalability of the synchronization framework. Therefore we performed two series of experiments, that are described in this chapter and in the following one. In this chapter we perform a small-scale case study on the well-known HotDraw framework, a framework for drawing editors. The goal is to show experimentally that the synchronization framework supports the different characterizations, and to assess the practical usability of our approach. The second series of experiments, that are described in the next chapter, test the usability and scalability of our approach on a large-scale industrial framework used in the television broadcasting industry.

6.1 Introduction

The claim of this dissertation is that when design is expressed as a logic meta program over implementation, a framework can be constructed that allows the design to check, generate or constrain the implementation, and vice versa. The proof of this claim is done *by construction*, meaning that we built the *synchronization framework*, a framework to synchronize design and implementation. As discussed in section 5.2.3, this framework conceptually supports the complete range of synchronization described by the characterizations of synchronization. However, we also want an experimental proof that the complete spectrum is supported, which also assesses the usability in the real world. Therefore this chapter performs experiments on HotDraw [Bra92, Joh92, BJ94, Cha94], a framework for structured drawing editors written in Smalltalk. HotDraw is used to build editors for specialized two-dimensional drawings such as schematic diagrams, blueprints, music, or program designs. The elements of these drawings can have constraints between them, they can react to commands from the user, and they can be animated. The editors can be a complete application, or they can be a small part of a larger system. We take this case because it is implemented in Smalltalk, fairly well-understood and documented, and not too large. In the next chapter we perform experiments on a larger, real-world case.

The *red thread* in the experiments is the building of a graphical editor for the constraint networks of the incremental solver. More specifically, we look at how to create an editor application and the appropriate figures representing *constraint variables* and *relations*. We have divided the experiments into two cases. In the first set of experiments, we show how *delayed synchronization* (where several changes are made to design and/or synchronization before being synchronized) is supported in our

approach. In the second series of experiments we show how *direct synchronization* (where changes to the implementation are directly propagated to the design and vice versa) is supported. We separated both cases from the beginning since the delayed synchronization can be done using an ordinary logic meta-programming language while we need the incremental solver to support direct synchronization. In both experiments, we use the same logic meta programs expressing the design notations used in previous section, but we store the actual design information differently. As is explained in section 6.6, both can be unified in one repository.

6.2 General setup of the experiments

Our experiments were performed in VisualWorks/Envy R4.01, on a Macintosh Powerbook G3 running at 250 Mhz, and with 64 megabytes physical memory. The version of HotDraw was the latest release for Envy (HotDraw R1.00). We used the latest version of SOUL/Envy (SOUL R1.55), including the synchronization framework. Since there was no design information available at the beginning of the experiments, the design repository was initially empty.

It is important to notice that we were not familiar with the HotDraw framework when we started experimenting with it. Hence, there was not only the issue of expressing part of the documentation of HotDraw and showing how it could be used by framework users, but there was also the problem of learning the framework. During the experiments we therefore wore two different hats, and frequently changed them: on the one hand we took the role of the *framework developer*, documenting the framework using logic meta programs in an effort to keep the framework documentation and the framework tied close together. On the other hand we played the role of *framework user* who tries to use the framework and the documentation to build an application.

6.3 Supporting delayed synchronization

In the first series of experiments we show how to support a development process where the synchronization of design and implementation is done from time to time. We follow a development scenario where we familiarize ourselves with the HotDraw framework and then instantiate it to build an editor for networks for our incremental solver. This shows how design can be extracted from implementation, how implementation can be generated from design and how design and implementation can be checked for conformance at any given point in time:

- the first experiment is to use logic meta-programming to better understand the implementation of HotDraw. This is actually a kind of reverse engineering step, where the existing documentation of HotDraw is used as blueprints of queries that are checked against the source code. It is thus concerned with generating (extracting) design from implementation. In this experiment we start out with no design documentation, and completely extract this information from the implementation using SOUL and the design expressed in the declarative framework;

- then we express design information that is specific to HotDraw. We focus on a particular hierarchy of composite figures, and on the relationships between the *editor* (the actual application), the *tools* (that create and manipulate figures) and the *figures* themselves. This step shows how design information that was extracted in the first experiment is made explicit, so that it can be used later on. Note that no synchronization is done in these experiments. We just show how we can instantiate and complement the general rules in the declarative framework with rules that

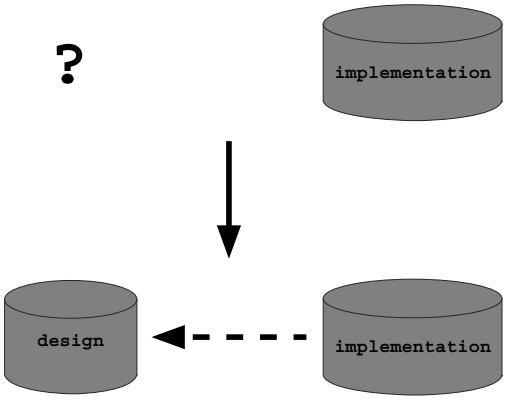

Figure 6.1: Extraction of design information from implementation.

express more specific design information. The next two experiments then show how we can synchronize this updated design information with the implementation again;

- this experiment shows how to synchronize the implementation with the updated design information from previous step. It shows a combination of using a query to find inconsistencies between design and implementation, and an action that automatically generate methods and classes if they are not implemented;
- then we see how to do conformance checking between design and implementation after changes have been made to design and implementation. In this experiment we do a conformance check between the updated design information from the second step and the modified implementation. In contrast with the previous experiment where a default action is provided that generates parts of the implementation, this consistency check results in a report that shows where the design and the implementation deviate.

6.3.1 Extraction of design information

Being a novice regarding the implementation of HotDraw, we started by reading the HotDraw documentation and papers mentioned on the website. The initial situation is depicted in figure 6.1: we had the HotDraw implementation, but no explicit design documentation. At the end of the synchronization we thus wanted to have a repository containing design documentation. We came quickly to the conclusion that most documents described a version of HotDraw *older* than the one we were using. However, it gave us enough overview of the implementation to get started. One of the first queries we ran extracted the *root classes* of HotDraw[1]. Based on the names of these classes, and on the HotDraw implementation, we came to the conclusion that three of these were of general interest: *DrawingEditor*, *Tool* and *Figure*. The *Figure* hierarchy proved very extensive, so in order to familiarize ourselves with it we used the UML class diagram rules from section 4.5.3 to extract a simple UML class diagram. These results are stored in the HotDraw design repository (indicated by the second argument to the predicate, *HotDrawDesign*):

[1] We found 18 root classes: Figure, LineAnnotation, DrawingController, TransitionTable, PositionConstraint, ButtonDescription, BoundaryConstraint, TransitionType, ToolStateCommandEditor, TransitionEditor, DrawingEditor, ToolbarController, ToolStateModel, EndToolState, ToolStateTransitionModel, FigureAttributes, ToolbarView, Tool

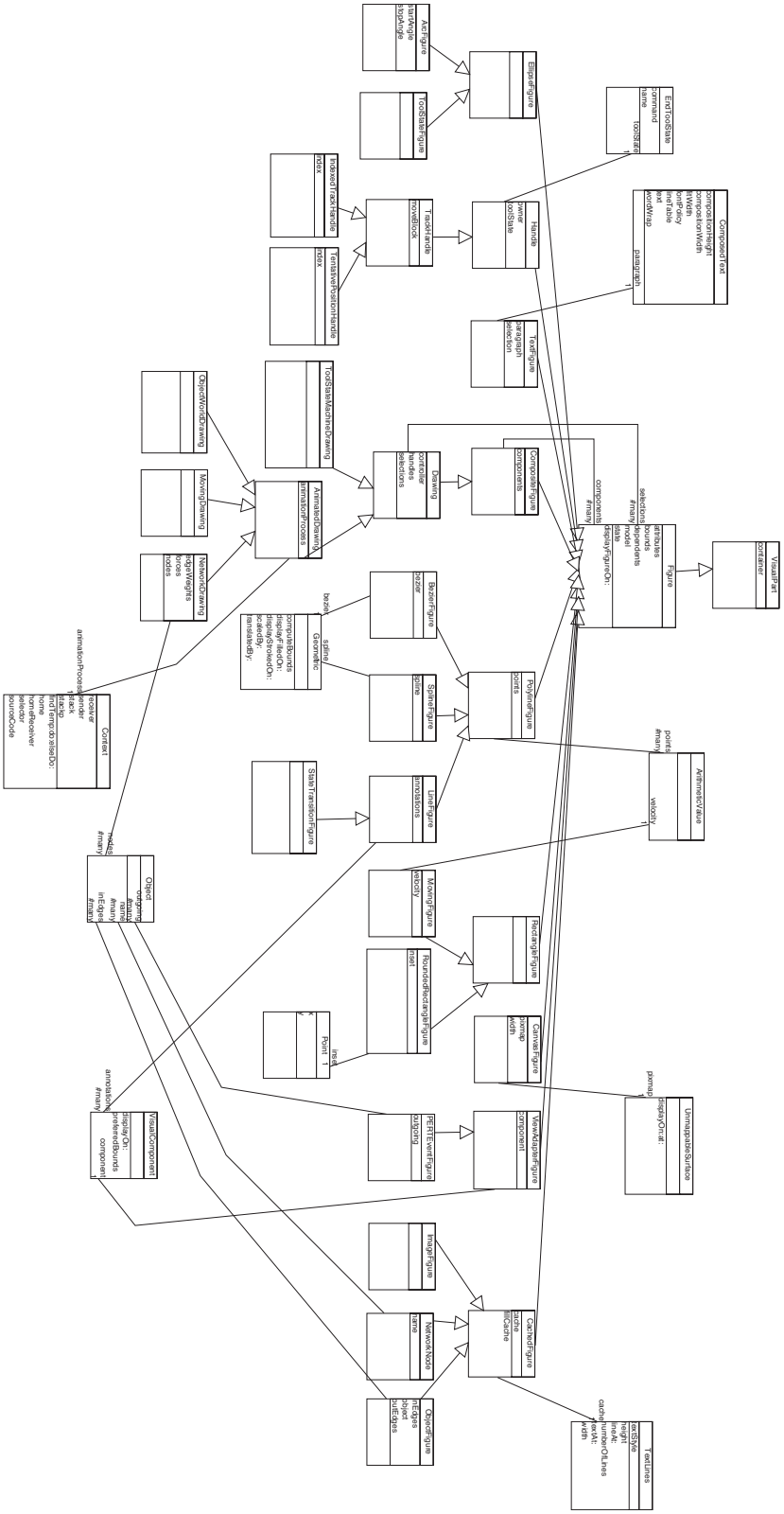

Figure 6.2: The extracted UML class diagram for class Figure.

predicate	description
composite	compositePattern([Figure],[LineFigure]) compositePattern([Figure], [Drawing]) compositePattern([Figure], [CompositeFigure])
visitor	none
singleton	none
bridge	none
factory method	base level: 41 meta level: 3
accessor methods	base level: 56 meta level: 1

Table 6.1: Extracting design from HotDraw

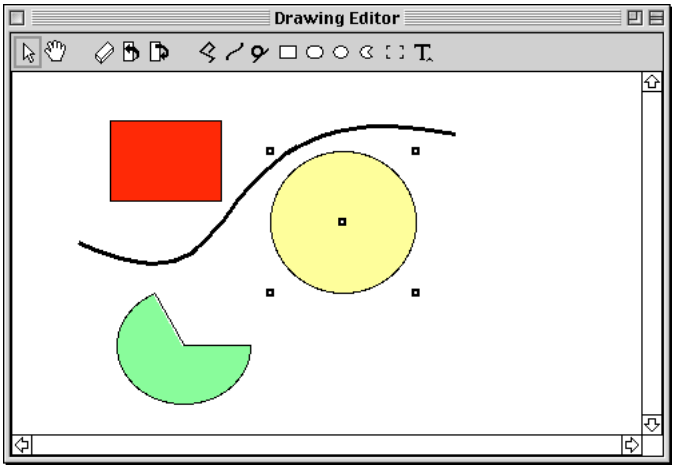

Figure 6.3: The standard HotDraw editor opened

Query initializeDrawing(),
assertUMLDiagram([Figure], [HotDrawDesign])

The extracted classifiers and generalization and association relationships are shown in figure 6.2. Besides the extraction of this UML information, we also ran queries to check what accessor methods were used where, and if we could detect design patterns. The results are shown in table 6.1. In this extraction phase we used all the SOUL development tools that were introduced in section 3.2.6.

This step shows how a lot of design documentation can be extracted from the implementation, helping a developer unfamiliar with a certain implementation to gain a better insight. This does not eliminate the need to read documentation, or to browse the source code manually to look at certain parts of the implementation. It merely complements it by providing advanced development tools that allow to browse the system on different levels of abstraction. Next we see how this extracted information can then be refined towards more specific documentation.

6.3.2 Complementing the extracted design with specific information

In the first experiment in section 6.3.1 we showed how to extract design from implementation. In this section we show how to complement this design with more specific HotDraw design information regarding editors and composite figures. Hence, in this experiment we start with the synchronized design and implementation from the first experiment, and complement the design information with more specific information. It is important to note that in this step we do *not* yet synchronize design and implementation. This will be done in the next two sections, 6.3.3 and 6.3.4.

Screenshot 6.3 shows a standard HotDraw editor where we have drawn some standard figures. Note the label of the window (that says *Drawing Editor*), and the toolbar. The toolbar displays a number of buttons that can be used to draw figures in the drawing area, to select figures (which displays their handles so they can be manipulated), and to delete figures. Not shown on the picture is a context sensitive menu that is associated with each figure, and allows us to set some properties (such as fill colours and line widths). Of course, specific figures can have specific menus by overriding the *menuAt:* method.

The implementation of the editor uses three main components: the *DrawingEditor*, the *Figure* and the *Tool*. From browsing the code (using a combination from the standard Smalltalk development tools and the SOUL development tools), we found some very important, yet undocumented properties and relationships between these components:

1. The interaction of the user with the editor is completely described using state diagrams. Class *Tool* uses these state diagrams to decide what action is performed when a user clicks in a drawing editor. Therefore, each tool needs to be added to this state diagram. This is done by adding a class method to class Tool that describes the state changes for this tool, and what figures it creates. The name of that state has to be used in the *toolNames* method in the editor;

2. the label displayed by the window is determined by the *windowName* method on *DrawingEditor*. Subclasses can override this to display other names;

3. the tools that are shown by the editor are enumerated in a method called *toolNames*. This method lists a number of *tool states*. Hence, every one of the tool states mentioned should be available in the state diagram offered by the class *Tool*;

4. every tool offered by the editor has a button on the toolbar. Therefore the editor has to offer icons to use as buttons for every tool name mentioned in the *toolNames* method. These icons are returned by methods that reside on the class side of the editor class. However, there has to be some mapping between the name of the tool, and the name of the method used to provide the icon. This mapping is actually a naming convention implemented in a method called *iconNameFor:*. Hence the toolbar is constructed by enumerating all the tool names provided by the *toolNames* method, and retrieving the icon for this tool using the naming convention.

Note that these dependencies are between three classes that are hierarchically unrelated, and are implemented using naming conventions, hardcoded references and Smalltalk meta-programming techniques. We only found these dependencies by using the SOUL tools and regular Smalltalk development browsers, as they were undocumented and scattered in different locations in the source code. Therefore we decided to make these conventions explicit as logic meta programs. The predicates we implemented are shown in table 6.2, and are described in detail in the rest of this section. We then used these predicates to generate code, for conformance checking and in a constraint network to guide development.

predicate	description
hdEditorClass	an editor belongs to the *DrawingEditor* hierarchy
hdToolNamesMethod	has a *toolNames* method
hdIconMethod	has icons for each tool
hdToolMethod	extends the state diagrams on class *Tool*
hdEditor	relates the editor and its tools and figures

Table 6.2: The predicates dealing with editors, tools and figures

hdEditorClass

The first predicate we would like is one to help us with constructing editors. As outlined in the above explanation about editors and tools, there are several classes and methods we have to generate in order to accomplish this. We have spread this code over several rules. First of all we implement the *hdEditorClass* predicate that describes what an editor class looks like. Actually, this is a very simple predicate, since it only needs to check that the class belongs to the *DrawingEditor* hierarchy:

Rule hdEditorClass(?name, ?editor) **if**
hierarchy([DrawingEditor], ?editor),
className(?editor, ?name),

This predicate can now be used to detect inconsistencies between design and implementation, and to extract information from the implementation. For example, when we have information that some class is an editor class, we can use the *hdEditorClass* to confirm or reject this. The name of the class is bound to *?name* when the existence of the editor in the implementation is confirmed:

Query hdEditorClass(?name, [FooEditor])

By replacing *FooEditor* with another variable, we can use the same *hdEditorClass* predicate to extract information about editors from the code. However, note that this predicate does not try to correct inconsistencies. For example, if there is no *FooEditor* in the implementation, the query just fails. Then it is again up to the user to determine the action to take to solve the inconsistency. In terms of the classification of synchronization as given in section 2.5.1, this comes down to the *report* action. Using some of the other predefined predicates in SOUL, we can also implement a version of the *hdEditorClass* rule that tries to automatically solve inconsistencies. We call this predicate *cpgEditorClass*, where the acronym *cpg* stands for checkPossiblyGenerate (which is used throughout our predicates):

Rule cpgEditorClass(?name, ?editor) **if**
cpgClass(?name, [DrawingEditor]),
className(?editor, ?name),

The *cpgEditorClass* predicate makes sure that there is an editor class with a certain *?name* (using the *cpgClass* predicate). If no such class exists, it is generated as a direct subclass of *DrawingEditor*. If it already exists in the hierarchy of *DrawingEditor*, nothing is generated and the rule succeeds. If it exists, but is not a subclass of *DrawingEditor*, the generation fails.

```
DrawingEditor>>toolNames
        "Return the list of names for the tools.
        'nil' represents a space between tools in the icon bar."

        ^#(     'Selection Tool'
                'Hand Tool'
                nil
                'Delete Tool'
                'Bring To Front Tool'
                'Send To Back Tool'
                nil
                'Polyline Tool'
                'Bezier Tool'
                'Spline Tool'
                'Rectangle Tool'
                'Rounded Rectangle Tool'
                'Ellipse Tool'
                'Arc Tool'
                'Image Tool'
                'Text Figure Creation Tool')

HotPaintEditor>>toolNames
        "Return the list of names for the tools."

        ^super toolNames
        , #(    nil
                'Hot Paint Canvas Tool'
                'Hot Paint Paintbrush Tool'
                'Hot Paint Mask Tool'
                'Hot Paint Image Tool'
                'Hot Paint Erase Tool')
```

Figure 6.4: The toolNames method for two classes: DrawingEditor *and* HotPaintEditor

hdToolNamesMethod

The second predicate we describe is concerned with the tools offered by the editor. As said before, the editor should override a method called *toolNames* to describe the tools it uses. There exist two typical implementations of this method. The first just returns an array with the names of the tools this editor offers. For example, the *toolNames* method of *DrawingEditor* shown in figure 6.4 lists the default tools that are used by all HotDraw editors. The second implementation strategy that is typically offered by specific editors is to get the tools used by their superclass, and append some specific tools to it. The second method of figure 6.4 shows the implementation of the *toolNames* method for class *HotPaintEditor*, one of the HotDraw examples. The following *hdToolNamesMethod* predicate describes the general structure of a *toolNames* method for a certain editor. Note that we have added a *?kind* variable, that can be used to differentiate different forms of *toolNames* when needed. The *hdToolNamesMethod* predicate first of all states that *?editor* should be an editor class. Therefore

we use the predicate introduced above, without supplying a particular name for the editor. Then we say that the method *?m* should be a method of class *?editor*, with the name *toolNames*. The statements of the method are described by an auxiliary predicate, *hdToolNamesStatements*. We provide two facts that implement this predicate, corresponding to the two ways of implementing the *toolNames* method described above. This could of course be extended to include other forms as well.

Rule hdToolNamesMethod(?editor, ?m, ?toolNamesList, ?kind) **if**
editorClass(_, ?editor),
classImplementsMethodNames(?editor, [#toolNames], ?m),
methodArguments(?m, <>),
methodTemporaries(?m, <>),
hdToolNamesStatements(?statements, ?toolNamesArray, ?kind),
array2List(?toolNamesArray, ?toolNamesList).

Rule hdToolNamesMethod(?editor, ?m, ?toolNamesList) **if**
hdToolNamesMethod(?editor, ?m, ?toolNamesList, _).

Fact hdToolNamesStatements(<return(literal(?toolsArray))>, ?toolsArray, [#enumeration]).

Fact hdToolNamesStatements(<return(send(send(variable([#super]), [#toolNames], <>),
[#,],
literal(?toolsArray)))>,
?toolsArray,
[#superAppendEnumeration])

The *hdToolNamesMethod* predicate describes the form of the toolNames method of a certain editor, if the method exists. This again allows to extract the tools from a certain editor, to find all editors using a particular tool or to check whether a particular editor uses a particular tool. However, the results are just reported as results from a query, and when an inconsistency is found, no attempt is made to solve it. Therefore we implement another predicate that generates an appropriate *toolNames* method in the case it does not exist, or that rewrites an existing *toolNames* method when one does exist but lacks certain tools we would like to offer in the editor (note that we could also have split the second rule in two):

Rule cpgToolNamesMethod(?editor, ?toolNamesList) **if**
hdToolNamesMethod(?editor, ?m, ?toolNamesList, _).

Rule cpgToolNamesMethod(?editor, ?toolNamesList) **if**
not(hdToolNamesMethod(?editor, ?m, _, ?k)),
xor(hdToolNamesStatements(_, ?existingToolNames, ?kind),
and(equals(?existingToolNames, <>),
equals(?kind, [#superAppendEnumeration]))),
difference(?existingToolNames, ?toolNamesList, ?newToolNames),
append(?existingToolNames, ?newToolNames, ?newNamesList),
hdToolNamesStatements(?newStats, ?newNamesList, ?kind),
generateMethod(?editor, [#toolNames], <>, <>, ?newStats)

Using the *hdToolNamesMethod*, we also implemented a *hdToolNameMethod* predicate that relates an editor and each of its tools separately instead of the editors and the list of all its tools.

```
iconNameFor: aString

    | iconName |
    aString isNil ifTrue: [^nil].
    iconName := aString select: [:each | each isAlphaNumeric].
    iconName := iconName copyFrom: 1 to: (iconName size - 4 max: 1).
    iconName at: 1 put: iconName first asLowercase.
    ^(iconName , 'Icon') asSymbol
```

Figure 6.5: The iconNameFor: *method on* DrawingEditor*, implementing the naming convention to get the icon for a tool*

hdIconMethods

For every tool enumerated in the *toolNames* method, there should be a corresponding icon that is used to build the toolbar. The *DrawingEditor* implements a default naming convention, that determines the selector to use for a given toolName. This naming convention is implemented in the method *iconNameFor:*, for which the code is given in figure 6.5. When this method is called, *aString* is bound to a string containing the name of the tool. When this string is not nil, its alphanumeric characters are used, the last four characters are chopped of, and the first character is ensured to be a lowercase character. At last, the string *'Icon'* is appended. For example, for the tool called *'Selection Tool'*, this yields the string *selectionIcon*, while *'Rounded Rectangle Tool'* yields *'roundedRectangleIcon'*. The resulting strings are used to retrieve the icons for these tools. The implementation of this method is interesting, because it reveals another naming convention: the last four characters are chopped of because every toolname ends on *'Tool'*. This naming can also be made explicit by providing a separate predicate for it, or adding some lines to the *hdToolNamesMethod* predicate explained in previous the section.

In short, every toolname should have a corresponding class method that returns its icon. The *hdIconMethod* predicate makes this relation explicit. For a given editor, it describes that a toolName should have a corresponding method on the class side. It uses an auxiliary predicate, *toolIconName*, that uses a `smalltalk term` to invoke the *iconNameFor:* method of the supplied editor to get the correct naming convention. Note that we could also have implemented this directly in SOUL, using the string handling predicates.

```
Rule hdIconMethod(?editor, ?toolName) if
    metaClass(?editor, ?editorClass),
    toolIconName(?editor, ?toolName, ?toolIconSelector),
    classImplements(?editorClass, ?toolIconSelector).

Rule toolIconName(?editor, ?toolName, ?toolIconName) if
    generate(   ?toolIconName,
                [(?editor new iconNameFor: ?toolName) asStream])
```

As in the previous sections, we also added a predicate *cpgHotDrawIconMethod*, that generates the necessary method to provide an icon in the case where it is absent. This is again an example of

a predefined action we built into the predicate, and that decides to generate the implementation in the case where it is missing. The method that has to return this icon uses two auxiliary methods, that provide the image and the mask to use. In the predicate the code is provided by the auxiliary predicates *toolIconImageCode* and *toolIconMaskCode*. Since these predicates essentially contain Smalltalk code that just return textual descriptions of the default icons, we do not show their implementation. The *cpgIconMethod* just uses these predicates to get the code, and generates the appropriate methods when needed:

Rule cpgHotDrawIconMethod(?editor, ?toolName) **if**
hotDrawIconMethod(?editor, ?toolName).

Rule cpgIconMethod(?editor, ?toolName) **if**
metaClass(?editor, ?editorClass),
not(hotDrawIconMethod(?editor, ?toolName)),
toolIconName(?editor, ?toolName, ?toolIconSelector),
toolIconImageCode(?toolIconSelector, ?imageSelector, ?imageCode),
cpgMethodInProtocol(?imageCode, ?editorClass, [#resources]),
toolIconMaskCode(?toolIconSelector, ?maskSelector, ?maskCode),
cpgMethodInProtocol(?maskCode, ?editorClass, [#resources]),
toolIconSelectorCode(?toolIconSelector, ?imageSelector, ?maskSelector, ?iconSelectorCode),
cpgMethodInProtocol(?iconSelectorCode, ?editorClass, [#resources])

While the *hdIconMethod* and *cpgIconMethod* predicates define the relationships between one toolName and its icons, we also added two predicates that do this for a list of toolNames (*hdIconMethods* and *cpgIconMethods*). The implementation simply enumerates all elements in the list and uses the single version predicates.

hdToolMethod

Previous sections describe predicates that all dealt with just the class or methods on the editor itself. Now we describe the methods on the *Tool* class that adds the necessary states to the general state diagram governing the behaviour of the editors. Like we said before, the *Tool* class uses a state diagram to implement the interaction between the user and the editor. Therefore, each tool offered by the editor has to be added to this state diagram. That way, when the user selects a tool in the buttonbar, and then clicks on the drawing area, the figure defined by the tool can be created.

When building the state diagram defined by class *Tool*, all methods from the protocol *tool states* of *Tool*'s metaclass are enumerated. Every method in this protocol adds certain states and transitions, that are used by *Tool* whenever the mouse is moved or clicked within the HotDraw editor. Therefore, every toolname listed in the *toolNames* method should occur in at least one method in the *tool states*. Also important is that these methods can be used to associate toolNames and figures, since the transitions are responsible for creating figures. We have expressed this information in the *hdToolMethod* predicate. This predicate associates a certain toolName with its figure and its initialization method on the *Tool* class. Therefore it enumerates the methods in the *tool states* protocol. For each of these methods, it checks which figures are referenced by looking at the referenced classes and only keeping the ones that actually belong to the *Figure* hierarchy. Then it enumerates the tool states, and keeps the ones that end on *'Tool'*. Like we said before, this naming convention of postfixing tools is used throughout HotDraw, and can here be put to good use. As with the previous predicates, we again

provide a *cpgToolMethod* predicate that generates a default initialization method adding states to create and select the figure. The name of this method is constructed by prefixing the name of the tool (without spaces) with the string *'initialize'*. The rest of the code is then given by an auxiliary predicate *toolInitializationCode* that we is not shown here because it contains just a Smalltalk description of the states and transitions that are added for the figure created by the tool.

```
Rule hdToolMethod(?toolName, ?figure, ?initMethod) if
        methodInProtocol([Tool class],[#'tool states'],?initMethod),
        classesUsed(?initMethod, ?classList),
        member(?figure, ?classList),
        hierarchy([Figure], ?figure),
        isSendTo( ?initMethod,
                  send(variable([#Tool]),[#states],<>),
                  [#at:put:],
                  <literal(?toolName), _>),
        patternMatch(?toolName, postfix(['Tool']))

Rule cpgToolMethod(?toolName, ?figure, ?initMethod) if
        cpgFigure(?figure),
        toolInitializationCode(?toolName, ?figure, ?code),
        generateMethodInProtocol(?code, [Tool class], [#'tool states'])
```

The *hdToolMethod* relates three variables: the tool used, the figure and the tool where the states are added. We have a very important remark to make here, that may not be visible because it is very natural: this rule actually configures several classes from completely different hierarchies to work together. This is a major difference with some of the other rules we have seen and that only express local properties of methods or classes. While this rule does not much different, it describes the interplay between some components in the HotDraw framework.

Note that we also added *hdToolMethods* and *cpgToolMethods* that handle lists of toolNames and figures.

When we used the *hdToolMethod* predicate to see which figures are initialized by which methods, we noticed that some figures were missing. This means that there exist HotDraw figures that are not created by tools in editors. We can divide these figures in some groups.

- first of all we have the class *Figure* itself. This is a template class that is not meant to be created by tools, so this is no problem;
- next we found *CachedFigure*, *ViewAdapterFigure*, *ToolStateFigure* and *StateTransitionFigure*. These classes were used in examples or to edit the state diagrams, but have no associated tools that allow them to be drawn in an editor;
- more interesting was that there is no tool to create a *LineFigure*. The reason is that instances of LineFigure are created directly when a user connects two figures. Instead, one would expect a state transition that defines that when a user clicks the connection point of a figure, and moves the mouse to another figure, a new LineFigure is created connecting the two figures;
- the same holds for the three figures that represent handles on figures (*TrackHandle*, *IndexedTrackHandle* and *TentativePositionHandle*). These handles are created and returned by the figures when they are selected in a drawing.

predicate	description
hdcClass	a composite figure is in the hierarchy below CompositeFigure
hdcConstraintsVar	has an instance variable to hold the constraints
hdcInstanceCreation	has a *createAt:* constructor
hdcInitialization	has a *initializeAt:* method
hdcCopying	has a *postCopy* method
hdcSetBoundsTo	has a *setBoundsTo:* method
hdcFigure	combination of all the previous predicates

Table 6.3: The predicates dealing with CompositeFigures

- grouping of figures is accomplished through the menu, and not through a tool, so *CompositeFigure* is also one of those classes for which there is no tool.

This is actually an example to show that, by making the design information explicit and using it to check the source code, we can make quality assessments. For example, in this case it shows places where refactoring of code is probably needed, or more explanations so that we can understand why these classes form exceptions and are not created by tools. We will see more examples when we express a specific set of programming conventions in the case study in the next chapter, and do a conformance check with the implementation.

hdEditor

We have now all predicates to define what a HotDraw editor class looks like, and to help us check and generate editors. Therefore we group the predicates from previous sections to define the *hdEditor* and the *cpgEditor* predicates:

Rule hdEditor(?name, ?editor, ?toolNamesList, ?figureList) **if**
hdEditorClass(?name, ?editor),
hdToolNamesMethod(?editor, ?m, ?toolNamesList, ?kind),
hdIconMethods(?editor, ?toolNamesList),
hdToolMethods(?toolNamesList, ?figureList, ?initMethod),

Rule cpgEditor(?name, ?editor, ?toolNamesList, ?figureList) **if**
cpgEditorClass(?name, ?editor),
cpgToolNamesMethod(?editor, ?m, ?toolNamesList, ?kind),
cpgIconMethods(?editor, ?toolNamesList),
cpgToolMethods(?toolNamesList, ?figureList, ?initMethod),

The predicates dealing with composite figures

Besides the information that deals with the interplay between the *Tool* class, the editor and the figure, we also expressed information regarding composite figures. The implementation of these predicates is not discussed, but they are listed in table 6.3.

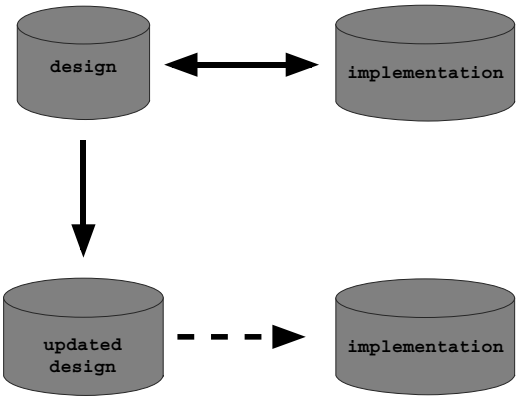

Figure 6.6: Generating missing parts of the implementation from the design.

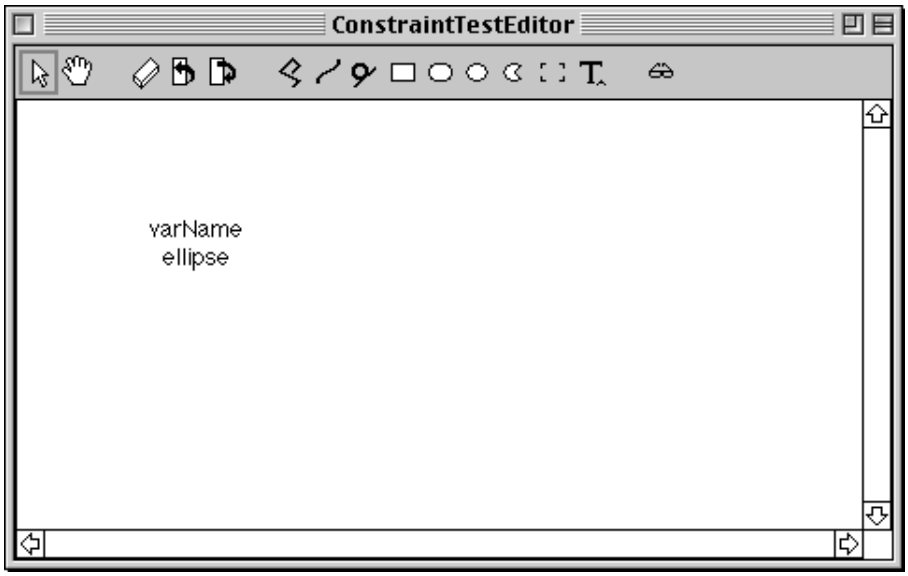

Figure 6.7: Screenshot of a freshly generated editor, where we created a freshly generated ConstraintVariableFigure

6.3.3 Generating missing implementation parts from the design

In the previous section we made HotDraw specific design information explicit. This means that we have a declarative framework that is extended to express some specific design information. This situation is depicted in figure 6.6. More specifically, the HotDraw specific design documentation describes that for each composite figure and editor the implementation has to implement classes with certain methods. In this experiment we show how this information is used to check the conformance between design and implementation. Moreover we do not only report the inconsistencies, but also implement an action that automatically generates missing implementation parts: whenever the design information specifies that a class or a method should exist, but it is not found in the implementation, it is generated. If it already exists, nothing happens. This actually means that this action leaves existing implementations *as is*, and does not override them.

For example, the following query expresses the design information that there is a composite figure called *ConstraintVariableFigure*, that has two component figures, *varName* and *ellipse*. It also specifies that there is an editor, called *ConstraintEditor*, with a tool that can draw *ConstraintVariableFigures*. In this experiment we are only interested in synchronizing this piece of design information with the implementation, hence we invoke the following query, that uses the design information expressed in section 6.3.2:

```
Query cpgCompositeFigure(   [#ConstraintVariableFigure],
                            <[#varName], [#ellipse]>),
      cpgEditor(    [#ConstraintEditor],
                    ?editor,
                    <['ConstraintVariableFigure Tool'] >,
                    <['ConstraintVariableFigure'] >)
```

In this experiment, neither the composite figure class, nor the editor class existed yet. Hence, they were completely generated, including their methods and the appropriate methods on the class *Tool*. Since no types were specified for the component figures, *TextFigures* containing their name are generated. Hence, the result of this synchronization example is that the implementation was made consistent with the design as specified in the query by generating it using the information contained in the declarative framework. The generated editor is showed in screenshot 6.7, where we have opened our new *ConstraintEditor* and created a *constraint variable* figure. Note again that, since we did not specify more specific information about the figures that need to be created, the names of the variables were used. Hence we see two TextFigures that use constraints to be positioned above one another.

We also want to stress that in the case the classes or methods specified in the design would have existed, they would not have been overridden. Hence, when changes are made to the implementation (for example to actually draw an ellipse figure around the name of the variable), and we re-synchronize design and implementation, these manual changes will not be lost. Of course, other actions can be implemented with other default behaviour, such as always regenerating the implementation, or prompting the user what to do.

6.3.4 Checking implementation and design

In section 6.3.1 we have shown how to use the synchronization framework to extract design from implementation. In section 6.3.3 we showed how to provide an action that automatically generates classes and methods when they are specified in the design but do not exist in the implementation.

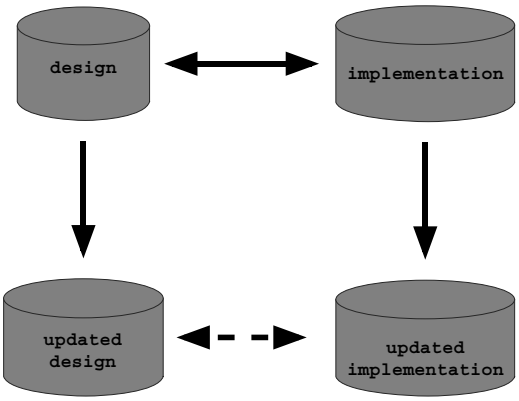

Figure 6.8: Two-way conformance check between design and implementation to synchronize them after both have been changed.

In this section we do a full conformance check between design and implementation that reports on any piece of design information that is not implemented, and any piece of implementation for which design information should be present, but is not. Hence, the situation of this synchronization is shown in figure 6.8: design and implementation were changed, and we are interested to see where they differ.

We have already discussed the updated design repository and mapping in detail in previous sections. In this section we manually make additions to the implementation, and then we synchronize this with the design information. The changes to the implementation were concerned with changing the implementation of *ConstraintVariableFigure* to draw an ellipsis around the variable name. More specifically, we changed the generated *initialize* method to initialize the second figure to be an ellipse. Then we changed the *setupConstraints* method (which was also generated as a result from the synchronization done in section 6.3.3), and changed the constraint that put the two text figures below each other to a constraint that ensures that the ellipse is centred around the text with the name of the constraint variable.

We now want to check whether the implementation and the design are still consistent after these changes to the implementation. Such conformance check actually consists of two parts, that can be combined. The first check is to see whether the current design information still holds for the implementation. The second check is to see whether the new implementation yields a new or changed design.

Checking whether the design information is still valid for the new implementation is relatively simple. Since all the design information is contained in the form of facts in a logic repository, we just have to fire each of these facts as a query, which checks that fact against the implementation. We are interested in all queries that return false, since this produces the pieces of design information that became invalid. This is shown in the following query, that first retrieves all the clauses of our design repository (using the *repositoryClause* predicate) and then checks all of them against the current implementation (by calling them as queries using the *call* predicate). The failed ones are enumerated in a list, which is the result of this query.

```
Query findall( ?failedInfo,
               and( repositoryClause(?failedInfo, [HotDrawDesign]),
                    not(call(?failedInfo))),
               ?failedList)
```

The result of this query is a list with the failed clauses. This list can then be processed manually (checking why a piece of design information does not hold anymore, using all facilities offered by the development environment and our tools), could be used in a browser that displays it more appropriately, or could form the foundation for a sophisticated tool that suggests solutions. Note that in the query we gave the complete design repository is rechecked. By slightly changing this query, we can recheck only certain parts of the repository (for example, to determine whether some specific design patterns still holds) to speed up the check.

The second check looks whether the new implementation also adds new design information. The general approach to do this is to regenerate (parts of) the design, and check whether they are contained in the repository. For example, we can extract the information regarding composite design patterns, and find all possible new instances:

```
Query findall( newCompositePattern(?comp, ?composite, ?sel),
               and( compositePattern(?comp, ?composite, ?sel),
                    not(repositoryClause( compositePattern(?comp, ?composite, ?sel),
                                          [HotDrawDesign]))),
               ?newComposites)
```

This query again returns a list containing all possible new composite design patterns. These can then be checked by the user and asserted in the design repository, or fed into a browser or a tool. Note that regenerating the complete design information can take rather long, and is therefore best performed in batch overnight. However, normally we would only extract new design information for the classes or methods we just implemented. Here we wanted to show how to do a complete synchronization check between the design and implementation. Hence this experiment shows that the design information cannot only be used as explicit documentation and to generate specific parts of the implementation, but also to check whether the implementation is still consistent with the design (and vice versa), and to show which parts differ.

6.4 Supporting direct synchronization

In this second series of experiments, we investigate direct synchronization, where changes of implementation or design are directly propagated to each other. Conceptually, supporting direct synchronization does not differ much from supporting delayed synchronization. The checks we described in previous sections just have to be done whenever something changes. For example, when we change the implementation of a method, a full consistency check between the design and the new implementation should be performed. It is clear that this approach has too much overhead to work in practice. What we need is a way of determining the impact of a small change (in this case the change of one method) in such a way that only the necessary information is rechecked or generated. This, in turn, could then be propagated further if necessary. So, while conceptually there might be no real difference in supporting direct synchronization, we need a practical system of determining the impact of small changes.

Therefore we use the incremental solver introduced in section 3.3. First of all, we use the synchronization tool framework to add special constraints that are dependent on changes in the system. Hence they receive notifications on system changes, and can propagate these changes, firing queries on the way to determine the impact. Hence the setup of these experiments is different from the setup for the

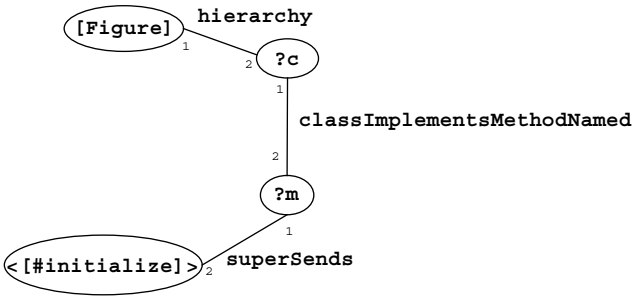

Figure 6.9: Incremental solving example

first series of experiments. In these experiments, the logic repository containing the design is replaced with a constraint network, so that changes can be propagated locally. We will first describe the addition to the incremental solver that allows constraints to be dependent on implementation changes. Then we describe the experiments in supporting the development process directly, giving feedback when a developer changes design or implementation.

Making the incremental solver dependent on system changes

When we introduced the incremental solver in section 3.3, we used a small example constraint network that expresses three relations between two variables. We have depicted the constraint network again in figure 6.9. When explaining the basic workings of the incremental solver, we already showed that the domains of the variables can be changed by the addition of relations. Since in this experiment the incremental solver has to react on changes to the implementation. Therefore we use the *synchronization tool framework* to register some constraints to receive implementation changes. Once they are registered for such changes, they receive notifications on additions, removals and changes in the implementation.

To show this on a concrete example, we create the same constraint network that we used before, but now we let the *classImplementsMethodNamed* constraint receive notifications of method changes. Therefore, every time a method changes in the implementation, this constraint is triggered and can take appropriate action. Because of the definition of the constraint, this consist of checking that, whenever the method is in the *Figure* hierarchy, and it is an initialize method, it has to use a super send. All other methods, or initialize methods in other classes, should not trigger violations. To create this network, we evaluate the following Smalltalk statement:

```
SOULIncrementalSolver new
        name: 'Dependent Network Example';
        add: 'hierarchy([Figure], ?c)';
        add: 'classImplementsMethodNamed(?c, [#initialize], ?m)' method: #m;
        add: 'superSends(?m, <[#initialize]>)'
```

These statements create a new incremental solver and, one by one, add the relations. The only difference is that the *classImplementsMethodNamed* constraint receives all changes in methods. We also indicate that the variable *?m* holds the methods.

At this moment, in our example of HotDraw, 10 subclasses of *Figure* implement a method *initialize*, and all of these methods do a super send. Suppose we now implement a method *initialize* on class *ConstraintVariableFigure* (the newly implemented class from the first series of examples):

initialize

super initialize.
self initInstVars.

When we accept this method, a changed message is triggered that indicates that a new method *initialize* was added to a class named *ConstraintVariableFigure*. This notification is intercepted by the *classImplementsMethodNamed* constraint. Because the change message indicates that it is a method addition, and because it knows which method was added in which class, and because we indicated the variable *?m* to be the method variable, this constraint tries to add this new result to the domain for *?m*. Therefore, it finds the other relations that use variable *?m*. In our example, there is only the *superSends* constraint. This constraint checks whether the newly accepted method indeed does a super send. Since this is the case, it then checks whether it is also a valid solution for *classImplementsMethodNamed* constraint, and if so, whether this changes the domain for variable *?c*. Therefore it constructs and solves a query, filling in the method body of the newly added method:

Query classImplementsMethodNamed([ConstraintVariableFigure],
method([ConstraintVariableFigure],
[#initialize],
arguments(<>),
temporaries(<>),
statements(<send(variable([#super]),[#initialize],<>),
send(variable([#self]),[#initInstVars],<>)>))

The result of this query gives us possible new values for the domain of *?c*, in this case the class *ConstraintVariableFigure*. This new value is checked against the current domain of class *?c*. Because it does not yet exist in this domain, it is added. Since there is another constraint that also uses variable *?c*, this constraint is checked too. Because class *ConstraintVariableFigure* is indeed a subclass of class *Figure*, it is also consistent with that constraint. Since there are no more relations or variables to check, the process stops. At the end of this change, all relations have one extra solution in their result. Changes or removals of methods (or classes) are handled analogously, growing or shrinking solutions of relations, and propagating the changes to dependent relations (that can in their turn grow or shrink their solutions, and propagate the changes further along).

6.4.1 Guiding development

Now that we have discussed the principle of using the incremental solver, it is time to put it to good use. Like we said before, we now want to support the development process directly, giving feedback when a developer changes design or implementation.

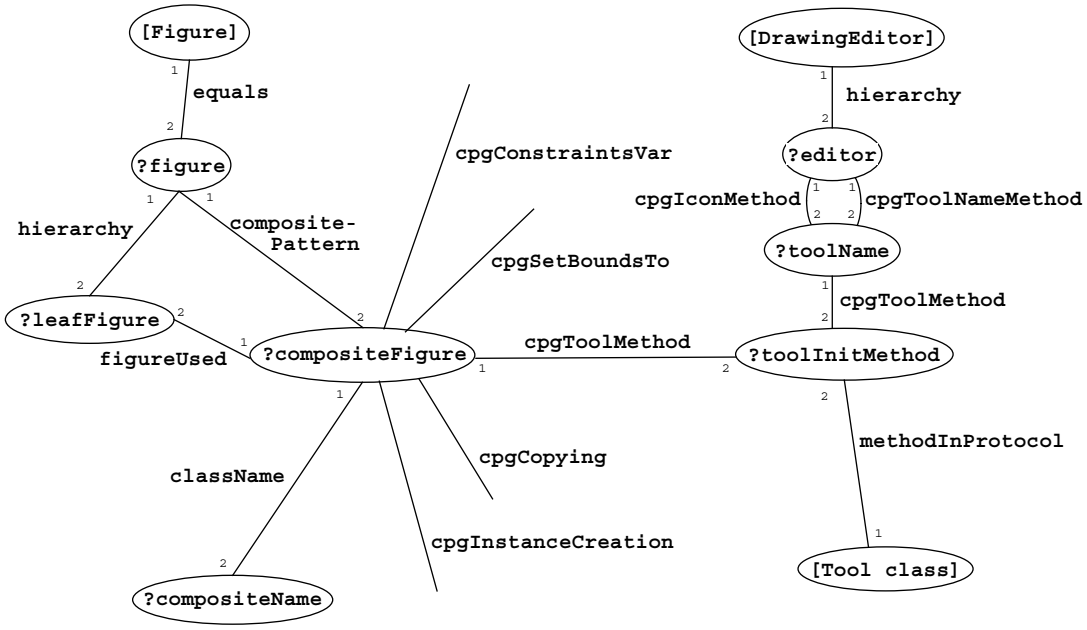

Figure 6.10: Constraint network to support the direct synchronization experiments

The constraint network we use for these experiments expresses the information obtained in the first series of experiments. It is depicted in figure 6.10, and is created by the following Smalltalk expression that, one by one, adds the relations to the network:

```
SOULIncrementalSolver new
        name: 'HotDraw Support Network';
        add: 'equals(?figure, [Figure])';
        add: 'compositePattern(?figure, ?compositeFigure)' class: #compositeFigure;
        add: 'className(?compositeFigure, ?compositeName)';

        add: 'cpgInstanceCreation(?compositeFigure)';
        add: 'cpgConstraintsVar(?compositeFigure)';
        add: 'cpgCopying(?compositeFigure)';
        add: 'cpgSetBoundsTo(?compositeFigure)';
        add: 'figureUsed(?compositeFigure, ?leafFigure)';
        add: 'hierarchy(?figure, ?leafFigure)';

        add: 'hierarchy([DrawingEditor], ?editor)' class: #editor;
        add: 'methodInProtocol([Tool class], [#"tool states"], ?toolInitMethod)' method: #toolInitMethod;
        add: 'cpgToolNameMethod(?editor, ?toolName)';
        add: 'cpgToolMethod(?toolName, _, ?toolInitMethod)';
        add: 'cpgToolMethod(_, ?compositeFigure, ?toolInitMethod)'
```

By creating the constraint network, three important things happen:

1. while building the network, we find values for the variables in the relations. All this information is not stored in one repository under the form of facts (as was the case in the first series of

experiments), but is instead stored in a distributed fashion (since the solver keeps the results locally for each constraint);

2. changes of the implementation trigger the network, which then determine the impact of these changes, and propagates them if needed. This propagation results in updating the design information wherever necessary, possibly checking or deriving additional information;

3. we can change design information by adding or removing results of relations. These changes are again propagated, and checked against the implementation and other design relations.

In the first series of experiments we created a composite figure to show constraint variables. In this experiment we create the *ConstraintFigure* figure to display the relations between those constraint variables. We start by making a change to the design: adding a new solution to the *className* constraint. Of course, there exists no class with that name in the implementation yet. Because of the definition of this constraint, it notifies us that the class does not exist, and fails[2]. As a result, we know that the class does not yet exist, and that we have to add it manually.

So next we open a standard development tool, and create a new subclass of the class *CompositeFigure*, called *ConstraintFigure*. Because this is a change to the implementation, the relations that are dependent of implementation changes are notified. Each of these relations checks whether this change is of interest by invoking a query. If this query fails, the change is not of interest. If the query does not fail, the change is propagated. In our example, only one constraint is interested in the change: the *compositePattern* constraint. The change is propagated, and so the relations on the *?compositeFigure* variable are checked one by one, to see if this new class conforms to them. Several of these relations automatically generate code (more specifically, the ones starting with *cpg*, as explained in section 4.4.1). The *figureUsed* constraint extracts the figures that are used in this composite which, at the moment, is none. Then the *toolInitializeFigure* constraint checks whether this figure has a tool that can create it. Of course, at the moment there is none, so this is reported. So, adding a subclass from *CompositeFigure* in a standard development tool results in an updated design documentation, the automatic generation of some parts of the implementation, and the notification that there is no tool that can create this kind of figure.

The next obvious step is to create a tool that allows our editor to create *ConstraintFigure* figures. Since we have documented this step before, we choose to evaluate a query that helps in doing this, and that generates a default method on class *Tool*:

Query toolInitializationCode([‘ConstraintFigure’], ?code),
generateMethodInProtocol(?code, [Tool class], [#’tool states’]))

Because the generation of the method by the query is a change in the implementation, this change is again intercepted by the constraint network. Now we get a notice by the *hdToolNameMethod* that there is no editor that uses the *ConstraintFigure tool*.

[2]We could also have made this constraint more specific, and automatically let it generate a subclass of *CompositeFigure* with the name we have given.

We then add the new tool to our editor using the following query that uses the *cpgToolNamesMethod* predicate defined in section 6.3.2:

Query cpgToolNamesMethod([ConstraintEditor],<['ConstraintFigure'] >)

This generates the necessary methods to add the tool to our editor. Again the constraint network is notified and updated, and as a result the *cpgIconMethods* constraint generates default icons for this new tool. The design documentation then indeed reflects the implementation. Of course, we now have to manually reimplement some methods in the *ConstraintFigure*, but whenever we change a method, we can be sure that the constraint network checks it to see whether it violates the design documentation we expressed in it, and updates this information when necessary. This concludes our experiment of using the incremental solver to support direct synchronization.

6.5 Experimental validation

We performed the experiments with a clear goal in mind: to show that the synchronization framework allows design to check, generate or enforce implementation, and vice versa. During the experiments, we encountered several problems that needed solving. First of all we came to the conclusion that there is a substantial practical difference between supporting delayed versus direct synchronization. While they are conceptually not that different, the performance penalty for doing a complete conformance check for every small change in design or implementation proves to be not very practical. Therefore we first performed a series of experiments using nothing but delayed synchronization, and then turned our attention to direct synchronization.

Supporting delayed synchronization was feasible and practical with our approach. We showed this using a fairly regular development process that synchronized design and implementation while not constraining either phase too much.

Supporting direct synchronization in such a way that it is usable and scalable in practice proved much harder. We built an incremental solver that allows us to filter and propagate the changes, thereby removing the need to synchronize all of the design information whenever something changed. The constraint network we showed allowed us to monitor changes in the implementation, and gave feedback about these changes. Also, when we changed the design information contained in the network, this was also checked against the implementation.

We now want to review the classifications of synchronization introduced in section 2.5.1, and show that each of the possibilities can be handled by our approach. This enumeration is complementary to the enumeration in section 5.2.3, where we discussed the instantiation of the *synchronization framework* from a general point of view.

direction of synchronization : since we expressed design as a relation in terms of the implementation, this relation can be used for checking (when both design and implementation exist), as shown in 6.3.4. It can also be used to generate one of the two (if the other exists), such as in 6.3.1 and in 6.3.3. Hence the different possibilities for this classification are accounted for;

action to be taken : when elements are found that are out of sync, there are two possibilities: the differences can be reported (allowing the user to take action later), or they can be fixed automatically. In most examples where we ran queries, the results showed us differences between the design information and the implementation. For example, in section 6.3.4 we run a query

that reported the design information which could not be found in the implementation, and another one that reported design information that was extracted from the implementation, but did not exist in the design data repository. We also implemented rules to automatically generate pieces of implementation when they deem this is necessary (for example, the relations starting with *'cpg'* in the direct synchronization experiments in section 6.4.1.

notification time : Making some changes and then doing a conformance check is an example of retroactive notification in delayed synchronization. When doing direct synchronization we used both retroactive notification and reactive notification (depending on the constraint used). We did not show proactive notification, because our system that notified us of changes did so after the changes were applied. However, this is fairly easy to modify such that the system notifies us before the change is actually committed. In that case, we could check relations and, depending on the result, choose to allow the action or refuse it;

trigger time : because the synchronization system is explicit, it can be triggered at any time by anyone. In the first series of experiments we showed delayed synchronization (where synchronization was done after several changes were made to design and/or implementation), and in the second series we supported direct synchronization (where it was the environment that invoked the synchronization system after each small change);

scope : since rules are not bound to certain pieces of the implementation, the rules are global in general. Of course, queries will typically restrict their search to certain hierarchies of classes. For example, in section 6.3.2 we show the *hdEditorClass* rule that uses the *hierarchy* predicate to restrict its work to the *DrawingEditor* hierarchy. In a system with a local scope, this would somehow have been bound explicitly to the class *DrawingEditor*. Another example that is very hard to express in a locally scoped system is discussed in section 6.3.2. The *hdToolMethod* predicate shown there acts on several hierarchies and methods at once. In a locally scoped system, we can only put it in one of those classes, which is not very natural.

implementation granularity : in these experiments we extensively used the fact that the *declarative framework* uses full parse trees. Lots of examples of this can be found throughout the *declarative framework* and the predicates expressed in this chapter. For example, the *hdToolNamesMethod* predicate in section 6.3.2 expresses the structure of the *toolNames* method of a *HotDraw* editor. This would be very hard to express when only partial representation is available;

static/dynamic : In these experiments we only used static information. Dynamic information could have been used mainly for typing, but like we already mentioned, it is not available in the current *synchronization framework*.

6.6 Discussion

6.6.1 Performance and scalability

From the beginning, SOUL was meant to be our experimental logic meta-programming language. As such, its primary concern was to allow us to experiment with a logic programming language integrated with an object-oriented programming language. Therefore, not so much the performance, but the extensibility was stressed. For example, on our 250 mhz Macintosh G3, deducing the type of an instance variable typically takes about one minute and a half, and extracting the UML class

diagram for the complete *Figure* hierarchy takes three hours[3]. While these figures are only meant to give an indication of the performance, we also compared timings for some of our queries with those of Kim Mens. In his dissertation research, that also deals with logic meta-programming, Mens uses a commercial Prolog system to reason about implementation. Whereas in SOUL the base predicates are mapped to Smalltalk objects, in Mens' approach they fetch results from a relational database using ODBC queries. This results in about the same performance penalty. For analogous queries, the commercial Prolog system was about forty times faster than SOUL. We can draw two conclusions from these benchmarks:

- even using the current, non optimized implementation of SOUL, querying the source code using a logic meta-programming language is feasible. While larger queries (with complicated mappings to the source code, such as in our paper expressing software architectures [MWD99]), queries might take longer than an hour, most queries take on the order of minutes. This is too slow to be truly interactive, but the speed of commercial Prolog implementations makes these kinds of queries possible;
- now that we have a clear view of the functionality that a logic meta-programming language should have, we can re-implement it and focus more on performance. The commercial Prolog indicates that the performance gains of such new implementation should be significant.

Last but not least we want to mention the *system dependent cache* that we use to cache the construction of logic representation of methods. As explained in this chapter, the logic programs reason about the source code in a logic form. This means that every time we ask for a particular parse tree of a method, the source code of this method needs to be fetched and parsed, and then the resulting parse tree needs to be enumerated to construct a logic representation of it. To make this more efficient we have implemented a *method cache* that holds a certain number of these method bodies. Noteworthy about the cache is its invalidation mechanism, that depends on system changes. Every time a method is changed or deleted, it is removed from the cache. This invalidation mechanism allows the cache to be very efficient. While we did not implement it, an analogous system could be used for the typing of instance variables, implementing an *instance variable type* cache that gets notified when methods and classes change. This has the potential to dramatically increase the performance for predicates that depend on typing of instance variables (such as the ones expressing UML class diagrams).

6.6.2 Combining direct and delayed synchronization

In the experiments we explicitly divided *direct* synchronization from *delayed* synchronization. For delayed synchronization, we could directly use the logic meta-programming language to synchronize design and implementation. In that setup, the design was stored as clauses in a design repository. For direct synchronization, we used the incremental solver to make the approach scalable and usable in practice. The setup there was to store the design data in a distributed fashion (divided over the relations) instead of putting it in one flat repository. It is important to note that both series of experiments used the same repository of clauses expressing the design notations. Combining them in one system is not very hard: we merely have to allow a network to be used as a repository. This is not very hard, since the network contains the relations, and for each of these relations the results. Therefore, the information in the constraint network can indeed be considered as one repository.

[3]Note that, since Smalltalk is dynamically typed, most of this time is spent trying to type the instance variables of the figures. Therefore, these results cannot be compared with commercial UML case tools that extract class diagrams from typed languages.

6.6.3 Symbiosis versus integration versus stand-alone

In our approach, we express design as a logic meta program of implementation, and then use a logic meta-programming language to synchronize design and implementation. In this section we want to make clear why we want this logic meta-programming language to be integrated with the development environment (and what kinds of synchronization are possible when it is not). We difference between the following modes of integrating a logic meta-programming language and the development environment:

1. *stand-alone*: the development environment and the logic meta-programming language are completely separate entities. They can only interact indirectly, for example through files, and are not aware of each other. Hence the logic meta-programming language has no language features or extra constructs to interact with a development environment. In this setup, we have all the benefits of a logic meta-programming language, but of course the integration is severely limited. As example, take any programming environment and a stand-alone Prolog interpreter. Since there is no relation between the two, the Prolog interpreter needs a logic repository with (aspects of) the source code in logic format. Keeping the source code synchronized with its logic representation has to be done delayed, since the development environment has no guaranteed mechanism to tell the logic interpreter that the implementation has changed. Using the characterizations of synchronization introduced in section 2.5.1, this means that only *delayed triggering* is supported, and that only retroactive notification is possible;

2. *integrated*: the development environment and the logic meta-programming language are aware of each other, and use facilities offered by the operating system to communicate. Typically this means that hooks to some interoperability mechanism provided by the operating system can be used by both the development environment and the logic meta-programming language (such as *AppleScript* under MacOS, or *DDE* and derived technologies on *Windows* systems). In comparison with the stand-alone situation this means that both delayed and direct synchronization can be supported, and that it is easier to keep the source code consistent with the repository containing its logic representation. However, it is still necessary to have a logic representation of the implementation language concepts;

3. *symbiosis*: the development environment and the logic meta-programming language are not only aware of each other, but entities from the development environment can be used directly in the logic meta-programming language and vice versa. To the integrated approach this removes the need for separate logic representations of the concepts reified by the logic meta-programming language.

6.6.4 SOUL and other object-oriented programming languages

SOUL is a logic meta-programming language implemented in Smalltalk, that reasons about Smalltalk source code. In this context, two different questions can be asked regarding other object-oriented programming languages:

1. can SOUL be used to reason about another base language then Smalltalk ?

2. can SOUL be implemented in another language than Smalltalk ?

SOUL reasoning about another base language

The answer to the first question is *yes*. SOUL can indeed be used to reason about another language then Smalltalk. However, note that this means that some of the benefits of the symbiosis are not useful then, and that the rules in the declarative framework need to be changed. Certain is that the rules in the representational layer need to be changed. However, two options are possible. The choice between the two depends on the differences between Smalltalk and the new base language, and on the intended usage:

1. the first option is to keep the same predicates that are currently used, but implement them to work with the base language. This means that the rest of the rules can be kept the same. Possibly these layers can be complemented by predicates that are specific to the new base language, or some groups of predicates might be replaced. For example, when reasoning about a statically typed programming language, the group of predicates implementing the type checking can be re-implemented to take advantage of the explicit types in the base language;

2. the second option is more drastic: change the representational layer. This implies typically that the complete (or at least most) of the predicates in the declarative framework should be rewritten.

Implementing SOUL in another language

Of curse SOUL can be implemented in another language. The core interpretation engine is a stream-based logic interpreter that can be implemented in any general-purpose programming language. The hardest part is probably the implementation of the up-down mechanism that enables symbiosis and reflection. However, there we see some precedents. At the lab, the up-down mechanism was used to implement *Agora*, a reflective, object-based language [Ste94]. The first language was implemented in Smalltalk, but the system was also implemented in C++ and Java [DM98]. Each of these systems required different implementation techniques to implement the reflection, but all were possible. Hence we see no reason why this should pose problems for the SOUL implementation.

Note however that, while SOUL can be implemented in another language, this is much harder for the synchronization framework. This framework extensively uses the facts that Smalltalk is reflective and that the environment is open source. Hence, integrating tools with the environment (a requirement for the synchronization tool framework that needs to capture changes to design and implementation) will be far more difficult.

6.7 Conclusion

This chapter is the first chapter with experiments to validate our claim. It shows experimentally that the synchronization framework indeed supports all the possibilities of synchronization of design and implementation as described in section 2.5.1 (except the *static/dynamic* characterization). Through an extensive case study of the *HotDraw* framework, we show how design can be extracted from the implementation (and vice versa), how to do a conformance check between design and implementation, and how design can be used to guide implementation (and vice versa). The experiments were performed for both *direct* and the *delayed* synchronization. Besides this experimental validation, this chapter also shows the usability of the synchronization framework to make some undocumented and hard to find relations between the *DrawingEditor*, *Figure* and *Tool* classes explicit. These relations are hardcoded in a number of methods on these three classes, and use several naming conventions

and low-level dependencies. Using SOUL we made these conventions explicit and use them to guide development. This shows the practical usability of synchronization, even for a mature and refined framework that formed the basis for several design patterns. In the following chapter we apply the synchronization framework on a large-scale industrial application to assess the usability and scalability in that context.

Chapter 7

Supporting real-world development

In the previous chapter we performed a number of experiments on the *HotDraw* framework. This allowed us to show the overall approach, demonstrating the uses of the synchronization framework in supporting delayed and direct synchronization, and different actions that can be taken. However, these experiments were done on a fairly small scale, and under laboratory conditions. We also wanted to test our system on a large, real-world application to validate its usability and scalability. This chapter discusses the experiments we performed on *MediaGeniX*'s *Whats'On* application, and the lessons learned from these experiments.

7.1 Introduction

The claim made in this dissertation is that expressing design as a logic meta program of implementation constitutes a framework to synchronize design and implementation. Up until now we have developed and tested our approach under laboratory conditions, applying them to small case studies. This showed us that we can support different forms of synchronization as described by the characterizations of synchronization. However, we were unsure whether our approach would be usable in practice. This chapter answers this important applicability question by describing the experiments we did on a large commercial Smalltalk system.

MediaGeniX develops tailor-made broadcast management systems for television stations. Their flagship product is *Whats'On*, that manages television channels, integrating programme scheduling and asset management needs. It centralizes all information related to the planning and management of television broadcasts, such as the schedules, storage media, contracts, license windows, author rights, programme information, press information, viewing figures, ... It is built as a core framework [VV96] of about 2000 classes that is then customized and instantiated for different clients that are located throughout Europe.

For our system we worked on the *Media Management* module of *Whats'On*, that handles everything that has to do with the actual management of the media used for broadcasting, such as tapes. Originally this functionality was offered for one customer only, and was heavily intertwined with the application logic. This dispersion made it very difficult to customize it for other clients that needed similar functionality. Therefore it was completely rewritten as an optional module, that could easily be configured for each client needing this functionality. The resulting *Media Management* module took about 1 man-year to implement, which was done by up to three developers. It consists of 441 classes.

Since the *Media Management* module is one of the newer parts of *Whats'On*, it is one of the first

to use the *MediaGeniX Application Framework* (MAF). The MAF is MediaGeniX' framework for building applications. The rationale to develop the MAF was to allow software developers to build applications easily and rapidly according to MediaGeniX' application design guidelines. The MAF is an elaboration of the application building classes provided by VisualWorks Smalltalk, and consists of five major parts:

1. a framework for building applications that MediaGeniX wants to build;
2. adapted and extended VisualWorks user interface components;
3. new user interface components that are typical for broadcasting software;
4. integration with other frameworks, such as the domain/persistency framework and the permission framework;
5. design and implementation rules for developers.

Besides the definition of the 'look and feel' of the applications MediaGeniX wants to build, the MAF lays down how those applications have to be implemented. In that view, the MAF is a crucial part of the software products of MediaGeniX that are realised in Smalltalk.

We performed two sets of experiments. The first was to synchronize the UML diagrams from the *Media Management* module with the implementation. The second was to make explicit the rules that MAF applications should comply with, and to check existing applications for conformance with these rules. The experiments were performed during 7 days that we stayed at MediaGeniX. Note that we have given timings for each of the queries we performed. The motivation for this is to give an *indication* of how long a certain query took. The queries were done on two machines: a Macintosh Powerbook, with a G3 processor running at 250 megahertz and 64 megabytes of memory, and a PC, with a Pentium running at 266 megahertz, and also featuring 64 megabytes of memory. The environment used was VisualWorks Smalltalk/Envy R4.01.

7.2 Setup of the experiments

To perform the experiments, we used the complete synchronization framework. First of all we instantiated the declarative framework, and extended it with two layers containing the rules specific to these experiments: the rules expressing MediaGeniX specific programming conventions, and the rules expressing the MAF programming conventions. These two layers were added as two separate repositories to the composite repository holding the declarative framework.

7.3 Synchronizing UML diagrams

The first set of experiments was to synchronize the UML diagrams documenting the *MediaManagement* module with the actual *MediaManagement* implementation. Hence the input for this experiment was the UML diagrams of the *MediaMangement* module, and the actual Smalltalk implementation of this module. The goal of the experiment was to assess whether the UML predicates as introduced in section 4.5.3 could be used on real-world UML class diagrams to detect inconsistencies between the diagrams and the implementation. Hence it thus assesses the quality of the UML rules and the usability of the synchronization framework in practice. In this context, we actually performed two kinds of experiments, since we had different implementation versions of the module: version 1.8 was the first

implementation to be released to customers. The latest release we used was 2.28, which contained some enhancements in functionality and bugfixes with respect to the first release. The UML diagrams were used in the design phase, before the first implementation phase. Between version 1.8 and 2.28, they were manually brought inline with the implementation. Hence the UML diagrams were fairly up to date. The goal of the experiments described in this section is to:

1. synchronize the existing UML diagrams against version 1.8 of the *MediaManagement* module;
2. synchronize the existing UML diagrams against version 2.28 of the *MediaManagement* module, and determine whether we can detect design changes between version 1.8 and version 2.28.

7.3.1 Expressing MediaGeniX programming conventions

In section 4.5.3 we saw how the rules expressing UML class diagrams build on the typing rules in order to describe associations. More specifically, associations between UML classifiers are mapped to instance variables of the appropriate type in the implementation. Hence, to use the default UML rules in the declarative framework, the classes should have an instance variable for each association at the UML level.

Throughout the implementation of *Whats'On* however, domain classes need to be *persistent*, meaning their state can be stored and retrieved from databases. Therefore a *persistency framework* is used, which imposes some constraints on the domain classes. First of all, every *domain class* has an associated *storage class* that is responsible for mapping values to an underlying database. Second, the domain classes delegate the retrieval and updating of their state to the storage classes through their accessing methods. As a result, domain classes do not use instance variables, but only provide the accessors which are responsible for getting and setting the values through the associated storage class[1]. This second constraint also implies that throughout *Whats'On*, accessing instance variables should always be done using the accessing methods.

Since there are no explicit instance variables in *Whats'On* domain classes, our rules expressing UML associations are not valid. Hence we need to complement them with rules expressing the specific *MediaGeniX* programming convention used in the persistency framework. We did this in such a way that the general framework was complemented with some rules, and that we did not have to change any rules from the UML layer. First of all we added another *accessorForm* rule, that expresses the specific form of an accessor method used to retrieve the value of an instance variable[2]. Throughout *Whats'On* this is done by sending the message *getValueOf:* to self, passing the name of the instance variable.

[1] The only exceptions to this general rule is the usage of instance variables to cache results in order to improve the efficiency.

[2] This complements the accessorForm predicates introduced in section 4.5.1 that describe direct and lazy implemented accessor methods. Hence the other predicates using accessors also automatically use this new form.

This method is implemented at the root domain class, and is responsible for fetching the value of the persistent instance variable.

```
Rule accessorForm(?method, ?iv, [#mgxPersistent]) if
        equals( ?method,
                method( ?class,
                        ?iv,
                        arguments(<>),
                        temporaries(<>),
                        statements( <return(send(     variable([#self]),
                                                      [#getValueOf:],
                                                      <literal(?iv)>))>))),
        classImplementsMethodNamed(?class, _, ?method).
```

The second predicate we complemented with MediaGeniX specific information is the *instVar* predicate. By default this predicate asks the class for information regarding instance variables. However, for domain classes the accessor method should also be checked since they define *persistent* instance variables. Therefore we introduced the *persistentInstvar* predicate that expresses that we can get the persistent instance variables for a MediaGeniX class by looking at the accessor methods in the *mgxPersistent* form described previously. Then we add another *instVar* rule that expresses that an instance variable can also be a persistent instance variable:

```
Rule persistentInstvar(?class, ?persistentInstvar) if
        mediagenixClass(?class),
        method(?class, ?m),
        accessorForm(?m, ?persistentInstvar, [#mgxPersistent]).

Rule instVar(?class,?var) if
        persistentInstvar(?class, ?var)
```

As we mentioned, the three rules described above were put in a separate layer, that was then added to the repository containing the rest of the declarative framework. This enhanced repository was then ready to reason about the *Media Management* module.

7.3.2 Conformance checking the UML diagrams and the implementation

The *Media Management* module's design comprises 7 UML class diagrams that depict the main classifiers and relations. To give an impression of these diagrams, figure 7.1 shows the one that describes the basic administration of media. We started by doing a conformance check of these diagrams against version 1.8 of the *Media Management* module implementation. We followed the process for conformance checking described in section 6.3.4 of the HotDraw experiments, building a design repository describing the elements in the UML diagram. Once the design repository was built, we checked for each of the elements whether we could find them in the source code. As a result we found minor discrepancies (such as typos in names of classes, instance variables and operations), missing information (such as attributes or operations that could not be found in the implementation). We also found a few larger problems, such as 4 missing classes, some associations between classes that could not be found in the code, and incorrect cardinalities of associations. In general, however, the design diagrams were fairly consistent with the implementation.

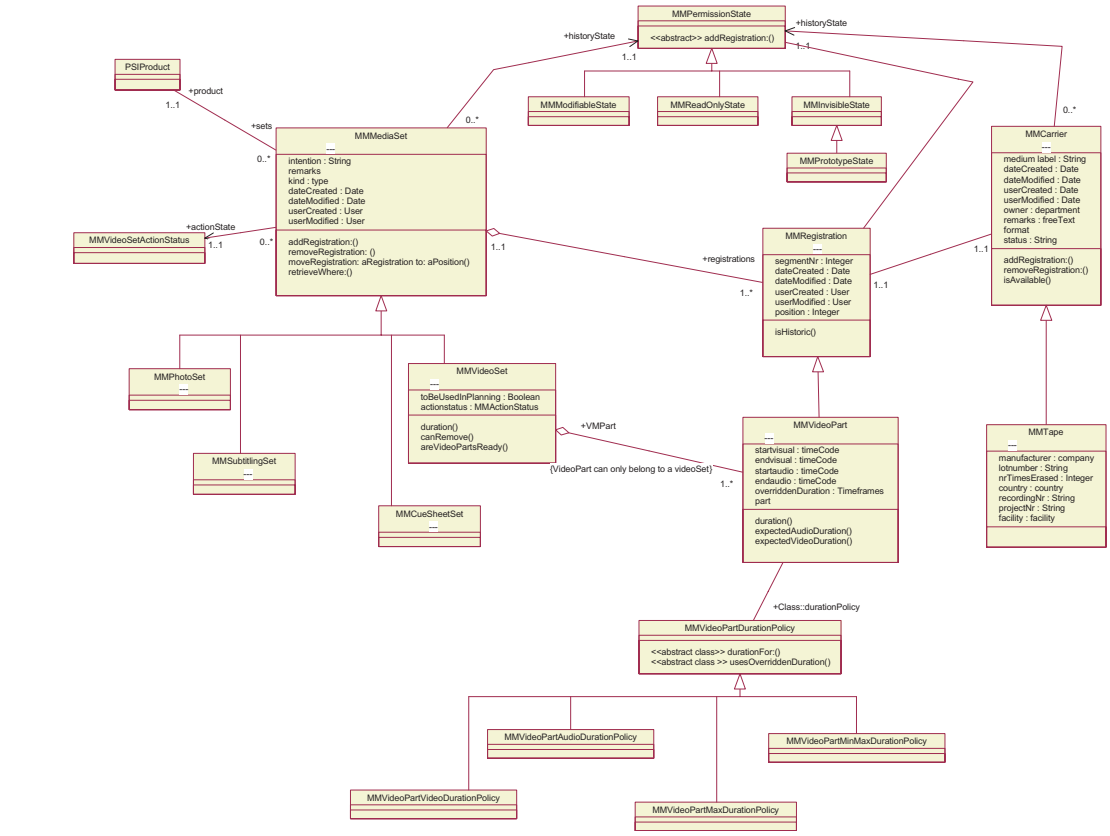

Figure 7.1: Part of the MediaManagement *UML diagrams*

classifier	**attribute**	change
MMCarrierLocationDescriptor		removed
MMCarrierLibraryPositionPolicyAssoc	basicPositionType	added
MMCarrier	locationDescriptor	removed
	parcelItems	added
	isBorrowed	added
	nrTimesErased	added
MMVideoSet	plannersAttention	added
	description	added
MMCarrierLibrary	defaultPositionType	added

Table 7.1: Classifier changes between versions 1.8 and 2.28 of the Media Management *module*

classifier	**association**	change
MMRegistration	owner-1-MMRegistration	added
MMCarrierLibraryPositionPolicyAssoc	positionPolicy-1-Object	added
MMMediaSet	remarks	removed
MMVisioningComment	videoPart	removed
MMCarrier	parcelItems-many-Object	added
	isBorrowed-1-Boolean	added
	nrTimesErased-1-ArithmeticValue	added

Table 7.2: Association changes between versions 1.8 and 2.28 of the Media Management *module*

The other way around we also looked for information we could extract from the implementation and which was missing in the source code. Therefore we extracted association relationships from the code, and compared them against the information in the design repository. While we extracted associations that were not on the design documentation, these proved to be left out intentionally from the design documentation. However, the original documents omitted a lot of role names, which we were able to extract. The result of this phase were some extended UML diagrams,

The overall results of these experiments were an updated set of UML diagrams, where the names of classes and roles now correspond to their implementation counterparts, and where the associations in the UML diagram we could check are consistent with the implementation.

7.3.3 Checking evolution in the implementation

The previous experiments were done with version 1.18 of the *Media Management* module, the first one to be released to the customers. Since then, the implementation has undergone some changes, mainly for maintenance and bugfixing, and for smaller, client specific updates. Since *Media Management* is fairly new, no large updates have been done. Still, in this experiments we want to see if we can find changes in the implementation that are not reflected in the design. Therefore we extracted the same design information as for the first experiment, but using version 2.28 of the *Media Management* implementation. We then compared the extracted information, noting the changes between both. First of all we compared the classifiers and their attributes. The results of this comparison are shown in table 7.1. Afterwards we also compared the extracted associations for both versions, for which the results can be found in table 7.2.

Overall we can say that the extraction process on the new version 2.28 generally gave better results. The reason is that, because functionality was added, more messages were send and thus that there was more information available for our type checking rules. Regarding the differences between the two versions, we found no big differences. This was expected since the implementation did not undergo major changes. We noted however one class that was removed. When checking with the architect of the *Media Management* module this was indeed a refactoring that was done. The removed class actually implemented application behaviour, and was thus not a domain class. Therefore it was removed, and its functionality was moved to a new application.

7.4 Supporting the *MediaGeniX Application Framework*

Together with the main architect of the *MAF*, we made a list of constraints that are imposed on to *Media Management* applications in order for the MAF to function properly. The goal of these experiments is to

1. make these programming conventions explicit;
2. check the source code to find violations against the MAF programming conventions.

7.4.1 Expressing the aspect and domain class rules

The MAF imposes certain constraints on the classes that use it. Currently these constraints are implicit, passed orally between developers. The first part of this experiment therefore was to get the rules (or at least the most important ones) from the architect of the MAF. The resulting list is displayed in figure 7.2. Note that a lot of the rules regulate the usage of *aspects*, a mechanism used in VisualWorks's model-view-controller implementation [KP88] to link models to applications. More specifically, an *aspect* is a model object for a widget provided by an application model. In other words, it provides a customizable channel the views use to get the information they have to display from the model.

Once we had the list, we expressed it as a number of predicates shown in figures 7.3, 7.4 and 7.5. Predicate *initializeAspectsMethod* expresses that the method *initializeAspects*, when implemented on MAF applications (classes in the hierarchy of the class *MAFApplicationModel*), should have a specific form. The first statement in a *initializeAspects* method should be the super send. Then it should be followed only by statements described by the auxiliary predicate *topInitAspectStatement*. The *topInitAspectStatement* predicate describes that an aspect can directly be assigned a value (the first rule), or that it can be assigned a value through a mutator message. In both cases, the statements describing the value that should be assigned are expressed by the auxiliary predicate *initAspectStatement*. This expresses the different values that can be assigned to aspects, both direct and indirect. For example, it takes factory methods and transitive closures of message sends into account.

Note that this heavy use of recursion (necessary to express the transitive closures) that is necessary to express these programming conventions means that less powerful approaches can not fully express these programming conventions. When these transitive closures cannot be expressed, a lot of false information will be detected when we start looking for violations using these rules.

7.4.2 Checking the aspect and domain class rules

Once made explicit, we did several conformance checks of the implementation, to find out where the MAF assumptions where possibly breached in the *Media Management* module code. The first

1. *for each aspect there has to be an aspect method*. An aspect method is an accessor method that returns an aspect. The MAF assumes that these aspect methods do not use lazy initialization (which is the default for aspect methods that are used outside the MAF), and that they are hence pure functional methods that do not store anything in instance variables but just return the aspect;

2. *accessing an aspect should only happen through its aspect method*. MAF applications should always use the aspect methods, and never access the instance variables directly;

3. *all aspects should be bound to instances of ValueModel, MAFSelectionInList, MAFNoteBook or MAFApplicationModel.*

4. *domainobjects should send change messages whenever they are changed*. Because the MAF applications need to know whenever a domain object changes, the domain objects are responsible for sending change messages. Because the persistency framework implements this behaviour, there is no problem for the majority of the domain classes. However, problems might arise when domain classes use instance variables to keep their own intermediary results, without using the persistency framework. This is mostly done by domain classes that provide different views on values of certain instance variables. For example, collections of values can be returned sorted by some key, or filtered. In that case, they should still use the *setValueOf:to* method;

5. *aspectadaptors on domainobjects should have 'subjectSendsUpdates: true'*: In general, a MAF application should not create AspectAdaptors on domain models explicitly. However, in the rare cases they do, they have to indicate to the AspectAdaptor that the domain objects sends updates. This is generally done by sending *subjectSendsUpdates: true* to the aspect adaptor;

6. *aspects should be initialized in the initializeAspects method* and not, for example, in an *initialize* method. The *initializeAspects* method should do a super send;

7. *all enabling and disabling of widgets should be done using MAFEnablingPolicy*. The messages *isEnabled:*, *isVisible:* and *readOnly:* should never be used;

8. initialization of an enabling policy is done in the *setupEnablingPolicy* method. This method should do a super send.

Figure 7.2: MAF constraints imposed on the implementation.

```
Rule initializeAspectsMethod(?class, ?m) if
        classImplementsMethodNamed(?class, [#initializeAspects], ?m),
        hierarchy([MAFApplicationModel], ?class),
        methodStatements(?m, ?stats),
        head(send(variable([#super]), [#initializeAspects], <>), ?stats),
        tail(?initAspects, ?stats),
        forall(     member(?initAspectStatement, ?initAspects),
                    topInitAspectStatement(?class, ?initAspectStatement))

Rule topInitAspectStatement(?class, assign(?leftHand, ?initStat)) if
        initAspectStatement(?class, ?initStat).

Rule topInitAspectStatement(?class, send(variable([#self]), ?mutatorSelector, <?initAspectStatement>)) if
        initAspectStatement(?class, ?initAspectStatement).

Rule topInitAspectStatement(?class, send(variable([#self]), ?delegatedInitSelector, <>)) if
        classImplementsMethodNamed(?class, ?delegatedInitSelector, ?m),
        methodStatements(?m, ?stats),
        forall(     member(?stat, ?stats),
                    topInitAspectStatement(?class, ?stat))
```

Figure 7.3: The initializeAspectsMethod *and* topInitAspectStatement *predicates*

thing we wanted to know was which *MediaManagement* applications violated one of the MAF rules regarding aspect and domain class usage. Therefore we launched a query to enumerate all classes in the *MediaManagement* module that implement an *initializeAspects* method with at least one aspect methods violating the MAF rules:

```
Query     findall(    ?c,
                      and(  mmClass(?c),
                            classImplements(?c,#initializeAspects),
                            not(initializeAspectsMethod(?c,?m))),
                      ?L)
```

Running this query over the implementation gave 22 results, each indicating a possible violation of a MAF rule [3]. Together with the MAF architect we looked through the results, and categorized them according to severity. The results are shown in table 7.3. We found 8 genuine errors, and 2 classes with a dubious implementation of the *initializeAspects* method (ending with the super send instead of starting with it). We also found two implementations that triggered violations, but were actually legacy implementations that were allowed to do this. The last 8 classes did specific aspect initializations which were not captured by our rules and hence resulted in errors. Given some more time, most of these cases could be eliminated by further extending the *initAspectStatement*.

[3] This took 11 minutes and 36 seconds on our testing machine. Note that 51 classes were checked, and that 29 classes passed the tests.

```
"domain object wrapped asValue"
Fact initAspectStatement(?class,send(_, [#asValue], <>)).

"explicit class"
Rule initAspectStatement(_, variable(?className)) if
        className(?aspectClass, ?className),
        validAspectClass(?aspectClass).

"instance creation of explicit class"
Rule initAspectStatement(_, send(variable(?className), _, _)) if
        className(?aspectClass, ?className),
        validAspectClass(?aspectClass).

"recursive"
Rule initAspectStatement(?class, send(?rec, _, _)) if
        initAspectStatement(?class, ?rec).

"factory method"
Rule initAspectStatement(?class, send(variable([#self]), ?facSel, <>)) if
        mafAspectFactoryMethod(?class, ?facSel).

"factory method on class side"
Rule initAspectStatement(?class, send(      send(variable([#self]), ?factoryMethodSelector, '<>),
                                            ?instanceCreationSelector,
                                            _)) if
        mafAspectFactoryMethod(?class, ?factoryMethodSelector).

"instance creation of class returned by factory method on class side"
Rule initAspectStatement(?class, send(      send(send(variable([#self]), [#class],<>),
                                                ?factoryMethodSelector,
                                                <>),
                                            ?instanceCreationSelector,
                                            _)) if
        metaClass(?class, ?mClass),
        mafAspectFactoryMethod(?mClass, ?factoryMethodSelector).
```

Figure 7.4: The initAspectStatement *predicate*

```
Rule mafAspectFactoryMethod(?startClass, ?factorySelector) if
        rootMinusOne(?startClass, ?minusOne),
        flattenedClassImplementsMethodNamed(?startClass, ?minusOne, ?factorySelector, ?m),
        returnStatements(?m, ?returnStatements),
        forall(     member(?stat, ?returnStatements),
                    initAspectStatement(?startClass, ?stat)).

Rule validAspectClass(?class) if
        hierarchy([ValueModel], ?class).

Rule validAspectClass(?class) if
        hierarchy([ApplicationModel], ?class).

Rule validAspectClass(?class) if
        hierarchy([MAFNoteBook], ?class).

Rule validAspectClass(?class) if
        hierarchy([MAFSelectionInList], ?class)
```

Figure 7.5: The mafAspectFactoryMethod *and* validAspectClass *predicates*

severity	**class**
Error	MMAbstractPrototypeListSelector MMActivitySetUpSelector MMCarrierSelector MMPlanningInfoList MMProductAndMediaSetRegistrationSelector MMRegistrationVisioningListSelector MMCarrierLocationDescriptorEditor MMViewProgramHistoryEditor
Dubious	MMParcelEditor MMParcelEditorTask
Legacy	MMCarrierLocationQuerySelector MMSearchParcelEditor
Ghosts	MMCarrierEditor MMAbstractHierarchicalList MMListDataBag MMMediaSetRegistrationCarrierEditor MMProductMediaSetSelector MMSearchParcelTask MMTaskBasedEditor MMVideoSetVisioningEditor

Table 7.3: Media Management classes violating initializeAspectsMethod

class	selector
MMAbstractList	basicListDataBagAspect:
MMParcelSelector	preferredSelectionChannel
MMCarrierSelector	initializeAspects
MMRegistrationOnExistingCarrierSelector	initializeCarrierList
MMLocationSelector	firmSelector
MMParcelEditor	parcelItemSelectionInListStatusChannel
MMSearchParcelEditor	release
MMViewCarrierHistoryEditor	initialize
MMViewProgramHistoryEditor	editeeChanged
	historyListFromProduct
MMCarrierTapeSpecificNoteBookPage	firmSelector
MMSearchParcelTask	initializeAspects
MMBarcodeReader	errorMessage

Table 7.4: MAF applications directly referencing instance variables

We also checked direct sends to instance variables holding aspect methods. Therefore we launched a query enumerating all 1495 methods of the 108 *MediaManagement* applications:

```
Query   hierarchy([MAFApplicationModel], ?c),
        mmClass(?c),
        instVar(?c, ?iv),
        and( classImplementsMethodNamed(?c, ?sel, ?m),
             newIsSendTo(?m, variable(?iv), ?msg, ?args),
             instVar(?c, ?iv),
             not(accessorForm(?m, ?iv, _)))
```

The results of this query are shown in table 7.4[4]. As can be seen, 13 methods directly reference instance variable instead of using their accessors, and are a possible source of bugs. They should be refactored to use the accessor methods for the instance variables they now reference directly.

Last but not least we were interested in finding aspects that used lazy initialization, which should not be done according to the MAF rules. To check this, we use a Visualworks Smalltalk programming convention that aspect methods are in a protocol called *aspects*, and the accessor rules as introduced in section 4.5.1. The resulting query checks all methods in the *aspects* protocol to see if they are in *lazy initialized accessor* format:

```
Query hierarchy([MAFApplicationModel], ?c),
      mmClass(?c),
      and( methodInProtocol(?c, aspects, ?m),
           lazyInitialisedAccessorForm(?m, ?iv, ?kind))
```

From the 145 aspect methods in the applications in the *Media Management* module, we found 18 that use lazy initialization[5]. They are in clear contradiction with the MAF rules, and should be fixed. They are listed in table 7.5

[4]This query took 1 hour and 48 minutes.

[5]The query took just over one minute to run on the testing machine.

class	selector
MMCarrierAndProductSelector	productsStatusBarHolder
	selectionInProducts
MMActivitySetUpSelector	projectChannel
MMParcelSearchResultViewer	selectedDescriptiveParcelItemHolder
MMBrowser	resultList
	resultListDefinitionChannel
MMMediaSetAndRegistrationListView	mediaSetListDefinitionHolder
	mediaSetListHolder
	mediaSetListSelectionInList
	mediaSetListStatusBarHolder
	registrationListDefinitionHolder
	registrationListHolder
	registrationListSelectionInList
	registrationListStatusBarHolder
MMCarrierBrowser	queryEditorBook
MMNewCommandDialog	command
MMNewMediaSetDialog	mediaSetPrototype
MMNewMediaSetDialog	mediaSetType

Table 7.5: MAF lazy initialized aspect methods

7.4.3 Expressing the enablingPolicy rules

Normally, enabling and disabling widgets in an application is done explicitly by sending *isEnabled:*, *isVisible:* and *readOnly:*, passing a boolean to indicate the desired state. Since this can get very complicated for large user interfaces with lots of interface elements that depend on each other's state, the MAF introduces an *MAFEnablingPolicy* class. The purpose of this class is to centralize all information regarding the enablement state of widgets, and making sure that this is handled appropriately. However, this means that the applications using the *MAFEnablingPolicy* class should never use *isEnabled:*, *isVisible:* or *readOnly:* messages themselves, nor the derived methods that send one of these messages internally. Therefore we made this information explicit in some predicates.

The first predicate, *widgetStateChangers* constructs a list of selectors that result in state changes of widgets. One possible implementation could be to explicitly enumerate this list. However, we chose to calculate it, so that possible extensions of widgets would also be included. Therefore we get all the local senders of the message *isEnabled:* and *isVisible:* on the class *WidgetWrapper* and its subclasses. These lists are concatenated, and contain all messages that send *isEnabled:* or *isVisible*. Then we complement this list with the three main messages to change the state (*isEnabled:*, *isVisible:* and *readOnly:*). This list, without duplicates, gives all selectors that change the state of widgets. It is used in the *enablingPolicyViolators* predicate to find all MAF applications that use the *MAFEnablingPolicy* and still send one of these 'forbidden' messages:

Rule widgetStateChangers(?messages) **if**
stLocalSenders([WidgetWrapper], [#'isEnabled:'], ?isEnabledSenders),
stLocalSenders([WidgetWrapper], [#'isVisible:'], ?isVisibleSenders),
append(?isEnabledSenders, ?isVisibleSenders, ?indirectMessages),
append(?indirectMessages, <[#isEnabled:], [#isVisible:], [#readOnly:]>, ?allMessages),
noDups(?allMessages, ?messages).

```
Rule enablingPolicyViolators(?violations) if
        widgetStateChangers(?forbiddenSends),
        findall(    violation(?sel, ?class, ?selector),
                    and(  member(?sel, ?forbiddenSends),
                          stLocalSendersComplete([MAFApplicationModel], ?sel, ?whoWhere),
                          member(<?class, ?selector>, ?whoWhere),
                          classImplements(?class,[#setupEnablingPolicy])),
                    ?violations)
```

7.4.4 Checking the enablingPolicy rules

Using the *enablingPolicyViolators* predicate, we find that the application *MMTaskBasedEditor* sends *disable* to items in its menu. This should be incorporated in the *MAFEnablingPolicy* used in the rest of this class. The other 46 classes that use *MAFEnablingPolicy* do not violate these rules.

7.5 Lessons learned

These experiments of our approach outside laboratory conditions proved worthwhile. It was interesting to use and see the applicability in a real-world context. In this section we want to enumerate general points of interest we learned during these experiments.

- In the *HotDraw* experiments described in chapter 6, we only had some vague information about the implementation to start with. Using SOUL to explore the system we were able to express important design information. In the *MediaGeniX* experiments the process was much simpler because we had access to clear design documents and to the architects and developers. Almost immediately after we started we could therefore extract UML information, and we expressed the MAF programming conventions. So, in order for this approach to work well from the start, the developers or architects should bootstrap the process by supplying as much design information as possible;

- the key point in making the approach efficient in a real world context is reduction of scope. This can be done by making use of the programming conventions, and by pre-filtering irrelevant information using a coarse grained (but efficient and inexpensive) approach, and then finecombing these results with the more expensive full logic programming approach;

- the reasoning power of SOUL was necessary in order to make the MAF programming conventions that deal with aspects explicit. The reason was that in practice transitive closures of methods sends need to be taken into account to express certain programming conventions. This is hard to express in approaches that do not support recursion (such as SmallLint, a tool that one might think could be used to support these programming conventions). However, to express some of the other programming conventions, no recursion is necessary and hence less powerful but faster reasoning engines could be used (such as SmallLint). We discuss this issue further in the future work in section 8.3.3, because we would like to combine solvers of different expressive power and performance.

- during the experiments with the UML schemas we lacked integration with the UML tool used regularly in the development process. As a result, we had to manually 'translate' the UML

schemes from their graphic description to our logic description. Vice versa, we had to manually update the UML diagrams with the results from our extraction process. So, in order to be fully usable in a practical setting, we have to integrate SOUL not only with the development environment, but also with external tools. This is possible by writing scripts in tools that support this, or using the interoperability mechanisms offered by operating systems (such as *AppleScript* under MacOS, or *DDE* and derived technologies on *Windows* systems).

- the declarative framework is the key mechanism in being able to adapt quickly to different implementations. As we saw in section 7.3.1, being able to complement the general rules with *MediaGeniX* specific rules meant we could reuse the declarative framework. Actually, two features are necessary: a composition mechanism of repositories, and a mechanism that clauses can easily use and override other clauses. While SOUL allows the first, the latter is currently very primitive. We discuss this together with other extensions of SOUL in section 8.3.2 of the future work.

7.6 Conclusion

This chapter describes the experiments we performed in a real-world context. It shows that the synchronization framework can be used in a practical setting to synchronize design and implementation. In a limited period of time we successfully applied the synchronization framework to express and synchronize design information with an implementation. More specifically, we did a conformance check of existing UML diagrams with the released implementation. We found some discrepancies between the two, most notable some classes and relations in the UML diagram that did not exist in the implementation. Also, we were able to complement the UML diagrams with information we extracted, most notably role names for associations. We also checked the evolution of the implementation with respect to this UML diagram. Besides these experiments with UML diagrams, we also made a set of programming conventions explicit, and used this to find violations against these programming conventions in the implementation. The results of these checks where a number of clear errors in some parts of the implementation that do not follow the programming conventions.

Overall, the experiments on *HotDraw* and *Whats'On* showed that the rules in the declarative framework, although lightweight, can be used to successfully express the design used in a particular context and that the synchronization framework successfully synchronizes design and implementation. In the next chapter we conclude this dissertation, list the major contributions and discuss future work.

Chapter 8

Conclusion and future work

8.1 Conclusion

The goal of this dissertation is to pave the road towards support for co-evolution. The thesis statement we defended was the following:

> **Thesis**
> *A framework for co-evolution of design and implementation, where design and implementation are related in such a way that the one can check, generate or constrain the other, can be achieved in a logic meta-programming language integrated with a software development environment.*

When we looked at supporting co-evolution, we came to the constatation that in order to support co-evolution, we had to synchronize the changes between design and implementation. Therefore, we first of all investigated the *characteristics* of synchronization between design and implementation, and we found that there is actually a broad spectrum. To describe this spectrum, we used the following characterization, which we applied on related work: *direction*, *action*, *notification time*, *trigger time*, *scope*, *implementation granularity* and *static/dynamic*. To construct a framework that supports all of these characteristics, we proposed and defended the following conceptual solution:

1. make the relation between design and implementation explicit by expressing design as a logic meta-program of implementation;

2. integrate the logic-meta programming language in the environment to capture changes of design and implementation;

3. use the logic meta-programming language to find differences between design and implementation, and define actions (SOUL).

We showed that this conceptual solution indeed supports all the characterizations. However, we also implemented a software artefact to show that this solution is not only conceptual, but can also be used in practice. The software artefact is called the *synchronization framework*. It is composed of two individual frameworks: the *declarative framework* that expresses design as a logic meta programof implementation, and the *synchronization tool framework*, that integrates synchronization tools

in the development environment. Last but not least, we also implemented a reflective logic meta-programming language that exploits its symbiosis with the base language to reason directly over the implementation of the base programs.

The synchronization framework was then applied to two different case studies to show its usability and scalability in practice. First it was experimentally shown on a smaller case study (the HotDraw drawing editor framework) that the synchronization framework indeed supports all different characterizations of synchronization. On the same case study, we also found that, even for the well-known and well-documented framework HotDraw is, there is need for synchronization of design and implementation and the synchronization framework can do so. Besides the experiments on HotDraw we also did experiments on a large industrial framework. Here we did conformance checks of UML diagrams against the implementation (complementing the diagrams with extracted information and detecting differences between the UML diagrams and the implementation). We also expressed programming conventions and found several violations in the implementation that needed to be fixed. These experiments strengthened our claim that the synchronization framework is usable in practice, and showed that it is scalable.

Overall, the conceptual and experimental validation proved our thesis statement, and the experiments showed the usability and scalability of our software artefact that implements our solution.

8.2 Contributions

While proving our claim, the following contributions were made:

- the first contribution is the study and characterization of synchronization mechanisms. These characterizations are used as the key variation points of our synchronization framework;

- the second contribution is the design of the logic meta-programming language, and more specifically its symbiosis with the underlying implementation language. This symbiosis allows the logic meta-programming language to wrap or evaluate expressions in the implementation language during the logic interpretation process;

- the third contribution is the incremental solver we built using techniques from the incremental symbolic constraint solving community. The main idea is to use the logic meta-programming language to express and solve the relations, and to use local propagation techniques to incrementally solve the network;

- the fourth contribution is the *synchronization framework* itself, that is used to build tools that need synchronization of design and implementation. It consists of the *declarative framework* and the *synchronization tool framework*. The declarative framework is a logic meta-programming framework that is used to map design to implementation in an explicit, customizable and expressive way. The synchronization tool framework is a Smalltalk framework that allows to monitor and act upon any change to design and implementation.

8.3 Future Work

8.3.1 Refining the synchronization framework

First of all we want to apply the synchronization framework to more cases. The goal is to ameliorate the synchronization framework, as this can only be done by applying it more. For the declarative framework, this should result in an extended set of logic meta programs expressing design. Possibly we can add some popular design notations, and extend the current ones. For the synchronization tool framework we want to fully implement the pro-active notification mechanism, and experiment with it.

8.3.2 SOUL-2

Because a suitable logic programming language integrated in a development environment did not exist, we implemented SOUL. SOUL has several very important and non-trivial features we want to keep (especially the symbiosis with Smalltalk). However, now that we have a better view on the requirements for a logic meta-programming language integrated in a development environment, we want to implement a new and improved version of SOUL, called SOUL-2. More specifically, we want to tackle the following areas:

- performance: the inference mechanism in the current version of SOUL is completely stream-based. While this had certain advantages when we started our experiments, this implementation is rather slow. As we already indicated in section 6.6.1, we have a performance loss of a factor of 40 when compared with commercial Prolog interpreters. So, we want to implement a new stack-based interpreter in order to boost the performance;

- incremental solver: we want to reimplement the incremental solver, using the experience gained by the first implementation and by the experiments. The resulting incremental solver can then be truly multi-way, and far more configurable with respect to constraint violations then the current one;

- SOUL as base language: Clauses written in SOUL are built using certain programming conventions and are also subject to evolution. Therefore we would like SOUL to also support SOUL as base language, next to object-oriented programming languages that are currently supported;

- further symbiosis with Smalltalk: in its current version, SOUL has a symbiosis with Smalltalk that allows full introspection and even reflection. We would like to extend this further. For example, we already performed experiments to write SOUL code in any Smalltalk method. When such Smalltalk methods are executed, and the SOUL clauses are encountered, they are evaluated by the SOUL interpreter, passing the context of the method. When this integration is completed, we will have a language with full reflection between an object-oriented programming language and a logic programming language. Compared with SOUL in its current state, this would mean for example that there would be a causal connection between Smalltalk methods and their logic form. This would make rewriting code much easier. It would also mean that dynamic information is supported;

- repository composition mechanism: In SOUL we can compose repositories using a nesting mechanism. This proved very important in practice. However, we would like to go much further in this context, and have a real composition mechanism on repositories. The idea is to have an object-oriented programming language like late-binding mechanism, where repositories

can be parametrized with other repositories. A crude, experimental version of such system was implemented that allows terms to be prefixed by a *connector variable* that has to be bound with a repository at interpretation-time. For example, this system allows us to write a rule in a repository that has a *basicLayer* connector that asks the repository bound to this connector for the *classImplements* predicate:

```
Rule sameSelectors(?c, ?d, ?sel) if
        @basicLayer.classImplements(?c, ?sel),
        @basicLayer.classImplements(?d, ?sel)
```

The *connector variables* have to be bound to a repository when it is interpreted.

8.3.3 Combining solvers

In the related work described in section 2.5.2, it is clear that there is a trade-off between *expressivity and computation power* versus *performance*. For example, an approach based on regular expressions allows to express fairly complicated patterns, and is performant. On the other, using SOUL we express much more complicated patterns (using abstraction mechanisms and recursion), but the interpretation is slower. Therefore, one way to increase performance is to integrate several solvers in one interpretation engine. The idea is to use the fastest approach possible depending on the expressivity that is needed. We implemented a number of ad-hoc rules that already do this, for example the rules implementing pattern matching operations (see the logic layer described in section 4.2). This combination should be refined and extended, and promises a high expressivity while remaining performant.

8.3.4 Full co-evolution support

In this dissertation we limited ourselves to the synchronization design and implementation, where the mapping between both has to be made explicit in logic meta programs. However, in order to fully support co-evolution, we have to take other phases in the development cycle into account and help the definition of the mapping between such phases.

First of all, we would like to generalize the approach as proposed in this dissertation to other cycles in the development process. More specifically, we think of supporting use-cases as well. Therefore, uses-cases need to be made explicit as well, and changes to use-cases have to be propagated to the design and vice versa.

Second, there should be support for building and maintaining the mapping between design and implementation. The synchronization framework assumes that the mapping is available, and uses it to determine the impact of changes to the design level on the implementation and vice versa. However, there is no real support for implementing and maintaining this mapping. The developer has to be an expert when describing the mappings, and a lot of experiments are needed in order to validate it. We would like to investigate whether this process can be simplified and supported. One research direction that is worth investigating is to use knowledge representation and machine learning techniques to help implementing and maintaining the mapping. For example, we want to have a mechanism where we provide the design and an implementation that conforms to this design, and where the mapping is extracted semi-automatically.

Bibliography

[BAFB96] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: a local propagation algorithm for inequality constraints. In *Proceedings of the ACM symposium on User interface software and technology*, pp. 129–136, 1996.

[BB98] G. J. Badros and A. Borning. The cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW Tech Report 98-06-04, University of Washington, 1998.

[BDMDV00] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. In *Proceedings of the ECOOP'2000 Workshop on Aspects and Dimensions of Concerns*, 2000.

[Bec97] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[BFB95] A. Borning and B. N. Freeman-Benson. The oti constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pp. 624–628. Springer LNCS 976, September 1995.

[BJ94] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, volume 821 of *LNCS*, pp. 139–149. Springer-Verlag, July 1994.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, 1996.

[Bok99] B. Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in java. In *Proceedings of ESEC/FSE'99*. Springer-Verlag, September 1999.

[Bor79] A. H. Borning. *Thinglab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, July 1979. Also available as Stanford Computer Science Department report STAN-CS-79-746 and as XEROX Palo Alto Research Center report SSL-79-3.

[Bra92] J. Brant. Hotdraw. Master's thesis, University of Illinois, 1992.

[Bri00a] J. Brichau. Declarative composable aspects. In *Proceedings of the ECOOP'2000 Workshop on Advanced Separation of Concerns*, 2000.

[Bri00b] J. Brichau. Declarative meta programming for a language extensibility mechanism. In *Proceedings of the ECOOP'2000 Workshop on Reflection and Meta Level Architectures*, 2000.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Method Language User Guide*. Addison-Wesley, 1997.

[Bro96] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 1996. TR-96-07.

[Bud94] D. Budgen. *Software Design*. Addison-Wesley, 1994.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Cha94] I. Chai. How to create a new figure with constraints. Technical report, University of Illinois, 1994.

[CM81] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

[Coh90] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33:52–68, 1990.

[Cop98] J. O. Coplien. C++ idioms. In J. Coldewey and P. Dyson, editors, *Proceedings of the Third European Conference on Pattern Languages of Programming*, 1998. To be published.

[Cre97] R. F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[DD99] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In *Proceedings of the ECOOP99 Aspect Oriented Programming Workshop*, 1999.

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

[DDVMW00] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *To appear in the proceedings of the international symposium on Software Architectures and Component Technology 2000.*, 2000.

[Def99] A. Defaweux. Modelling co-evolutionary algorythms through coupled fitness landscapes. Master's thesis, Vrije Universiteit Brussel, 1999.

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

[DM98] W. De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*. Springer Verlag, 1998.

[DV98] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[DVD99] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, LNCS 1616, pp. 250–272. Springer-Verlag, 1999.

[DVFW00] K. De Volder, J. Fabry, and R. Wuyts. Logic meta components as a generic component model. In *Proceedings of the ECOOP'2000: Fifth International Workshop on Component-Oriented Programming*, 2000.

[DW00] D. Deridder and B. Wouters. The use of an ontology to support a coupling between software models and implementation. In *Proceedings of the ECOOP'2000 Workshop on International Workshop on Model Engineering*, 2000.

[FB89] B. N. Freeman-Benson. A module mechanism for constraints in smalltalk. In *OOPSLA 89 Proceedings*, pp. 389–396, 1989.

[FBMB90] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33, Issue 1:54–63, 1990.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *OOPSLA 89 Proceedings*, pp. 327–335, 1989.

[FMvW97] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In M. Aksit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pp. 472–495, Jyväskylä, Finland, 9–13 June 1997. Springer.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.

[GM90] P. Gray and Mohamed. *Smalltalk-80 : A Practical Introduction*. Pitman, 1990.

[GR95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management*. Addisson-Wesley, 1995.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora (eds.)*, volume I. World Scientific Publishing, 1993.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28 of *ACM SIGPLAN Notices*, pp. 411–428. ACM Press, Oct. 1993.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[JL87] J. Jaffar and J. Lassez. Constraint logic programming. In *Proceedings of the fourteenth ACM Symposium of the Principles of Programming Languages*, pp. 111–119, 1987.

[Joh77] S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, Dec. 1977.

[Joh92] R. E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27-10, pp. 63–76, 1992.

[Jon87] W. C. Jones. *Modula2: Problem Solving and Programming with Style*. Harper & Row, 1987.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, pp. 220–242. Springer Verlag, 1997. LNCS 1241.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Technical report, Palo Alto, 1988. IB-D913170.

[Lew95] S. Lewis. *The art and science of Smalltalk*. Hewlett-Packard professional books, 1995.

[LH89] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pp. 38–48, September 1989.

[Luc97] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1997.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Dept. of Computer Science, AI-Lab, Vrije Universiteit Brussel, Belgium, 1987.

[MDR93] S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, Apr. 1993.

[Mei96] M. Meijers. Tool support for object-oriented design patterns. Master's thesis, Utrecht University, August 1996.

[Men99] T. Mens. *A Formal Foundation for Object-Oriented Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.

[Men00] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science, C.A.R. Hoare, Series Editor. Prentice Hall, 1988.

[Mey00] B. Meyer. *Eiffel : The Language*. Prentice Hal, 2000.

[Min96] N. H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.

[MMW00] T. Mens, K. Mens, and R. Wuyts. On the use of declarative meta programming for managing architectural software evolution. In *Proceedings of the ECOOP'2000 Workshop on Object-Oriented Architectural Evolution*, June 2000.

[MN95] G. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 116–127. ACM Press, 1995.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18–28. ACM Press, 1995.

[MP97] N. H. Minsky and P. P. Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Systems*, 3(2):87–101, 1997.

[Mur96] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[MWD99] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pp. 33–45, June 1999.

[Pfl98] S. L. Pfleeger. *Software Engineering : theory and practice*. Prentice-Hall, 1998.

[PW92] D. Perry and A. Wolf. Foundations for the study of software architectures. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[RBJO96] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[RDW98] T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs with declarative event analysis. In *Proceedings of ECOOP'98 Reverse Engineering Workshop*, June 1998.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Rob99] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana Champaign, 1999.

[RWL96] T. Reenskaug, P. Wold, and O. Lehne. *Working with objects: the Ooram software engineering method*. Manning Publications, Greenwich, CT, 1996.

[San94] M. Sannella. The skyblue constraint solver and its applications. In *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*. MIT Press, 1994.

[SG96] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pp. 268–285. ACM Press, 1996.

[SMFBB93] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software–Practice and Experience*, 23 No. 5:529–566, May 1993.

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley, 1996.

[SS88] L. Sterling and E. Shapiro. *The art of Prolog*. The MIT Press, Cambridge, 1988.

[Ste94] P. Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.

[TDM99] T. Tourwé and W. De Meuter. Optimizing object-oriented languages through architectural transformations. In *8th International Conference on Compiler Construction*, pp. 244–258, 1999.

[Tho94] J. Thompson. *The Coevolutionary Process*. University of Chicago Press, 1994.

[Tou00] T. Tourwé. Framework optimization through declarative program specialization. Technical report, Vrije Universiteit Brussel, 2000.

[VV96] A. Vercammen and W. Verachtert. Psi: From custom developed application to domain specific framework. In *Addendum to the proceedings of OOPSLA '96*, 1996.

[WDVP00] B. Wouters, D. Deridder, and E. Van Paesschen. The use of ontologies as a backbone for use case management. In *Proceedings of the ECOOP'2000 Workshop on Objects and Classifications, a natural convergence*, 2000.

[Web96] *The New International Webster's Comprehensive Dictionary of the English Language*. J.G. Ferguson Publising Company, 1996.

[Wuy96] R. Wuyts. Class-management using logical queries, application of a reflective user interface builder. In I. Polak, editor, *Proceedings of GRONICS '96*, pp. 61–67, 1996.

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

[Wuy00] R. Wuyts. *The Implementation and Usage of SOUL.* Vrije Universiteit Brussel, Brussels, Belgium, 2000.