Université de Nantes
École doctorale
Sciences et Technologies
de l'Information et des Matériaux
Année 2004

Universidad de Chile
Escuela de postgrado
Facultad de Ciencias
Físicas y Matemáticas
Año 2004

PhD thesis

of the University of Nantes, France
and the University of Chile, Chile

in Computer Science
presented and defended publicly by

Éric Tanter

on November 8th, 2004
at the Computer Science Department of the University of Chile

From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming

in front of the jury composed of:

Reporters:	Shigeru Chiba	Associate Professor, Tokyo Institute of Technology, Japan
	Theo D'Hondt	Professor, Vrije Universiteit Brussel, Belgium
Examiners:	Nancy Hitschfeld	Assistant Professor, Universidad de Chile, Chile
	Éric Monfroy	Professor, Université de Nantes, France
	Jacques Noyé	Assistant Professor, École des Mines de Nantes, France
	Gustavo Rossi	Professor, Universidad Nacional de La Plata, Argentina
Advisors:	Pierre Cointe	Professor, École des Mines de Nantes, France
	José Piquer	Associate Professor, Universidad de Chile, Chile

Hosting group and laboratory in France
OBASCO Project – EMN / INRIA
LINA / EMN
University of Nantes

Hosting group and laboratory in Chile
Center for Web Research
Computer science department
University of Chile

Acknowledgments

First of all, I am grateful to the institutions that made this PhD work possible: the computer science departments of the University of Chile and of the Ecole des Mines de Nantes, on behalf of the University of Nantes. I am especially thankful to the Milenium Nucleous Center for Web Research, Mideplan (Chile), for providing me a generous grant that contributed to making this research experience pleasant, and the EMN-INRIA (France) OBASCO project for welcoming the possibility of a co-advised thesis by financing various travels between Chile and France, as well as assistance to some conferences. This PhD work is also debtful to: the Accessnova program (Chile), for financing this work in its early phase when I had no grant; CONICYT (Chile), for providing financial support in various opportunities, in particular to participate to conferences; the CONICYT-INRIA (Chile-France) ProXiMoS and OSCAR projects, which have made it possible to enjoy a fruitful collaboration with the OASIS project at the INRIA Sophia Antipolis in France; and more recently, the Chile-Korea IT Collaboration Center, to which I wish the best.

I am really thankful to my thesis advisors, José Piquer and Pierre Cointe. Pierre is the person to blame for my initial fascination for reflection and metaprogramming. José has made considerable efforts to make it possible for me to attend conferences and other events all around the world, in spite of many difficulties: without this support my experience and interest in research would have been greatly reduced.

I am overwhelmed with gratitude for the inconditional support provided by Jacques Noyé all along these four years. Although Jacques does not get any official honor for my PhD thesis, he is the most fundamental person in this work (our common publications are the best testimony of this fact). I have learnt enormously by interacting with you, on topics as vast as critical mind, self questioning, research attitude, scientific writing and ethics. I sincerely wish our collaboration will continue for long.

I would like to thank the different students I had the honor of advising since I started my PhD: Marc Ségura-Devillechaise, Michael Vernaillen, Peter Ebraert, François Nollen, Angel Núñez and Rodolfo Toledo, as well as the students of the courses I gave. Teaching and advising are definitely rich activities whereby one learns as much, if not more, than he teaches.

I am really grateful to the different colleagues –should I say friends– with whom I collaborated in various opportunities: Isabelle Attali, Noury Bouraqadi, Denis Caromel,

Shigeru Chiba, Stéphane Ducasse, Rémi Douence, Luis Mateu, Leonardo Rodríguez, Marc Ségura-Devillechaise, Michiaki Tatsubori, Julien Vayssière, and many others, from EMN, INRIA, DCC, VUB, and other places around the world. These interactions made research look like a “living” activity to me. I would certainly not have been able to survive these years of research without them, and I really hope that they will go on and that new ones will appear.

I finally would like to express my gratitude to all the members of my jury, for their efforts in reviewing and commenting on this work, as well as for coming, most from far away, to Santiago in order to attend my defense.

Last but not least, my warmest feelings go to my family and friends. I know that having a son living at the other corner of the world is difficult for parents. I just want to say that you are the ones who gave me the strength to pursue my own path, beyond pre-established roads: this is definitely the most valuable gift parents can offer their children. Amongst my friends, which I cannot fully enumerate here, I would like to deeply thank the crazy ones that are, or have been, sharing the “Chilean experience”: Louis Foucart, Nicolas Mroczko, Olivier Motelet, and Guillaume Pothier. You do not know how important you are to me.

"The emotional reaction is all that matters.
As long as there is some feeling of communication,
it isn’t necessary that it be understood."
– John Coltrane, 1964

Contents

Chapter 1

Introduction

Programming languages are at the heart of software engineering. Since the advent of high-level programming languages, all improvements have been motivated by the need to be able to build better software more rapidly. What "better software" actually means is not formally defined, although many good properties of software have been identified (*a.k.a.*, the "-ities"), among which the crucial ones are understandability, maintainability, reusability, and evolvability. These properties are tightly linked to the issue of modularization, which can in essence be seen as the possibility to cleanly encapsulate concerns of the software in separate modules. A module is basically a work assignment, whose role is to localize and hide design decisions. This principle is known as the Separation Of Concerns (SOC) principle [Dijkstra, 1968; Parnas, 1972].

Research in programming languages therefore aims to provide proper means of separating concerns into modules. Object-oriented programming (OOP) was a major breakthrough in the history of modularization, by promoting the encapsulation of both code and state in a given modular unit, called an object. However, during the last two decades, with the convergence and massification of information technologies, object-oriented programming has been pushed to its limits in more and more complex software systems as its industrial acceptance has grown [Hayes, 2003]. In complex situations, it is hard to ensure proper modularization of a piece of software, thereby compromising its desired properties. Furthermore, with the advent of ubiquitous computing, a piece of software is no longer meant to be used in a known, fixed context. New requirements for *adapting* software to changes in its execution context, possibly dynamically, have naturally started to appear [Weiser, 1993; Cheverst *et al.*, 2000]. As a matter of fact, being able to adapt concerns of an application intrinsically depends on the possibility of properly modularizing such concerns. The time at which adaptation can be performed is then a matter of the time at which the binding [Halpern, 1993] between modules is done.

In this context, various approaches to software modularization and adaptation together with supporting technologies have been explored, and are still under exploration. Among these approaches, a notable attempt has been made with the study of *computational reflection* [Smith, 1982]: the ability of a program to describe and modify, at the *metalevel*, its own state and behavior. Computational reflection is a very general approach, which originally focused on program adaptation and has been progressively applied to separation of concerns [Zimmermann, 1996].

Problem Statement

The acceptance of reflection as a major modularization technology has been limited, due to several issues related to its applicability:

- Reflective computation is expensive: apart from optimizing the implementation of reflective mechanisms as such, it is necessary to allow for the selective use of reflection.
- In OOP, reflection is accessed via metaobject protocols (MOPs) [Kiczales *et al.*, 1991], which are paradoxically hardwired and hence not adequate in all situations. Since developing reflective systems from scratch is a demanding task, apart from being expressive and flexible, reflective systems should be tailorable to specific needs.
- Using reflection is complex: appropriate means should be provided in order to be able to properly *structure* the metalevel and control the genericity of this approach.

Several research efforts have been made to address them, but they have been incomplete with regard to the three points highlighted above, usually by not considering each of them simultaneously. Other efforts have resulted in the formulation of new approaches, in particular Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 1997b; Elrad *et al.*, 2001], which is deeply rooted in reflection. Still, the specificity of AOP contrasts with the genericity of reflection. Furthermore, although AOP provides more adequate support for modularization by advocating the use of dedicated aspect languages, most AOP proposals do so by sacrificing flexibility and extensibility.

Thesis

This thesis reconciles both reflection and AOP by proposing a model of reflection that supports both the genericity of reflective systems and the specificity of AOP. Furthermore, in order to combine the power and generality of reflection and the guidance of aspect languages, we propose a versatile substrate for AOP based on an extension of our model of reflection. A prototype implementation and significant applications validate our approach.

Opening reflective systems. Considering as a reflective system the part of a language environment that allows for reflective programming, as well as extensions that complement the reflective abilities of a language, *opening* a reflective system means exposing specific interfaces for its adjustment. We provide insights on how to open reflective systems, by iteratively refining the customization interfaces provided in particular to the roles of *metalevel architect* and *assembler*. This work is in the line of the Open Implementation research area [Rao, 1991; Kiczales, 1992; Kiczales *et al.*, 1997a].

Partial reflection. We propose a model of reflection that paves the way to selective, fine-grained, and flexible reflection. The model includes *links* as first-class entities representing the binding between base programs and metaobjects. The precise definition of links addresses efficiency by applying reflection only at appropriate places, implementing a form of *partial*

reflection. It also tackles complexity by supporting fine-grained specialization of metaobject protocols and flexible structure of the metalevel. This model is first proposed in the context of behavioral reflection (*i.e.* reflection about the execution of a program). It is later complemented with structural reflection in a unified model.

Versatile kernels for aspect-oriented programming. In order to combine the power and generality of reflection and the guidance of aspect languages, we introduce the idea of a versatile substrate for AOP, making it possible to combine various approaches to AOP. We extend our model of partial reflection to include support for the detection and resolution of interactions among links. By appropriately supporting the definition of various aspect languages, such a kernel makes it possible to experiment with different approaches to AOP and to compose aspects written in different (possibly domain-specific) aspect languages.

Reflex and applications. An open implementation of our model of partial reflection, tailorable and extensible, is developed for Java. It is then evolved into a versatile kernel for AOP. Furthermore, significant applications of our prototype in different contexts, such as distributed and concurrent programming, empirically validate our argument.

As a result, our proposal bridges the gap between metaobject protocols and AOP in a way that is fruitful to both.

How to Read this Dissertation

Chapter 2 is dedicated to the presentation of the necessary background concepts and to the discussion of existing related work in the area of reflection (Sections 2.1 to 2.4), open implementations (Section 2.5) and aspect-oriented programming (Section 2.6). This chapter concludes this introduction by elaborating on the problem statement of this thesis work (Section 2.7).

The core of this dissertation is then divided in two parts. Part I presents the steps of this work related to the opening and enhancement of runtime behavioral reflection (Chapters 3 and 4), while Part II addresses the connection of our work to aspect-oriented programming (Chapters 5 to 8).

Chapter 3 presents the first step of our work: motivating the need to open reflective system, and proposing an initial approach. This approach relies on opening the transformation process so that a metalevel architect can specialize MOPs.

Chapter 4 represents a major step towards our objective: we expose a model of partial behavioral reflection and refine the customization interfaces offered to metalevel architects and assemblers, so that they are finer-grained and easier to use. This chapter highlights the underlying connection between partial behavioral reflection and aspect-oriented programming, which we further explore in the following chapters.

Chapter 5 reports on a case study to support dynamic crosscutting –a major feature of aspect-oriented programming– within partial behavioral reflection. It suggests slight enhancements to the model presented in Chapter 4 and illustrates their use.

In the next three chapters, we dive into our proposal of a common substrate for aspect-oriented programming, called a versatile AOP kernel. Chapter 6 motivates the need for such a kernel and establishes a set of requirements that an AOP kernel should fulfill.

Chapter 7 details our approach to a versatile AOP kernel for the Java programming language. The kernel is based on a unified model of partial reflection, including structural reflection. We explain the evolution of Reflex to an AOP kernel, including comprehensive support for composition among aspects written in different languages.

Chapter 8 presents a case study for our kernel approach, based on the case of Sequential Object Monitors (SOM), a model and library for concurrent programming. It explains how SOM is implemented on top of our kernel and exemplifies interaction scenarios.

Finally, Part III concludes with a summary of the contributions of this thesis and perspectives for future research.

Note to the Reader: It has been recognized that open implementation design is an inherently *iterative* process [Kiczales *et al.*, 1993]. Our experience with building an open reflective system, Reflex, is another testimony of this fact. As the understanding of the issues at stake evolves, and the domain supposedly covered by the implementation is extended, both the customization interfaces and the framework are adapted and refined. Therefore, this dissertation reports on successive iterations in the design and implementation of Reflex. Therefore, the APIs and frameworks presented in the first chapters of this dissertation do not correspond to the current version of Reflex. As a matter of fact, Reflex is still evolving at the time this thesis is being finished and is bound to continue evolving in the future.

What this Thesis is Not About

Although this thesis is concerned with addressing applicability issues of reflection and metaprogramming, methodology issues that appear when dealing with reflective programming in-the-large, such as reflective analysis and design, are out of the scope of this thesis. Also, this thesis is definitely experimental, rather than theoretical, and hence does not propose a formal treatment of the subject.

Finally, we assume that the reader is well aware of object-oriented programming in general. All our experiments are carried out within the Java programming language. Therefore we also assume the reader to be fluent in Java. Although most of our results are not Java-specific, some are, and we do not explicitly explore the application of these results to other object-oriented languages. This is left as a promising perspective, particularly when considering more dynamic and uniform programming languages, such as Scheme or Smalltalk.

Chapter 2

Reflection, Open Implementations and Aspect-Oriented Programming

In this chapter, we review the state-of-the-art of the three themes surrounding this thesis work: reflection and metaprogramming, open implementations, and aspect-oriented programming.

The first four sections of this chapter are dedicated to reflection. Section 2.1 introduces the concept of reflection and its application to programming languages. Section 2.2 discusses reflection in the particular context of object-oriented programming languages. Then, since we are interested in this work in addressing issues in the concrete applicability of reflection, we dedicate Section 2.3 to the structuring and engineering of metalevel architectures, while implementation considerations are dealt with in Section 2.4. After this comprehensive review of reflection, the last two sections discuss the related areas of open implementations (Section 2.5) and aspect-oriented programming (Section 2.6). Finally, in Section 2.7, we revisit the problem statement of this thesis in the light of the concepts, approaches and issues presented in this chapter.

2.1 Reflection in Programming Languages

This section introduces the concept of *reflection* and its application to programming languages. What reflection actually means is pretty well embodied in the following explanation of the word *reflect*:

> *"One meaning of the word reflect is to consider some subject matter. Another is to turn back something (e.g. light or sound). Punning on these two meanings, we get the notion of turning one's consideration or considering one's own activities as a subject matter."* [Rao, 1991]

This section is structured as follows. Section 2.1.1 gives a brief historical introduction to the difference between programs and data and concludes with the appealing idea of conceiving pro-

grams as data for other programs [1]. Then, the notions of *metaprogramming* and *reflection* are defined (Section 2.1.2). Section 2.1.3 exposes the seminal experiments in reflection in programming languages, based on the idea of *reflective towers*. Finally, Section 2.1.4 discusses characteristics of reflective languages as well as some of these languages.

2.1.1 Programs and Data

When considering an automatic information processing system, the distinction between *programs* and *data* naturally appears. Data represent the information to process, while programs represent the processing to apply to such data.

This distinction existed well before modern computers: indeed, in the 1830's, scientist Charles Babbage had conceived a calculating engine whose internals could be adapted to a particular processing [Bromley, 1987]. This machine, called the *Difference Engine No.2*, was made of a *store* where data was kept and a *mill* which was in charge of processing the data. In this architecture, no confusion was possible between data and programs since a program was not stored in the same place than data, but rather represented by the internals of the calculating engine.

More than one century later, in 1958, the American mathematician John von Neumann first described the architecture of numeric computers and introduced the idea to store program instructions in memory, that very same memory in which data is kept. The possibilities offered by the manipulation of a program as data to another program fascinated him, as mentioned in his book *The Computer and the Brain* [von Neumann, 1958].

Interestingly, the idea to represent programs as data possibly processed by other programs also appeared in theoretical computer science. For instance, in Church lambda calculus [Barendregt, 1984], both programs and data are represented by higher-order functions. Similarly, a special kind of Turing machines [Turing, 1936], called the *universal Turing machine* [Turing, 1937], is indeed able of processing any other Turing machine.

Therefore, both experimental and theoretical approaches to computer science meet to consider that representing programs as data that can be manipulated by another program is a legitimate idea that is worthwhile studying.

2.1.2 Metaprogramming and Reflection

Considering that a program can act upon another program leads to the introduction of a number of concepts, defined by Pattie Maes in [Maes, 1987a], and illustrated in Fig. 2.1:

Definition 2.1 – Computational system

A system that acts and reasons about a *domain*. □

[1] Credits for this historical introduction to the distinction between programs and data go to Julien Vayssière [Vayssière, 2002].

Steyaert makes clear the difference between a program and a computational system [Steyaert, 1994]: a program is a textual description, while a computational system is a running program. A program *describes* a computational system. To act and reason about its domain, a computational system (or program) holds a *representation* of its domain (Fig. 2.1(a)). In order to be indeed useful, this representation should be *effective* in the sense that it is both always up-to-date with respect to the domain, and capable of triggering changes in the domain. This two-way connection is known as the *causal connection*:

Definition 2.2 – Causal connection

Property that ensures that changes in the domain are reflected in the computational system, and vice-versa. □

With these two definitions, it is possible to define a metasystem (Fig. 2.1(b)):

Definition 2.3 – Metasystem

A computational system whose domain is another computational system. □

The domain of a metasystem, a computational system, is called its *base system*. An evaluator is a particular metasystem that turns a program into a computational system (*i.e.* by running it). The program of the evaluator, or any other metasystem, is a *metaprogram*.

Reflection then appears when considering a metasystem whose domain is itself (Fig. 2.1(c)):

Definition 2.4 – Reflective system

A metasystem causally connected to itself. □

Therefore, a reflective system is characterized by its ability to act and reason about itself. A reflective program is a program describing a computational system that accesses its own metasystem. This ability opens a wide range of practical applications, as will be discussed further in this chapter. A more complete definition of reflection was given by Brian Cantwell Smith in [Ibrahim, 1990]:

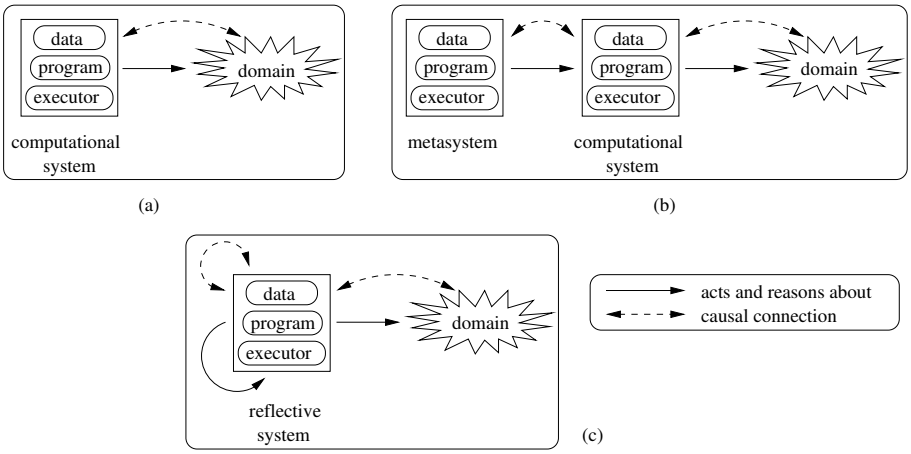

Figure 2.1 – Computational system (a), metasystem (b) and reflective system (c).

Definition 2.5 – Reflection

An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on, and deals with its primary subject matter. □

2.1.3 Reflective Towers

Brian C. Smith is a philosopher, considered as the pioneer of the field of computational reflection. In the early 1980's, he proposed and defined what it means for a system to be reflective, presented the general architecture of *procedural reflection*, and illustrated it through the implementation of a reflective dialect of Lisp, called 3-Lisp [Smith, 1982; Smith, 1984; des Rivières & Smith, 1984].

As a reflective language, 3-Lisp embodies self-knowledge in the domain of metacircular interpreters. Every 3-Lisp program is interpreted by a (continuation-passing) metacircular interpreter, also written in 3-Lisp. This gives rise to a potentially *infinite tower of metacircular interpreters*, each being interpreted by the one above it. Crucial to this architecture is the causal connection between the interpretation levels, characterized by Smith as *"meta-ness"*. A program running at one level can provide code to be run at the next higher level, hence gaining *explicit* access to the formerly *implicit* state of the computation [des Rivières & Smith, 1984]. Such code is provided by calling *reflective procedures*, a special class of procedures. A reflective procedure (or reflective function) can be viewed as a local procedure running at the level of the interpreter, that therefore manipulates data representing the code, the environment, and the continuation of the current (base level) computation. More generally, the structures at any given level represent the state of the

computation one level below.

Actually, because the levels in the tower need not be based on interpretive techniques, Smith and des Rivières use the term *reflective processor program* (RPP) instead of interpreter. Fig. 2.2 illustrates the levels of processing in the infinite reflective tower.

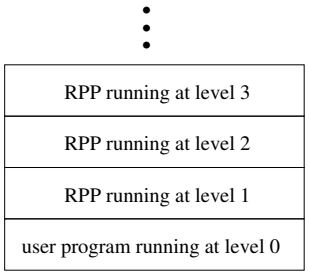

Figure 2.2 – Processing levels in the reflective tower.

The reflective tower is a special case of processing towers, in the sense that it is infinite and homogeneous. Finite heterogeneous processing towers are actually commonplace: consider a Java program at level 0, run by the Java Virtual Machine which is a machine language program running at level 1, which in turn is run by the hardware at level 2, thereby stopping the tower. In a reflective language, user code may not only run at level 0, but at any level above, hence gaining power to direct the course of its own execution.

2.1.3.1 Dealing with Infinity

To deal with infinity, des Rivières and Smith introduced the notion of the *degree of introspection* of a program: in any single program p and input i, only a *finite* number of levels n are needed to run the program; this number is the degree of introspection of the considered program. Hence, given n, the level $n + 1$ interpreter can be replaced by an implementation processor G, which is a real, non-reflective processor.

Since n is unlikely to be determined without actually running program p, the implementation processor G is proposed to be a *level-shifting processor* (LSP): such a processor is able, when it is determined (dynamically) that a new level of processing is required, to create the explicit state of the LSP on the fly as if it had run since the beginning of the program, and to resume the computation from this state. Therefore, along the execution of a program, G will *shift up* levels, progressively climbing to higher and higher reflective levels. Recall that shifts up are triggered by calling reflective procedures. In order to be efficient, however, a LSP should never run at any higher level than necessary, hence requiring the ability to *shift down* as soon as possible (Fig. 2.3).

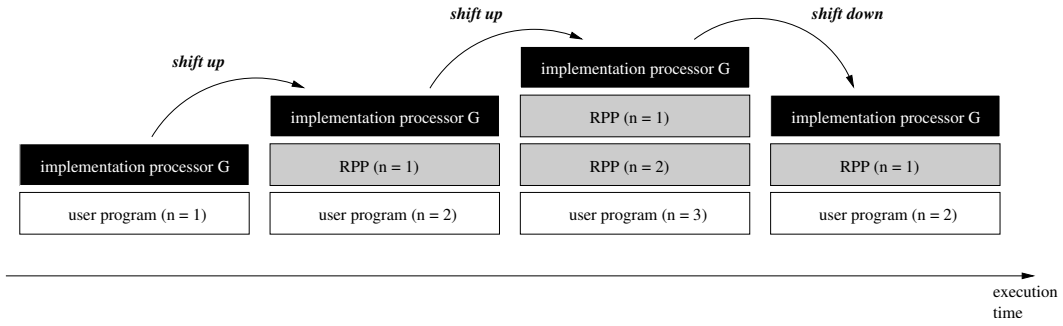

Figure 2.3 – The level shifting processor.

2.1.3.2 Reification and reflection

Wand and Friedman bring some more light on what shifting up and down actually means, in two major papers that attempted at giving a more formal, denotational account of reflection. They show in [Friedman & Wand, 1984] that the concept of reflection as formulated by Smith can be decomposed in two processes, called *reification* and *reflection*, which respectively correspond to shifting up and down:

Definition 2.6 – Reification

The process by which the *state of the interpreter* is passed to the program itself, suitably packaged (*reified*) so that the program can manipulate it. □

In the context of a conventional operational semantics model, the *state of the interpreter* is defined as interpreter registers holding an expression, an environment and a continuation. As further mentioned, the process of reification can be thought of as converting *program into data*. The data representing the piece of program is also called a *reification*.

Definition 2.7 – Reflection

The process by which *program values* are re-installed as the state of the interpreter. □

In this context, the *program values* are defined as values for an expression, an environment, and a continuation. The process of reflection can therefore be seen as converting *data into program*. It is

also sometimes referred to as *absorption* [des Rivières & Smith, 1984; Steyaert, 1994; De Meuter, 1999] or *deification* [Yokote, 1992; Douence & Südholt, 2001].

Following their quest for a formal understanding of reflection, Wand and Friedman manage to give a semantic account of Smith's reflective tower. Using a meta-continuation semantics, their account of the reflective tower does not employ reflection to explain reflection. It is presented in an appropriately-named paper: *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower* [Wand & Friedman, 1988].

2.1.4 Reflective Languages

As we have seen, the process of *reification* makes it possible for a program to gain access to a representation of (a part of) itself or to some aspect of the programming language, which were otherwise implicit. Smith mentioned two important requirements a language must conform to in order to be reflective [Smith, 1984]. First, the language needs *"an account of itself embedded within it"*; that is to say, some representation of the language must be accessible from within itself. Second, this self-representation must be causally connected to the system, as discussed at the beginning of this chapter. Another characterization of a reflective language is that of a language that provides its programs with (full) reflection [Maes & Nardi, 1988; Malenfant *et al.*, 1996b]. Full reflection here means that true reflection ideally does not impose any limit on what the program may observe or modify. However, it is inherently impossible to reify strictly *all* parts of a reflective system. This impossibility was mentioned in a theoretical context in [Wand & Friedman, 1988], and in an experimental one in [Douence & Südholt, 2001]. Therefore, the precise point at which a language with reflective mechanisms becomes a reflective language is not well defined [Rivard, 1996]. This distinction is however useful to contrast reflective languages with programming languages that only provide (or are extended with) *some* reflective mechanisms.

2.1.4.1 Reflective mechanisms

A reflective mechanism is defined in [Malenfant *et al.*, 1996b] as *"any means or tool made available to a program P written in a language L that either reifies the code of P or some aspect of L, or allows P to perform some reflective computation"*. Reflective mechanisms are called reflective *operators* in [Steyaert, 1994] and defined as: *"language facilities, offered by the programming language, that allow programs to access the metasystem with which they are executed"*. In order to better characterize reflective mechanisms, several distinctions should be introduced. There are potentially many "things" that can be reified, and the possible actions the program is allowed to carry over these reifications can also vary. Therefore, a first distinction is made between *introspection* and *intercession*:

Definition 2.8 – Introspection

The ability of a program to simply *reason about* reifications of otherwise implicit aspects of itself or of the programming language implementation (processor). □

In analogy with file systems, introspection can be seen as a *read access* to reifications.

Definition 2.9 – Intercession

The ability of a program to actually *act upon* reifications of otherwise implicit aspects of itself or of the programming language implementation (processor). □

Following the same analogy, intercession corresponds to a *write access* to reifications. The causal connection property ensures that changes made to reifications are indeed effective.

Another distinction is made between *structural* and *behavioral* reflection, depending on the representation reifications give access to:

Definition 2.10 – Structural reflection

The ability of a program to access a representation of its structure, as it is defined in the programming language. □

For instance, in an object-oriented language, structural reflection gives access to the classes in the program as well as their defined members.

Definition 2.11 – Behavioral reflection

The ability of a program to access a dynamic representation of itself, that is to say, of the operational execution of the program as it is defined by the programming language implementation (processor). □

In an object-oriented language, behavioral reflection could for instance give access to base-level operations such as method calls, field accesses, as well as the state of the execution stack of the various threads in the program.

Behavioral reflection was actually pioneered by Smith, as discussed previously, and is much more difficult to implement than structural reflection since it is not restricted to the static representation of programs. We will come back on implementation issues later in this chapter.

The distinction between introspection and intercession and that of behavioral and structural representations are indeed orthogonal: the former determines the kind of access given to the representation, whose type is determined by the latter. Moreover, these distinctions are completely valid in the context of metaprograms, not only reflective ones. For instance, a preprocessor is a metaprogram that uses both structural introspection and intercession. Conversely, an Integrated Development Environment (IDE) only needs structural introspection to provide a class browser. Finally, a debugger is a metaprogram that introspects both structure and behavior, and that may, in some cases, actually change the execution of the program (behavioral intercession).

2.1.4.2 Some reflective languages

As defined by Pattie Maes [Maes, 1987b], *"a programming language is said to have a* reflective architecture *if it recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly"*. Various languages with reflective architectures have been proposed. We have already mentioned 3-Lisp [Smith, 1982]. Another reflective variant of Lisp is Brown, which served as the basis for the formal work of Friedman and Wand [Friedman & Wand, 1984]. These languages are examples of procedure-based languages with a reflective architecture. Languages with reflective architectures have also been proposed in other paradigms, such as logic-based languages (*e.g.* Fol [Weyhrauch, 1980] and Meta-Prolog [Bowen, 1986]), rule-based languages (*e.g.* SOAR [Laird *et al.*, 1986]). Finally, many reflective object-oriented languages have been proposed, basically because of the apparent good match between object orientation and reflection. The next section will explore this relation further. Examples of reflective object-oriented languages are 3-KRS [Maes, 1987b; Maes, 1987a], Agora [Steyaert, 1994; De Meuter, 1999], ObjVLisp [Cointe, 1987], Smalltalk [Rivard, 1996], Classtalk [Briot & Cointe, 1989] and CLOS [Kiczales *et al.*, 1991].

Smalltalk is in fact not fully reflective due the pragmatic reason of efficiency [Goldberg & Robson, 1983], which made its designers choose not to reify messages and message lookup as such. Still, Smalltalk presents so many reflective mechanisms and is so deeply rooted in reflection that it is often considered as being able to provide much of the power of full reflection [Foote & Johnson, 1989]. Indeed, Smalltalk is almost entirely written in itself, and supports both introspection and intercession of its structures (classes) and its behavior (by reifying both the compiler, and message sending and control state) [Rivard, 1996]. Conversely, other industrial languages like C++ [Stroustrup, 1997] have no reflective or metalevel features[2]. The more recent Java programming language [SUN, 1996a] actually started without any reflective mechanism, but has been progressively updated with more and more reflective mechanisms. It now basically supports structural introspection and a limited form of behavioral intercession. We will come back to Java in Section 2.4.2.4.

[2] Actually C++ *templates* are a textual compile-time metaprogramming facility [Czarnecki & Eisenecker, 2000].

2.1.4.3 From reflective languages to reflective applications

As we have seen, reflection was first introduced in the context of reflective *languages*, *i.e.* languages that indirectly give access to the otherwise implicit aspects of themselves, through reflective metacircular interpreters. As a consequence, this makes it possible for programs (applications) to be reflective as well, since they can access their own representation in the language. However, as argued in [Douence & Südholt, 2000], most useful reflective applications rely only on limited reflective capabilities. Douence and Südholt therefore advocate the use of reflective applications, rather than (fully) reflective languages. Such applications then exhibit reflective capabilities that are specifically-tailored to their needs and are only present at well-chosen places in the code. We shall come back on this in Section 2.4.4 when discussing partial reflection.

2.2 Reflection and Object Orientation

As we have seen, reflection in programming languages started to gain a lot of attention in the early 1980's. Be it a coincidence or not, this period of time was also the beginning of the advent of object-oriented programming. Rapidly, most of the work in reflection was formulated in the context of object orientation. As acknowledged by the reflection community, the reason seems to be a good match between both [Ibrahim, 1990]. We will discuss this match in Section 2.2.1. In Section 2.2.2, we will introduce *metaobject protocols*, the fruit of the wedding between reflection and object orientation. The various reflective object-oriented models that have been proposed in the literature will be reviewed in Section 2.2.3

2.2.1 The Good Match

Object orientation as a programming paradigm appeared as a means to solve some of the issues with procedural programming. Mainly, the issue of keeping procedures and data structures coherent which each other: in procedural programming, both are defined separately, although they are inherently interdependent, since changes in one usually affects the other. Object-oriented programming addresses this issue by packing data and procedures together in entities called *objects*, that communicate through *messages*. The fact that data and procedures are distributed in separate objects brings interesting properties, such as *abstraction* and *encapsulation*. Furthermore, object-oriented languages have quickly integrated means for localized extensions of behavior, through *overriding*.

As Pattie Maes remarks, abstraction in object-oriented languages makes reflection naturally fit in this spirit. Since an object is free to realize its role in the overall system in its own way, *"it is natural to think that an object not only performs computation about its domain, but also about how it can realize this computation"* [Maes, 1987b]. Moreover, since abstraction and encapsulation promote minimum coupling between communicating objects by relying on well-defined *interfaces* or *protocols*, it is possible to program base computation independently of meta computation. Therefore base objects (performing base, or domain, computation) and so-called *metaobjects* (performing meta computation) can be made to cooperate through a well-defined interface and therefore

it is possible to change implementations of one or another independently. And precisely, through extension mechanisms provided by object-oriented languages such as delegation and overriding, it becomes feasible to develop libraries of metaobjects. Such metaobjects can be reused and extended in turn. Eventually, the very motivations that led to the advent of object orientation as a major programming paradigm do explain why reflection is also expected to profit from it:

> *"What reflection on its own doesn't provide, however, is flexibility, incrementality, or ease of use. This is where object-oriented techniques come into their own."* [Kiczales *et al.*, 1991]

In other words, object orientation seems promising to address issues related to the *structure* of the metalevel and the *locality* of reflective computation. By locality we refer to the scope that changes done at the metalevel have on base-level computation. We will be refining this notion along this chapter, since it is one of the key elements that drove a major track of research in metalevel architectures, open implementations, and aspect-oriented programming.

2.2.2 Metaobject Protocols (MOPs)

Object orientation allows for the independent programming of base and meta computation, since base objects and metaobjects communicate through a well-defined interface. Such an interface, which is the equivalent of standard interfaces between objects, transposed in the realm of reflection and metaprogramming, is called a *Metaobject Protocol*, abbreviated as *MOP*. As Kiczales *et al.* put it:

> *"Metaobject protocols are* interfaces *to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language."* [Kiczales *et al.*, 1991]

This notion is refined later, in the context of the CLOS MOP, a metaobject protocol for CLOS [Bobrow *et al.*, 1993] (an object-oriented variant of LISP):

> *"First, the basic elements of the programming language - classes, methods and generic functions - are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of* metaobjects. *Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects - a* metaobject protocol. *Third, for each kind of metaobjects, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol."* [Kiczales *et al.*, 1991]

Maes observed that reflective computation may be caused either by an object itself or by the interpreter [Maes, 1987b]. On the one hand, an object can trigger reflective computation by specifying reflective code, *i.e.* code that explicitly mentions metaobjects. On the other hand, the interpreter may cause reflective computation for an object whenever it determines that it is needed. At such time, the interpretation of the object is delegated to some metaobject. Such

kind of reflection has been called *implicit* [Maes & Nardi, 1988]. This observation leads to the distinction of different types of MOPs, depending on whether they are *explicit*, implicit or related to communications between metaobjects [Zimmermann, 1996]:

Explicit MOPs are used by base objects to communicate with the metalevel. This can concretely be done by sending messages to a given metaobject or by actually *changing* a metaobject by another one. This usually results in explicit changes in the behavior of base objects. For instance, changing the status of an object from volatile to persistent may be done either by informing the metaobject in charge of this concern that, from now on, the object should be persistent, or by effectively changing the default metaobject implementing the volatile semantics by one implementing the persistent semantics.

Implicit MOPs take place transparently: the base object does not know about the "jump to the metalevel". For instance, each time an object is created, its state may be transparently initialized by retrieving it from storage, and when it is destroyed, storage is transparently updated with the new state. As Maes mentioned, this transparency comes from the fact that the interpreter itself triggers the meta computation.

Inter-metaobject Protocols are used by metaobjects to communicate with each other. An inter-metaobject protocol is also explicit (implemented through standard method calls), but usually not visible to base objects.

It is interesting to note that explicit and implicit MOPs usually collaborate to achieve a given behavior: base objects can use an explicit MOP to *specify* the required semantics (*e.g.* saying that an object should be persistent), which is then *implemented* through the implicit MOP (*e.g.* intercepting object creation and destruction to retrieve and store the object state).

2.2.3 Reflective Models for Object-Oriented Languages

The first object-oriented languages to incorporate some reflective facilities, such as Smalltalk-72 [Goldberg & Kay, 1976] and Flavors [Weinreb & Moon, 1981], did so in ad hoc ways. A first step towards a cleaner handling of reflective facilities was the introduction of *metaclasses* by Smalltalk-80 [Goldberg & Robson, 1983], further studied by Cointe in ObjVLisp [Cointe, 1987] and Classtalk [Briot & Cointe, 1989]. Metaclasses basically serve the purpose of specifying the internal structure and behavior of a class. In uniform object-oriented languages, everything is an object. In a pure class-based language, a class is therefore just an object, that has the particularity that it can generate objects, its *instances*. And a metaclass is a class whose instances are classes. This approach to reflection was brought to the fore by Pierre Cointe with ObjVLisp. This model mainly allows for extension of the static part of object-oriented languages, although it can serve as a basis for behavioral reflection, as discussed hereafter.

The second model of reflection for object-oriented languages was introduced by Pattie Maes in 3-KRS [Maes, 1987a; Maes, 1987b], in the line of Smith's work on 3-Lisp. In this model, there is a 1-to-1 relation between an object and its metaobject. The metaobject represents the otherwise implicit information of its so-called *referent*: its structure as well as its way of handling

messages. However, this model was formulated in the context of the prototype-based language KRS [Marke, 1988], which does not support the notion of classes. Ferber [Ferber, 1989] studied the transposition of this model to class-based languages, comparing it to the metaclass model. Finally, a third model, based on message reification, is presented.

2.2.3.1 Metaclass model

In this model, the meta relation is merged with the type relation: the class of an object is considered as its metaobject, since classes actually describe the structure and behavior of their instances. Metaclasses are therefore the metaobjects of classes, because of their ability to describe the internal structure and behavior of a class. Note that some languages only provide one metaclass that describes all classes in the system. This was the case of Smalltalk-76, and is still the case in Java, which furthermore closes the door to extension since the unique metaclass cannot be subclassed. A unique metaclass is actually not convenient to allow semantic variations because it propagates changes to all classes in the system. Conversely, in languages like ObjVLisp and Smalltalk-80, each class is the unique instance of its own metaclass.

From a behavioral point of view, it is equivalent for an object `o` to receive a message `m`, or for the class of `o` to receive a message `handleMessage` (that takes as parameters both `o` and `m`). Thus, the default handling of a method is described in the metaclass, since it is where the `handleMessage` method is defined. Specializing the default interpretation of a message implies a substitution or an alteration of the metaclass. This model presents the drawback that all instances of a class share the same message interpreter: there is no possibility to specialize the interpreter for a unique object[3]. Furthermore, metaclass substitution is dangerous and can quickly lead to inconsistencies. Finally, it is not possible for the metaobject (the class) to keep personal characteristics of objects. These limitations are summarized by Ferber saying that *"metaclasses are not* meta *in the computational sense, although they are* meta *in the structural sense"* [Ferber, 1989].

2.2.3.2 Metaobject model

In this model, the metalink is different from the instance-of link: classes and metaobjects are distinct objects. Each object has its own metaobject.

The main distinction between this model and the metaclass model lies in the separation of structural and computational (behavioral) reflection: in the metaclass model, classes are used for both structural description (definition of the instance structure and the set of applicable operations) and computational description (how a message is interpreted and a method is applied); conversely, the metaobject model splits them apart: classes handle the structural part, while metaobjects handle the behavioral part. This model presents many advantages. First, it is easy to modify the metaobject of a single object; second, an object can be monitored by its metaobject; finally, defining new ways of handling messages simply consists in defining new classes of metaobjects

[3] Unless a dictionary is kept in the class in order to distinguish between instances, but then this is really close to the metaobject model, presented afterwards.

(*e.g.* by subclassing a default metaobject class).

2.2.3.3 Message reification

Ferber introduces yet another model that consists of reifying the communication itself. In this model, each communication is an object, instance of a message class, that can react to the `send` message. It is therefore the responsibility of the message to interpret itself. Specialization of message sending semantics therefore implies subclassing the default message class. In the model presented by Ferber, a message object encapsulates only the receiver, the selector (of the message) and the arguments. Cazzola has further extended this model by including the sender object as well, leading to more expressive power [Cazzola, 1998], which is particularly useful in a distributed setting [Ancona *et al.*, 1998; McAffer, 1995b]. The main disadvantage of this model, apart from efficiency considerations, is that it does not say anything about the objects of the application. However, as Ferber outlines, this model can be used in conjunction with the metaobject model.

2.2.3.4 Where is the reflective tower?

The three models presented above are mainly based on the *lookup/apply* protocol of object-oriented languages. The CLOS MOP [Kiczales *et al.*, 1991] is rather based around the generic function model, where the application of a generic function is reified as a generic function [Malenfant *et al.*, 1996b]. As discussed by Malenfant *et al.*, reflective towers appear in both kinds of approaches. Des Rivières [des Rivières, 1990] has pointed out the existence of the reflective tower in the CLOS MOP: the tower appears because, when invoked, the generic function that describes how generic functions are applied is itself a generic function, and therefore must invoke itself. This infinite meta-regression is however simply avoided (by not reifying the application of the MOP generic function). Besides, in [Malenfant *et al.*, 1996a], the existence of the reflective tower in the *lookup/apply* model is shown. The tower appears because apply methods are themselves methods, which much have their own apply method.

As argued in [Douence & Südholt, 2001], since these approaches to reflection do not feed higher-level interpreters with the code of lower-level interpreters, they have no semantic foundation, but allow for more efficient implementations. In contrast, MetaJ [Douence & Südholt, 2001], 3-KRS [Maes, 1987a] and Agora [De Meuter, 1999] are semantics-based, following Smith's seminal work on 3-Lisp. According to [Malenfant *et al.*, 1996b], the most important difference however is that the 3-Lisp (and alike) tower is potentially infinite, whereas in object-oriented models, the towers are finite by construction, and the languages always provide mechanisms that stop the tower at some fixed level, possibly differing from methods to methods.

2.2.4 MOPs for Separation of Concerns

Reflective systems have interesting practical applications. In particular, in the context of programming languages, reflection has served a lot as a means to rapidly experiment with variations on language semantics, in particular for object-oriented programming, at a time where basic elements were still being defined [Ibrahim, 1990]. Furthermore, from a software engineering point of

view, reflection is interesting because it introduces a separate level, the metalevel, at which several concerns can be addressed, in a manner that is mostly transparent for the base level.

As a matter of fact, one of the major quest of programming language research is that of being able to develop software systems while preserving a good *Separation of Concerns* (SoC) [Dijkstra, 1968; Parnas, 1972]. The idea of SoC basically consists in assigning particular concerns to separate modules. Object-oriented programming was itself a step forward in this direction. However, as software systems are applied in more and more complex situations and demanding environments, it becomes difficult to maintain a good separation of all concerns. For instance, when considering the behavior that a cleanly-designed system should adopt when facing exceptional situations, the concern of exception handling tends to be spread over many modules, thereby violating the SoC principle.

Behavioral reflection and metaobject protocols actually provide means to achieve a cleaner separation of concerns in complex software systems, since the metalevel can actually address the issues related to *how* a system should do its job, letting the base level only focus on *what* it should do: this is the idea of separating functional concerns, handled at the base level, from non-functional concerns, handled at the metalevel. Furthermore, behavioral reflection supports separation of dynamic concerns as well, thereby offering a modular support for adaptation in software systems [Blair *et al.*, 2000; Redmond & Cahill, 2002]. These strengths of reflection have been exercised in a wide range of domains, including distribution [Stroud, 1993; Caromel *et al.*, 1998; Ledoux, 1999; McAffer, 1995b], mobile objects [Blair *et al.*, 2000; Ledoux & Bouraqadi-Saâdani, 2000; Tanter *et al.*, 2002b], concurrency [Masuhara *et al.*, 1994], fault-tolerance [Fabre *et al.*, 1995] and atomicity [Stroud & Wu, 1995].

In [Stroud & Wu, 1996; Briot *et al.*, 1998], a comparison is made between three approaches to separation of non-functional concerns: system-based approaches, language-based approaches, and MOP-based approaches. System-based approaches basically consist in handling non-functional requirements such as persistent data storage, data sharing, and distributed programming, directly in the operating system. This approach has the advantage of being efficient, but offers no means of adaptation or customization.

Language-based approaches consist in extending the semantics of the language by providing a range of building blocks. This can either be done by adding semantics directly to the programming language, by extending the runtime support mechanisms or by adding object definitions to object libraries. Typical tools are therefore preprocessors, specially-tailored interpreters, and/or reusable objects. Some examples of this approach are: Arjuna [Shrivastava *et al.*, 1991], which provides persistence and atomicity by inheritance, and distribution transparency via a preprocessor; PC++ [Wu, 1994], which provides atomic data types via a combined use of inheritance and preprocessing; or SOS [Shapiro *et al.*, 1989], which adds persistence and migration to C++ objects with a special compiler and a runtime object management system. The limitations of these approaches is that they are usually not transparent at all for programmers: they require “stylized” code that obscures base functionality. For instance, in Arjuna, explicit lock manipulation code must be mixed with base functionality. Furthermore they require a specialized implementation of a language that is hard to customize: for instance, the PC++ processor generates code that might not be adequate in some situations; changing the generation scheme actually involves changing

the preprocessor.

MOP-based approaches, on the other hand, provide both transparency and flexibility. Non-functional requirements are implemented as metaobjects, by system developers. Then, adding non-functional properties to objects is done by binding them to appropriate metaobjects: *e.g.* a persistence metaobject, a replication metaobject, etc.. Defining new kind of behaviors can be done by incrementally extending metaobject classes. Furthermore, metaobjects may in some cases be reusable: indeed, standing above objects from a meta viewpoint makes it possible for metaobjects to be generic, and hence adequate on different types of objects. As Stroud and Wu conclude, the key of MOP-based approaches among others is their *flexible approach to reification* [Stroud & Wu, 1996]. Indeed, dedicated preprocessors and the like also use a form of reification to operate, but they do so in a very ad hoc way, that makes them unsuitable for handling other concerns or being extended.

2.3 Structuring the Metalevel

As reflective approaches matured, attempts to concretely apply them to various domains have progressively brought to the fore the need for further investigating structuration aspects of the metalevel. It is therefore no coincidence that the main elements discussed in this section emerged from applied work, *e.g.* in the field of concurrency, distribution, and operating systems. In particular, the nature and arity of the metalink was further explored, leading to more flexible models. This is explored in Section 2.3.1. Besides, the nature and coordination of metalevel entities themselves have been subject to deeper studies, as the need for advanced engineering techniques at the metalevel started to appear. This is reviewed in Section 2.3.2.

2.3.1 Nature of the Metalink

As we said earlier, the notion of *metaobject* was first introduced by Pattie Maes in the context of 3-KRS [Maes, 1987b]. In this model, a metaobject is an object which reflects the structural, and possibly also the computational aspect of a *single* object.

Still, in 3-KRS, several metaobjects participate in the representation of a single object[4]: the metaobject of an object has slots that are filled by primitive metaobjects. These primitive metaobjects together represent the complete 3-KRS interpreter. ABCL/R [Watanabe & Yonezawa, 1988], a reflective version of the object-oriented concurrent system ABCL/1 [Yonezawa, 1990], is another example of such an architecture. In ABCL/R, the arity of the metalink is also 1-to-1, and the different aspects of a base object (in this case, variables, scripts, local evaluator and message queue) are held in the state variables of the metaobject.

Such architectures are qualified as *individual-based architectures* [Matsuoka *et al.*, 1991] since a single object is the unit of computation at the base level (from a metalevel point of view). The

[4] A distinction is introduced in [Matsuoka *et al.*, 1991] between metalevel objects and metaobjects. While metaobjects are metalevel objects, the reverse is not true: some metalevel objects simply *reside* at the metalevel, without actually being bound to a base object.

reflective tower (that comes from the fact that a metaobject is also an object) is called an *individual tower*. The limitation of such an architecture lies in its lack of a *global view* of computation. Since each metaobject is self-contained (in the sense that it only "sees" its referent), other parts of the base computation are only accessible through explicit access to their metaobjects.

To address this issue in scenarios dealing with resource management where a more global view of the computation is needed, the idea of *group-wide reflection* was introduced [Watanabe & Yonezawa, 1990]. In group-wide reflective architectures, the collective behavior of a group of objects is represented as coordinated actions of a group of metalevel objects, called the *metagroup*. The reflective tower, that comes from the fact that a metagroup is itself an object group, is called a *group tower*. In this model, there is no intrinsic relation between a particular object and a metaobject. Rather, the entire object group is the unit of base-level computation (from a metalevel point of view). The disadvantage of such a model is that the identity of a given base object is lost at the metalevel, and must therefore be reconstructed manually.

Naturally, Satoshi *et al.* proposed the amalgamation of both architectures, called the *hybrid group architecture* [Matsuoka *et al.*, 1991], implemented in ABCL/R2. In this architecture, both the individual tower and the group tower are preserved. In their application context, coordinated resource management, they observe that the hybrid group architecture does not merely combine the benefits of both architectures. Rather, it enables advanced coordinated resource management schemes to be modeled, which would hardly be feasible with previous architectures. This work therefore brings a first justification to the interest of a more flexible metalink. However, in the proposed hybrid architecture, a limitation is that an object cannot belong to more than one group: although this limitation does make sense in the precise application context of ABCL/R2, it is, in a more general setting, questionable. Satoshi *et al.* actually end up arguing that architectural issues are fundamental, and that research on more effective architectures should be pursued.

In their work on the Iguana language (a fully-featured MOP for C++ [5]), Gowing *et al.* expose a fairly flexible approach to the metalink [Gowing & Cahill, 1996]. Metaobject instances may be shared by several objects, and an object may be controlled by several metaobjects. In their model, there is one metaobject per *reification category* (*i.e.* features of the object model of the language that can be reified, such as object creation and deletion, activation frames, state access, etc.). This has an interesting impact on the modularity of the metalevel, exploited in particular by McAffer [McAffer, 1996], as discussed hereafter.

2.3.2 Metalevel Engineering

Applying metalevel architectures in complex domains raises the need to be able to properly engineer the metalevel. Jeff McAffer made a significant contribution to this issue in a paper called: *Engineering the Metalevel* [McAffer, 1996]. His observations are the result of his work on a metalevel architecture for distributed object systems, CodA [McAffer, 1995a; McAffer, 1995b]. As he states:

[5] The issue of bringing reflection to a compiled language such as C++ will be discussed later in Section 2.4.

> *"[...] the metalevel has been thought of as a place for making small changes requiring small amounts of code and interaction. We believe that the metalevel should be viewed as any other potentially large and complex application – it is in great need of management mechanisms."* [McAffer, 1996]

McAffer is therefore concerned by the need to bring traditional engineering techniques to the metalevel, such as decomposition, combination, abstraction and reuse. According to him, the desirable properties of expressiveness (as the possibility to express a wide range of computational behavior), extensibility and programmability are lacking in previous work, which solely concentrated on the clear separation of base level and the metalevel.

2.3.2.1 Operational decomposition

Most reflective systems are based on reification of the structural concepts offered by the language (classes, methods, objects, slots, etc.). McAffer characterizes this approach as a *top-down* approach: taking high-level concepts (those of the language) and breaking them into their constituent pieces. Although this approach presents the advantage of structuring the metalevel in terms of a limited and particular set of concepts that are usually well-understood (since they closely match those of the base language), it is hard to integrate new concepts or behaviors which have no foundation in the base language. This limitation compromises the desired expressiveness and extensibility properties.

The approach promoted by CodA in this regard is therefore to *separate the description of the computational behavior of an object from that of its base language*. McAffer therefore formulates a *bottom-up* approach, which consists in starting from the basic *operations* (*e.g.* message send and receive, field access, object creation, etc.) defining the computational behavior of an object. This approach strongly diverges from the top-down approaches that we have been discussing until now: the interpreter-based approaches, such as 3-Lisp [Smith, 1982], 3-KRS [Maes, 1987a] and MetaJ [Douence & Südholt, 2001], where metaobjects match the structure of the interpreter, and the language-centric approaches, such as the CLOS MOP [Kiczales *et al.*, 1991], ObjVLisp [Cointe, 1987] and Classtalk [Briot & Cointe, 1989], which provide representation of the structural elements of the language.

Actually, the top-down and the bottom-up approach described by McAffer could alternatively be referred to as a structural and a behavioral approach, respectively. Other pieces of work adopt a similar philosophy, and interestingly enough, they all come from concrete applications of reflective techniques (and not from the pure language community): work on atomic data types [Stroud & Wu, 1995], concurrency [Ichisugi *et al.*, 1992], distribution [Okamura & Ishikawa, 1994; Okamura *et al.*, 1992] and operating systems [Yokote, 1992]. This *operational decomposition* of the metalevel is shown to be both expressive and extensible. Such an architecture concentrates on what occurs, not how the description of this is organized. Object systems are therefore reduced to a set of conceptual operations, whose *occurrences "can be thought of as the events which are required for object execution"* [McAffer, 1996].

2.3.2.2 Fine-grained MOPs

McAffer also argues for the freedom to design metaobjects at the appropriate level of *granularity*. This is indeed just standard object-oriented programming practice, that leads to better robustness, encapsulation, and modularity.

In this direction, a significant improvement in the modularity of a metalevel architecture is the concept of *fine-grained MOPs*, introduced in Iguana [Gowing & Cahill, 1996]. This proposal was motivated by the will to make metaobject protocols practical for operating systems [Gowing & Cahill, 1995], more precisely, as a mechanism for adaptable system components. The fine-grained MOPs of Iguana are an enhancement, in terms of fine granularity and combination possibilities, of the Multi-Model Reflection Framework developed for AL-1/D [Okamura *et al.*, 1992; Okamura & Ishikawa, 1994].

A MOP essentially specifies a reflective object model: the object model specified by a MOP is implemented by metaobjects. The idea of fine-grained MOPs is to allow multiple reflective object models to coexist in a given application. For example, in an application, a distributed object could use a distributed object model while other objects of the system use the standard (local) object model. Furthermore, if an object subsequently needs to modify its object model (that is, its metalevel implementation), it can do so knowing that any changes will not affect other object models: this is called *metalevel locality of change* [Gowing & Cahill, 1996]. Iguana hence provides a very elegant and flexible way of structuring customized metalevels from elementary building blocks, with the protocols themselves providing higher-level building blocks. The Iguana approach to fine-grained metalevel structuring was later on ported to Java, with Iguana/J [Redmond & Cahill, 2000; Redmond & Cahill, 2002].

2.4 Implementing Reflection

The implementation of reflection poses a number of challenges, which this section surveys. The first one, quite easily addressed, is that of the potential *infinite* in the reflective tower and the associated issue of metaregression (Section 2.4.1). Another one is how to actually *provide* reflection in existing, non-reflective languages both interpreted and compiled. This issue is discussed at length in Section 2.4.2. The case of Java is treated apart in more details, since it is the language with which we will be experimenting. Finally, we will consider the issue of the *efficiency* of reflection, which has seen the emergence of two trends: the first one, presented in Section 2.4.3, attempts to make reflection efficient by anticipating reflective computation; the other approach, explored in Section 2.4.4, rather considers that the cost of reflection may not be such an issue if we are able to selectively apply it only when needed.

2.4.1 Infinity

The issue of the potential infinity present by essence in reflective architectures appears more problematic than it really is in practice. It is either addressed by arbitrarily fixing the number of metalevels (such as in the Apertos operating system, where the number of metalevels is fixed to

four [Yokote, 1992]), or more generally, by relying on *laziness*, as in all other reflective systems mentioned until now.

Infinite meta-regressions, also called *circularities*, can also be easily discharged. As explained in [Kiczales *et al.*, 1991], there are two kinds of circularity issues: *bootstrapping issues*, which are involved with how to get a reflective system up and running in the first place, and are usually easily tackled in an ad hoc manner; and *metastability issues*, which have to do with how a reflective system manages to run, and to stay running even while fundamental aspects of its implementation are being changed. Metastability issues require a bit more care and anticipation, however. By noticing that they are indeed similar to recursion, similar practices apply: typically, stopping on some special cases, like in well-founded recursion (think of the factorial function, for instance, which stops recursion for $n = 0$). Indeed, like for recursion, the particular way regression is stopped depends on each particular case (see for instance [Kiczales *et al.*, 1991], Appendix C, for the CLOS MOP, or [Chiba, 1995], for the OpenC++ MOP).

2.4.2 Reflection for Interpreted and Compiled Languages

Reflection and metaobject protocols have first been mainly studied in the context of interpreted languages. The reason for that is that an interpreter is the good place to look for metalevel information about a running program[6]. Since reflection occurs when a program has access to such metalevel information about itself and can manipulate it, having a reflective interpreter only involves exporting this information and providing the base-level program with means to access and modify the information.

However, this approach is not adequate in two (widely-spread) situations: *(a)* for compiled languages, such as C++, where source code is turned into code directly executed by the machine, since there is no interpreter at all; *(b)* for interpreted languages whose standard interpreter is non-reflective and hardly extensible or modifiable, like Java virtual machines. Actually, the Java language is *compiled* into an intermediate language, the *bytecode* language, which is *interpreted* by a Java Virtual Machine. As a matter of fact, the bytecode is very close to the original source code, in the sense that most semantic information is preserved at a sufficiently high-level of abstraction. Approaches to bring reflection in these cases –called *reflective extensions*– are discussed hereafter. The two following sections discuss the issue of providing reflection in compiled and interpreted languages respectively. Before presenting the case of the Java programming language in Section 2.4.2.4, we will discuss the notion of *binding time*, which is helpful to understand and characterize the various approaches to implement reflection.

2.4.2.1 Compiled languages

With compilers, the metalevel information that is constructed at compile time is usually not kept beyond the compilation phase. In the generated code, the object model of the language is com-

[6] We use the term *metalevel information* as introduced in [Gowing & Cahill, 1996]: *"to describe both the tables of data associated with interpretation/compilation and the implicit knowledge maintained by the interpreter/compiler to order its decision making process regarding the behavior of the base-level program"*.

pletely implicit and can not be accessed[7]. Therefore, as discussed in [Gowing & Cahill, 1996], adding reflection to a compiled language entails maintaining the metalevel information beyond the compilation process and also transforming the generated code with the appropriate links to the metalevel information that controls its behavior. Concretely, this widely-used technique consists in transforming code to introduce so-called *hooks* to the metalevel, also known as *metalevel interceptions* (MLIs) [Zimmermann, 1996].

Typically, hooks are pieces of code in charge of the reification process: inserted in the code, they trigger a shift to the metalevel when reached by the execution flow. Therefore, even if the fact that some metalevel behavior should occur is statically determined (*i.e.* anticipated), the precise metaobjects implementing that behavior can still be accessed and changed dynamically, thus supporting dynamic adaptation of behavior. Obviously, this technique involves a significant execution overhead if used at each and every place in the code, since the program must evaluate both the hooks (*i.e.* code that builds a representation of the intercepted piece of program) and the metalevel code. This is where considerations related to *partial reflection* come into play, discussed in depth in Section 2.4.4.

Note furthermore that we hereby considered the implementation of dynamic behavioral reflection, provided by so-called *runtime MOPs*: in fact, the observation that some metacomputation need not happen at runtime led to the introduction of *compile-time MOPs* and other related techniques, as discussed in Section 2.4.3.

2.4.2.2 Interpreted languages

To introduce reflection in languages for which a non-reflective interpreter is available, the logical solution is to make the interpreter reflective. Douence and Südholt proposed, in the context of object-oriented languages, a significant improvement over early proposals such as 3-Lisp and Brown (where the interpreter is fully reflective). Their generic reification technique makes it possible to build, from a non-reflective metacircular interpreter, a specially-tailored reflective interpreter in which only required elements are reflective [Douence & Südholt, 2001]. This technique based on transforming the code of the interpreter presents nice properties of completeness and sound semantics. However, for this approach to be applicable, a metacircular interpreter must be available. In the context of an industrial language like Java or C#, this is not the case: production-quality virtual machines are not metacircular interpreters.

In such a situation, two alternatives are available. The first one is to introduce the interception and redirection mechanisms in the program code (source or binary), as in the case of compiled languages discussed previously. The interpreter is untouched. The second alternative is to leave program code as it is, but to modify or extend the interpreter to create the interception mechanisms. The limitation of the first approach is that it requires a static transformation of the application, which limits its support for dynamic adaptability. Furthermore, it is limited in expressiveness to what can actually be found in the code. Its great advantage is that it remains compatible with the

[7] If compiling with debugging attributes, some semantic information can still be obtained, but it is really hard to use in order to affect the behavior of the program. Still, Marc Ségura-Devillechaise has recently been able to get convincing results in the case of Linux elf binaries [Ségura-Devillechaise *et al.*, 2003].

standard interpreter for the language. It can thus benefit, at no cost, from the evolution of virtual machine technologies, and its use is also facilitated.

The second approach, on the other hand, presents the advantage of having direct access to the internal structure of the interpreter and therefore provides greater flexibility and expressiveness for supporting dynamic adaptation. The major disadvantages are the loss of compatibility with standard environments, which often results in particular tools becoming obsolete quickly, and the complexity of the implementation. Indeed, modifying a production virtual machine is not an easy task, and it is subsequently difficult to keep up-to-date with new versions and technologies (all the more that virtual machines are usually updated more often than language specifications). Just as an illustration, consider the case of Iguana/J [Redmond & Cahill, 2002]: it was implemented as a native dynamic library integrated very closely with the interpreter, via the Java Just-In-Time (JIT) compiler interface [SUN, 1996b]. At that time, Sun started its integrated HotSpot technology [SUN, 2004], and stopped supporting the JIT compiler interface.

2.4.2.3 Binding times and modes

When starting to consider implementation techniques and approaches to reflection, it is important to look at reflection under the viewpoint of *binding times*.

Looking at the history of programming languages, one can notice that it has been driven by the quest for ever late binding time, responsible for most of the advances in software design [Halpern, 1993; Kay, 2001; Hayes, 2003]. Binding basically means associating a value to a name. The following definition, taken from [Halpern, 1993], expresses binding from a lower-level point of view:

> Definition 2.12 – Binding and binding time
>
> Binding means translating an expression in a program into a form immediately interpretable by the machine on which the program is run; binding time is the moment at which this translation is done. □

For instance, consider the binding of a procedure call to the address of the code to be run: while the binding is done at compile time in procedural languages, object-oriented languages have postponed it to runtime.

Postponing binding times brings more flexibility at the expense of performance penalties. If we make the distinction between a *formal* binding time (as being the latest moment at which the binding can be done, in general) and the *actual* binding time (as being the actual moment at which a particular binding is done), we can say with Malenfant *et al.* that:

> *"The general trend in the evolution of programming languages has been to postpone formal binding times towards the running of programs, but to use more and more*

> *sophisticated analysis and implementation techniques to bring actual times back to the earlier stages.*" [Malenfant *et al.*, 1996b]

Seen in this light, reflection in programming languages naturally fits in the trend of ever late binding times, by postponing the binding of almost all elements of programs and languages to the runtime.

The actual trade-off between functionality and cost in reflective architectures is further clarified by introducing the notion of *binding modes* [Czarnecki & Eisenecker, 2000]. While the binding time describes when an association occurs, the binding mode describes the *permanency* of the binding. This mode may be either static, if it cannot be undone, or dynamic, if it may be undone and redone. Systems can therefore be characterized according to their position in the range going from static binding at compile time, to dynamic binding at runtime. The hook insertion technique presented above imposes binding to be done at compile time (or load time) and may support both static and dynamic binding. Conversely, the interpreter-based approaches support dynamic binding at runtime (and are hence better suited for *unanticipated* software adaptation). A characterization of many systems in light of this range can be found in [Redmond & Cahill, 2002].

2.4.2.4 The Java case

The Java programming language first started without any reflective mechanisms, and has been successively updated, until JDK version 1.3. The standard Java reflection API [SUN, 1999c] mainly supports structural introspection. Indeed, the ability to obtain metaobjects representing classes, methods, fields and constructor is restricted to introspection: it is not possible to modify a given class through such an API. It is, however, possible to instantiate a class through its representation (an instance of class `Class`) or to invoke a method (through an instance of class `Method`). This limited support for reflection called for many proposals of reflective extensions to appear. Still, the standard reflection API is really useful and widely used, for instance for serialization [SUN, 1998a], remote method invocation [SUN, 1998b] and component architectures [SUN, 2000]. Consequently, its implementation has been aggressively optimized since its first versions. It is interesting to note that it is also useful for implementing reflective extensions, since it provides basic (and necessary) features for providing behavioral reflection. Structural intercession, on the other hand, is not supported in Java, except in a limited manner when the virtual machine is running in debug mode.

The fact that so much information is present in the Java bytecode motivated many reflective extensions to be based on bytecode transformation (*e.g.* Dalang [Welch & Stroud, 1998], Kava [Welch & Stroud, 2001], Jinline [Tanter *et al.*, 2002a], Javassist [Chiba, 2000; Chiba & Nishizawa, 2003]). Transforming bytecode presents several advantages over source code transformation, as done by OpenJava [Tatsubori, 1999] and Reflective Java [Wu, 1998] for instance. First, the source code is not always available, in particular when considering binary COTS (commodity off-the-shelf) components or distributed systems, and second, since Java supports *dynamic class loading*, this makes it possible to transform classes lazily as they are loaded. An exhaustive discussion of Java bytecode transformation approaches can be found in [Tanter *et al.*, 2002a].

Apart from approaches based on source code transformation or bytecode transformation, some

approaches are based on a modified or extended virtual machine. We mentioned the case of Iguana/J in the previous section. Guaraná [Oliva & Buzato, 1999] and MetaXa [Kleinöder & Golm, 1996; Golm & Kleinöder, 1999] are other examples. Some approaches also use the fact that Java features Just-In-Time (JIT) compilation to operate at this level (*e.g.* [Matsuoka *et al.*, 1998; Redmond & Cahill, 2002]). Finally, it is also possible to use the debugging interface of Java [SUN, 1999b], although this interface is only available when the virtual machine is run in a (costly) debug mode.

Among code transformation approaches providing runtime behavioral reflection, it is interesting to note that some approaches are based on the use of *interception objects* rather than direct code transformation: this is the case of Dalang, the MOP of ProActive [Caromel *et al.*, 2001], and the standard *dynamic proxies* introduced with the JDK 1.3 [SUN, 1999a]. The major inconvenient of this approach is to introduce two objects (the interceptor and the original object) when conceptually there is only one: this gives rise to the famous "self problem" first discussed in [Lieberman, 1986]. Other disadvantages of this approach are discussed in [Welch & Stroud, 1999], where Welch and Stroud motivate the evolution of Dalang, based on interceptor objects, to Kava, based on bytecode rewriting.

2.4.3 Techniques for Efficient Reflection

Making reflective systems efficient is a truly hard challenge. Since reflection is interpretative by nature, it is highly inefficient. Therefore, if it is to be used intensively in real-world systems, its applicability is compromised. Not surprisingly, a lot of research efforts have been devoted to tackle the efficiency issue of reflective systems. There are indeed two major approaches: the first one, explored in this section, relates to techniques that basically try to anticipate execution in order to replace interpretation by *compilation* whenever possible – at the price of dynamicity; the second, that will be discussed in Section 2.4.4 rather focuses on a *partial* use of reflection, for instance by providing means for users to precisely select where reflective computation is required. The underlying intuition of this second approach is that, if reflection is seldom used, at a few appropriate places, then its inefficiency may not be such a big issue.

In this section, we discuss the three main implementation techniques for efficient reflective languages, following [Chiba, 1997]: currying, partial evaluation, and compile-time MOPs. This discussion includes a formal representation of the execution model advocated by each technique. We will extend this in Section 2.4.4 by proposing a formal representation of partial reflection, that will clarify the complementarity between the two approaches.

First of all, let us give the execution expression of a standard (*i.e.* non-reflective) program. Let P denote the text of this program. In order to execute the program, we need a compiler or an interpreter capable of doing so, that is, giving meaning to the program text. Let ξ be a semantic function, which intuitively denotes a compiler or an interpreter capable of executing a program text. The result of $\xi[\![P]\!]$ (the application of ξ to the program text denoted by P) is a *function*, in other words, a directly executable program. To model the input and output of a program, we will consider that this function takes as input an *initial environment* and produces as output a *final environment*. This is a slight shortcut, since what typically occurs is that the input data of a program

is first used to build the initial environment, and similarly, the final environment is transformed to an output result (*e.g.* by printing). Therefore, we assimilate data and environments.

The execution equation of a standard program is:

$$\xi[\![P]\!](D) \quad where\ \xi : Prog \rightarrow (Env \rightarrow Env) \tag{NR}$$

$Prog$ is the set of program texts (programs for short), and Env is the set of environments (data for short). The result of $\xi[\![P]\!]$ is thus a function that takes D (initial environment) as input to produce the result (final environment).

Now, let L be a metaprogram that also includes the program of the interpreter, P be a base-level program, and D be the base-level data given to P. Then the execution of a reflective program is described as follows:

$$\xi[\![L]\!](P, D) \quad where\ \xi : Prog \rightarrow (Prog \times Env \rightarrow Env) \tag{R}$$

The result of $\xi[\![L]\!]$ is now a function that takes both P and D as input. In other words, the metaprogram L is executed to interpret the base-level program P with the data D.

2.4.3.1 Currying

Currying is a method to change the arity of a function, named after the logician H. B. Curry. The technique of currying was applied to the CLOS MOP [Kiczales *et al.*, 1991]. The idea is to make $\xi[\![L]\!]$ return a function that takes as single parameter the program P (expression) and that returns yet another function that takes as single parameter D (environment):

$$\xi[\![L]\!](P)(D) \quad where\ \xi : Prog \rightarrow (Prog \rightarrow (Env \rightarrow Env)) \tag{C}$$

Currying by itself does not improve performance, and actually involves changing the MOP. But this technique allows the protocol implementor to cache the intermediate result $\xi[\![L]\!](P)$ and reuse it later. This way, less metacomputation is executed at runtime. Note that $\xi[\![L]\!](P)$ may even be computed in advance at load or compile time.

The currying technique has its disadvantages: it requires transforming a protocol, hence making it difficult to use, and requires the language to provide efficient lambda functions (so that applying cached functions actually represents a gain).

2.4.3.2 Partial Evaluation

Partial evaluation [Consel & Danvy, 1993; Jones *et al.*, 1993] is a technique developed to deal with late bindings that can be computed at compilation time. Given a program and some statically-known input values, partial evaluation propagates this knowledge within the program and computes an optimized version of the program, semantically equivalent, that works on the unknown input values. For instance, if D_1 is the static input to P and D_2 the unknown input, then applying a partial evaluator P_E yields a program $P' = \xi[\![P_E]\!](P, D_1)$ such that: $\xi[\![P']\!](D_2) = \xi[\![P]\!](D_1, D_2)$.

Several attempts have been made to apply this technique to "compile away the metalevel" as much as possible. For instance, [Asai *et al.*, 1996] propose an approximation of the reflective tower of metacircular interpreters that relies on duplication and sharing of environments among interpreters, which is then optimized through partial evaluation. In [Braux & Noyé, 2000], partial evaluation techniques are applied to eliminate the use of the reflection API of Java as much as possible, resulting in notable improvements in the execution of the serialization framework, for instance.

The basic idea is to partially evaluate the metalevel program with respect to the base program. The execution model of the partial evaluator is:

$$\xi[\![\xi[\![P_E]\!](L,P)]\!](D) \tag{PE}$$

Note that $\xi[\![P_E]\!](L,P)$ is equivalent to $\xi[\![L]\!](P)$ in the currying technique (expression *(C)*), except that it is a program text (to which ξ must be applied in order to be executed), not a (directly executable) function. Apart from the benefit of not requiring lambdas, this approach has the benefit of not requiring to change the protocol. However, this technique is extremely difficult to implement, as acknowledged by all researchers in this field. For instance, Asai *et al.* are unable to reach fully automatic partial evaluation in their model. Much work remains to be done for partial evaluation to become widely applicable.

2.4.3.3 Compile-time MOPs

The compile-time MOP is a technique developed by Shigeru Chiba, originally for bringing efficient reflective abilities to the compiled language C++, called OpenC++ [Chiba, 1995] (version 2). This technique was later on transposed in the Java world with OpenJava [Tatsubori *et al.*, 2000] and Javassist [Chiba, 2000; Chiba & Nishizawa, 2003]. The idea of a compile-time MOP is to generate a new program (text) from the application of the metaprogram. A compile-time MOP hence *substitutes* the result of the metaprogram (applied to the original program) for the original program:

$$\xi[\![\xi[\![L]\!](P)]\!](D) \tag{CM}$$

Compared to currying, a compile-time MOP does not *cache* the function returned from the application of the metaprogram. It is therefore much more efficient, but is less dynamic, since it generates a program: the result of $\xi[\![L]\!](P)$ is a new program text. The difference with partial evaluation is just that a general-purpose partial evaluator is not used. Rather, this technique acts as a partial evaluator specialized for L.

As mentioned in [Malenfant *et al.*, 1996b], although being a *static* metaprogramming technique, compile-time MOPs do not imply that everything is done prior to execution. For instance, Javassist exploits the fact that Java gives access to parts of program text (class definitions) at *load time*, which occurs during execution. Javassist is therefore a compile-time MOP operating at load time, also called a *load-time MOP*. Metacomputations can also be moved to runtime, if dynamic compilation is available. As argued in [Keppel *et al.*, 1991], *"although it costs something to run the compiler at runtime, runtime code generation can sometimes produce code that is enough faster to pay back the dynamic compile costs"*. The great impact of just-in-time compilers in modern

virtual machines, like for Java, actually confirms this fact, all the more as their techniques are ever improving. Logically, attempts have been made to build dynamic compile-time MOPs [Matsuoka *et al.*, 1998], operating as JIT compilers.

Finally, it is interesting to notice that compile-time MOPs can be used to implement runtime MOPs: hook introduction can indeed be viewed as a static metaprogramming technique. Chiba has shown how Javassist can be used to quickly implement a simple runtime MOP [Chiba, 2000]. In fact, compile-time MOPs are advanced macro processing systems, which have the particularity that the data structures used for processing are metaobjects, rather than abstract syntax trees. Chiba has highlighted that this very difference makes compile-time MOPs particularly well-suited to easily implement a large range of transformations of object-oriented programs [Chiba, 1998].

2.4.4 Partial Reflection

The idea of *partial reflection* was first motivated in the 1990 OOPSLA/ECOOP workshop on Reflection and Metalevel Architectures in Object-Oriented Programming [Ibrahim, 1990]. First of all, Brian Smith asserted that there exists a continuous spectrum of causal connection, between a base level and a metalevel. One end of the spectrum represents traditional, non-reflective, systems, while at the other end lie systems where the causal connection is full, like in 3-Lisp. In between are *partial connections*, and Smith argued that this is where most real world problems lie, and that research efforts should focus on this middle range. During this workshop, the inefficiency of reflection was discussed. Reflection was said to be inefficient because, as opposed to compilation, which consists in *embedding* a set of assumptions, reflection *retracts* some of these assumptions. Having such retractions everywhere, to achieve *full reflection*, is the cause for inefficiency. Therefore the idea that careful consideration must be taken when choosing what needs to be reflected upon was suggested: this is partial reflection. At that time, nothing was indeed said about what it means for a system to be partially reflective, or what means should be provided to specify the partiality of reflection.

In this section, we first propose a formal definition of the execution model advocated by partial reflection. Then we discuss approaches to selective reification (Section 2.4.4.2). The issue of how selectivity is defined is the subject of Section 2.4.4.3, while Section 2.4.4.4 examines the problem of specifying the actual protocol between base objects and metaobjects, including the shape of reifications.

2.4.4.1 Execution model

We hereby formulate an attempt to describe the approach of partial reflection with execution expressions, as in Section 2.4.3, with the objective to clarify the difference and complementarity between partial reflection and other approaches to efficient reflection. The idea of partial reflection is to precisely select a subset of a program P to be reflected upon, say P_r. Conceptually, P_r is actually interpreted by a (localized) metalevel program L, while the rest of the program, say P_{nr}, is executed directly. Considering $|P|$ as denoting the *size* of a program P (*e.g.* in terms of structures and execution points), we can introduce ρ as the *degree of reflectivity* of a partially-reflective

program:

$$\rho = \frac{|P_r|}{|P|} \quad \rho \in [0,1]$$

This arbitrary measure is an intuitive formalization of the fraction of reified structures and execution points in a program. Since $|P| = |P_r| + |P_{nr}|$, it follows that $|P_{nr}| = (1-\rho)|P|$. In the partial reflection approach, execution is therefore described by two *coexisting* expressions:

$$\xi[\![P_{nr}]\!](D) \quad and \quad \xi[\![L]\!](P_r, D) \tag{PR}$$

The left expression of *(PR)* describes the part of the program that is directly executed and is therefore the same as *(NR)*. The right expression describes the reflected part of the program, and is therefore similar to *(R)*. This double expression illustrates the fact that implementation techniques presented in the previous section are not incompatible with partial reflection, since they can be applied to make $\xi[\![L]\!](P_r, D)$ more efficient. The difference in perspective is also highlighted, since partial reflection allows an *hybrid execution model*, made up of two simultaneous execution expressions.

Approaches to partial reflection can therefore be discriminated based on the possibilities they offer to specify ρ, as well as on the permanency of ρ (see Section 2.4.2.3). Dynamic approaches that support dynamic binding at runtime will typically make it possible to highly control the degree of reflectivity of a partially-reflective application during execution. Conversely, fully static approaches will provide means to fix ρ once and for all before execution.

2.4.4.2 Selective reification

A major dimension of specifying the degree of reflectivity of an application lies in selectively specifying what should be reified in an application. Considering a program P as a set of structures $\{s \in Struct\}$ and a set of execution points $\{ep \in ExecPoints\}$, ρ is determined by the function:

$$\pi : \{Struct, ExecPoints\} \rightarrow \{True, False\}$$

that specifies which structures and execution points of P should be reified. It has to be noted that this definition of π is theoretical: most approaches do not discriminate execution points as such but are restricted to selecting expressions in the code (which can be seen as families of execution points). Some approaches do not take structures into account, and yet others do not even give explicit control over reified expressions.

For instance, Iguana [Gowing & Cahill, 1996] –the first approach to our knowledge aiming at offering selective reification in a systematic, fine-grained, and flexible manner– makes it possible to select program elements down to expressions (not execution points). Metaobject protocols can be defined for some reification categories, and be attached to structures or expressions in program code.

Conversely to Iguana, a large number of runtime reflective extensions are only targeted at controlling method invocation. Most of these extensions, like Dalang [Welch & Stroud, 1998], Reflective Java [Wu, 1998], the ProActive MOP [Caromel *et al.*, 2001], MetaXa [Kleinöder & Golm, 1996; Golm & Kleinöder, 1999] and Guaraná [Oliva & Buzato, 1999] (all in the context

of Java), only make it possible to select which classes are made reflective. However this selection does not mean that a class is reified as such, but that all method invocations on (instances of) this class will be reified: the set of execution points is implicit, determined at a coarse-grained level, the class.

2.4.4.3 Definition approach

Specifying what should be reified in an application consists in telling the implicit MOP what to do. This can be done *intrusively*, *i.e.* directly in base code, using an explicit MOP (Section 2.2.2) or annotations, or *non-intrusively*, for instance via configuration files. This specification can be either *extensional* or *intentional*.

An extensional definition means that the programmer is responsible for explicitly identifying particular elements that are reified by the implicit MOP. This is usually done using an explicit MOP or annotations. For instance, the MOP of ProActive [Caromel *et al.*, 1998] only gives access to the implicit MOP via the explicit MOP. In Iguana [Gowing & Cahill, 1996], the task of (meta-object) *protocol selection* is done by placing annotations in the application source code in order to indicate reflective elements: reifying the expression `obj->method()` using protocol `P` is done by manually wrapping this expression as `(obj->method() ==> P)`. OpenJava [Tatsubori, 1999] also relies on a kind of annotations placed in the source code.

A less intrusive, more declarative approach is used in, *e.g.* Kava [Welch & Stroud, 2001], Reflective Java [Wu, 1998], and Iguana/J [Redmond & Cahill, 2002], with configuration files. For instance, Reflective Java provides a small script language to specify links between base and metalevels. But still, most of the declaration is extensional (though not intrusive, as opposed to annotations). Iguana/J allows a small level of intentionality by allowing wildcards in its association declarations. Nevertheless, a fully-intentional definition would rather require the possibility to define *predicates* over program elements. Logic Metaprogramming [Wuyts, 1998; Wuyts, 2001; De Volder & D'Hondt, 1999] (LMP) is a brilliant example of intentional definition, where logic facts and inference rules are used to determine elements to reflect upon and associated metalevel computation. Intentional definition has been widely accepted in approaches to aspect-oriented programming, presented in Section 2.6.

2.4.4.4 Actual MOP definition

With *actual MOP definition*, we refer to the definition of the actual *interface* of metaobjects used by an implicit MOP. For instance, an Iguana [Gowing & Cahill, 1996] metaobject controlling the invocation of methods has a method named `invoke` that takes as parameters an object, a method pointer, and actual invocation arguments packed in an array. Iguana/J [Redmond & Cahill, 2002] adopts a similar interface too, except that the method name is `execute`, the order of parameters is not the same, and the method pointer is rather a `Method` object, as provided by Java. In Kava [Welch & Stroud, 2001], the metaobject controlling field read accesses must have both a `beforePutField` and an `afterPutField` method that take as parameters the name of the accessed field and the field value.

All reflective systems therefore provide *fixed MOP*s: as flexible as they may be, they impose the actual interfaces of metaobjects. Furthermore, since they do not allow the precise selection of reified information, they all adopt a general approach, reifying all the information describing the intercepted operation (although the shape may change, between plain parameters, arrays, or object wrappers). This is unfortunate when a metaobject actually does not need some part of this information (or worse, if it does not need any information at all), because reification has a cost, since it may imply repetitively constructing arrays or instantiating some wrapper classes. Furthermore, these MOPs only reify the information that describes the intercepted operation as implicitly conceived by their designers.

Consider the case of Reflective Java [Wu, 1998]: it interestingly introduces the notion of *method categories*, as a way to distinguish between reified method invocations. In their script language, users can for instance specify that *get* methods are of one category, while *put* methods are of another category. This kind of classification is obviously interesting for better metalevel engineering. However the way that it is done in Reflective Java is quite questionable: the category information is passed at runtime to metaobjects as an extra parameter, which is typically tested via a switch statement in order to determine the corresponding metabehavior –a not-so-nice object-oriented programming practice indeed. Therefore, although Reflective Java already represents a progress over other approaches that do not support such classification within a bunch of reified operations, it further highlights the limitation of fixed MOPs. In the case of categories, one would like to be able to have different metaobject methods invoked depending on the category of the reified method invocation.

High flexibility in specifying the MOP –in its most essential meaning of *information bridge* between base objects and metaobjects– has therefore not been addressed by runtime reflective systems. A possible reason for this, apart from the fact that it may not have been identified as a key issue by runtime MOP designers, may reside in the difficulty of making this decision accessible to users. If we consider compile-time and load-time MOPs, only Javassist version 2 and above [Chiba & Nishizawa, 2003] actually makes it possible to extract whatever piece of information from a base-level program in a convenient manner.

2.5 Open Implementations

Since the beginning of this chapter, reflection has always been considered in the context of programming languages. Even though we have discussed the notion of *reflective applications* that exhibit specifically-tailored reflective capabilities, as motivated by [Douence & Südholt, 2000], the focus has always been to reify elements of programs related to their structure (as defined by the language) or their execution semantics. This section discusses a more general notion of reflection, which led to the introduction of *open implementations*. Section 2.5.1 is an introduction to the concept, Section 2.5.2 explains why open implementations are important, and finally, Section 2.5.3 surveys techniques and approaches for providing open implementations.

2.5.1 Implementational Reflection and Open Implementations

Ramana Rao first established that the conceptual framework of reflection may be useful not only for building programming languages but also for building malleable systems of all kinds [Rao, 1991]. Indeed, as he remarks, most significant systems do not only depend on the language constructs and semantics, but also on the other systems they make use of. Rao therefore reformulates the framework of reflection in terms of a system's *implementation*. To the concept of computational reflection, he opposes that of *implementational reflection*, and to that of reflective architecture, that of *open implementation*:

Definition 2.13 – Implementational reflection

Reflection that involves inspecting and/or manipulating the implementational structures of other systems used by a program. □

Two observations can help understand the relation between computational reflection and implementational reflection [Rao, 1991]. On the one hand, a language interpreter is itself the implementation of a language: this suggests that computational reflection is a special case of implementational reflection. On the other hand, the interface of any system can be seen as a language[8], and the implementation of the system as an interpreter for that language: this now suggests that implementational reflection is a special case of computational reflection. Rao logically suggests that computational reflection and implementational reflection are just different characterization of the same essential capability.

As with computational reflection, basic access to implementational aspects of a system does not make it an open implementation as such. The difference is similar to that made between reflective facilities and fully reflective architectures (Section 2.1.4). Recall that a language with a reflective architecture allows much more open-ended access to the implementation of a language [Maes & Nardi, 1988], in particular by allowing users to write code that is called by the language interpreter. Therefore the concept of a reflective architecture can be reformulated in terms of the implementation of a system, and leads to the concept of *open implementation* [Rao, 1991]:

Definition 2.14 – Open implementation

A system with an *open implementation* provides (at least) two linked interfaces to its clients: a *base-level* interface to the functionality of the system similar to the interface of other such systems, and a *metalevel interface* that reveals some aspects of how the base-level interface is implemented. □

[8] This statement by Rao, also mentioned elsewhere [Ibrahim, 1990], might be a slight overstatement indeed: the interface of a system may be more precisely viewed as a *vocabulary* rather than as a language as such, although there is a language behind the correct use of this vocabulary.

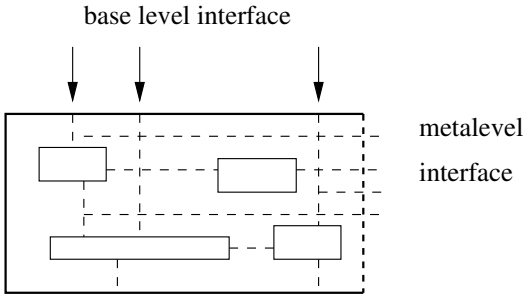

Figure 2.4 – A system with an open implementation. (From [Rao, 1991].)

One of the roles of the metalevel interface is to specify points at which users can add code that implements some base-level behavior with different semantics and/or performance (Fig. 2.4). Rao notices that the causal connection requirement of reflection is straightforwardly met since metalevel code actually directly implements aspects of the base level. Indeed, reflective systems providing metaobject protocols are open implementations of interpreters (*e.g.* [Kiczales *et al.*, 1991; Douence & Südholt, 2001]), and systems providing compile-time MOPs are open implementations of compilers (*e.g.* [Lamping *et al.*, 1992; Chiba, 1995]).

Patrick Steyaert has proposed a very interesting account of reflection [Steyaert, 1994], which is entirely based on open implementations rather than on a tower of metacircular interpreters. This work is focused on open implementations for programming languages, proposing a framework for object-based languages. In [de Volder & Steyaert, 1995], this approach is used to construct the reflective tower based on open implementations, bringing a cleaner understanding of reflection. In his PhD dissertation, Steyaert actually uses open implementations as a criteria to differentiate a language with *reflective facilities* from a language with a *reflective architecture*: the former only requires implementational access to the metasystem, while the latter derives from access to the metalevel interface of an open-implemented metasystem. The open implementation of a programming language is implemented in one language, called the *implementation language*, and actually implements a *set* of languages (depending on the metalevel interface), called the *engendered languages*. He further argues that not every open implementation is suitable as the basis for a reflective architecture, introducing the notion of open implementations with *reflective potential*. Such potential consists in that *"all first class values (primitive values, functions, objects, etc.) can freely travel between implementation language and engendered language, and that both languages can transparently use each others first class values"*. This property is known as *linguistic symbiosis* [Ichisugi *et al.*, 1992].

The idea of metacircularity, which refers to the fact that an interpreter is written in the same language that it interprets, does not really fit well in the realm of open implementations. Rao discusses this issue in the context of Silica, a window system: although the metalevel of Silica is written in the same language used to implement its base level, it is indeed not written in the base-level "language" that it provides. Actually, it does not even make sense to write a window system in the "window system language" that it implements. This observation seems to apply as well to all systems that are not programming language interpreters.

2.5.2 Why Implementations Should Be Opened Up

The idea to expose implementation details to clients may seem in a first place highly contradictory to traditional software design principles. The so-called *black-box abstraction* principle is a basic tenet of software design that states that a module should expose its functionality but hide its implementation. Following this principle, issues of the implementation of an interface are not part of client's concerns, and should therefore be completely hidden from them.

However, as argued in [Kiczales, 1992], any concrete implementation of a high-level system requires fixing a number of tradeoffs. In addition, the higher the level of a system is, the more tradeoffs there are [Kiczales *et al.*, 1993]. As a matter of fact, it is *not* possible to provide a single, fixed, closed implementation of such a system that will satisfy all users. This is particularly true when considering performance characteristics. In the context of programming languages, this was first noticed by Wirth:

> *"I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details."."* [Wirth, 1974]

A classical example used to illustrate the need to control implementation strategies is that of the way instances are implemented in a class-based language [Kiczales *et al.*, 1991]. Consider a class `Position` with two instance variables `x` and `y`, and a class `Person` with potentially a thousand instance variables, corresponding to the many properties that can actually describe a given person. It is clear that the ideal implementation strategy for these two classes are completely different. For `Position`, an array-like strategy is ideal, providing compact storage and quick access to both variables. For `Person`, on the other hand, a hashtable-like strategy would be more appropriate, avoiding to allocate a high amount of memory when it is highly probable that not all variables will be used.

A nefast effect of the black-box abstraction is that when facing similar issues, clients usually "code around" the problem either by re-implementing an appropriate version of a module or by using existing modules in contorted ways [Kiczales, 1992]. A reverse approach to the black-box abstraction is the so-called *white-box* approach, which consists in exposing each and every detail of a system's implementation. For instance, an object-oriented program distributed under an open source license makes it possible for users to tune the system according to their needs. However, giving access to the source code, although object-oriented, is not a guarantee that the implementation is well-enough structured to allow users to benefit from accessing it [Rao, 1991].

The open implementation approach therefore advocates to open up the implementation, but to do so in a *principled*, *disciplined* way [Rao, 1991; Kiczales, 1992]. The idea is not to make it possible for users to arbitrarily alter the implementation of a system. Using reflection parlance, an open implementation *reifies* some aspects of implementation, leaving others implicit [Rao, 1991]. An open implementation actually makes it possible to "re-make" some of the tradeoffs in the system to better suit their needs [Kiczales *et al.*, 1993], as well as customizing behavior. As a matter of fact, object-oriented programming, thanks to inheritance and polymorphism, is a particularly useful paradigm for developing open implementations. Therefore, an open implementation provides a *well-defined* interface to the implementation of the system. This interface can be exploited to

create either useful semantical variations or efficient implementations for particular situations. As Rao argues, *"explicitly focusing on the metalevel as a separate and first-class interface to export to the user forces a greater attention to exposing important design and implementation choices"*.

Kiczales has coined this framework as the *dual interface framework*. Under this framework, the client first writes a base program through the traditional interface, and then, if necessary, writes a metaprogram through the "adjustment interface" to customize the underlying implementation to meet the needs of the base program. He makes an enlightening digression to explain the intuition behind this model, based on a parallel between programming and physics, introducing the notion of *physically correct computing* [Kiczales, 1992]:

> *"There is a deep difference between what we do and what mathematicians do. The 'abstractions' we manipulate are not, in point of fact, abstract. They are backed by real pieces of code, running on real machines, consuming real energy and taking up real space. To attempt to completely ignore the underlying implementation is like trying to completely ignore the laws of physics; it may be tempting but it won't get us very far.*
> *Instead, what is possible is to temporarily* set aside *concern for some (or even all) of the laws of physics. This is what the dual interface model does: In the base-level interface we set physics aside, and focus on what behavior we want to build; in the meta-level interface we respect physics by making sure that the underlying implementation efficiently supports what we are doing. Because the two are* separate, *we can work with one without the other, in accordance with the primary purpose of abstraction, which is to give a handle on complexity. But, because the two are* coupled, *we have an effective handle on the underlying implementation when we need it. I like to call this kind of abstraction, in which we sometimes elide, but never ignore the underlying implementation 'physically correct computing'*."

Finally, another interest of opening up implementations is that the system need not provide direct support for functionalities that only some users want. Users can provide these for themselves using the metalevel interface. This point was particularly critical for Kiczales *et al*. as they were working on the CLOS standard, facing a traditional dilemma: needs of backward compatibility that were contradictory to important goals of an improved design, allowing both extensibility and efficiency. Opening up CLOS, thanks to the CLOS MOP, made it possible to support a "CLOS region", rather than a single "CLOS point" [Kiczales *et al.*, 1991], hence solving the dilemma they were facing.

2.5.3 Providing Open Implementations

Providing an open implementation of a system is naturally a more challenging task than simply providing a standard, closed implementation. In this section we first discuss the particularity of open implementation design. Then we review open implementation interface styles that have been identified in the literature. We subsequently discuss the kind of techniques that can be useful to open implementations. Finally, the crucial issue of *locality* is analyzed.

2.5.3.1 Designing the metalevel interface

In [Kiczales *et al.*, 1993], interesting elements about the particularities of MOP design versus traditional language design are discussed. We hereby generalize this discussion, by contrasting open implementation (rather than just MOP) design versus traditional system (rather than just programming language) design.

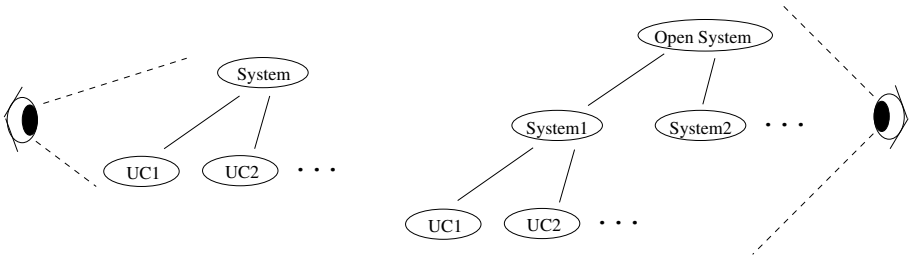

Figure 2.5 – Contrasting traditional system design (left) and open implementation design (right). (Adapted from [Kiczales *et al.*, 1993].)

A system designer typically considers a range of use cases and features the system should support elegantly. The system is designed accordingly. This process will generally be iterative and ad hoc, but the point is that the designer is working with two different levels of design at the same time: the level of designing particular use cases in terms of a given system, and the level of designing the system itself to support the lower-level design processes. This is illustrated on the left part of Fig. 2.5. Open implementation design is similar, with the addition of yet one more level of design process. In this case, the designer is not thinking about a single system that can be used to handle the use cases, but rather a whole *range* of systems, that can support an even wider set of use cases. This is illustrated on the right part of Fig. 2.5.

Therefore, the first question that pops up when designing an open implementation is *what range* of implementations users should be able to specify [Lamping *et al.*, 1992]. As Kiczales puts it, "*opening an implementation critically depends on understanding not just one implementation the clients might want, but also the various kinds of variability around that point they might want*" [Kiczales, 1992]. Not surprisingly, getting a clear understanding of the desired *implementation space* is inherently *iterative*. It is indeed difficult to reach enough generality and fine enough granularity to generate a wide variety of implementations. This fact tends to sustain the idea that there may not be any single perfect open implementation design. Iterative refinement is the way to go, but as Kiczales highlights, user feedback and complaints about previous systems and implementations take on tremendous value in this quest. The CLOS MOP is acknowledged to be the fruit of continuous refinement based on the feedback from a large community over five years [Kiczales *et al.*, 1991; Kiczales *et al.*, 1993].

An important objective of an open implementation is that users should be able to describe the aspects of the implementation that they care about, without talking about others: this is *locality*, an important issue that we actually already discussed in the context of reflective systems (Section 2.3).

For instance, metaobjects per object or per class, group-wide reflection, hybrid group reflection, fine-grained MOPs and other approaches to the metalink can all be viewed as experiments with locality [Kiczales, 1992]. We shall come back on this issue at the end of this section on open implementations.

2.5.3.2 Open implementation interface styles

In a study on open implementation design, Kiczales *et al.* presented three open implementation interface styles that are recurrent alternatives when trying to conceive open implementations [Kiczales *et al.*, 1997a]. These three styles for module interface design are given unfortunate names of B, C, and D (when style A is the black-box design). We will rather refer to them as *declarative* style, *strategy* style and *layered* style, respectively (we believe these names match the characterization done by the authors):

declarative style: the interface makes it possible for the user to describe, in a simple *declarative* language, the expected usage of the module.

strategy style: the user is given the possibility to *choose* the appropriate strategy in a fixed list of available strategies.

layered style: the user may, in addition to selecting one of the built-in strategies, *provide* a new strategy.

It has to be noted that all these styles are optional, in the sense that the user is left with the possibility of not declaring, selecting or providing anything, and get a default implementation strategy.

[Kiczales *et al.*, 1997a] then discusses the associated tradeoffs and types of appropriate situations of these styles. The declarative style has the advantage of not constraining the implementation, but it makes it difficult for the client to know how the provided information influences the module strategy. This approach is most appropriate when it is easy to choose an effective implementation strategy if the client behavior is known. Conversely, with the strategy style, the client precisely selects the strategy to use. However, he might choose badly. This style is appropriate when there is a few candidate implementation strategies, but it is difficult to choose among them automatically. Finally, in the layered style the module adopts the strategy provided by the client. It has the same advantage as the strategy style, but introduces a higher level of complexity for both parties: designing a module to support replaceable strategies might be difficult, and it might be hard as well for the client to build a new strategy. Therefore this approach is most adequate when it is not feasible for the module to implement all strategies that might be useful to the clients. It actually has the advantage of being a *layered* interface design, in the sense that it subsumes both the black-box style (if the user does not specify anything), and the strategy style (if the user is satisfied with a built-in strategy). This makes layering a good technique to balance ease of use and power.

2.5.3.3 Reflection and open implementations

The different interface styles we have just discussed may typically be implemented explicitly, via the use of *design patterns*: for instance, the strategy design pattern [Gamma *et al.*, 1994] is a good candidate for implementing the strategy and layered open implementation interface styles. However, as Rao mentions, the architecture or facilities prescribed by the metalevel interface must not prevent efficient and effective implementation of the base level [Rao, 1991]. What Rao is pointing at is that providing *explicit* representations of any aspect of a system's implementation (*e.g.* through strategy objects) may have consequences for the resulting efficiency of the system. Therefore, explicit reification may not always be appropriate. Rao hence underlines that *lazy* reification, or reification on demand, is a typical strategy for making implementation state explicit.

An interesting observation is then made: reflective capabilities of an object-oriented language is a possible implementation technique for lazy reification, as we have seen since the beginning of this chapter. In other words, metaobject protocols are a possible approach to selectively reify some elements of a system's implementation in order to open it. This idea is further discussed in [Gowing & Cahill, 1996]. The MOP-based approach is contrasted to an extremist open implementation approach where most of the implementation aspects are reified. Using fine-grained MOPs (Section 2.3.2.2), metalevel objects are exposed for certain features of the underlying (black-box) system, precisely selected by the client of the system. Furthermore, since metaobjects can be changed at runtime, dynamic adaptation of the selected features can be achieved through dynamic rebinding of metalevel objects. Therefore, the loop is closed: open implementations come from a generalization of the ideas of computational reflection to any kind of system, and computational reflection can be used to achieve open implementations.

2.5.3.4 Mastering Locality: Towards Aspect-Oriented Programming

The issue of locality has been recurrently mentioned since the beginning of this chapter. In [Kiczales *et al.*, 1993], five coarse notions of locality are discussed, which are neither sharp nor orthogonal, but yet useful to talk about this rather intuitive notion. Again, they are discussed in the context of metaobject protocols, but we generalize to the case of open implementations:

- *feature* locality refers to the fact that an open implementation should provide access to individual features of the base system;
- *textual* locality means that convenient means should be provided for users to indicate what behavior of the base system they would like to be different;
- *object* locality is the possibility to affect the implementation on a per-object basis;
- *strategy* locality refers to the possibility to affect individual parts of an implementation strategy;
- *implementation* locality points to the fact that a simple customization should be simple to implement: good default implementation must be provided, along with means for *incremental* deviation from that default.

In an obstinate effort to understand the issue of locality, Kiczales actually ended up proposing Aspect-Oriented Programming, discussed in the next section. We hereby would like to report on the early manifestations of the underlying intuition, which is made explicit in [Kiczales, 1992]. Kiczales makes an analogy with a discussion between humans to get some insight on the problems that appear when trying to concretely work with the dual interface framework: when a provider and a client discuss, they most of the time do so "at the base level", *i.e.* they actually talk about the functionality of a system. But from time to time, they go "meta" and start talking about *how* functionality should be achieved, and other non-functional requirements. The following insightful observations are hence made:

> *"[...] very often, the concepts that are most natural to use at the meta-level* cross-cut *those provided at the base level."*
> *"[...] We are, in essence, trying to find a way to provide two effective views of a system through cross-cutting* localities*."*
> *"[...] The structure of complex systems is such that it is natural for people to make this jump from one locality to another, and* we have to find a way to support that."
> [Kiczales, 1992]

At that time, only the problem was formulated. Apart from recognizing that the dual-interface framework might therefore very well evolve to a *multi-interface framework*, no sketch of a solution was given to this locality issue. A few years later, and after a bunch of concrete case studies, the first paper presenting Aspect-Oriented Programming [Kiczales *et al.*, 1997b] was published at ECOOP.

2.6 Overview of Aspect-Oriented Programming

Although the first sections of this chapter can be considered as a fairly comprehensive introduction to the area of reflection, this section does not pretend to be one for Aspect-Oriented Programming (AOP). Rather, we will limit ourselves to a brief overview of the major constituents of this approach. Chapter 6 of this dissertation includes a more complete analysis of aspect-oriented approaches, since it serves as the basis for one of our contributions.

Section 2.6.1 introduces the origins and concepts of AOP. We will take a closer look at aspect-oriented languages in Section 2.6.2. This will lead us in Section 2.6.3 to a first highlighting of the inherently deep relation between AOP and reflection. Finally, we briefly present some AOP systems in Section 2.6.4.

2.6.1 Origins and Concepts

Aspect-Oriented Programming (AOP) has emerged as a programming technique for better separation of concerns. As discussed at the end of the previous section on open implementations, one of the challenging issue is that of supporting *crosscutting localities* (Section 2.5.3.4). Although metaobject protocols and open implementations can be helpful at better modularizing software,

the problem that complex systems require different, crosscutting descriptions remains. This first section on AOP reports on the key concepts exactly as they have been introduced in [Kiczales *et al.*, 1997b].

2.6.1.1 Crosscutting, code tangling and aspects

In [Kiczales *et al.*, 1997b], a shift in terminology is exposed. The term *aspect* is proposed to denote the issues addressed by design decisions which are hard to cleanly capture in the existing entities provided to modularize code (*e.g.* classes and methods). Kiczales *et al.* argue that such issues are inherently hard to cleanly modularize because of their nature: they, by essence, *crosscut* the basic functionality of the system. As a result, code is *tangled*: this means that modules of the system get "polluted" with code dedicated to address such aspects. This leads to less understandable and reusable code. Aspect-Oriented Programming is therefore proposed as a programming paradigm to cleanly express programs with such aspects, including isolation, composition and reuse of aspect code.

Underneath this work lies the obsession to resolve the tension between understandability and efficiency. Recall that this was already the *motto* of Kiczales and colleagues in their work on open implementations: an open implementation makes it possible to retain understandability at the base level, while addressing "realistic" issues of performance via the metalevel interface. An example is given to illustrate that implementing a system with traditional, generalized procedure (GP) languages, one can either obtain an understandable but inefficient solution, or an efficient one that is unfortunately complex to develop and understand. This tension comes from crosscutting, defined as follows:

> *"[...] whenever two properties being programmed must compose differently and yet be coordinated, we say that they* cross-cut *each other.*" [Kiczales *et al.*, 1997b]

To compose modules, GP languages offer a single mechanism: procedure call. Therefore, when facing the implementation of crosscutting properties, the programmer must *co-compose* them manually, leading to complexity and tangled code.

Having said this, the authors precise the meaning of two important terms: *component* and *aspect*. Considering a system and its implementation using a GP language, a property that must be implemented is:

- *"A component, if it can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)"*.
- *"An aspect, if it cannot be cleanly encapsulated in a generalized procedure"*. [Kiczales *et al.*, 1997b]

What it means for a property to be "cleanly" encapsulated is further precised: well-localized, easily accessed and composed as necessary. It is furthermore noticed that components tend to be units of the system's functional decomposition, whereas aspects tend not to be so. Aspects rather tend to be properties that affect the performance or semantics of the components in systemic ways. The precise goal of AOP is therefore:

> *"To support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system."* [Kiczales *et al.*, 1997b]

An interesting remark is made that helps understand the idea of additional composition mechanisms that could aid programmers to develop complex systems. Indeed, most programmers are used to at least one of such mechanisms: the *catch and throw* mechanism. Such a mechanism actually provides a different composition mechanism that helps programmers implement certain aspects of their systems (in this case, the behavior the system should adopt in exceptional situations).

2.6.1.2 Elements of an aspect-oriented implementation

The implementation of an application typically consists of three ingredients: a programming language, an interpreter or compiler for that language, and a program. An aspect-oriented implementation is analogous. It consists of: a *component language* to program components, one or more *aspect languages* to program aspects, an *aspect weaver* to combine the languages, and finally, a component program, and one or more aspect programs [Kiczales *et al.*, 1997b]. As with GP-languages, aspect languages and aspect weavers can be designed to operate at runtime or at compile time.

As further discussed, designing an AOP system implies understanding well what should be the component language, what should be the aspect languages, and what must be shared between them. Component languages are typically GP-languages. Care must be taken however not to preempt anything the aspect programs need to control. Aspect languages should preferably be high-level and more tailored to the specific aspect they are tackling, in order to be easier and safer for the programmer to use. As argued in [Kiczales *et al.*, 1997b]:

> *"Proper use of AOP means that [the programmers] are expressing implementation strategies at an appropriately abstract level, through an appropriate aspect language, with appropriate locality."*

Aspect weavers must process the components and aspects and co-compose them properly to produce the overall system. Essential to the function of a weaver is the notion of *join points*, which are *"those elements of the component language semantics that the aspect programs coordinate with"* [Kiczales *et al.*, 1997b]. The job of the weaver is therefore to integrate the aspect programs in the component program at appropriate places, according to the *join point representation* of the component program.

2.6.2 Aspect-Oriented Languages

We now enter in more details concerning aspect languages. We review the notion of join points and join-point models further, before detailing the widely-used *Pointcut and Advice* model. We end this section by contrasting domain-specific aspect languages and general-purpose aspect languages.

2.6.2.1 Join-point models

There are various models of join points. As discussed in [Masuhara *et al.*, 2003], a join-point model consists of three elements:

- The *join points*, which are the points of reference that aspect programs can affect;
- A *means of identifying* join points;
- A *means of effecting* execution at join points.

Join points may be *dynamic* join points if they denote runtime actions, such as method calls, method executions, object creations, etc. They are called *lexical* join points if they simply refer to locations in the code (*e.g.* expressions). Join points may denote something else, for instance data flows in the component program [Kiczales *et al.*, 1997b], or elements of a class graph, such as in the work on adaptive programming and Demeter based on graph traversal [Lieberherr & Silva-Lepe, 1994]. But as a matter of fact, most of the work on AOP is based on a join-point model similar to that of AspectJ [Kiczales *et al.*, 2001] –characterized as the *dynamic crosscutting* mechanism–, where join points are well-defined execution points in the execution of a program. This model has a lot in common, to say the least, with the operational decomposition advocated by McAffer (Section 2.3.2.1).

An interesting dimension of AOP has been to draw special attention on how to identify, or select, join points. In the widely-accepted AspectJ terminology, a set of join points is called a *pointcut*. The introduction of control-flow based crosscutting, in which a pointcut depends on a control flow relation with respect to another pointcut, is very valuable. AspectJ first introduced it [Kiczales *et al.*, 2001], but in a somehow restrictive manner. Douence and Teboul proposed much more expressive means to reason on control flow [Douence & Teboul, 2004]. The will to explore data-flow based crosscutting is also mentioned in [Kiczales *et al.*, 2001]. To the best of our knowledge, only Masuhara and Kawauchi explored a concrete approach to data-flow based crosscutting, by proposing a dataflow pointcut [Masuhara & Kawauchi, 2003]. As a matter of fact, they mention an implementation based on a simplified version of AspectJ, hence suggesting that dataflow is a higher-level abstraction built from standard execution pointcuts. Identifying join points is typically done in a declarative manner, as we will illustrate in the next section. De Volder has proposed a logic meta-programming (LMP) approach to AOP, where logical queries can be used to specify crosscuts [De Volder & D'Hondt, 1999]. This approach has many advantages since it can benefit from unification to define parametric crosscuts.

2.6.2.2 The Pointcut and Advice model

Masuhara *et al.* proposed a semantics-based compilation model for an aspect-oriented language that follows the Pointcut and Advice (PA) model, based on its operational semantics [Masuhara *et al.*, 2003]. A formalization of a procedural subset of PA is presented in [Wand *et al.*, 2004]. PA is the most widely-adopted AOP model today.

PA is however not the only model of AOP. Approaches like traversal specification in Demeter [Lieberherr & Silva-Lepe, 1994], class composition as in Hyper/J [Ossher & Tarr, 2001]

and open classes as in MultiJava [Clifton *et al.*, 2000] and AspectJ, are also mechanisms for modular crosscutting. Masuhara and Kiczales studied these models and proposed a characterization of what is required to support crosscutting structure, basically, a common frame of reference that two or more programs can use to connect with each other and each provide their semantic contribution [Masuhara & Kiczales, 2003].

In PA, a join point is an *action* during program execution. The model presented by Masuhara *et al.* is restrained to method calls, method executions, object creations and advice execution. The *kind* of a join point is defined as the kind of action (*e.g.* `call`). The means of identifying join points is the *pointcut* mechanism: a predicate on join points, used to specify the join points that a piece of advice applies to. Primitive pointcuts either correspond to the join point kinds (*e.g.* `call(`m`)`) or match join points with values of a given type (*e.g.* `args(`$t\ v$`,...)`). They can be combined using logical operators (`&&`, `||` and `!`). High-order pointcuts are provided, such as `cflow(`p`)`, which matches join points that have a join point matching their sub-pointcuts in the call stack (this is control-flow based crosscutting as supported by, *e.g.* AspectJ [Kiczales *et al.*, 2001]).

In PA, the means of effecting the execution at join points is the *advice* mechanism. An advice contains a pointcut and a body expression, executed when a join point matches the associated pointcut. In AspectJ, there are five kinds of advice: `before`, `after`, `around`, and two particular cases of after, `after returning` and `after throwing`. The PA model studied in [Masuhara *et al.*, 2003] only supports `before` and `after`, however.

For instance, in AspectJ syntax, the following advice definition:

```
before: call(void Point.set(int)) && args(int z)
{ write("set:"); write(z); }
```

will, before calls to method `set` of class `Point`, print a message with the value of the argument passed to `set`, which is bound to `z` thanks to the `args` sub-pointcut.

Masuhara *et al.* expose an interpreter for this model and discuss its compilation following an approach based on partial evaluation [Masuhara *et al.*, 2003]. In their approach, partial evaluation of an AOP interpreter with respect to a subject program and advice definitions generates a compiled, or *statically woven*, program. In their discussion, the interesting issue of how a pointcut match can be determined is highlighted. The matching of some pointcuts can be statically determined, while others imply *residues* to be left for runtime evaluation. The notion of *join point shadow* is introduced: they are points in the program text, where the compiler actually operates. Join point shadows indeed correspond to expressions, whereas join points correspond to execution points. Masuhara *et al.* also discuss two alternative implementations of control-flow based crosscutting: a naive stack-based implementation (where a stack of join points is continuously maintained in the interpreter), and an efficient state-based implementation where only the state of `cflow` pointcuts is maintained (and queried at appropriate places).

Douence *et al.* also presented an operational semantics of an AOP system [Douence *et al.*, 2001], labelled Event-based AOP (EAOP). In their system, a monitor pattern matches a stream of events from a program execution, and applies aspect code when a match is detected. The great expressive power of EAOP makes it possible to reason about event patterns which are much more

general than the `cflow` presented above. The advanced event pattern matching facilities contribute to raising the level of abstraction, with explicit means of composing aspects, and facilities for the detection and resolution of conflicts between aspects [Douence *et al.*, 2002].

2.6.2.3 Aspect languages

As first argued in [Kiczales *et al.*, 1997b], aspect languages should preferably be high-level and tailored to the specific aspect they are tackling, in order to be easier and safer for the programmer to use. This is why the first aspect languages with which Kiczales and colleagues experimented were domain specific [Lopes, 1997; Mendhekar *et al.*, 1997; Irwin *et al.*, 1997]. To our point of view, this represents a major strength of AOP: because a domain-specific aspect language (DSAL) is specific to a particular domain or aspect, it fits better the needs of the specialized programmers. The right concepts can be directly manipulated, additional support provided in terms of verification, debugging... However, focusing on DSALs further complicates aspect interaction and composition. Therefore much of the research effort around AOP has shifted to general approaches. The evolution of the AspectJ [Kiczales *et al.*, 2001] language is a good example of this trend, since it has quickly evolved to a general-purpose aspect language. Two notable exceptions are XAspects [Shonle *et al.*, 2003] and Aspect-Oriented Logic Meta-Programming (AOLMP) [De Volder & D'Hondt, 1999]. XAspects is an extensible system for DSALs, but it unfortunately does not say much about aspect composition. AOLMP has been shown to be a good generic metaprogramming framework to define domain-specific aspect languages and still be able to declare rules about interactions and compositions of aspects [Brichau *et al.*, 2002].

The problem of general-purpose languages is that they provide, in the end, very little guidance to programmers. Domain-specific AOP presents the advantage of being *problem-oriented* since it is expressed in terms of the problem space (the abstract specification of the problem); conversely, general-purpose AOP remains *solution-oriented* since it is expressed in terms of the solution space (the operations of the language used to build the solution). This balance of pros and cons between general-purpose and domain-specific aspect languages is also valid in the context of programming languages in general, not only aspect languages.

2.6.3 AOP and Reflection

> "*AOP is a goal, for which reflection is one powerful tool.*" [Kiczales *et al.*, 1997b]

From the previous presentation of AOP, it should be clear that there is an apparent deep relation between AOP and reflection. We hereby report on some elements about this relation that can be found in the literature.

Considering the join point model presented in Section 2.6.2.2, the join point representation that is needed for weaving can be generated at runtime using a reflective runtime for the component language [Kiczales *et al.*, 1997b]. Similarly, static weaving can be done with compile-time MOPs, such as OpenC++ [Chiba, 1995] or Javassist [Chiba, 2000; Chiba & Nishizawa, 2003]. Kiczales *et al.* discuss the fact that metalanguages crosscut base level computation, since they provide views of the computation that no one base language component could ever see. As they put it, "*in AOP*

terms, metalanguages are lower-level aspect languages whose join points are the 'hooks' that the reflective system provides" [Kiczales *et al.*, 1997b]. They actually explain that they exploited this connection when first prototyping AOP systems: they often started developing simple metaobject protocols for the component language, and prototyped imperative aspect programs using them; then, they developed more explicit aspect language support for them. Also, both the work on Demeter [Lopes & Lieberherr, 1996] and composition filters [Bergmans & Akşit, 2001] were first described as being reflective facilities, and are now being described as being AOP approaches.

Kiczales *et al.* warn however that reflective access is so powerful that it can be dangerous or difficult to use [Kiczales *et al.*, 1997b]. This was indeed a motivation for introducing domain-specific aspect languages. We totally agree with this remark, since it is what makes AOP *problem-oriented* rather than *solution-oriented*, as discussed previously. But when considering general-purpose AOP, the difference is shallow, since the advantages of domain-specific programming are lost. In the presentation of AspectJ [Kiczales *et al.*, 2001], it is said that the semantics of advice is neither a metaprogramming nor a reflective semantics, since it does not involve reification[9]. We do not really agree with this statement, since AspectJ still makes it possible to access information that was otherwise implicit in the program (*e.g.* the fact that there is a call to some method occurring with such and such values as parameters). We rather consider that the *syntax* of AspectJ makes it possible to *hide* the "meta". In the presentation of the AOP interpreter for the PA model [Masuhara *et al.*, 2003], one can clearly see that the interpreter is especially built around the notion of join point, as a new structure used and accessed at various points in the interpreter. This interpreter extension actually hides the fact that modifying the interpreter is necessary to get access to (*i.e.* to trigger advice execution upon) occurrences of otherwise implicit information. We therefore rather agree with Kiczales when he defines AOP as a *principled subset of reflection* [Kiczales, 2001].

Kojarski *et al.* further explore the relation and interactions between reflection and generic AOP [Kojarski *et al.*, 2003], in the line of AspectJ or AspectS, an AOP system for Smalltalk, actually built around the Smalltalk MOP [Hirschfeld, 2002]. They show and discuss the tradeoffs in implementing AOP with reflection and, reciprocally, to implement reflection via AOP. A result of this study is that a MOP should be rich enough to enable AOP, while AOP makes it possible to apply reflection more selectively. Selective application of reflection is also the object of partial reflection (Section 2.4.4).

2.6.4 AOP Systems

There has been a lot of AOP systems developed in the last years. We have already mentioned a few, some being domain-specific [Lopes, 1997; Mendhekar *et al.*, 1997; Irwin *et al.*, 1997; Shonle *et al.*, 2003; Brichau *et al.*, 2002], others general-purpose [Kiczales *et al.*, 2001; Hirschfeld, 2002; De Volder & D'Hondt, 1999; Bergmans & Akşit, 2001]. Many systems were actually proposed as extensions of Java. Within these, the very large majority are general-purpose, and hence have a lot in common with reflective extensions (Section 2.4.2.4). They are therefore similarly subject to characterization according to the range of binding times and modes presented in [Redmond & Cahill, 2002]. We hereby only mention just a few.

[9] With the exception of the reflective access to the current join point object that is offered.

JAC [Pawlak *et al.*, 2001] is a dynamic AOP system based on a reflective infrastructure set up at load time. PROSE [Popovici *et al.*, 2003] is a dynamic AOP system based on both a modified JIT compiler and a modified JVM. PROSE deliberately sacrifices portability to achieve high performance. Another system, Steamloom [Bockish *et al.*, 2004], relies on a dedicated virtual machine, in order to experiment with fully-dynamic weaving. These approaches, as well as AspectJ, provide fixed languages for pointcut definitions, mainly syntactic-based, and support a fixed and limited number of base operations. Chiba, inventor of OpenC++, OpenJava and Javassist, has recently proposed Josh [Chiba & Nakagawa, 2004], an open AspectJ-like language, that supports the definition of new pointcuts and means of describing advices. Josh is a static AOP system. Event-based AOP (EAOP) [Douence *et al.*, 2001] has been prototyped as a Java extension relying on code transformation to generate events (reifications) for the execution monitor (metaobject). Again, the interesting part of the work on EAOP is the way their "metalevel" is structured, allowing reasoning about sequences of events. Recently, due to the great interest AOP is gaining in the industry, providers of Java application servers have developed their AOP systems: this is the case of BEA with AspectWerkz [AspectWerkz, 2002], and JBoss with JBoss.AOP [JBoss.AOP, 2004], among others.

2.7 Problem Statement

The two underlying motivations of our work are *(i)* to enhance the applicability of reflection, and *(ii)* to understand, clarify and exploit the connection between reflection and aspect-oriented programming by trying to reconcile both approaches.

First of all, there has been a lot of promising work regarding the applicability of reflection. In particular the operational decomposition advocated by McAffer [McAffer, 1996] and the concept of fine-grained MOPs as proposed by Gowing *et al.* [Gowing & Cahill, 1996], both appear to be of great value for structuring and engineering the metalevel appropriately; and the ideas on partial reflection, discussed in Section 2.4.4, are attractive to make reflection efficient by ensuring that the price of reflection is only paid when really needed. In these two lines of work, however, each proposal has its limits, as we will discuss during this dissertation. Also, all reflective systems, although they make it possible to open applications by providing metaobject protocols, are themselves *closed* systems. This makes them unsuitable in certain application contexts, and prevents from further experimenting with new approaches to, for instance, metalevel engineering. Therefore, Part I of this dissertation presents our work on *opening* reflective systems, and progressively, through the iterative process inherent to open implementations, refine the ideas of partial reflection, fine-grained MOPs and intentional definition of reflective needs.

Part I finishes by suggesting that an appropriate model of partial reflection can support aspect-oriented programming in an open and flexible manner. This suggestion is further explored in Part II. We report on our experiments to use reflection as a realistic implementation basis for AOP. After experimenting with general-purpose AOP, which can be seen as a proper interface to reflection, we explore the use of reflection as a substrate for hybrid aspect-oriented programming. By hybrid we mean the possibility to use different aspect languages and approaches on top of a common substrate. The motivation is to combine the power and generality of reflection with the

appropriate barriers and syntactic support provided by (domain-specific) aspect languages. We then study the extent to which our approach can make it possible to compose aspects defined in different languages.

As a result, this dissertation shows that by opening reflective systems, enhancing the underlying model of partiality and fine granularity, and providing open aspect language support with compositional means, the gap between metaobject protocols and aspect-oriented programming can be filled, in a way that is fruitful to both.

Part I

From Metaobject Protocols...

Table of Contents

Chapter 3

Opening Up Reflective Extensions

This chapter presents the first step of our work on opening reflective systems. It is based on the first version of our *open reflective extension of Java*, Reflex [Tanter *et al.*, 2001]. Beside providing application developers with a reflective extension of Java, Reflex allows *meta-architects* to extend or tune the reflective infrastructure in order to best fit the needs of their target applications. We qualify reflective extensions supporting this tuning as being *open*. After presenting our approach, we include illustrative examples dealing with mobile code.

3.1 Introduction

Our initial objective was to apply a reflective extension of Java to enhance mobile agent systems with respect to the way the resources attached to a mobile agent are handled upon migration (see [Tanter, 2000]). Using a reflective extension in such an application domain implied several strong requirements such as portability (an agent playground is in most cases heterogeneous).

Indeed, as discussed in the previous chapter, a number of systems (*e.g.* MetaXa, Guaraná, Dalang, Kava) have addressed the limitation of Java with respect to reflection by providing reflective extensions with richer MetaObject Protocols (MOPs). However, all these Java extensions provide a particular, fixed, MOP that reflects the commitment of the designer to particular trade-offs between efficiency, portability, expressiveness and flexibility. Unfortunately, these trade-offs are not satisfactory for all applications, since different applications may have different needs. In particular, when looking around for an appropriate reflective extension applicable to mobile agent systems, we could not find one that would fit our needs.

We therefore started implementing our own simple extension based on the load-time MOP Javassist [Chiba, 2000]. In this extension [Tanter, 2000], a reflective object was attached to a unique metaobject, which understood a single MOP (MetaObject Protocol) method for trapping method invocations. That is, the metaobject was activated on each invocation of a public method of its reflective object through *hooks* introduced via code transformation. These hooks were looking as follows:

$metaobj$`.trapMethodcall(`$args$`);`

Later on, we discovered that, in some cases, it was necessary to give control to metaobjects when their base object was being serialized. However this feature was not offered by our simple reflective extension. The code transformation process was extended in order to add to each reflective class a method of the Java serialization API, `writeReplace`, automatically invoked when serialization occurs, which was made to invoke another method on the metaobject, in the following way:

$metaobj$`.trapSerialize(`$args$`);`

This means that the MOP was extended with a new method. Unfortunately, the previously developed metaobjects were not compatible with the new MOP, since they did not implement the new method. It was all the more problematic as this extended MOP was only required for some particular objects and metaobjects.

In fact, the issue we encountered there is a recurrent one. On the one hand, there are *high-level* reflective extensions providing hardwired choices about MOP definition and hook introduction, as well as about some important trade-offs such as performance vs. portability. What happens then if these choices are not compatible with the application requirements? On the other hand, there are *low-level* bytecode manipulation APIs allowing the definition of a custom-built reflective extension, at some non-negligible development cost. There is no middle ground, no reflective extension that would both limit the number of hardwired choices and allow seamless customization and extension in order to suit the requirements of a particular application or class of applications.

This chapter suggests that providing such a reflective extension, which we arguably qualify as *open*, is a worthwhile task. It presents Reflex, a prototype *open reflective extension of Java*. Reflex is a working reflective extension implemented by composing basic building blocks organized according to a predefined framework. Reflex comprises the definition of the framework, default generic components and some specialized components. This introduces, besides the classical roles of metaobject programmer and end-user, a new role in the development of a reflective application: the *architect of the metalevel*, who is responsible for defining, based on the framework as well as a number of existing building blocks, a fully-defined reflective extension.

The main ideas on which this version of Reflex is based are:

- the definition of a generic MOP;
- the reification of the code transformation process as an extensible entity.

The idea of a *generic MOP* replaces the idea of a global, all-encompassing, MOP since needs in this regard are unpredictable. It is of course possible to offer an extensive MOP, but it will never cover all possible needs. In general, a MOP method is devoted to handling a particular operation of the base language, like method invocation, serialization, creation... Even if these operations could be completely identified, the way they should be dealt with cannot be predicted. For instance, in the case of method invocation, there is a possibly infinite set of ways to deal with it: one could want to handle accessor methods in a particular way (therefore requiring a method like `trapAccessor`), or to handle methods distinctly, depending on some *method categories* (hence requiring methods like `trapMethodCategoryA`,...). This is the issue of *actual MOP*

definition as discussed in Section 2.4.4.4. Our first solution to this issue is to base Reflex on a generic and minimal MOP consisting of a single method, called `perform` due to its similarity with the *perform* method of Smalltalk. This method takes as its first argument a string describing the operation. Metaobject invocations, and therefore hooks, look as follows:

$metaobj$`.perform(`$operation$`, `$args$`);`

To put the generic MOP into practice, these hooks must be inserted where needed. The corresponding code transformation is reified as an extensible entity, which we call a *class builder*, in order to set up the appropriate hooks within a given class, using *subclassing* when the class cannot be modified.

Section 3.2 comes back on some requirements that should be met by an open reflective extension. Section 3.3 describes how this first version of Reflex supports the role of the metalevel architect, via a description of its main elements: the generic MOP and class builders. Section 3.4 presents a simple walk through the development process, from metalevel design to application programming, while Section 3.5 reports on a more consequent application of Reflex to mobile code. Finally, we discuss limitations of this work.

3.2 Requirements

This section reviews the basic requirements in terms of portability, expressiveness, and efficiency that led to the design of this first version of Reflex. Apart from the fact that portability is not compromised over, great care is taken not to discard any option too early.

3.2.1 Portability

A major benefit of Java is its *portability*. In our opinion, this major benefit should not be lost when considering a reflective extension. It would be somehow contradictory to provide an extension which would include portability restrictions! We have previously mentioned that the initial target application of Reflex, mobile agents, requires portability. We expect many applications of a reflective extension of Java to share such a requirement.

This discards the idea of relying on a specific virtual machine or just-in-time compiler, as considered by systems such as Guaraná [Oliva & Buzato, 1999] or MetaXa [Golm & Kleinöder, 1999], as well as extending the VM with native, and therefore platform-dependent, code through the Java Native Interface (JNI), as in Iguana/J [Redmond & Cahill, 2000; Redmond & Cahill, 2002]. This also means that the hooks intercepting base-level operation occurrences should be introduced through code transformation (either bytecode or source code transformation). Moreover, in order to be 100% Java compliant this transformation should be restricted to application code. Java core classes should be kept untouched.

3.2.2 Expressiveness of the MOP

Let us first consider how the link between the base and the metalevel is handled. For the sake of generality, we shall assume that the metalink is instance-based (rather than type-based), has cardinality n-n (that is, a metaobject can be associated to several objects and, conversely, an object to several metaobjects), and is dynamic, making it possible to dynamically adapt the behavior of a base object.

A second important issue is the definition of operations. A quick review of the literature on the applications of reflection (see, for instance, [Briot *et al.*, 1998; Stroud & Wu, 1996; Okamura & Ishikawa, 1994]) shows that a simple MOP providing only control of method invocation covers the needs of a large range of applications. However, application developers may need metaobjects that control other operations than method invocation (*e.g.* object creation), or may need to handle these operations in an adapted manner (*e.g.* for performance enhancements, or to introduce method categories). We have previously mentioned our need to control object serialization (illustrated in section 3.5).

This means that the architect of the metalevel should be able to define new operations together with the corresponding hooks. This includes the definition of the hooks, the definition of the code transformation responsible for their introduction and the time when this code transformation takes place. Note that, in a given application, the set of operations of interest may vary, at least from one type of object to the other, if not from one object to the other. In the latter case, the introduction of different hooks should nevertheless keep the objects *type-compatible*. Actually, the operation occurrences of interest could even vary along the lifetime of an object. We shall assume here that all the potential interesting operation occurrences are known when the hooks are introduced. However, we do not assume any specific time (compile time, load time or object creation time) for hook introduction. This means that a standard, *non-reflective*, object (an object which is not linked to the metalevel) may be turned into a *reflective* one.

Finally, expressiveness also covers metaobject composition, *i.e.* assembling the metaobjects attached to a given base-level object. Let us note that this goes beyond the provision for a metalink of cardinality n-n as soon as the different metaobjects associated to a given base object interact. A *composition policy* describes the strategy according to which metaobjects are assembled. As there is no universal composition policy, an open reflective extension should make it possible to support different composition policies and allow the definition of new policies. For the sake of reusability, composing metaobjects designed to handle different operations should be possible.

3.2.3 Efficiency

When using reflection via hook introduction, there are two kinds of performance overhead to consider. One kind of overhead is due to the introduction of reflection, more precisely the introduction of hooks in base-level code. There is another kind of overhead which is due to the execution of the hooks. Paying these costs for any class and object would be an overkill. This is why partial reflection suggests that these costs should only be paid when needed. In this work we consider partiality at the level of objects and classes (we shall go further in partiality as this dissertation progresses). This has a number of implications:

- Reflective and non-reflective objects may live together within an application. The execution of the standard objects should not be affected by the presence of the reflective objects. Note that, with respect to the previous discussion on portability, this requirement is difficult to guarantee when modifying the standard Java compilation and runtime support.
- It should be possible to apply reflection on an instance basis. Care must be taken to keep the reflective and non-reflective instances of a given base class type-compatible.
- Even if the interesting hooks are known at object creation time, it makes sense to delay their introduction so that a non-reflective object which is bound to become a reflective one is not slowed down until really needed. Conversely, if the hooks are not needed any longer, it would be nice to be able to transform the reflective object back into a non-reflective one. The decision of delaying hook introduction may depend on the respective overheads of executing the hooks (with a useless indirection via a dummy metaobject immediately transferring the control back to the base object) and introducing the hooks after object creation time.
- It should be possible to choose to perform hook introduction at compile time (dealing with source code or bytecode) so that the cost of hook introduction is not incurred at run time (if it matters), or even to merge the base level and the metalevel as is done in the so-called compile-time MOPs [Tatsubori *et al.*, 2000], when both levels and their links are known at compile time. Much more aggressive optimization techniques could actually be envisioned, based on runtime code generation [Dynamo 2000, 2000] and partial evaluation [Jones *et al.*, 1993; Braux & Noyé, 2000], but being able to seamlessly combine well-known compile-time and runtime reflection techniques would already be quite a progress.

3.3 The Architect of the Metalevel

This section presents how Reflex supports the role of the metalevel architect. Reflex defines a framework for its components that can be specialized to meet particular needs. The already implemented components respect this framework and can either be extended or simply replaced with other components fitting into the framework. Such concrete components include hook introduction for the generic MOP and specialized MOPs (Sect. 3.4 and 3.5), and metaobjects implementing a composition scheme (Sect. 3.3.3 and 3.4).

The upper part of Fig. 3.1 shows the main elements of the Reflex framework. We shall come back to the lower part of the figure, showing an actual specialization, in Section 3.4. These elements are:

- `ReflexMetaobject`: an interface defining the basic services provided by a metaobject;
- `ReflexClassBuilder`: an interface defining the basic protocol for hook introduction;
- `Reflex`: a class that defines the explicit creation protocol, to obtain reflective instances of a class;

- `ReflexObject`: an interface that defines the protocol to access the metaobjects linked to a given object. Class builders are responsible for making *reflective classes*, *i.e.* classes defining reflective objects, implement this interface.

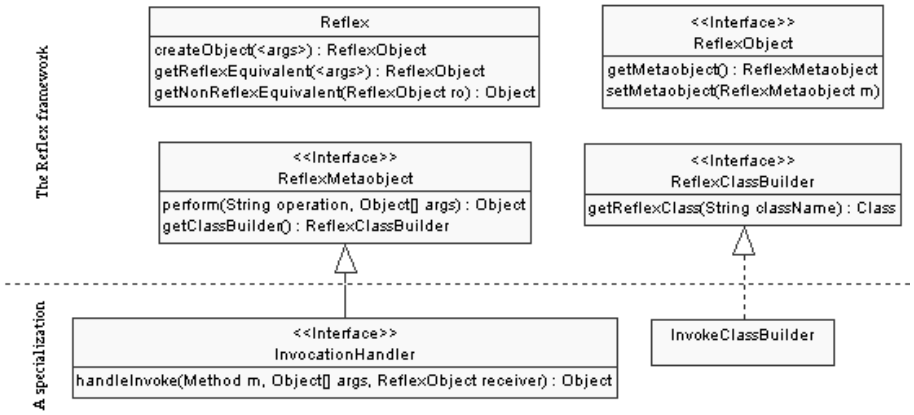

Figure 3.1 – The Reflex framework and one specialization.

The main responsibility of the metalevel architect consists in defining a MOP: this task is supported by the *generic MOP* and the reification of the class transformation process as *class builders*, presented respectively in Sect. 3.3.1 and 3.3.2. Section 3.3.3 discusses the task of defining a metaobject composition framework.

3.3.1 The Generic MOP

Fig. 3.1 shows that, by default, a metaobject implements the interface `ReflexMetaobject`, which defines two methods. The Reflex package includes a default implementation of such a metaobject. The method `getClassBuilder` returns the class builder implementing hook introduction (see below) as required by this metaobject. A hook is an invocation of `perform` with two arguments: a description of the type of the current operation occurrence, and an array of additional parameters depending on the operation.

As an example, let us consider hooks for method invocations. In a more "traditional" MOP, trapping a method invocation would be done using the following hook:

```
metaobj.handleInvoke(method, args, receiver);
```

While, with the generic MOP, trapping the same method invocation can be done using the following generic hook:

```
metaobj.perform("invocation", invokeArgs);
```

This hook could result, for instance, in the invocation of a method `handleInvoke` of the metaobject. The argument $invokeArgs$ describes the intercepted base-level operation occurrence: method, arguments, and receiver.

Such a MOP makes it apparent that a metaobject *interprets* base-level operation occurrences. It is very flexible in that new operations can be introduced without redefining the MOP (including generic class builders), which also means that metaobjects defined in such a context are quite easy to reuse even in the presence of new operations. We shall see in section 3.3.3 that it may also help when implementing a composition policy.

There are however some drawbacks associated to this flexibility. First, the use of a generic hook has some cost (an additional indirection with a dispatch on the operation type and some argument packing and unpacking), which may turn out to be prohibitive. Second, the choice of a uniform metaobject type may be counterproductive in a context where many *intended* types of metaobjects are manipulated, with the risk of preventing early type mismatch detection.

This could indeed be a serious problem if Reflex were a closed reflective extension, which it is not. The architect of the metalevel may choose to extend the MOP, as will be illustrated in Sections 3.4 and 3.5.

3.3.2 Class Builders

A crucial element of our approach to opening reflective extensions lies in the reification of the class transformation process as *class builders*. Class builders are responsible for hook introduction. A class of class builders implements the `ReflexClassBuilder` interface (Fig. 3.1), *i.e.* a class builder takes as input a standard class and returns a *reflective* class. This reflective class is then instantiated in order to get reflective objects that are type-compatible with instances of the initial class.

Therefore, a class builder corresponds to the set of operations (and more precisely, occurrences of such operations) of interest in the input class. It operationally defines these operations as a program transformation introducing the proper hooks at the proper places in the code. Depending on the class builder, this code may be either source code or bytecode and the transformation may happen in place (the input class is destructively turned into a reflective class) or not. In the latter case, *subclassing* should be used in order to get type compatibility between instances of the input class and the output class. Different class builders corresponding to different operations can then produce different compatible reflective classes that can coexist in a running system. Class builders can therefore be built according to the precise requirements (performance, dynamic adaptability, operations of interest...) of the target applications. They play an essential role with respect to the flexibility of the framework.

A class builder may first generate an implicit subclass of the original class if needed (using a `SubclassCreator` object). It then performs a sequence of transformational operations on this class. Such transformations are reified as entities, defined in classes such as `MetalinkInserter` to insert the metalink in the class, `MethodcallHookInserter` to wrap methods in order to transfer control to the metalevel, `MethodCopier` to copy compiled methods, etc. These transformation components are implemented with the Javassist API [Chiba, 2000] and therefore operate

on bytecode.

For instance, the generic class builder available in the Reflex package allows metaobjects to trap, through the generic MOP, invocations of public methods. This builder first creates a subclass, overrides all inherited public methods, and wraps them using the generic hook (`perform`) to give control to the metalevel. It then inserts the metalink (reference to a metaobject). Then the generated subclass can be stored on disk or not, before being returned for instantiation. Defining a new class builder is a matter of extending this predefined process (*e.g.* by subclassing) or defining a new one (for instance to avoid the generation of the subclass).

Any reflective class, created via a class builder, implements the `ReflexObject` interface, which defines the `getMetaobject` and `setMetaobject` methods for respectively retrieving and setting the metaobject attached to a base reflective object. This means that, in this still-immature version of Reflex, the metalink does not present a n-n cardinality: a base object is linked to a single metaobject (although several metaobjects can cooperate using a composition scheme).

Note that there is another classical way of introducing hooks: using object wrappers. This is the technique used in Dalang [Welch & Stroud, 1999] and the dynamic proxy classes offered by JDK1.3 [SUN, 1999a]. As discussed in the previous chapter, such a scheme suffers from a number of well-identified problems [Welch & Stroud, 1999], in particular from the identity problem and the lack of type compatibility between a standard, non-reflective, object and its wrapper, making it reflective. These problems can be circumvented either by merging the object and its wrapper, as in the successor of Dalang, Kava [Welch & Stroud, 1999], or by making the wrapper inherit from the wrapped object, as in the MOP used by ProActive [Caromel *et al.*, 1998]. In the former case, the issue is then to combine reflective and non-reflective instances of the same class. [Welch & Stroud, 1999] mentions the introduction of hooks on the sender's side, without more details. In the latter case, there is the cost of an additional object.

The idea of relying on inheritance in order to control invocations is not new [Ducasse, 1999; Foote & Johnson, 1989]. As for Java reflective extensions, the idea has already been used by Reflective Java [Wu, 1998]. However, Reflective Java does not provide any creation protocol (see below), which requires, when programming at the base level, to know the reflective classes attached to a base class and to manipulate these classes explicitly.

3.3.3 Composition framework

Reflex does not enforce the use of a particular composition scheme, nor does it enforce the use of a composition scheme at all. Still, Reflex includes a generic composition framework.

This framework simply makes explicit the distinction that we see between three types of metaobjects, namely, *composers*, *extensions*, and *interpreters*:

- A composer acts as a *facade* [Gamma *et al.*, 1994] of the set of composed metaobjects. It defines the composition policy and is in charge of managing the construction and evolution of the composition set.
- An interpreter defines a complete meaningful *interpretation* of a base-level operation (*e.g.* method invocation). A remote call metaobject is an example of an interpreter.

- An extension simply *extends* the interpretation of such operations with some extra behavior. A trace metaobject is an example of an extension.

The generic composition framework therefore consists of three empty interfaces, `Composer`, `Interpreter`, and `Extension`, deriving from the root interface `ReflexMetaobject`. Section 3.4 mentions a concrete composition scheme deriving from this framework.

3.4 Illustration: Simple Walk Through

This section walks through the different stages of the development process, showing how Reflex is used successively by the metalevel architect, the metaprogrammer, and finally the application programmer.

3.4.1 Perspective of the metalevel architect

The task of the metalevel architect consists in first defining a MOP, and then, possibly, defining a composition scheme.

3.4.1.1 Defining the MOP

The Reflex framework can be specialized in order to fit application requirements. The architect specializing the framework can introduce a new MOP, with a default implementation, define which hooks will be introduced and how. The lower part of Fig. 3.1 shows a specialization of the Reflex framework that we developed for applications for which trapping method invocations is a major feature. This specialization consists of definitions of:

- an extended MOP (interface `InvocationHandler`) that defines a method for handling method invocations, `handleInvoke`,
- a specific class builder (class `InvokeClassBuilder`) that introduces hooks for trapping invocations of public methods, through the new MOP method `handleInvoke`.

The hypothesis here is that performance is an issue with metaobjects performing, on average, very little work, which means that the cost of jumping to the metalevel should be minimized.

3.4.1.2 Defining a composition scheme

The architect of the metalevel can then design and implement a composition scheme. We already extended Reflex with such a scheme, inspired by Mulet et al. [Mulet *et al.*, 1995].

In this scheme (see Fig. 3.2) each *extension* metaobject performs its metaprocessing and then gives control to the next metaobject in the cooperation chain. The chain composer ensures that there is always one and only one interpreter placed at the end of the chain. Therefore, extensions

in this scheme are directly linked to another metaobject (which can be another extension or an interpreter) and explicitly cooperate with it.

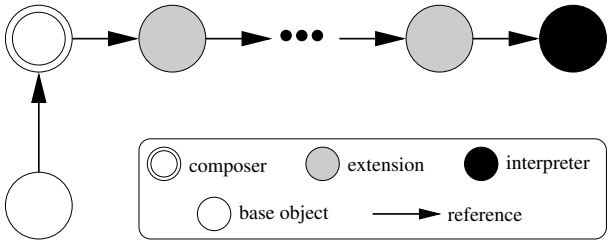

Figure 3.2 – Composition chain.

3.4.2 Perspective of the metaprogrammer

Now that the metalevel architecture has been set up, the metaobject programmer can start implementing metaobjects. These metaobjects conform to the MOP defined by the architect of the metalevel, and possibly make use of a composition framework. We illustrate here the development of a configurable trace metaobject for the MOP defined by the `InvocationHandler` interface, using the composition framework presented above.

The `Trace` class is declared as implementing the base interface for extensions in the chain composition scheme. The role of such a metaobject is to trace, on a given output, the method invocations that occur on a set of base objects. This trace is selective: it applies to some methods only. To this end, the trace metaobject aggregates a hash set containing the names of the methods to trace. This hash set and the target print stream on which the trace is performed can be given at instantiation time or later, and can be updated dynamically. The simplified (without exception handling) implementation of the `handleInvoke` method is as follows (`out` is the target print stream object, and `toTrace` is the hash set containing the names of the methods to trace):

```
public Object handleInvoke(Method m, Object[] args,
                           ReflexObject receiver){

    if(toTrace.contains(m.getName()))                          (1)
      out.println("call: " + m.getName() + " with: " + args ); (2)

    return this.getComposed().handleInvoke(m, receiver, args); (3)
}
```

First, the trace metaobject checks if the hash set contains the name of the invoked method (1). If it does, a trace is produced, giving the name of the method and its arguments (2). The invocation is then forwarded to the next metaobject in the chain (3).

3.4.3 Perspective of the application programmer

Once the library of metaobjects is ready, the application programmer can introduce metaobjects in the application. The role of the application programmer is basically that of identifying which objects should be reflective, and which metaobjects should be attached to each of these objects (even if this is actually hidden behind code transformation tools and wizards). Let us illustrate the creation of a reflective vector to which a trace metaobject is attached.

The first step is to create and set up the metaobject to be attached to the base object:

```
ChainComposer composer = new ChainComposer();  (1)
Trace trace = new Trace();                     (2)
composer.addExtension(trace);                  (3)
```

First, the composer is instantiated (1). Since no interpreter is specified in the constructor, the composer automatically instantiates a default interpreter, with default semantics. Then, the trace metaobject is created (2). Finally, the trace metaobject, which is an extension metaobject, is inserted into the composition chain managed by `composer` (3).

Once this is done, the reflective object can be instantiated, using the services of the `Reflex` class:

```
Vector v = (Vector) Reflex.createObject("java.util.Vector",
                                        composer);
```

The `createObject` arguments indicate that a reflective instance of `java.util.Vector` should be created, with the metaobject `composer` attached to the instance. When this statement is executed, the `createObject` method queries the composer for the class builder to use, and then delegates the task of retrieving the reflective `Vector` subclass to the class builder. At this point, if the reflective subclass has already been created, it is loaded if needed. Otherwise, it is created and loaded dynamically. The `createObject` method instantiates the reflective class, and attaches the metaobject to the created instance. Finally, it returns the reflective object. Now, `v` is a reference of (declared) type `Vector` that points to a reflective object, instance of a reflective subclass of the `Vector` class.

3.5 Illustration: Reference Management in Mobile Code

In this section, we present an application of Reflex to the issue of reference management in mobile code. Apart from showing the interest of using reflection to handle this concern of mobile code, this section illustrates a concrete case of MOP extension. The extension deals with controlling serialization, which is necessary in order to support some reference management policy. The content of this section is adapted from [Tanter & Piquer, 2001].

In mobile object systems [Cardelli, 1997], there are theoretically different strategies to manage the references attached to a migrating object (by copy, by move, remote reference, and rebinding). However, as clearly pointed out by Fuggetta et al. in [Fuggetta *et al.*, 1998], the strategy is generally determined by the language definition or implementation, not by the application programmer

(or with heavy restrictions, like in Java RMI [SUN, 1998b], discussed hereafter). The aim of this work is to use Reflex to give application programmers the means to easily specify which reference management policy should be used, on a per-instance basis. Furthermore, since the semantics of a given strategy is inherently generic, strategies should be implemented as basic blocks that can be reused and programmatically attached to any object, without any static constraint upon its type.

After a quick presentation of the issue of reference management in mobile code, we briefly expose our approach to a flexible provision of the rebinding and remote reference strategies. This shows how Reflex can be used in such a context, and provides an example of a MOP extension, in order to support serialization.

3.5.1 Reference Management

When an object is transmitted from one site to another, it is considered as a *mobile object*, that is *migrating*. Generally, such an object has a set of references to other objects, forming a graph of objects. When a mobile object migrates, the set of bindings to passive objects (also referred to as *resources*) has to be rearranged. This is what we call *reference management*, also referred to as *data space management* in [Fuggetta *et al.*, 1998]. The way this set is rearranged depends on the nature of the resources (if they can – or should – be migrated over the network or not), the type of the binding to such resources (by identifier, by value, or by type), as well as requirements set by the application. The very fact that it eventually depends on application specific requirements makes it impossible to fully automate the choice of the adequate strategy, entailing the need for its programmatic specification.

A major application of reference management is in the *mobile agent paradigm* [Lange, 1998], a particular distributed computing paradigm making great use of code mobility. In this paradigm, the mobile object considered is a mobile agent. A mobile agent is basically an active mobile object, which has some features such as proactivity. Resources are passive objects shared by mobile agents. Sites often called *places* constitute the computational environments that host mobile agents. Agents may move from one place to another, communicate, and access possibly shared resources. This work deals with reference management in general, but is presented in the context of mobile agents, thereby making use of the associated terminology.

3.5.1.1 Reference management strategies

Let us consider a migrating agent M_A that holds a binding B to a resource R.

- in a ***by move*** strategy, the resource R is transferred along with the agent M_A to the destination place and the binding B is not modified. There are two alternatives concerning the other existing bindings to R that other agents may hold: they can either be removed or be converted to a remote reference. This strategy is applicable only if transferring R over the network is possible (technically or semantically).

- in a ***remote reference*** strategy, the resource R is not transferred and once M_A has reached the destination place, B is modified to remotely reference R in the source place. Then,

each time M_A accesses R through B, some communication over the network occurs. This strategy is applicable in every case, under the condition that the price to pay for remote communication is acceptable, for instance if the resource is a huge data store or contains sensitive data, or if it is a resource that cannot be migrated (*e.g.* a printer handle).

- in a ***by copy*** strategy, a copy R' of R is created, B is modified to refer to R', and then R' is transferred to the destination place along with M_A. This strategy can be applied if the binding is not by identifier (*i.e.* it is no issue if M_A references a copy of R instead of R itself) and if R (and consequently R') can be transferred over the network.

- in a ***rebinding*** strategy, B is turned to void and re-established after migration of M_A to another resource R' on the destination place, where R' has the same type as R. Rebinding is suited to by-type bindings, exploiting the fact that the only requirement posed by the binding is the type of the resource. It avoids resource transfer across the network as well as creation of inter-places bindings (remote references). This mechanism requires that a resource of the same type of R exists at the destination place. Typical examples are a local display, a local naming service, or a local log file.

3.5.1.2 Standard reference management in Java

In Java, reference management is done with Java *Remote Method Invocation* (RMI) [SUN, 1998b]. RMI allows two different policies: by copy and remote reference. By default, any object is passed by copy: when transmitted, a copy of the object and all the objects accessible from this object is generated as a sequence of bytes (this is *serialization*). Then the sequence of bytes is transmitted to the destination site where the graph of objects is rebuilt (through a process called *deserialization*). RMI allows programmers to define classes of *remote objects*. When transmitted, such an object is not serialized like a normal object: instead of the object itself, a stub to the object is serialized. Then, on the destination site, invocations of methods on the stub entail network communication with the remote object. This way the remote reference policy is achieved.

The problem with RMI is that it forces *strong static constraints* on the remote objects. A remote object must be instance of a class that (1) *implements* the `Remote` interface and that (2) *extends* a base class of remote objects such as `UnicastRemoteObject`. Therefore it is impossible to specify the strategy (by copy or remote reference) dynamically on a per instance basis. Furthermore, the policy has to be statically defined. We use Reflex to overcome this limitation and bring more strategies to the programmer, still using RMI as a backbone for implementing our solutions.

3.5.2 The Rebinding Strategy

To achieve the rebinding policy, we have to assume that hosts on the network provide a way of getting a reference to a local resource based on some identifier. This makes it possible for an agent to rebind this resource each time it arrives to a host, and unbind it each time it leaves. There are several ways to implement such a system for local resources management. Let us consider a simple resource management system that allows any program to publish an object as a local

resource, associating a string identifier to it. One can then query the *resource manager* in order to get a reference to a previously published resource based on a string identifier. A resource manager is instance of the `ResourceManager` class, which defines basically the same services as a RMI registry (publish, republish, unpublish, get). There is a unique resource manager per JVM. It is obtained from the static method `getResourceManager` defined in the `ResourceManager` class.

The design principle of the rebinding strategy is illustrated in Figure 3.3. A "dumb" object acts as a proxy for a concrete local resource published by the resource manager. It is controlled by a `Rebinder` metaobject. The metaobject is created with the string identifier of the resource that it needs to retrieve from the resource manager when needed (Fig. 3.3a).

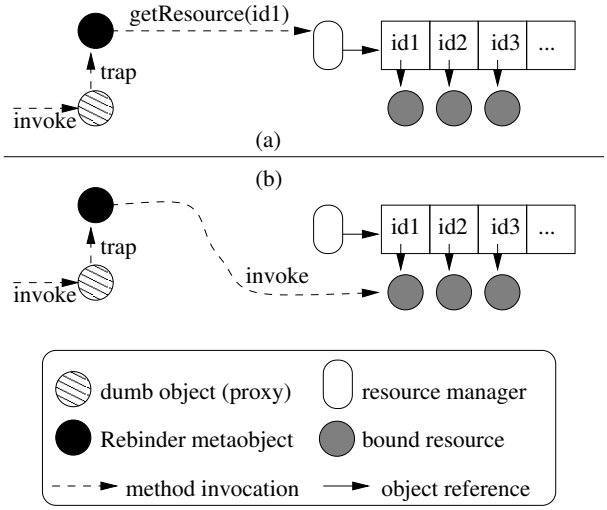

Figure 3.3 – The rebinding strategy.

When the binding to the resource is set up, the metaobject traps each method invocation on the dumb object and forwards them to the local resource (Fig. 3.3b). Upon migration, the binding is cleared and reestablished after migration when first needed (lazy initialization). The main point of using Reflex here is that the actions of binding and unbinding the resource obtained from the resource manager are handled transparently and automatically by a (reusable) metaobject.

3.5.3 The Remote Reference Strategy

We have also developed metaobjects for the remote reference strategy, which is the other interesting policy that is not flexibly provided by Java (see section 3.5.1.2 for a critical review of Java RMI). With Reflex and our metaobjects, the remote reference strategy can be applied to *any object, without any static constraint upon its type*, which is a notable enhancement of what RMI provides. This strategy requires a MOP extension, presented hereafter, so that metaobjects can control serialization. We finally give an overview of our solution.

3.5.3.1 A MOP extension

We now explain how the generic hook of Reflex can be used to control a new operation: object serialization.

We would like a more flexible system than RMI, allowing any object to be passed by reference, without any constraint on its class. Upon migration of the agent, passing the object by reference means passing a proxy to the object rather than the object itself. This can be done by making the object reflective and attaching to it a particular metaobject, informed when *serialization* occurs. Such a metaobject can then specify an alternate object for serialization (the proxy). Once passed, the proxy is controlled by a metaobject that performs all method invocations remotely through the network.

We therefore need to give the control of serialization to metaobjects. The Java serialization API [SUN, 1998a] offers a method, `writeReplace`, which can be used to specify an alternate object for serialization. Therefore, reflective objects should implement this method and a hook has to be inserted so that when this method is invoked automatically by the serialization process, the metalevel is informed and has the possibility of specifying an alternate object (such as a proxy).

We have implemented a new class of class builders, `SerializeClassBuilder`, which extends `InvokeClassBuilder`. The `SerializeClassBuilder` adds a `writeReplace` method to each generated class using a `MethodCopier` object. This method, defined below, plays the role of a generic hook:

```
private Object writeReplace()
    throws ObjectStreamException {

 (1)   ReflexMetaobject mo = this.getMetaobject();
 (2)   Object replace = mo.perform("serialize",
                                   null);
 (3)   if(replace == null)
 (4)       return this;
 (5)   return replace;
}
```

When serialization on the base object occurs, this method is invoked. It first retrieves the metaobject associated to the base object (1), and delegates to the metaobject the interpretation of the `"serialize"` operation occurrence (2). If the result is null (3), *i.e.* no metaobject understood the request, then the default policy applies (4). Otherwise, the alternate object is returned (5). In our framework, if a metaobject wants to be informed when serialization occurs, it simply has to implement the `handleSerialize` method. Metaobjects that do not understand `handleSerialize` can still be used, along with metaobjects newly developed for handling serialization.

3.5.3.2 Overview

Figure 3.4 illustrates the architecture of the remote reference strategy. In the first JVM, the object to be passed by reference is controlled by a `SendProxy` metaobject (Fig. 3.4a). This metaobject

makes use of the Reflex ability to provide control of object serialization. When the base object is about to be serialized (some mechanism lets it determine whether the occurring serialization corresponds to a migration or not), the `SendProxy` metaobject takes control and returns a proxy on the base object, which will be serialized instead of the base object.

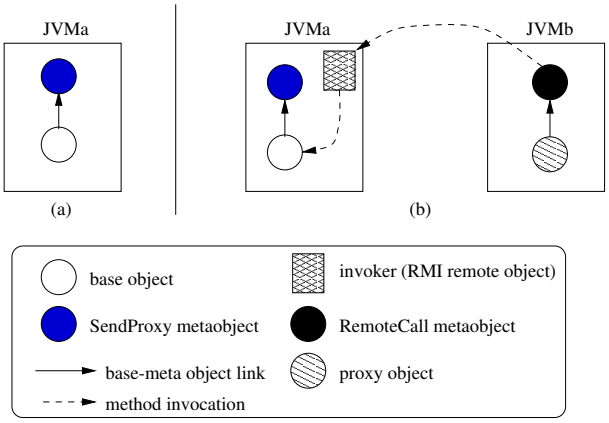

Figure 3.4 – The remote reference strategy.

Therefore, after the object that owned a reference to the base object has arrived at the destination JVM, it owns a reference to a type-compatible proxy object that is in turn controlled by a metaobject, of type `RemoteCall`. When a method invocation occurs on the proxy object, the `RemoteCall` metaobject forwards the invocation to the original base object through an *invoker* (a standard RMI remote object created by the `SendProxy` metaobject). Figure 3.4b illustrates the situation after migration. Thanks to this policy, we have been able to allow migrating objects to hold references to supposedly unserializable objects like file descriptors, and then seamlessly use these references with the remote access semantics.

3.5.3.3 Transparent adaptation of migration policies

This work on mobile code makes it possible to statically specify which migration policy should be used on a fine-grained basis. We have done some work in extending these results to achieve transparent adaptation of migration policies in the context of *ubiquitous computing* [Weiser, 1993].

In ubiquitous computing, software is used by roaming users that interact with the electronic world through a collection of devices of very disparate characteristics (PCs, laptops, PDAs, cellular phones, home devices). Thanks to its portability and support for code mobility, the Java programming language is seen as the best candidate for such a setting. As Katz pointed out, mobility requires *adaptability*, which means that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion [Katz, 1994]. There are many levels at which adaptation can be fruitfully applied [Blair *et al.*, 2000]. In this work we address the issue of adapting migration policies within an application.

Take the example of a Personal Information Management (PIM) application, such as an agenda. The user of this application typically owns a workstation at the office, a laptop for home use, and probably a PDA and a cellular phone. To avoid replication of code and data, we can design a centralized solution, in which the agenda application can be *migrated* from a main server to any of these devices.

When an application (or a collection of objects) migrates from one execution environment to another, there are a number of parameters that can be taken into account in order to optimize migration behavior. For instance, parts of the object graph could be left on the source environment and then be accessed remotely, or on the contrary, it could be desirable that migration take more time but then avoid remote accesses. Such a decision comes from reasoning about information on the infrastructure: this can be static data such as the types of the source and destination platforms, and dynamic data such as the state of the network in terms of available bandwidth, expected latency, and so on. At a finer-grained level, the decision can come from application-specific criteria, such as the expected size of a particular instance, constraints on the object states, etc.. To preserve transparency, reasoning about upcoming migrations is done at the metalevel.

This approach has been applied to a prototype agenda application, so that for instance, when migrating the application from the server to a PDA, metalevel reasoning could result in deciding to transfer only frequently accessed data, or the data for the current week only, instead of transferring everything [Tanter *et al.*, 2002b; Vernaillen, 2002].

3.6 Discussion

Concerning Reflex and its motivation, it has to be noted that the work on MetaJ [Douence & Südholt, 2001] shares with ours the objective of making it possible to tailor reflective extensions to the specific constraints of its target applications. However, MetaJ only deals with meta-circular interpreters. On the one hand, this makes it possible to adopt a semantics-based approach, and therefore to be very systematic and deal with formal correctness. On the other hand, the gap with practical considerations such as performance is far from being bridged. A first attempt was formulated in [Douence & Südholt, 2000], where interpretation was replaced by compilation as much as needed, but this work has not been, to our knowledge, taken to real applicability.

On the opposite, all the practical Java reflective extensions [Wu, 1998; Welch & Stroud, 1999; Golm & Kleinöder, 1999; Oliva & Buzato, 1999] provide a fixed MOP with some universal decisions made on the trade-offs between portability, expressiveness, and performance. The only point on which Reflex does not offer any freedom is portability, on the basis that this decision is actually set by the very definition of Java. In spite of its fixed MOP, the case of Iguana/J [Redmond & Cahill, 2000; Redmond & Cahill, 2002] is peculiar. Indeed, its MOP seems to cover all the elementary operations, for which metaobjects have already been designed. As we have seen in the previous chapter, Iguana/J follows the Iguana [Gowing & Cahill, 1996] approach and adopts the idea of combining these metaobjects through protocol declarations, protocols which can then be, again declaratively, associated to base classes. The implementation is based on the Java Native Interface, which has been abandoned, and is therefore not portable, although it benefits from an efficient capture of the basic operations (at the price of more implementation work).

As for composing metaobjects, with the exception of MetaXa [Golm & Kleinöder, 1999] and Guaraná[Oliva & Buzato, 1999], the other Java reflective extensions do not offer any help. In MetaXa, the composition scheme is fixed. The VM systematically organizes the metaobjects in a chain of metaobjects, following the order of introduction of these metaobjects. Guaraná is more open in that it offers metaobjects similar to the composers introduced in the specialization of Reflex as well as an extensible communication protocol similar to the propagation of events realized through the use of `perform`.

As far as the application of Reflex to mobile code is concerned, it can be said that although the idea of applying reflection and metaprogramming to the domain of distributed systems and mobile code is not new, the literature on the subject is rather scarce. In [Okamura & Ishikawa, 1994], the authors present an application of metalevel programming to control object location in a distributed system. Objects can be migrated based on metalevel decisions. CodA [McAffer, 1995a; McAffer, 1995b] provides distributed systems with a computational model supporting migration, remote references and replication. The main problem with CodA is that it is an entire execution environment where all objects are controlled by the metalevel. The ProActive library [Caromel *et al.*, 1998] aims to simplify concurrent and distributed programming thanks to metaprogramming. However, the reflective model is based on a wrapper approach inspired by the classical stub/proxy architecture, suffers from annoying limitations (such as constructor chaining), and is not very suitable when used locally. Ledoux and Bouraqadi analyze in [Ledoux & Bouraqadi-Saâdani, 2000] the possibilities of adaptability that reflection could bring to mobile agent systems. They mention adaptability of resource management mechanisms, similarly to what we presented in this chapter, and of migration mechanisms (strong migration vs. weak migration). However, only concepts are formulated in this position paper.

Summary

In this chapter, we have exposed our first steps towards achieving open, applicable and configurable reflective systems. We have introduced the role of the *metalevel architect*, responsible for specifying the structure of the metalevel, depending on the requirements of the target application domain. In standard, closed, reflective extensions, this role is simply played by the designer of the extension. We have also presented how the first version of Reflex provides support for the metalevel architect: through a *generic MOP* that can be specialized and extended, and through *class builders*, as extensible entities reifying the class transformation process. The following chapter presents a refinement of these ideas, based on a new model of behavioral reflection and a better overall architecture.

Chapter 4

A Model of Partial Behavioral Reflection

In this chapter, we extend the ideas of the previous chapter, based on the experience gained when applying the first version of Reflex in different contexts. The customization interface that makes Reflex an open system is consequently refined in order to make it more usable. Furthermore, we present a comprehensive approach to *partial behavioral reflection*, an approach to more efficient and flexible behavioral reflection. We expose its *spatial* and *temporal* dimensions, and propose a model of partial behavioral reflection. In the context of Java, we describe a reflective architecture offering appropriate interfaces for *static* and *dynamic* configuration of partial behavioral reflection at various levels. We present a new version of Reflex, our open reflective extension for Java, implementing this architecture. Reflex fully supports partial behavioral reflection in a portable manner, and seamlessly integrates load-time and runtime behavioral reflection. This chapter also shows micro-benchmarks and examples supporting our approach. The three illustrations, that respectively deal with the observer pattern, asynchronous communication via transparent futures, and runtime monitoring, also highlight the interest of partial behavioral reflection as a tool for open dynamic Aspect-Oriented Programming. Most of the content of this chapter is an updated version of [Tanter *et al.*, 2003].

4.1 Introduction

In the previous chapter, we have introduced a new role in the whole process of developing a reflective application, the *metalevel architect*. This architect, using the framework provided by Reflex, as well as a number of existing building blocks, is responsible for specifying the structure of the metalevel, depending on the requirements of the target application domain. Then, the process proceeds as usual: the *metaprogrammer* implements the building blocks of the metalevel as *meta-object* classes obeying the structure defined by the architect, the *base programmer* develops the base application, and the *assembler* links both levels by implementing the causal connection.

4.1.1 Limitations of the First Version

Our first piece of work suffers from several limitations and drawbacks. First of all, the idea of a generic MOP, though appealing, is open to criticism: it seems too permissive and therefore hard to correctly use; it is inefficient and will, most of the time, require a MOP specialization. Therefore, existing class builders are very likely to be extended and new ones created.

Unfortunately, the model of class builders as presented in the previous chapter is too *coarse-grained* to be really appropriate for user extensibility. Indeed, it does not scale well when considering multiple requirements simultaneously. For instance, we have, as of now, only considered specialization of MOPs in terms of the metaobject method to invoke: other issues such as the precise pieces of information that are passed to metaobjects, or the precise selection of base-level operation occurrences subject to interception ought to be taken into account as well.

Assigning the resolution of these many issues to a unique entity, the class builder, results in a big, unmanageable and hardly extensible entity, thereby compromising the interest of the approach versus the alternative of building ad hoc extensions. This problem is exacerbated when considering the support of several operations in the base application, since the basic means for class builder composition in this model is the single inheritance offered by Java. Furthermore, the dependence of metaobjects on class builders is too rigid.

As far as metaobject composition is concerned, our initial scheme does not really survive the introduction of several operations either. Indeed, the notion of interpreter is operation-specific and hence separate metaobject chains should be available for the different operations. This means that we need to truly support a n-n metalink cardinality. Supporting a separate link to metaobjects per operation would already be a progress, but would not be enough: it can very well make sense to have separate (chains of) metaobjects controlling different operation occurrences (although of the same operation) in a given object. All in all, we need a more flexible notion of what a metalink can be.

Finally, in the previous version, Reflex only offered an explicit MOP for specifying reflective needs, although the model did not prevent more implicit, declarative specification.

4.1.2 Contributions

This chapter builds on the two fundamental ideas introduced in the previous chapter: supporting the role of the metalevel architect and reifying the transformation process as user-specified entities. However it goes much further, first of all by making MOP specializations and extensions more lightweight: it uses a single generic class builder with pluggable hook installers, which are smaller entities. Also, the core of the Reflex framework is more clearly separated from user-defined entities and MOPs. As regards configuration, this second version of Reflex offers extensive static and dynamic APIs, with the possibility of making classes reflective directly through static configuration, hence making it possible to leave the base application code intact.

More fundamentally, this chapter addresses issues in flexibility and selectivity of traditional behavioral reflection, by proposing a comprehensive approach to *partial behavioral reflection*. The chapter includes the following contributions:

- We present a comprehensive approach to partial behavioral reflection, highlighting its spatial and temporal dimensions. Our model of partial behavioral reflection generalizes the classical view of metalinks by grouping execution points into composable sets (*hooksets*), possibly crosscutting the object decomposition, and attaching some metabehavior to these sets through a highly configurable *link*. The related ideas and techniques are applicable to a wide range of object-oriented languages.

- We describe, in the context of Java, an open architecture supporting this approach and making it possible to combine static and dynamic configuration of reflection.

- The architecture has been retrofitted into Reflex. Configuration work is shared between the metalevel architect and the assembler. In particular, we introduce an expressive reification selection framework that allows intentional, possibly declarative, description of the MOP entry points. Spatial and temporal selection of reflection can be done by combining static and dynamic configuration. Micro-benchmarks show the benefit of the approach.

  Finally, we illustrate, through examples, how these ideas facilitate the definition of crosscutting metaobjects, clarifying some links between (partial) reflection and Aspect-Oriented Programming.

The rest of this chapter is structured as follows: in Section 4.2, we introduce partial behavioral reflection in a general setting. Section 4.3 presents the second version of Reflex and micro-benchmarks. Finally, Sections 4.4, 4.5 and 4.6 illustrate how Reflex can be used to achieve separation of concerns on concrete problems.

4.2 Partial Behavioral Reflection

Partial behavioral reflection is an approach to more efficient and applicable behavioral reflection that relies on avoiding useless reifications. To this end, high flexibility in specifying reflective needs is required.

4.2.1 Spatial and temporal selection

Partial behavioral reflection addresses the issue of flexibility vs. efficiency by limiting to the greatest extent possible the number of control flow shifts occurring at runtime. Indeed, shifting to the metalevel is powerful but costly. Recall that such a shift consists of first reifying the operation occurrence, and then delegating (at least part of) its interpretation to the metaobject. A hook is the base level piece of code responsible for performing a reification and giving control to the associated metaobject. Reification is a significant cause for performance degradation. For instance, in the case of a method invocation, reification usually implies wrapping all the arguments of the invocation into an array of objects and retrieving a reference to a method object. This data may further be encapsulated into a unique method call object. From an efficiency viewpoint, it is therefore crucial to limit the number of shifts and pay the price of reification only when it is effectively needed. Another issue lies in the precise selection of what piece of information should be reified.

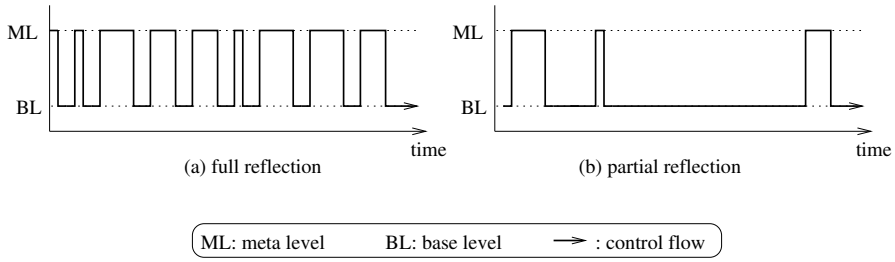

Figure 4.1 – "Reflectogram" of a reflective application: illustration of the evolution of the control flow in a reflective application.

Fig. 4.1 represents the *"reflectogram"* of a reflective application. A reflectogram illustrates the control flow between the base level and the metalevel during execution. Using full reflection (Fig. 4.1a), any operation at the base level is reified and therefore many –possibly useless– shifts occur. This does not occur with partial reflection (Fig. 4.1b). The execution of the application is otherwise unchanged, and hence does not suffer any performance overhead.

We separate two dimensions of the careful selection of reification: the *spatial* dimension, and the *temporal* one.

4.2.1.1 Spatial selection

Spatial selection consists of selecting *what* will be reified in an application. Spatial selection can be done statically or dynamically. We distinguish between three different levels of selection:

- ***Entity selection*** refers to the selection of the reflective classes and objects. For instance, we may want to specify that classes `A` and `B` are *fully* reflective (all their instances are reflective), and instance `c` of class `C` is reflective (class `C` is *partially* reflective). Other classes and objects are left intact.
- ***Operation selection*** refers to the possibility of selecting which operations are reified for a given reflective entity (*i.e.* a class or an object). For instance, we may want to reify message receive and field access for class `A`, and only message send for class `B`.
- ***Intra-operation selection*** refers to the possibility of performing fine-grained selection with respect to a particular operation. This selection may be based on characteristics of specific occurrences; for instance, we may want to limit message receive reification to message `foo` in class `A`. For caller-side operations, the selection can also be based on the method/constructor where such an occurrence is found; for instance, we may want to reify only the message sending of `foo` (to instances of class `A`) that occurs in all public methods of `B`.

Intra-operation selection is a crucial property of our approach to partial behavioral reflection. Indeed, it is the finest grain of control over the spatial dimension of the reification process, making it

possible to ignore most of irrelevant operation occurrences during transformation, so that metaobjects do not have to do such selection at runtime.

4.2.1.2 Temporal selection

Temporal selection consists of selecting *when* reifications are effectively active. It optimizes the overall performance of a system making use of reflection a step further. During the lifetime of an object, the reflective needs may change: a reification may have to be turned off so that metalevel behavior does not apply anymore, or conversely, some external condition may require activating a reification. Obviously, for temporal selection to be worthwhile, the cost of a deactivated reification should be less than the cost of an activated reification with an empty metaobject. Our benchmarks (Sect. 4.3.4.3) demonstrate this fact in the context of Java.

4.2.1.3 A Model of Partial Behavioral Reflection

Spatial and temporal selection consists of precisely selecting the execution points that need to be reified, in order to apply metalevel behavior when needed. The issue is then to manage the hooks, in order to define how they are linked to the metalevel. Traditionally, behavioral systems adopt a metalink that gathers hooks on a per class or per object basis. However, we feel that such a classical view on metalinks is limited. For the sake of generality, we propose the notion of *hooksets*: hooksets may gather execution points scattered in various objects. This has the nice property of making it possible to apply a metaobject modularly implementing a concern that *crosscuts* the object decomposition. Also, a given object may be involved in several hooksets. This allows for a better modularity of the metalevel, since metaobjects may be assigned a single responsibility. Since the sets of hooks are not restrained to a particular object or class, they need to be named in order to be manipulated.

Hooksets can be composed using the standard set operators: a hookset can be defined as the union of two hooksets, their intersection, their difference, or any combination of these. This represents a powerful reuse mechanism for hookset definitions.

In our model, the metalink, to which we will refer simply as the *link*, can be described by several attributes, among which:

- ***scope***: determines whether there is one global metaobject controlling each hook in the link hookset (*global scope*), or if each selected class has a particular metaobject handling hooks occurring within its instances (*class scope*), or if each object has a dedicated metaobject (*object scope*).
- ***activation***: a dynamically-evaluated activation condition that can be specified to achieve expressive temporal selection. The activation condition may be set at various levels (link, class, object).
- ***control***: determines whether the metaobject is given control *before*, *after*, *before and after* an operation occurrence or if it can *replace* it.

- ***mintypes***: makes it possible to impose type restrictions on the metaobjects that are linked to the hookset.
- ***updatable***: specifies whether the link may be dynamically modified (*i.e.* the hookset is linked to another metaobject).

In this model, spatial selection is done by defining hooksets, while temporal selection is done through the activation condition of the link. This flexible model is similar to the Event-Condition-Action model used for example in active databases [Dittrich *et al.*, 1995]. The event layer is realized by hooksets (hooks are indeed event sources), the condition layer is realized by the activation condition attached to the link, while actions are implemented by metaobjects (Fig. 4.2).

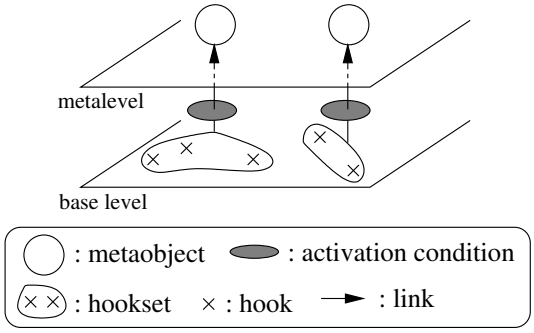

Figure 4.2 – The model of hooksets.

We believe that it is important to allow for the *separate* definition of hooksets, metaobjects, and links in order to maximize their reuse, and to decouple the work of the assembler from that of the metaprogrammer.

4.2.2 Static and dynamic configuration

As we can see, many aspects of behavioral reflection can be configured. Some can be configured statically, and others need to be configured dynamically. For instance, the *assembler* may know that a given reification needs to be activated all the time. On the contrary, a *metaprogrammer* may want to activate some reification on runtime events. As a consequence, both a static configuration API and a runtime API, for dynamic configuration, are needed. The resulting application can then be configured by a combination of static and dynamic configuration.

4.3 Partial Behavioral Reflection for Java

The second version of Reflex supports partial behavioral reflection as presented in Section 4.2. Although the architecture presented here is dedicated to Java, the concepts and underlying ideas of

this architecture are language-independent. Reflex is said to be *open* because, as opposed to other reflective extensions, it does not impose any specific MetaObject Protocol (MOP) [Kiczales *et al.*, 1991], thanks to a layered architecture. Indeed, Reflex allows metalevel architects to define their own MOP, based on the framework provided by *Core Reflex*, possibly reusing parts of a *standard MOP* library. This library, built on top of the Core, turns Java into a ready-to-use behavioral reflective system.

In this section, we first introduce the core architecture of Reflex, and then present a standard MOP built on top of it. Based on these elements, we present the interfaces provided by Core Reflex for configuration (both static and dynamic). Finally, we describe some aspects of the implementation of Reflex, and discuss the results of our micro-benchmarks.

4.3.1 Core Reflex architecture

We previously introduced the role of metalevel architect in the whole process of developing a reflective application. This architect is responsible for defining a specific MOP reflecting a chosen compromise between efficiency, expressiveness and flexibility. This definition is based on Core Reflex.

The overall execution-time picture assumed by Core Reflex is the following (Fig. 4.3): when a class is about to be loaded, it goes through a selection process whereby Reflex determines whether some links apply to the class. If this is the case, the class is transformed into a reflective class before being loaded. Otherwise, the class is loaded as usual.

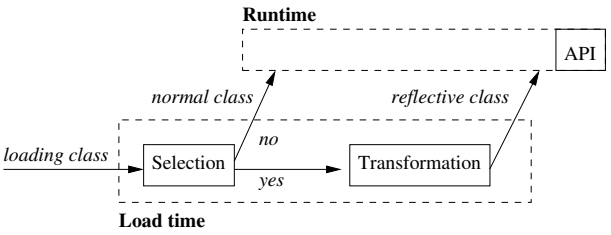

Figure 4.3 – Overview of Core Reflex.

Load-time transformation is discussed in the following section. The selection process is controlled by the assembler: it is driven by defining hooksets, whose definition rely on spatial selection entities (Sect. 4.3.1.3), and links (Sect. 4.3.1.5). Temporal selection (Sect. 4.3.1.4) is initialized statically. A runtime API (Sect. 4.3.3.2) is provided to the metaprogrammer in order to dynamically control links.

The transformation process is controlled by the meta-architect: the architect defines the available MOP. Defining a MOP basically consists of specifying which operations are reifiable (*e.g.* message sending), what the interfaces of metaobjects are, and providing code transformation entities able to insert hooks (Sect. 4.3.1.2). Several MOPs can coexist in a given reflective application.

4.3.1.1 Load-time bytecode transformation

Because of our concern with the concrete applicability of behavioral reflection, the concretization of our architecture takes the form of a portable Java library. Reflex relies on load-time bytecode transformation, in order to be applicable to binary components and in settings where all classes are not known until they are actually loaded (*e.g.* open distributed systems)[1].

To make classes reflective, Reflex uses the Javassist framework for load-time structural reflection [Chiba, 2000], as mentioned in the previous chapter. Javassist relies on a specific class loader [Liang & Bracha, 1998]. When requested by the virtual machine, the loader forwards the request to a *class pool*, in charge of locating class definitions. When a class is located and about to be loaded, the class pool can notify a *translator*, which can then modify the class. To this end, the translator can obtain reification of classes as `CtClass` objects. `CtClass` offers the same introspection capabilities as those of the standard reflection API of Java, plus intercession capabilities (*e.g.* adding/modifying a member, changing the superclass, altering method bodies...).

Reflex is connected to Javassist through a particular translator, a class builder. As a consequence, classes in this version of Reflex are represented at load time as `CtClass` objects. Conversely to the previous chapter, a class builder is now part of Core Reflex: users do not control the transformation process by providing their own class builder. Rather, the generic class builder of Reflex delegates the task of hook insertion for a given operation to finer-grained entities, called *hook installers*, presented hereafter. The generic class builder factors out common infrastructure setting work, while hook installers are restricted to operation-specific details. Their composition is managed by the framework via delegation. This is more flexible than what we presented in the previous chapter, where class builders were composed by the architect through inheritance.

4.3.1.2 Defining MOPs

Reflex is an open platform on top of which specific MOPs can be defined. In itself, Reflex does not support any language operation nor does it impose any specific interfaces for metaobjects.

The `OperationSupport` class is used to define the support for an operation in a MOP. An instance of this class encapsulates a *static operation class* and a *hook installer*.

Operations. The first step when defining a MOP consists of specifying the operations that are supported. For instance, a basic MOP might only offer support for the message receive operation, while a well-furnished MOP might support almost all operations available in the language.

Operations are represented at load time by *static operation classes*. Core Reflex only provides a marker interface for these classes, `StaticOperation`. An instance of a static operation class represents the static occurrence of an operation in a base-level class definition. Hence, by defining operation classes, the metalevel architect controls the exact contents and interface of these load-time representations.

Similarly, so-called *dynamic operation classes* represent language operations during execution. Instances of such classes are *runtime* objects representing an operation occurrence during

[1] Load-time transformations can also be performed at compile time in order to avoid load-time overhead, provided that classes that need to be processed are available.

execution.

Metaobjects. The second step consists of specifying the *interface* of metaobjects, that is to say, what data is reified and how this data will be passed to metaobjects at runtime upon occurrence of language operations. Within the base program, much information is potentially reifiable. All this information is not necessarily of interest to the metalevel architect. For instance, a general-purpose MOP might provide full reifications (including all reifiable data), while a more specific MOP might only provide a simple string description of the operation occurrence.

There are three alternatives for a MOP to pass reified data to metaobjects during execution. Reified data can be passed as a set of arguments, as a single array, or encapsulated within a dynamic operation instance. Specification of parameter passing is driven by a trade-off between abstraction and efficiency.

All these decisions shape the specific interfaces that metaobjects are expected to implement. Core Reflex only provides marker interfaces for distinguishing metaobjects: `Metaobject`, and its subinterfaces, `Before`, `After`, `Replace`. Section 4.3.2.4 presents the hierarchy of metaobject interfaces of the standard MOP.

Hook installers. Hook installers are bytecode transformation entities that have basically two responsibilities: *parsing* a class definition to find occurrences of a given static operation class, and *generating* the appropriate hooks to install, if any. This includes generating the code that will build the reification, and the code that will do the delegation to the metaobject, through the appropriate interface. Hook installers must therefore generate metaobject method invocation and parameter passing according to the expected interface of the metaobject. Reflex provides the `Installer` interface for hook installers. These entities can be implemented using tools such as Javassist, Jinline [Tanter *et al.*, 2002a] or BCEL [Dahm, 1999].

4.3.1.3 Spatial selection

In Reflex, spatial selection is done at load time. Each time a class is loaded in a JVM, Reflex determines which links have to be installed by applying *class selectors*, and then which hooks should be inserted in the class bytecode by applying *operation selectors*. Hooksets are indeed defined *intentionally*, in the sense that the set of points is deduced from the application of predicates (embodied in the selectors).

A **class selector** is responsible for selecting classes (*entity selection*). It implements the `ClassSelector` interface:

```
public interface ClassSelector {
  public boolean accept(CtClass aClass);
}
```

The method `accept` returns true if the class `aClass` should be selected. Having a reification of the class (as a `CtClass`) allows the class selector to select a class on any introspectable characteristics (*e.g.* the class hierarchy, the parameter types of its public methods, etc.), as illustrated in Section 4.5.

An **operation selector** is responsible for selecting operation occurrences (*intra-operation selection*). It implements the `OperationSelector` interface:

```
public interface OperationSelector {
  public boolean accept(StaticOperation anOp,
                        CtClass aClass);
}
```

An operation selector can select operation occurrences based on the characteristics of the static operation instance, and on the class to which the selector is applied. For caller-side operations, the operation selector can also limit the scope of reification by selecting operation occurrences based on the specific method/constructor in which they appear.

The use of selectors can be heavyweight if one needs to code specific selectors for each particular case. To facilitate specification of spatial selection, some general-purpose selectors are provided with Reflex, such as the `NameCS` class selector used in the forthcoming examples, which accepts any class whose name was given as a constructor parameter. Providing advanced selectors, able to interpret more expressive parameters (*e.g.* regular expressions) would be of great value.

4.3.1.4 Temporal selection

Since spatial selection is done statically based on the program code, we need to provide control for activation at the object level, in order to support reification on a per-instance basis. In other words, instance-based temporal selection (activation) is used to support per-instance entity selection as well. Furthermore, to be more expressive, activation conditions should possibly be user-defined. Hence, an activation condition may either be a constant, `ON` or `OFF`, or a user-defined condition. Such a condition is an object implementing an interface, `Active`, which declares an `evaluate()` method that receives as argument the current object:

```
public interface Active {
  public boolean evaluate(Object o);
}
```

In addition to providing fine-grained control on activation, it is very convenient to be able to control activation more globally, that is to say, at the class level and at the link level. In order to provide these functionalities in our API, we have two implementation alternatives. The first one consists of keeping only an activation condition in each object, but then to control activation globally we need to be able to retrieve each class involved in a link and each instance of a given class. This obviously has a huge overhead in terms of memory usage. Hence, we adopt a solution based on an *activation hierarchy*, in which the link level is given priority over the class level, which in turn has priority over the object level. A new condition, `SUB`, is provided to indicate that a given level delegates the responsibility of determining activation. `SUB` is not meant to be evaluated, and is prohibited at the object level, since it does not make sense.

As a result, a hook belonging to a hookset of link l in a given object o, instance of class C, will be active if and only if:

- the activation condition of l evaluates to `true`, or

- the activation condition of l is set to `SUB` and the activation condition of C evaluates to `true`, or
- both the activation condition of l and of C are set to `SUB` and the activation condition of o evaluates to `true`.

4.3.1.5 Defining hooksets and links

Hooksets. Since hooksets can be composed using the standard set operators (Sect. 4.2.1.3), we distinguish between *primitive* and *composite* hooksets (Fig. 4.4).

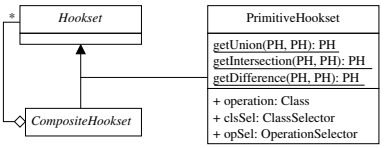

Figure 4.4 – UML class diagram of hookset definitions.

A primitive hookset is an operation-specific set defined by a triple *[operation, class selector, operation selector]*. If an execution point is affected by several links, there is a composition issue. This issue is left open for the moment, it will be addressed in the second part of this dissertation.

Links. Links are described at load time by `Link` objects, and make it possible to specify and characterize the association between hooksets and metaobjects. A link is identified by a unique identifier, and associates a hookset with a *metaobject definition*. It is characterized by a *set of attributes* (Fig. 4.5). At runtime, links are represented by `RTLink` objects, which provide means for the metaprogrammer to access metaobjects and activation conditions (see Sect. 4.3.3.2).

Metaobject definition. In Reflex, metaobjects can be created in a *lazy manner* (depending on an *initialization* attribute of the link). This has the advantage of avoiding unnecessary creations (*e.g.* if a link is never activated), while allowing metaobjects to control the execution of constructors. A `MODefinition` object specifies how the metaobject should be obtained. There are basically two means of obtaining metaobjects: either by instantiating a metaobject class, or by querying a *metaobject factory* (Fig. 4.5). In both cases, custom parameters can be specified. They will be passed as an array of strings.

A metaobject factory is responsible for returning the initial metaobject for a link:

```
Metaobject getMetaobjectFor(Object target, RTLink link, String[] args)
```

The factory method is called whenever the control flow reaches a hook whose metaobject has not yet been set (or has been reset). If the scope of the link is object, the `target` argument is the instance to which the metaobject should be linked. If it is class, then `target` is the concerned class object, and if it is global, `target` is simply `null`.

The factory method does not only return the metaobject for the object which is passed as a parameter. It may also, for instance, link this metaobject to other objects, making it a very convenient way to set up an instance-based *crosscutting metaobject*.

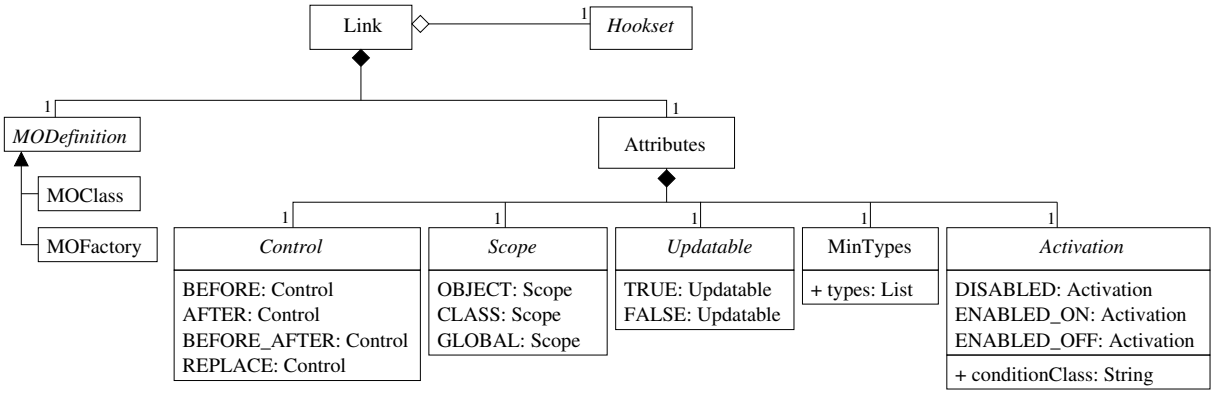

Figure 4.5 – UML class diagram of link definitions.

Attributes. The various link attributes presented in Section 4.2.1.3 are available in Reflex. More link attributes have been added to Reflex as new needs are being discovered. All link attributes are encapsulated within an `Attributes` object (Fig. 4.5). Control, scope, and updatability are represented by enumeration classes. Mintypes is simply specified as a list of type names. For better flexibility when dealing with composite hooksets, the control attributes may be set to a specific value for each composed hookset individually.

Activation support is optional in Reflex. If the activation attribute is `DISABLED`, then the link is not activatable, *i.e.* it is always active. If activation support is needed, then it is necessary to specify what activation condition is associated to the link when the application starts:

```
link.setActivation(new Activation("MyCond"));
```

All levels of activation will be set to the same instance of the class `MyCond` (which implements `Active`). For convenience, `ENABLED_ON` and `ENABLED_OFF` are provided. They correspond respectively to `Active.ON` and `Active.OFF` as initial activation conditions.

4.3.2 Standard MOP

This section describes the standard MOP that is provided by default with Reflex. This MOP is designed to be general-purpose and expressive.

4.3.2.1 Static operations

Operations are represented by a hierarchy of static operation classes, which are subclasses of the abstract `StdStaticOperation` class (Fig. 4.6). Each static operation class defines accessors to the various data that can be extracted from the code at load time. For instance, `MsgSend`, which represents the message send operation, defines accessors to the receiver type, the name

of the method, the argument types, and so on. Caller-side operation classes have a `getWhere` method to retrieve the method or constructor where the operation occurrence was found, making it possible for operation selectors to refine their selection.

4.3.2.2 Dynamic operations

Dynamic operations are represented by a hierarchy of dynamic operation classes (Fig. 4.7). A dynamic operation object encapsulates all the *runtime* information available describing the characteristics of the corresponding operation occurrence.

4.3.2.3 Static / dynamic operation dependencies

When a shift to the metalevel occurs, the target metaobject receives as argument the identifier of the hookset whose member triggered the shift, along with the array of objects referencing the various runtime values of interest. Access to the elements of this array is eased by the fact that a static operation class defines a set of constants (final class variables) that correspond to the indices of the information in the array (Fig. 4.6).

An instance of a dynamic operation class is an object representation of the array, which is passed to the metaobject at runtime. Therefore, a dynamic operation class *depends* on its associated static operation class in the sense that it should define information that maps the information declared in the static operation class. To convert an array into a dynamic operation object, dynamic operation classes implement a static method `convert(Object[])`, which metaobjects can use as follows:

```
class MyMetaobject implements ReplaceCast {
  Object replaceCast(Object[] data){
    DCast cast = DCast.convert(data);
    make use of the DCast object
} }
```

This scheme is provided since manipulating dynamic operation instances is convenient but costly. We obtain both flexibility and abstraction (it is easy to get a dynamic operation as an object), and efficiency (one does not pay for it if direct access to the array is enough).

4.3.2.4 Metaobjects

Metaobjects are instances of classes implementing one of the marker interfaces, `Before`, `After`, and `Replace`, which serve as superinterfaces to operation-specific interfaces (Fig. 4.8).

The method signatures of these interfaces look as follows:

```
void before<i>OP</i>(Object[] data)
void after<i>OP</i>(Object[] data)
Object replace<i>OP</i>(Object[] data)
```

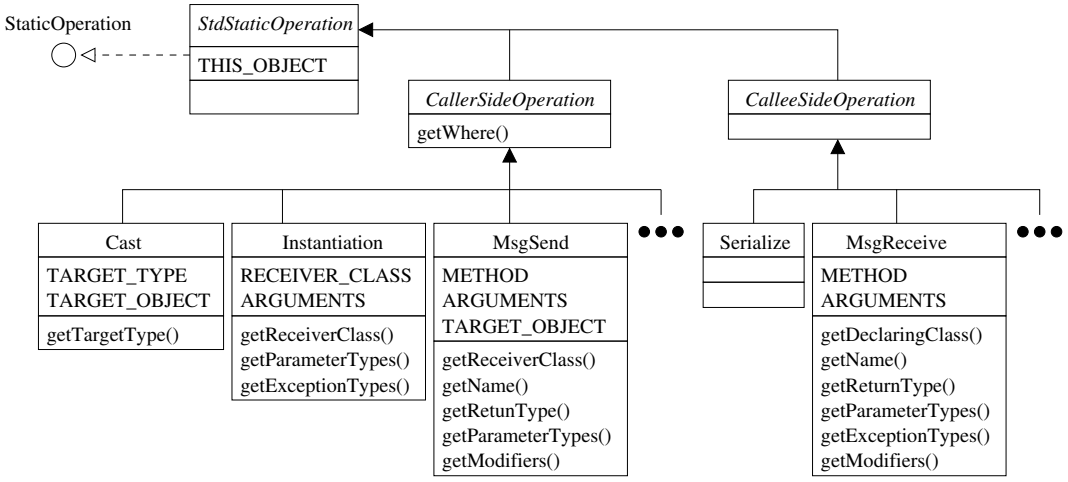

Figure 4.6 – Part of the hierarchy of static operation classes (standard MOP).

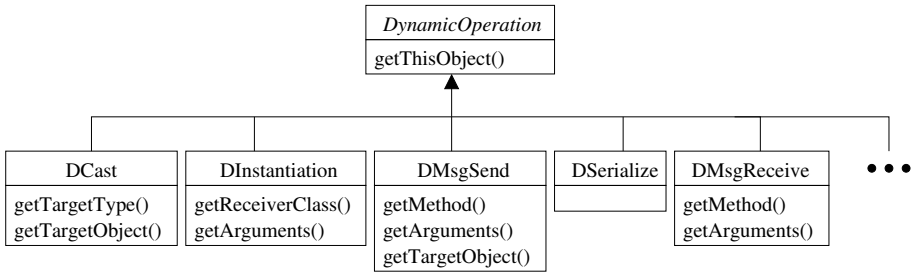

Figure 4.7 – Part of the hierarchy of dynamic operation classes (standard MOP).

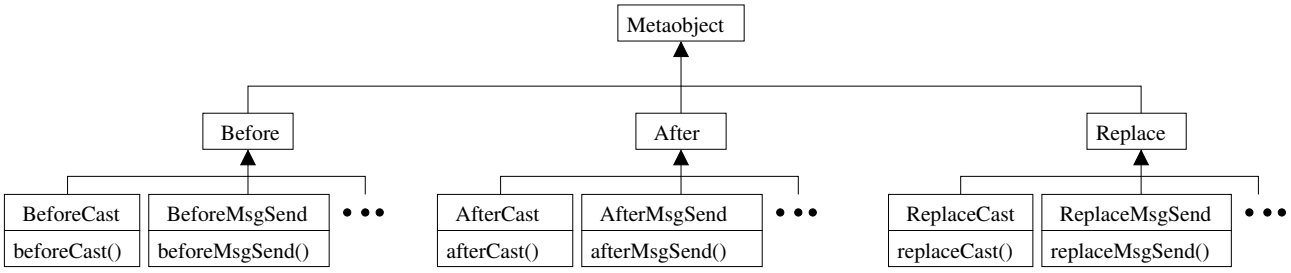

Figure 4.8 – Part of the hierarchy of metaobject interfaces (standard MOP).

Only *replace* metaobjects can change arguments and/or the return value of an operation, while the other types of metaobjects are only able to provoke side effects. Actually, only `replaceOP` methods have return type `Object`, while the others have return type `void`.

Since a metaobject class can be multi-operation, it can implement any combination of interfaces. Indeed, dynamic operations are very much like *events*, and metaobjects like *listeners* of various kinds of such events.

4.3.3 Configuration

Reflex explicitly supports the roles of metalevel architect and assembler. Hence, Reflex provides configuration interfaces for them (Fig. 4.9). Access to these interfaces can be done in various ways, statically and dynamically, as explained later in this section.

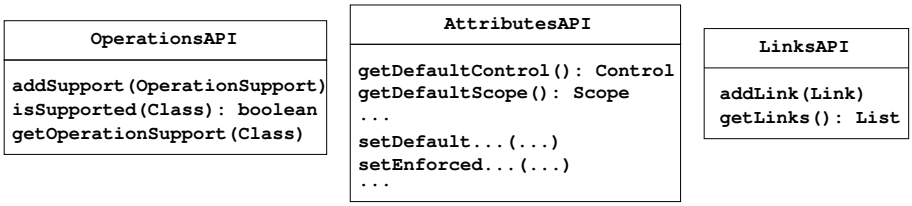

Figure 4.9 – Interfaces for configuration.

In addition to defining supported operations through the `OperationsAPI`, the metalevel architect can provide default values for the various attributes, using the `AttributesAPI`, and also restrict the permitted values for the various attributes. For instance, by setting the mintypes attribute, the metalevel architect can enforce the use of a particular metaobject framework. This can be useful to impose a security framework [Caromel & Vayssière, 2001], or to provide a common metaobject composition framework [Mulet *et al.*, 1995; Oliva & Buzato, 1999; Tanter *et al.*, 2001]. The assembler mainly uses the `LinksAPI` to define links.

Configuration constraints (*e.g.* uniqueness of link identifiers, respect of enforced attribute values, consistency) are checked when new definitions are made; thus these methods throw exceptions in case of violations.

4.3.3.1 Static configuration

In this version, Reflex supports two forms of static configuration: using *configuration classes*, and using *XML configuration files*. Creating a convenient domain-specific language (DSL) for such configuration is an interesting perspective, and is explored in the next part of this dissertation.

Configuration classes. Configuration of Reflex can be done by providing configuration classes. A configuration class is a class that implements at least one of these two methods:

```
static void initReflexArchitect() // for the architect
static void initReflex()          // for the assembler
```

Architect configurations are applied before assembler configurations. As an illustration, the following class implements an architect configuration that adds support for the message send operation, using the standard MOP of Reflex:

```
public class SampleArchitectConfig {
  public static void initReflexArchitect(){
    OperationSupport msgSendSupport =
      new OperationSupport(
        reflex.std.operation.MsgSend.class,
        new reflex.std.installer.MsgSendInst());

    OperationsAPI.addSupport(msgSendSupport)
}}
```

Hookset and link definitions can be done in the same manner. The following class illustrates the possibility of embedding the hookset and link definitions within a metaobject class (see the discussion on locality in Section 4.4.2):

```
public class SampleMO implements BeforeMsgSend {
 public static void initReflex(){
   PrimitiveHookset hs =
    new PrimitiveHookset(MsgSend.class,
                    new NameCS("Foo"),
                    new MsgNameOS("run"));
   Link l =
     new Link(hs,
              new MOClassDefinition("SampleMO"));
   l.setScope(Scope.GLOBAL);
   l.setControl(Control.BEFORE);

   LinksAPI.addLink(l);
 }

 public void beforeMsgSend(Object[] data){
   DMsgSend msgSend = DMsgSend.convert(data);
   System.out.println("About to send message: " +
                      msgSend.toString());
}}
```

This class defines a primitive hookset `hs` that gathers all calls to a `run` method occurring within a `Foo` class. These operation occurrences will be reified and passed to a unique metaobject (scope is *global*), which will be given control before such occurrences. The defined metabehavior is a simple trace.

XML Configuration files. Reflex also includes an XML parsing module for configuration files. The syntax is a straightforward mapping of the object-oriented configuration presented above. XML configuration is used in the examples of Sections 4.4 and 4.5.

Running Reflex. To run an application with Reflex using given configuration files and/or classes, the command:

`java` *Application arg1 arg2...*

should be replaced by:

```
java reflex.Reflex -configClasses Config1:Config2...
                   -configFiles file1:file2...
                   Application arg1 arg2...
```

4.3.3.2 Dynamic configuration

The runtime API of Reflex allows for dynamic configuration by the architect, the assembler, and the metaprogrammer. The APIs for the architect and the assembler (Fig. 4.9) can be accessed at runtime. All changes made through these interfaces (*e.g.* newly defined links) *only apply to classes loaded afterwards* (see the discussion in Section 4.3.4.2).

The metaprogrammer API is implemented as a set of static methods of the `Reflex` class. It offers a service to create particular reflective instances (similar to the `createObject` method of the first version of Reflex). The new part of this API makes it possible to retrieve runtime link objects:

```
RTLink getRTLink(String linkID);
```

Runtime link objects are used to control temporal selection by giving access to activation conditions, and metabehavior, by giving access to metaobjects.

Activation API. Using a `RTLink` object, it is possible to set and retrieve the activation condition associated to an activatable link, at the desired level (link, class, object). The methods to set an activation condition are:

```
setActiveForLink(Active a)
setActiveForClass(Class c, Active a)
setActiveForObject(Object o, Active a)
```

The `set` methods have `get` counterparts to retrieve an activation condition.

Metaobject Access API. A link object also makes it possible to retrieve and set metaobjects at the desired level (depending on the scope attribute of the link). To set metaobjects, the following methods are provided:

```
setMetaobject(Metaobject m)
setMetaobject(Class c, Metaobject m)
setMetaobject(Object o, Metaobject m)
```

These methods have `get` counterparts to retrieve metaobjects. The Reflex runtime ensures consistency and enforces the restriction rules that may have been given (*e.g.* mintypes, updatable).

4.3.4 Implementation

4.3.4.1 Support for hook activation

The portable support for hook activation/deactivation is implemented with activation conditions, as presented in Section 4.3.3.2. Classes and objects involved in an activatable link include an extra field to hold their activation condition. Classes involved in links own references to the corresponding `RTLink` objects.

An activatable hook differs from a normal one in that a check of the activation conditions (in the order: link, class, object) is performed before building the reification and invoking the metaobject. Hence, the code of any activatable hook is as follows:

`if(`*determine activation*`)` *reify and delegate*
`else` *original code*

where the `else` clause is not present for before or after hooks.

4.3.4.2 Limitations

Our implementation is constrained by the necessity of remaining portable and compatible with standard Java. This precludes, at least in a first step, the possibility of resorting to features of the Java Platform Debugging Architecture (JPDA), such as class reloading, which is not meant to be turned on in a production environment. Investigating the benefits of the JPDA in prototyping or debugging environments is certainly valuable and may be part of future work.

Therefore, in the current version of Reflex we are neither able to define new links dynamically for already loaded classes, nor to make activatable a non-activatable link. This also explains our implementation of hook activation. In particular, note that, even in the case of a deactivated hook, the activation conditions are still checked repeatedly during execution. *Once loaded in the execution space, class definitions are never altered.*

Furthermore, there is a one-to-one relation between a `Link` object and a `RTLink` object: the `RTLink` object is created when first needed, and references its associated `Link` object. Due to the restrictions of Java entailing that a class definition cannot be changed at runtime, the causal connection between these two representations is incomplete: a runtime change in the definition of a link, for instance of the boundaries of the hookset, is only reflected for newly loaded classes. Moreover, changing the definition of a link at runtime may lead to inconsistencies. For instance, changing the scope of a link dynamically would invalidate the metaobject access scheme for already-loaded classes. Therefore, we forbid the change of constituents of a link definition that cannot be safely changed at runtime. This issue actually deserves further study and may be solved by considering a more dynamic implementation of our model.

4.3.4.3 Benchmarks

We have performed micro-benchmarks to validate the interest of partial behavioral reflection. The machine used runs under Linux (kernel 2.4), with an Intel Celeron 1.2GHz processor and 760MB

of RAM, using the HotSpot client VM from Sun Microsystems version 1.4.1 (build 1.4.1-b21, mixed mode).

The benchmarks are based on the *message send* operation, as implemented in the default MOP provided with Reflex (v1.0 alpha 4). We have measured the execution time of a `loop` method that calls a given number of times a `test` method. Both methods are defined in class `Sample`. In order to associate some computation to the `test` method, the body of this method is a loop incrementing a counter. The aim of the benchmarks is to measure the overhead of applying reflection to calls to the `test` method.

Our micro-benchmark suite consists of the following test cases:

- In the first test case (*non-Reflex*), we measure the execution time of the application without Reflex.
- The second test case (*non-reflective*) consists of running the application with Reflex, including parsing the configuration files and transforming class `Sample` to reify the sending of `test` occurring in `loop`. However, in this case, we use a normal, *i.e.* non-reflective, object. This lets us measure the sole cost of load-time transformation, without measuring the runtime cost of reflection.
- The following three test cases measure the cost of an empty before/after metaobject respectively when activation is disabled (*bef/aft no act*), when it is enabled and the link is activated (*bef/aft act on*), and finally when activation is enabled but the link is deactivated (*bef/aft act off*).
- We finally considered three similar cases, using replace control instead of before/after (*replace no act*, *replace act on*, *replace act off*). Recall that in these cases, method invocation is done using the Java Reflection API.

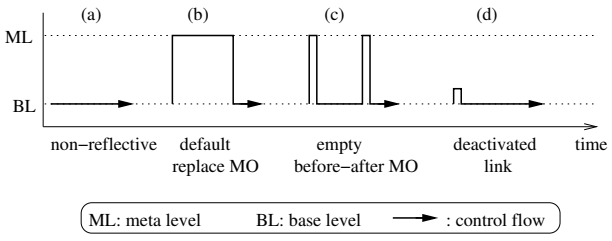

Figure 4.10 – Representation in a reflectogram of the various types of test settings.

Fig. 4.10 illustrates the various test cases in terms of reflectogram: *non-Reflex* and *non-reflective* are of type *(a)*, *replace no act* and *replace act on* are of type (b), *bef/aft no act* and *bef/aft act on* are of type *(c)*, while the others are of type *(d)*.

We have measured the cost of running the whole application (*i.e.* starting a new JVM each time) with the Linux `time` command. We have checked that all test cases got a full control of the CPU (99%) and that garbage collection was not triggered.

invocations	1000	5000	10000	20000	50000
non-Reflex	11	37	71	149	401
non-reflective	24	51	85	152	354
bef/aft no act	25	59	102	187	443
act on	26	59	102	188	444
act off	25	55	93	169	396
replace no act	26	61	105	193	455
act on	27	62	106	194	457
act off	25	55	93	169	396

Table 4.1 – Micro-benchmarks results (time in 1/10s).

The two first lines of Table 4.1 illustrate the overhead of load-time transformation. Plotting these figures gives two parallel straight lines from 1000 to 10000 invocations. The slope of these lines gives the cost of the elementary test: 0.67ms. The value of the ordinate at the origin is 0.4s without Reflex, and 1.8s with Reflex. That is, bytecode transformation has an important impact on start-up time. Partial reflection makes it possible to reduce this impact. For a higher number of invocations, it turns out, quite unexpectedly, that Reflex code becomes more efficient than the Java code. This does not seem to be an epiphenomenon (resulting, from instance, of some HotSpot deoptimization) as the same effect can be observed with interpreted code.

All the other figures correspond to straight lines: there is no runtime disturbance (for instance coming from memory management, or HotSpot), and it is therefore easy to compare the various overheads.

- Comparing non-reflective and reflective (replace) execution, we can deduce an overhead of $2\ 10^{-4}$s per reflective invocation. This overhead is indeed pretty high: we have benchmarked standard invocation cost in the same configuration, obtaining a cost of $6\ 10^{-8}$s per invocation. This measurement was done in interpreted mode only, hence it actually represents an upper bound for the invocation cost with HotSpot enabled. The fact that the overhead of reflective invocation is between 3 and 4 orders of magnitude greater than a standard invocation validates the need to precisely select where and when reification occurs.

- The results also show that before/after control is less expensive than replace control, with an overhead of $1.8\ 10^{-4}$s per invocation compared to non-reflective execution. The gain is however quite moderate. The situation is fairly different in interpreted mode where another series of measurements have shown that the overhead of before/after control was half the overhead of replace control.

- Finally, deactivation is also worthwhile as it reduces the cost of replace and before/after

control to $8.4\ 10^{-5}$s (these figures could probably be further improved). On the other hand, one can easily see that the cost of activation is negligible when enabled.

These tests call for further measurements. In particular, benchmarking reflection on sizeable applications, while reifying various operations, would be of great interest. These results may also evolve as we believe that the implementation of Core Reflex can still be improved in a number of ways.

4.4 Illustration: Observer Pattern

In this section we illustrate the implementation of the Observer design pattern [Gamma *et al.*, 1994] with Reflex, and contrast it with the corresponding Java and AspectJ [Kiczales *et al.*, 2001] implementations, in the line of the work presented at OOPSLA 2002 by Hanneman and Kiczales [Hanneman & Kiczales, 2002].

The Observer design pattern is one of the patterns that involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern. As clearly shown in [Hanneman & Kiczales, 2002], the Java implementation of this pattern leads to mixing functional code of the participating classes with pattern-specific code, which we would like to avoid.

Let us give the Reflex implementation of the Observer pattern for the simple scenario presented in [Hanneman & Kiczales, 2002]. This scenario includes `Point` objects that are observed by some `Screen` objects. Furthermore, in order to illustrate composability of patterns, some of the screen objects play both the role of *observers* (of the point) and *subjects* for another screen object.

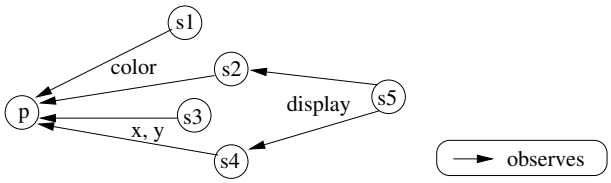

Figure 4.11 – Observation relations between the point object and the various screen objects.

The scenario includes one point `p` and 5 screen objects, `s1`, `s2`, `s3`, `s4` and `s5`. The observation relations are as follows (Fig. 4.11):

- `s1` and `s2` observe *color changes* in `p`,
- `s3` and `s4` observe *coordinate changes* in `p`,
- `s5` observes *changes in the display* of `s2` and `s4`.

4.4.1 Implementation with Reflex

4.4.1.1 Configuration

We have chosen to illustrate the configuration with XML files (Fig. 4.12).

The architect configuration defines support for the two operations that are used in the present example, (*field write* and *message receive*), using the standard MOP, along with default attribute values. Notice that, by default, reifications are activatable with activation initially off.

The definition of three links is required (Fig. 4.12):

1. `coordinateObserved` applies to instances of class `Point`; notification is triggered upon occurrences of a *field write* operation, where the name of the field is either `x` or `y`.
2. `colorObserved` also applies to instances of class `Point`, and notification is triggered upon occurrences of a *field write* operation, where the name of the field is `color`.
3. `displayObserved` applies to instances of class `Screen`; notification is triggered upon occurrences of a *message receive* operation, where the name of the message is `display`.

4.4.1.2 Metaobjects

In each of the three links, the metaobject is an instance-specific metaobject that informs the relevant observers *after* the corresponding operation occurrences. To this end, each metaobject holds a list of the observers attached to a particular subject.

The generic part of the pattern behavior (maintaining a list of observers, notifying each of them when needed) has been factored out in the abstract `Observer` metaobject class. This class also offers a service to create observing relations (`map`). The case-specific part of the update logic is left to concrete subclasses: `Observer` defines an abstract `updateObserver` method (Fig. 4.13).

We need to define three metaobject classes for this example: `CoordinateObserver` and `ColorObserver` (Fig. 4.13) for observing changes in point coordinates and color, respectively, and `DisplayObserver` for observing display changes on screens.

4.4.1.3 Running the scenario

The `Main` class runs the proposed scenario by first creating the point and screen objects, and then makes the mapping observers-subject as explained above (Fig. 4.14). Fig. 4.15 illustrates the configuration at runtime. For instance, when `p` changes color, this change is reified and passed to metaobject `2` that then notifies each observer (`s1` and `s2`).

```
<reflexConfig-architect>
  <operations>
    <operation class="reflex.std.operation.FieldWrite" name="fieldWrite"
               installer="reflex.std.installer.FieldWriteInst" />
    <operation class="reflex.std.operation.MsgReceive" name="msgReceive"
               installer="reflex.std.installer.MsgReceiveInst" />
  </operations>
  <defaultAttributes scope="object" activation="enabled-off" control="after" />
</reflexConfig-architect>
```

```
<reflexConfig-assembler>
 <link id="coordinateObserved">
  <hookset operation="fieldWrite" classSelector="NameCS"
           argsCS="observer.Point" operationSelector="FieldNameOS"
           argsOS="x | y"/>
  <metaobjectDefinition class="observer.CoordinateObserver"/>
 </link>
 <link id="colorObserved">
  <hookset operation="fieldWrite" classSelector="NameCS"
           argsCS="observer.Point" operationSelector="FieldNameOS"
           argsOS="color"/>
  <metaobjectDefinition class="observer.ColorObserver"/>
 </link>
 <link id="displayObserved">
  <hookset operation="msgReceive" classSelector="NameCS"
           argsCS="observer.Screen" operationSelector="MessageNameOS"
           argsOS="display"/>
  <metaobjectDefinition class="observer.ScreenObserver"/>
 </link>
</reflexConfig-assembler>
```

Figure 4.12 – XML configuration of the metalevel architect (*top*) and the assembler (*bottom*) for the Observer design pattern implementation.

```
public abstract class Observer {
 protected List itsObservers;
 public void addObserver(Object o){ itsObservers.add(o); }
 public void removeObserver(Object o){ itsObservers.remove(o); }
 protected void updateObservers(Object subject){
   Iterator theIter = itsObservers.iterator();
   while (theIter.hasNext())
     updateObserver(theIter.next(), subject);
 }
 protected abstract void updateObserver(Object obs,
                                        Object sub);

 public static void map(String linkID, Object sub, Object obs){
   RTLink link = Reflex.getLink(linkID);
   ((Observer) link.getMetaobject(sub)).addObserver(obs);
   link.setActiveForObject(sub, Active.ON);
}}
```

```
public class ColorObserver extends Observer
                           implements AfterFieldWrite {
  public void afterFieldWrite(Object[] data){
    updateObservers(data[0]);
  }
  protected void updateObserver(Object obs, Object sub){
    ((Screen) obs).display("Screen updated " +
                           "because color changed.");
}}
```

Figure 4.13 – Metaobject classes for the Observer design pattern example. The `Observer` class is generic and reusable, while the `ColorObserver` class is specific to the example.

```
public class Main {
 public static void main(String[] args){
   Point p; Screen s1, s2, s3, s4, s5;
   create the objects
   Observer.map("coordinateObserved", p, s1);
   Observer.map("coordinateObserved", p, s2);
   Observer.map("colorObserved", p, s3);
   Observer.map("colorObserved", p, s4);
   Observer.map("displayObserved", s2, s5);
   Observer.map("displayObserved", s4, s5);
   play with p
}}
```

Figure 4.14 – Main program for the Observer design pattern example.

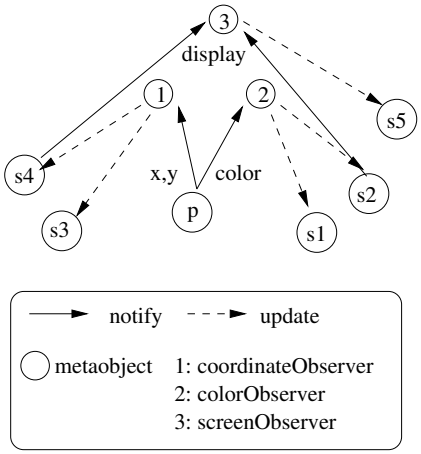

Figure 4.15 – Illustration of the configuration at runtime for the Observer scenario.

4.4.2 Assessment

The Reflex implementation of the Observer pattern presents the same modularity properties than the AspectJ implementation [Hanneman & Kiczales, 2002]:

- *Locality* – All of the code that implements the Observer pattern is located in the abstract and concrete metaobject classes; participant classes are left intact. This issue is further discussed below.
- *Reusability* – The core pattern code is abstracted and can hence be reused in other situations. For each pattern instance, we only need to define concrete metaobject classes and set the configuration files as required.
- *Composition transparency* – Since there is no coupling between a participant and the pattern, a subject or observer can take part in multiple observation relations.
- *(Un)pluggability* – Since participants are not *aware* of their role in a pattern instance, it is possible to switch between using and not using a pattern.

As far as locality is concerned, it is important to notice that both the Reflex implementation and the AspectJ implementation rely on an abstract class/aspect, three concrete subclasses/aspects, and a dynamic configuration to setup the scenario. A difference between Reflex and AspectJ is that, with AspectJ, the pointcut declaration and the link are embedded within the aspect definition. Although this can be seen as a nice property, it is also an example of tangled implementation: defining a set of points, a behavior, and the link between them are different concerns that should possibly be specified separately. Reflex allows for this separation, which enhances hookset, link and metaobject reuse. In cases where this level of separation is not needed, metaobject classes can embed their configuration (playing the role of configuration classes, as in Section 4.3.3.1), which is then equivalent to the AspectJ case.

Furthermore, Reflex preserves a pure object-oriented implementation style: it uses metaobject classes and configuration classes, both of which are implemented as standard Java classes. The configuration files are optional declarative alternatives to configuration classes. Conversely, aspects in AspectJ *are not* standard classes. Hence, Reflex can directly benefit from advanced support for Java development (*e.g.* incremental compilation in the Eclipse IDE [2]). On the other hand, AspectJ offers a nicer syntax for expressing aspects.

From a functional point of view, (un)pluggability can be taken a step further in Reflex than in AspectJ since the support for link (de)activation makes it possible to switch off/on the use of a pattern dynamically. Moreover, the sources of the underlying events can be (de)activated locally or globally very easily.

[2] `www.eclipse.org`

4.5 Illustration: Transparent Futures

We now show how to use Reflex to implement an important feature of distributed object systems that offer asynchronous communication: transparent futures (see for instance ProActive [Caromel *et al.*, 1998]). This example illustrates the use of runtime activation/deactivation, the need to control the cast operation, and the interest of our selection framework.

4.5.1 Transparent futures

Futures were first introduced in MultiLisp [Halstead, Jr., 1985]. In a distributed object system with futures, a call between two processes returns immediately. The client process does not need to wait for the value returned by the server process (unlike, for instance, Java RMI [SUN, 1998b]). The returned object, called a *future*, is in fact just a *place holder* for the actual result. While the server process executes, the client can continue its activity, and may even pass the future by reference to other objects/processes. The client process must block and wait for the server process to terminate only when the future is effectively accessed. This is called *wait-by-necessity*.

4.5.2 The problem of futures with simple MOPs

Using a runtime MOP is a common and easy way to implement transparent futures, as done in ProActive. A future is in fact a reflective object whose metaobject implements the wait-by-necessity strategy. However, apart from the well-known identity issue (the future *is not* the result), this approach suffers some limitations in a strongly-typed language like Java, due to the widespread use of downcasts. A simple MOP supporting only message receive is not sufficient. Indeed, when the future is created, the declared type of the result is known, but not its runtime type. This implies that the future might not be of an adequate type when downcasted later on.

Let us suppose that class `B` is a subclass of class `A`. Consider the following method `foo` of a class `Server`:

```
public A foo(){
  ...
  finally returns a B object
}
```

When this method is called asynchronously, the system creates a future, type-compatible with `A`, and returns it to the caller. With typical client code such as:

```
A a = Server.foo();
...
((B) a).m();
```

an exception will occur on trying to downcast the future before the result has been received.

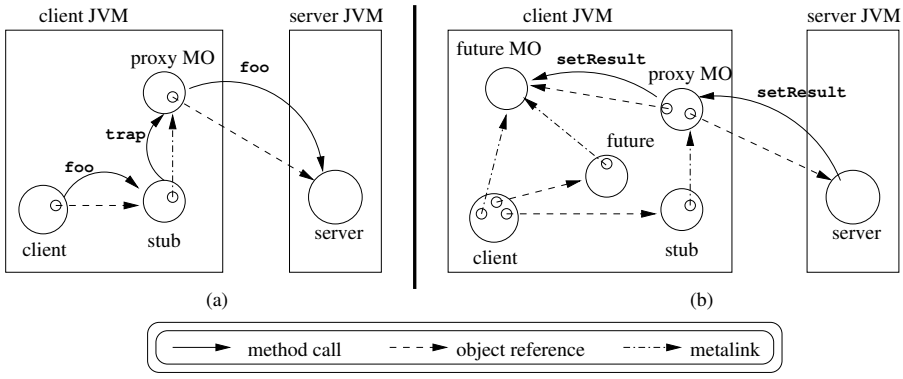

Figure 4.16 – Illustration of the future scenario. (a) The client calls `foo` on the server. (b) The future has been returned to the client and later, the result is delivered to the future metaobject.

4.5.3 A solution with Reflex

The problem presented above can be solved using an expressive MOP supporting cast in addition to message receive, as does the standard MOP provided with Reflex. Also, the expressive power of the selection framework of Core Reflex as well as the activation scheme provided are of great help in this case. This section explains how to design the solution, and the following one shows how to configure Reflex in order to implement it.

Figure 4.16 illustrates the situation at the time `foo` is called (Fig. 4.16(a)), corresponding to the classical scenario of transparent remote calls with a runtime MOP, and once the future has been returned (Fig. 4.16(b)). At this time, the client and the future share a common metaobject, the future metaobject. This metaobject controls both *casts* occurring in the client and *message receptions* occurring in the future. While the result has not been delivered yet (delivery is done by the proxy metaobject, Fig. 4.16(b)), if the future metaobject intercepts either a cast of the future within the client or a message reception on the future, then execution is blocked until the result is available.

Once the result has arrived, two cases have to be considered:

- the result is of type `A`: the metaobject fills the future with all the fields of the result so that the future becomes (a clone of) the result. At this time, controlling message reception on the future is no longer needed, hence the future is deactivated. From now on, the future is a standard object.
- the result is of type `B`: the metaobject keeps a reference to the result. Then, each time it intercepts a message receive on the future, it forwards it reflectively to the result. Later, if it intercepts a cast by the client, then it returns the result instead of the future, and execution proceeds.

4.5.4 Implementation

The first step consists of defining the desired `futureHS` hookset. This hookset is defined as the union of two primitive hooksets (Fig. 4.17):

- `futureReceives` is a set including all calls to public methods of future classes. This set is defined by a simple operation selector selecting all public methods for the message receive operation (`PublicOS`), and a class selector matching all future classes (`FutureCS`).
- `clientCasts` is a set including all casts to a future type (`FutureCastOS`) occurring in any class (`AnyCS`).

We then define the `future` link, linking this composite hookset to the metaobject class `FutureMO`. The scope is set to global, so that only one shared instance of `FutureMO` is created. The metaobject class implements both `ReplaceCast` and `ReplaceMsgReceive`. It holds a table of associations between asynchronous call identifiers and future objects.

```
<hookset id="futureHS">
 <union>
   <hookset id="fReceives"  operation="MsgReceive"
     classSelector="FutureCS"
     operationSelector="PublicOS" />
   <hookset id="clientCasts"  operation="Cast"
     classSelector="AnyCS"
     operationSelector="FutureCastOS" />
   </union>
</hookset>

<link id="future">
 <hookset id="futureHS">
 <metaobjectDefinition class="FutureMO"/>
 <attributes control="replace" scope="global"/>
</link>
```

Figure 4.17 – XML configuration for the future example.

The definition of `FutureCastOS` does not present any particular difficulty. In contrast, the definition of the `FutureCS` is not easy. Recall that the role of this class selector is to select the future classes. A straightforward solution consists of selecting any class that is a possible result type of a public method of a class on which an asynchronous call is performed[3]. Obviously, in such a case, `FutureCS` might end up selecting almost all classes in the system (for instance, if a method has return type `Object`).

We are actually facing a trade-off between transparency and partiality. Fortunately, Reflex makes it possible to choose and implement the appropriate trade-off depending on requirements. The straightforward solution is totally transparent, but it may lead to a heavy use of reflection.

[3]This is a class of *active objects* in ProActive.

A more elaborate alternative can be based on some heuristics or static analysis to determine the future classes. Finally, one can initialize `FutureCS` with a list of future classes names, or make all future classes implement a marker interface. This last alternative is definitely not transparent, but is optimal in terms of partiality of reflection.

4.5.5 Assessment

This example shows a concrete case where controlling the cast operation turns out to be interesting. Furthermore, link activation is very useful here, since we are facing a scenario where beyond a certain point in time reification is not needed any longer.

This example also illustrates the interest of a powerful selection mechanism, as opposed to type patterns in AspectJ, and other purely syntactic-based approaches. Let us consider, for instance, the class selector we mentioned as a possible alternative, which precisely selects those classes that are *the result types of the public methods of the classes on which asynchronous calls are performed*. Such an advanced criterion cannot be expressed using patterns (even compound ones) on type names.

Also, the possibility of adopting various selection strategies (based on syntax, program analysis, introspection, etc.) is a great advantage in terms of flexibility since system designers are not constrained by a particular, closed, way of specifying selection. In the future example, the adopted solution eventually depends on the particular requirements of the target distributed object systems in terms of transparency and partiality.

4.6 Illustration: Runtime Inspection

In this section, we sketch how partial behavioral reflection can be applied to provide an interactive environment for runtime inspection, which, in particular, could be used to assist in reflective and aspect-oriented programming [Tanter & Ebraert, 2003; Ebraert, 2003].

4.6.1 Requirements for runtime inspection

An interactive environment for runtime inspection has several requirements related to software visualization issues [Stasko *et al.*, 1998]. Although the core of this work is not about visualization itself (these aspects would require more research), we believe that at least two important issues deserve early consideration: visual load, and synchronization.

As soon as we are interested in realistic applications, a tough issue in visualization is the control of the *visual load*, that is, the possibility to ensure that not too much information is displayed at a given time, so that the user cognitive charge is not excessive and the user can avoid getting lost. A user should be able to select what exactly is of interest to him and possibly be given the chance to adjust the visualization layout. Another important requirement deals with the *synchronization* between the executing application and the inspection environment. Our approach is based on a synchronous *means of control* [Mehner & Rashid, 2002]. The idea here is to provide the user

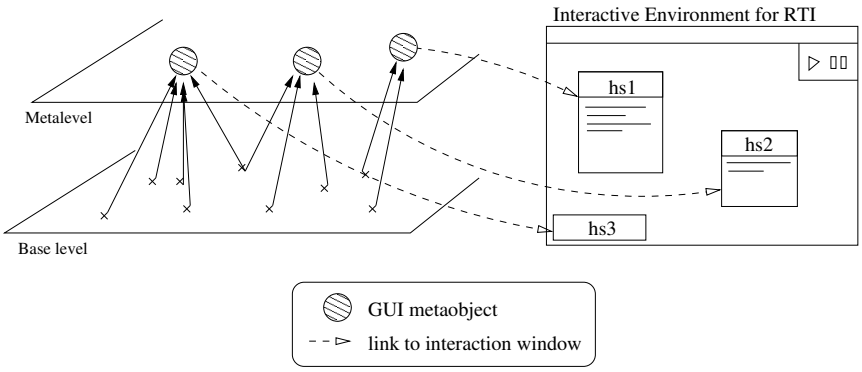

Figure 4.18 – Schema of the interactive environment.

with a feeling of direct interaction with the running application, offering the possibility to suspend execution, to adjust settings or interact with the application, before resuming execution. To sum up, it seems fundamental to provide a very fine-grained control over what is inspected, along with a high-level of interactivity.

4.6.2 Approach to Application Inspection

According to the above-mentioned requirements, we are convinced that our approach to partial behavioral reflection is particularly well-suited to provide an interactive environment for runtime inspection. In order to back up that statement, we have developed a first monitoring tool that uses Reflex for allowing runtime inspection [Ebraert, 2003].

First of all, to specify which parts of a base application a user wants to monitor, Reflex configuration is used to define *hooksets*. In addition to this static configuration, the runtime API of Reflex can be used to control link activation. This makes it possible for the user to precisely control, down to the finest granularity level, which parts of the system and which particular execution points will be observed/manipulated and when. For instance, a user can specify that monitoring a given part of the system becomes interesting whenever some dynamically-evaluated activation condition holds.

Visually, the user gets confronted with a multi-window system where a small control window is attached to every hookset (a *GUI metaobject* is associated to each hookset). This permits the visualization layout to be customized (*e.g.* windows can be minimized, resized, closed and moved). In our prototype, a control window simply offers terminal-like output and basic interaction features:

- inspection of reified operations with a notion of time line: to indeed introspect the base application execution,
- custom speed execution of the base application: to control the amount of information presented to the user per time unit, and

- fine-grained control over hookset activation conditions: to limit the set of displayed events.

For each link, a window is created, responsible for the monitoring of its operations. Depending on the scope attribute of the link, different types of windows are used:

- for a global scope, the window is a simple hookset window that monitors all the operations affected by the hookset of the link.
- for a class scope, the hookset window contains a class window for each of the classes affected by the hookset. Each class window monitors the operations related to the considered hookset occurring within its instances.
- for an object scope, the hookset window also contains a class window for each of the classes affected by the hookset. But in this case, each class window contains an object window monitoring only the operations related to the considered hookset occurring within a particular instance.

Closing a window on a certain level closes all its nested windows and stops the monitoring of that hookset, class or object. This allows visual load to be limited as the user can really select what to monitor. Figure 4.18 shows how the interface looks like. Obviously, further versions could illustrate program dynamics in a more elaborated fashion.

Synchronization between the user and the running application is made at each entry to the metalevel. Whenever a reification occurs, the interactive environment can synchronize with the running application. For instance, if the user asks to suspend the program execution, this suspension will take effect upon the next entrance to the metalevel. Customizing the base application execution speed is also managed this way.

4.6.3 Perspectives for Concerns Inspection

Reflex is a general tool for supporting separation of concerns (SOC) through behavioral reflection. Indeed, it can serve as a generic platform for aspect-oriented development as argued in the first sections of this chapter. In such a case, an application runs at the base level while crosscutting and/or non-functional concerns are implemented modularly as metalevel entities (*SOC metaobjects*).

In order to help with the prototyping, development, and debugging of such a *concern* metalevel, our approach to runtime inspection could provide a valuable support. In such a scenario, the inspection environment actually runs at the meta-metalevel, allowing for the manipulation of the metalevel (Fig 4.19). The interaction with the runtime API of Reflex would then be much more powerful than in standard application inspection, since it would not simply include link activation and basic synchronization features, but would also provide the means to dynamically change the bindings between base execution points and SOC metaobjects. Note that plugging a metalevel on top of an existing metalevel does not raise any problem since metaobjects are implemented with standard Java classes, and are thus also subject to selective reification. This line of work is being pursued, in an effort to study the dynamic evolution of applications [Ebraert & Tanter, 2004].

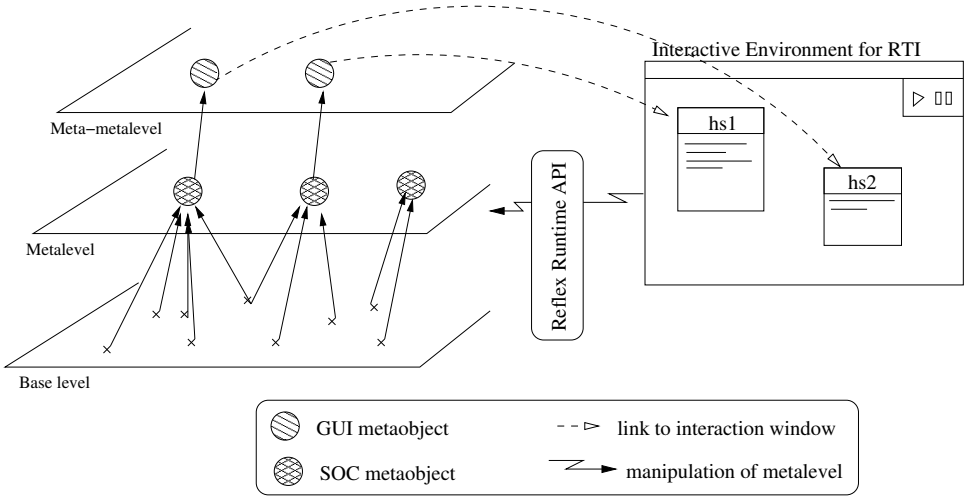

Figure 4.19 – Applying the interactive environment to inspect a *concern* metalevel.

4.7 Discussion

There are several behavioral reflective systems that can be considered to provide some sort of *limited* partial behavioral reflection: Reflective Java [Wu, 1998], Dalang and Kava [Welch & Stroud, 1999], MetaXa [Golm & Kleinöder, 1999], Guaraná [Oliva & Buzato, 1999], and Iguana [Gowing & Cahill, 1996], first in the context of C++, and its Java incarnation, Iguana/J [Redmond & Cahill, 2002]. Apart from Iguana, all these systems only provide the possibility to choose which classes are made reflective. The Iguana approach is the closest to ours in its philosophy. A limitation of Iguana is that it makes it possible to restrict the set of reified operations, but not to extend it. Furthermore, it does not support intra-operation selection, which is a crucial feature of partial behavioral reflection, as presented in this chapter. Iguana/J [Redmond & Cahill, 2002] is more powerful than Reflex with respect to temporal selection, since it does not require hooks to be specified and introduced at load time. Actually, in the range of approaches discussed in [Redmond & Cahill, 2002], Reflex can be characterized as supporting dynamic binding at load time, meaning that bindings (*i.e.* hook introduction) are made at load time and can be undone/redone at run time (*i.e.* changing metaobjects, deactivating links). As for configuration, no other reflective architecture offers the same range of static and dynamic configuration possibilities. The explicit support for the role of metalevel architect is also a distinguishing property of our work. Finally, the mentioned systems all adopt a classical view on the metalink and have not considered the handling of crosscutting metaobjects, which is made possible in Reflex thanks to the model of hooksets.

As regards AOP, general-purpose runtime AOP is a subset of partial behavioral reflection. For instance, Event-based AOP (EAOP) [Douence *et al.*, 2001] can be seen as a particular instantiation of partial behavioral reflection, where all hooks forward control to a unique omnipotent metaobject called the monitor. Building a general-purpose aspect language on top of a generic reflective

system actually makes it easier to provide guarantees in terms of aspect behavior and to lower the complexity of programming. But the issue is then how to develop specific languages on top of such generic infrastructures. Reflective approaches *can* support AOP development using multiple aspect languages. Aspect-Oriented Logic Meta Programming (AOLMP) [De Volder & D'Hondt, 1999; Wuyts, 2001] is a proof of feasibility. AOLMP offers a general declarative framework for programming aspects (the base level can be a Java or Smalltalk program, while the metalevel is programmed in a logic programming language). This framework can be used to declare rules about interactions and compositions of aspects defined in ASLs [Brichau *et al.*, 2002].

Summary

We have presented a comprehensive approach to partial behavioral reflection, including its spatial and temporal dimensions. Then we have described how Reflex supports partial behavioral reflection in an open and portable manner. Reflex seamlessly integrates load-time and runtime behavioral reflection. The main contributions of this work are:

- presentation of a model of partial behavioral reflection, which generalizes the classical view of metalinks, offering support for relations between several execution points and capacity to handle crosscutting concerns,
- full integration of many reflective aspects in a generic model and an open implementation,
- clearly separated specification of hooksets, links and metaobjects, which can still be embedded in one single place,
- intentional, possibly declarative, description of the MOP entry points, using an expressive reification selection framework (class and operation selectors),
- open MOP support, which provides extensibility in terms of supported operations,
- high level of configurability, both at load time and runtime,
- explicit and enhanced support for the roles of assembler and metalevel architect in addition to the role of metaprogrammer.

Finally, we have illustrated the interest of our approach in several settings: design patterns, distributed programming, and runtime inspection. These examples highlight the applicability of partial behavioral reflection, and establish a first connection between our work and Aspect-Oriented Programming.

This connection is further explored in the next part of this dissertation. In the following chapter, we study a mapping of the general-purpose aspect language AspectJ. And then, since we are particularly interested in the possibility of keeping this infrastructure open but still providing its users with structure and guidance as well as guarantees on the resulting programs, we focus on the issue of developing specific languages on top of a generic architecture, introducing the notion of *versatile kernels* for AOP and studying the evolution of Reflex towards such an infrastructure for hybrid AOP.

Part II

...to Versatile Kernels for AOP

Table of Contents

Chapter 5

Partial Behavioral Reflection and AOP

Chapter 4 establishes an initial connection between partial behavioral reflection and aspect-oriented programming. The intuition that partial behavioral reflection may be appropriate to support AOP in an open manner naturally calls for further exploration.

In this chapter, we report on a first experiment to support general-purpose AOP with partial behavioral reflection. More precisely we study the mapping of dynamic crosscutting as supported by AspectJ to our model of partial behavioral reflection. The result of this study, exposed in Section 5.1, is to slightly refine our model in order to support dynamic crosscutting more effectively. In Section 5.2 we illustrate the new features by revisiting the implementation of the observer pattern in Reflex.

5.1 Supporting Dynamic Crosscutting

In order to concretely validate our intuition that partial behavioral reflection is appropriate for supporting AOP approaches in an open manner, we need to carry out consequent case studies of partial behavioral reflection. We have chosen to study the support of dynamic crosscutting in partial behavioral reflection because it is the most notable feature of AOP proposals. Among these, AspectJ [Kiczales *et al.*, 2001] is the most successful: it is a simple, well-designed and production-quality extension to the Java programming language, that allows the modular implementation of crosscutting concerns. The AspectJ language is a general-purpose aspect language, as opposed to domain-specific aspect languages [Shonle *et al.*, 2003; Brichau *et al.*, 2002; Lopes, 1997; Mendhekar *et al.*, 1997]. This section reports on the mapping of AspectJ's dynamic crosscutting mechanism. It is based on [Rodríguez *et al.*, 2004]. First, we give an overview of AspectJ in Section 5.1.1, before explaining how to support its dynamic crosscutting features with Reflex 5.1.2. This initial approach leads us to identify two enhancements to our model. In Section 5.1.3, we revisit the mapping with these enhancements. Finally, we report on benchmarks comparing the performance of standard AspectJ and of the AspectJ translator based on Reflex (Section 5.1.4).

5.1.1 AspectJ

AspectJ extends the Java language with a new unit of modularity, *aspects*, to implement crosscutting concerns modularly. AspectJ supports two kinds of crosscutting: *dynamic crosscutting* makes it possible to define additional behavior to run at certain well-defined points in the execution of the program; *static crosscutting* makes it possible to modify the static structure of a program (*e.g.* adding new methods, implementing new interfaces, modifying the class hierarchy)[1]. This study is concerned with the most distinguishing mechanism of AspectJ: dynamic crosscutting. AspectJ follows the Pointcut and Advice model for AOP presented in Chapter 2, Section 2.6.2.2.

5.1.1.1 AspectJ basics

In AspectJ a join point represents a well-defined point in the execution of a program, where program behavior can be extended with a crosscutting behavior. AspectJ supports different kinds of join points, which correspond to different operations of the underlying language, Java: method call, field set, handler execution, etc. Since a join point is an execution point, it has a static counterpart at the code level, called the join point *shadow* [Masuhara *et al.*, 2003]. A join point may further be discriminated by a dynamically-evaluated condition, called the join point *residue* [Hilsdale & Hugunin, 2004], in order to determine whether a runtime occurrence of the join point shadow actually is an expected join point.

The AspectJ language provides the means to group join points of interest into a *pointcut*, in order to specify the places where an aspect actually affects a base application. A pointcut definition may also specify the *context information* that should be passed to the aspect (*e.g.* the arguments of the current join point). Pointcuts are specified using several primitive *pointcut designators* (PCDs) which can be combined using the standard logic operators. For instance, the following AspectJ code:

```
pointcut move(int x, int y):
    call(* Point.moveXY(int,int)) && args(x,y);
```

defines a pointcut named `move` that combines two primitive PCDs in order to select all calls to method `moveXY` of class `Point`, and expose both method parameters as context information.

Finally, the crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice*. An advice is a method-like construction that defines the additional behavior to execute at certain join points. When defining an advice, one must explicitly bind it to a pointcut. There are five *kinds* of advice, which differentiate the moment at which the advice is executed with respect to the join point execution: before (before the join point execution), after (after the join point execution), after throwing (after the join point execution, returning with an exception), after returning (after the join point execution, returning normally), around (replace the join point execution). An around advice can include a special `proceed` statement to trigger the execution of the join point it replaces. Advices may

[1] The terminology of static and dynamic crosscutting was introduced in [Kiczales *et al.*, 2001]; we could alternatively use the terms structural and behavioral crosscutting.

have parameters, in which case they must be bound to the pointcut context exposure parameters. For instance, the following AspectJ advice:

```
void around(int x,int y): move(x,y){
  proceed(max(0,x), max(0,y));
}
```

simply ensures that a point cannot be moved to negative values of `x` or `y`.

5.1.1.2 Properties of PCDs

Pointcut designators in AspectJ do not all have the same properties. They can be characterized based on the following properties[2]:

- *statically matched*: some PCDs can be resolved completely by looking at the program text. Such PCDs may express:
  - a *kind restriction*: match only certain kinds of join point, for instance `call`, which matches method or constructor calls (caller side), or `execution`, which matches effective method or constructor executions (callee side).
  - a *signature restriction*: restrict join points based on their signatures. AspectJ offers wildcarding in signatures for convenience; for instance, `call(* *.move())` restrict join points of kind `call` to calls to a no-arg `move` method.
  - *location restriction*: restrict join points based on the location in source code where they occur. Examples are `within` and `withincode`.
- *dynamically matched*: such PCDs require runtime information to determine whether they match a candidate join point or not. Such checks are implemented by dynamically-evaluated conditions, called *residues*. AspectJ supports three types of residues: *control flow* residues for expressing control-flow based crosscutting, *instanceOf* residues for runtime type checking, and *if* residues for evaluating arbitrary (though `static`) boolean expressions.
- *context exposure*: such PCDs expose join point information to the context. Examples are `arg` and `this`.

5.1.2 Mapping Dynamic Crosscutting

In this section, we present how partial behavioral reflection can support the dynamic crosscutting mechanism of AspectJ. The presentation is informal, example-based, and does not enter into details. It just explains how such a mapping is achieved, in order to make clear some limitations of the current model. We end this section by presenting two minor extensions to the current model of partial behavioral reflection, which lead us to a much better mapping, presented in more details in Section 5.1.3.

[2] A more exhaustive presentation of pointcut designators can be found in [Kiczales *et al.*, 2001] and on the AspectJ website [AspectJ Website, 2002].

5.1.2.1 Scope of the mapping

We hereby only focus on the dynamic crosscutting mechanism of AspectJ. In order to keep the argumentation clear and concise, static crosscutting is not considered. Concerning dynamic crosscutting, we limit ourselves to the main concepts –join points, pointcuts and advices–, and do not address more advanced features such as aspect instantiation, privileged aspects and so on.

5.1.2.2 Running example

The mapping is presented using a simple *shape editor system* (Fig. 5.1). The system manages three kinds of shapes: `Line`, `Point` and `Composite`. A `Line` has two edges points. `Composite` is a shape container. All of them are subclasses of `Shape`, which is an abstract class with one method, `moveXY`. This method is meant to move the center of a concrete shape to the specified coordinates, therefore, for `Point` it moves the point to the new coordinates, for `Line` it moves the middle point of the line to the new coordinates, consequently moving its edges, and for `Composite` it move the center of the overall shape to the new coordinates, consequently moving all its inner shapes.

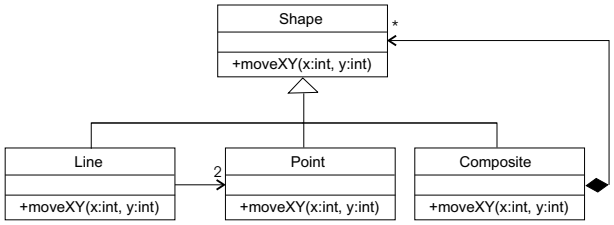

Figure 5.1 – Simple shape editor system

5.1.2.3 Initial approach

In this section we present an overview of the mapping of AspectJ dynamic crosscutting to the model of partial behavioral reflection as presented in Chapter 4. Conceptually, an aspect is a metalevel entity, since its primary subject matter is to affect the execution semantics of another program.

A first possible mapping of AspectJ on top of behavioral reflection would consist in reifying whatever occurs in the base program (similarly to a debugging mode) and pass it to some “aspect controller” (a global metaobject) that will dynamically determine which pointcuts match, and execute the corresponding advices. Needless to say, such an approach would be far from satisfying in terms of efficiency.

Rather than entirely matching pointcuts dynamically, partial behavioral reflection gives room for a more staged approach whereby the static projections of pointcuts (their shadows) are represented as hooksets. Hence only execution points of potential interest are reified. Links are

configured to delegate control to the appropriate metaobject when needed. The metaobject is then in charge of completing pointcut matching and, if appropriate, executing the advice. This section gives more details on this approach, evaluates it, and concludes with a suggestion of how to improve the model of partial behavioral reflection.

Pointcuts

As seen in Section 5.1.1.2, pointcut designators may impose statically and dynamically matched restrictions over join points. The main matter of pointcut mapping lies in how such restrictions are expressed within Reflex.

Let us first consider static restrictions, with the following simple user-defined pointcut designator, `move`:

```
pointcut move(): call(* *.moveXY(int, int));
```

This pointcut designator uses AspectJ wildcard mechanism: it refers to any invocation of a method `moveXY(int,int)` occurring in any class. In Reflex, mapping the `move` PCD is done simply by defining a primitive hookset characterized by:

- the `MsgSend` operation class;
- a class selector that selects all classes;
- an operation selector that selects only message send occurrences for a method named `moveXY` that receives two `int` arguments.

Note that the `call` PCD can entirely be matched statically, based on program text. In other words, it is fully determined by its shadow. In Reflex vocabulary, a join point shadow is a hook. By extension, a pointcut shadow is a hookset. Hence, mapping PCDs that are statically determined is straightforward in Reflex.

Now consider the following extension to the previous pointcut designator:

```
movePoint(): call(* *.moveXY(int, int)) && target(Point);
```

This pointcut adds the restriction that the target object of a call to `moveXY` be of type `Point`. Such a restriction cannot be completely resolved statically (if it can) without using an expensive static analysis of concrete types. In program text, it is only possible to select calls that are done on an object whose declared type is either `Point`, or any super and subclass of it. Since the concrete type of a variable is only determined at runtime, this pointcut requires a residue that dynamically checks the type of the target of the call (via `instanceof`). In Reflex, residues do not have a direct counterpart[3]. However, they can be handled as an extra condition checked at the metalevel. When a reification occurs, the context information is all passed to the metaobject controlling the considered hookset. Such a metaobject starts by checking the residue condition before continuing.

[3]Only activation conditions are available, which are not meant to reason about context information other than the currently executing object (Section 4.3.1.4).

Residues are all checked in the same way. However, a control flow residue deserves special attention. The pointcut designators related to control flow (`cflow` and `cflowbelow`) allow picking out join points based on whether they are in a particular control-flow relationship with other join points. Checking this relationship represents nothing new compared to other residues, however, setting up this relationship implies exposing the control flow information of other join points.

Let us discuss this with an example:

```
moveSinglePoint(): movePoint() && !cflowbelow(move());
```

This PCD further restricts the `movePoint` PCD, matching only those invocations that are not made below the control flow of any join point matched by the `move` PCD. In other words, it matches only the moving of a stand-alone point (top-level calls). Note that this pointcut definition implies the definition of two nested pointcuts: the one inside the `cflowbelow` (`move`) and the one affected by a control flow restriction (`movePoint`).

To be able to determine whether `movePoint` is matched in the control flow of `move`, we first need to expose the control flow information of `move`: this is done using the notion of *event collectors* [Tanter & Noyé, 2004b]. Event collectors gather execution events to expose parts of a program execution (nesting, sequences, etc.), under any structure (counter, stack, tree, DAGs, graphs, etc.), for dynamic introspection. In particular they are used to expose control flow information. Indeed, event collectors are just like metaobjects, except that their purpose is only to *expose* elements of program execution, rather than to *affect* program execution. In order to support `cflow` and `cflowbelow`, exposing a simple counter that keeps track of entries and exits in a pointcut suffices. Consequently, mapping a control flow PCD implies two separate tasks:

- defining a separate link for an event collector exposing control flow information of the pointcut passed as argument to `cflowbelow`;
- defining a condition that checks the exposed control flow information.

Hence, in our example, the metaobject must check two residues: the control flow condition, and the instanceof condition.

Pointcut parameters

Consider the following extension of the pointcut designator `moveSinglePoint`:

```
moveSinglePointArg(int x, int y): moveSinglePoint() && args(x,y);
```

This pointcut designator extends the previous example by exposing the arguments of the invocation of method `moveXY(int,int)` in `Point`. These arguments will then be available when defining an advice bound to this pointcut.

In Reflex, the information that is reified is defined at the operation level: when support for a particular operation (*e.g.* message sending) is added to Reflex, an entity responsible for the

low-level code transformation is given. Such an entity (called a hook installer, Section 4.3.1.2) determines which information is reified and how. This unfortunate coupling implies that we either need to define a new operation support each time a different piece of context information is needed, or always reify all context information and let the metaobject extract the needed pieces.

Advices

Since an aspect may contain any legal Java class member, it must be mapped to an ordinary class, following the singleton pattern. It could be mapped to a metaobject class, however, due to the extra work we leave in metaobjects (checking residues, extracting context information), it is preferable to get a better separation and stick to a clean aspect class.

Advices are transformed into methods of the aspect class. Its name is generated by the translator and its arguments are the same as in the advice definition. For instance:

```
aspect MovingPoint {
  // -- pointcut definitions --

  after(int x, int y): moveSinglePointArg(x,y){
    log.println("Point moved to: " + x + "," + y);
} }
```

This logging aspect is transformed into a class with the same name, with a method corresponding to the advice (Fig. 5.2). Besides, the pointcut `moveSinglePointArg` results in one hookset `hs-move` (since the shadow of this pointcut is defined by the pointcut `move`) and the metaobject `moveSPA`. The associated metaobject is responsible for checking the residues (for `cflowbelow` and `target`), collecting the values of the arguments, and finally invoking the aspect advice method. The binding between the metaobject and the hookset is done through a link. The kind of the advice (before, after, around)[4] is mapped to the control attribute of the link (in this case, `after`).

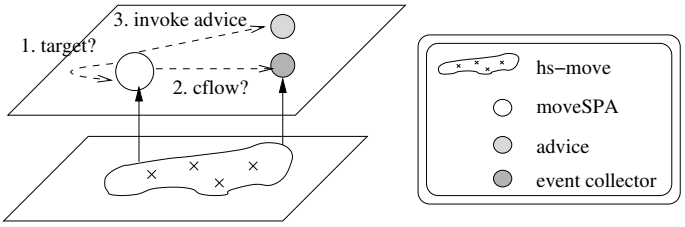

Figure 5.2 – Initial mapping of the aspect `MovingPoint`

Since `moveSinglePointArg` is affected by a control flow restriction over the pointcut `move`, an additional link is required: this link binds the same hookset `hs-move` to an event

[4]Reflex does not yet support *after throwing*, although there are plans to support it in a near future.

collector metaobject. Such a metaobject is a simple counter that keeps track of entries and exits in the hookset. Therefore the link control is set to `before_after`.

5.1.2.4 Issues of the Mapping

Our initial approach shows that the model of partial behavioral reflection as presented in Chapter 4 is expressive enough to handle the dynamic crosscutting mechanism of AspectJ in a non-naive manner (conversely to the all-dynamic approach consisting of a centralized pointcut matching process occurring at runtime). However, this experiment also highlights interesting issues that could be resolved to achieve a better mapping.

First of all, pointcuts are not handled as efficiently as they could, since we are not able to embed residues in the code: they need to be checked at the metalevel. This has the bad property of forcing reification even in cases where the residues reject a particular occurrence. And reification is a major source of overhead in reflective systems.

Furthermore, the handling of selective pointcut parameter exposure is not satisfactory: one of the alternatives mentioned in section 5.1.2.3 implies reifying all available pieces of information, which is highly inefficient, and the other one implies defining new operation supports each time a different piece of information is required. Recall from Chapter 4 that an operation support is made up of an operation class and a hook installer. This last alternative is efficient since hook installers determine the required pieces of information, but would be really cumbersome in practice: it might result in a myriad of defined operation supports for what is conceptually the same operation (which just happens to be reified differently).

Finally, these two limitations forced us to introduce an extra indirection from metaobjects to aspect objects, in order to cope with them. In the next section we introduce two extensions to the model of partial behavioral reflection that make it possible to circumvent these limitations. We aim at a mapping that is both efficient and clear. Clarity would be obtained by having metaobjects simply implement advices, embedding other concerns within hooksets and links.

5.1.2.5 Two model extensions

In order to solve the issues presented above, we propose two extensions to our model of partial behavioral reflection:

MOP descriptors – to describe, *at the link level* (as opposed to the operation level), which information has to be reified, and how, upon occurrences of a given operation;

hookset restrictions – to embed dynamically-evaluated, but fixed, conditions in generated hook code.

MOP descriptors

A MOP descriptor is an object that describes how an operation should be reified: it defines the expected type of the metaobject as well as the method to invoke. This description is completed

by a specification of the parameters that should be passed to the metaobject method. The metaobject class should obviously be compatible with the specified type and implement the method corresponding to the given name and parameters.

Parameters are simply objects that implement a dedicated interface, `Parameter`:

```
interface Parameter {
  public String getCode(Operation aOp);
}
```

The role of a parameter object is to generate the source code that, when executed, results in the reference to the desired information. Standard parameters are provided, such as `CONTEXT`, which refers to the currently executing object (or class if within a static member). Another example is a parameter that resolves to the reference to the metaobject associated to a given link. Operation-specific parameters are also provided by hook installers via *parameter pools*. For example, for the `MsgReceive` operation, the parameter pool makes it possible to obtain a parameter object for a method object, or a parameter (at a given index) of the invocation. Finally, custom parameters can be specified. The user then needs to give the source code, in the extended Java language supported by Javassist [Chiba & Nishizawa, 2003], that should be evaluated.

A MOP descriptor makes it possible to specify whether a set of parameters should be passed as plain arguments to the metaobject, packed into an object array, or encapsulated into some object, instance of a user-defined class.

Since MOP descriptors completely decouple the specification of the MOP from the capability of reifying an operation (implemented in hook installers), it is now possible to use different MOP descriptors for the same operation in different links, which greatly simplifies the efficient mapping of the pointcut context exposure feature of AspectJ.

Hookset restrictions

A hookset restriction is a dynamically-evaluated condition that should be true in order for a hook to trigger reification and metaobject invocation. Such a restriction is hardwired in the hook code at generation time to improve performance. Hookset restrictions are specified when creating a link. In case the hookset bound to a given link is compound, it is possible to set a restriction that applies to all or only some of the sub-hooksets.

The restriction is specified as a static method and hence can be computed based on globally-available information (*i.e.* static fields or methods), as well as parameters if needed. Parameters are specified when declaring the restriction in a similar manner as for MOP descriptors. The list of parameters must be compatible with the signature of the restriction method.

For example, pointcut `movePoint` implied a restriction that the target object of a `moveXY` call be of type `Point`. This pointcut introduces the need for an `instanceOf` residue, which can be specified as the following restriction, to which the parameter `CONTEXT` is given:

```
public static boolean accept(Object o){
  return o instanceof Point;
}
```

When attached to the hookset selecting calls to `moveXY` this restriction discards, at runtime, calls to objects which are not of a subtype of `Point`.

5.1.3 Revisiting the Mapping

In this section we revisit the mapping of the dynamic crosscutting mechanism of AspectJ to Reflex, taking into account the two extensions to the model of partial behavioral reflection presented in Section 5.1.2.5. This section illustrates that these extensions allow us to greatly improve the quality of the mapping, both in terms of efficiency and clarity. Besides explaining how such extensions do enhance the mapping, we give a more detailed presentation of the involved translation.

5.1.3.1 Pointcuts

Thanks to the extensions presented above, we are able to completely map a pointcut declaration without having to postpone extra responsibilities to the metaobject. In other words, the result of translating a pointcut is a (possibly composite) hookset, along with hookset restrictions and appropriate information for the MOP descriptor of the link. We hereby give an overview of the pointcut translation process.

The translation process

Once parsed, a pointcut is represented by a tree with PCDs as leaves, and logical operators as nodes. Our translation algorithm is pretty straightforward. The first step consists of building an intermediate isomorphic tree in which each leaf is replaced by a quadruple. Such a quadruple is generated based on the properties of the PCD (see Section 5.1.1.2). It is of the form (PH, SR, DR, CE), where:

- PH represents a primitive hookset. PH is non-empty only if the given PCD restricts the join point kind, such as `call`, in which case PH holds the operation that corresponds to the PCD kind. If the PCD is further restricted by a signature pattern, such a pattern is mapped to class and operation selectors.
- SR is a static restriction expression. SR is non-empty only if the given PCD serves for stating location or signature restrictions, such as `within`.
- DR is a dynamic restriction expression that corresponds to the residues of a PCD, if any. Recall that PCDs such as `this` and `cflow` do require residues, whereas `call` does not.
- CE holds the list of context parameters exposed by the PCD, if any.

The second step of the translation algorithm is to reduce the tree as much as possible, before converting it to elements of Reflex. This reduction is done by applying a set of reduction rules that basically discard nodes and compose the quadruples accordingly. The principle followed by the reduction process is simple: eliminate all the nodes with `!` or `&&` operators. To this end,

reduction rules are applied in a given order, and most of them are straightforward; for instance, "in case at least one child of an && operator is an || operator, perform distributive composition of quadruples", or, "the negation of a quadruple reduces to the negation of all its components".

Distributive composition of quadruples is done as follows: PHs that relate to the same operation are composed together via primitive hookset composition operators (union, intersection, etc.); SRs, as well as DRs, are composed together using the usual logical operators; and CEs are composed together by simply merging their lists. Negating PHs implies negating corresponding class and operation selectors, while negating SRs and DRs simply means negating the expressions (or nothing if empty). Once the reduction rules have been applied the PH is composed with the SR by merging the additional static restriction expression in SR into additional class and/or operation selectors in PH.

If the resulting tree is a single leaf, then it represents a single primitive hookset. Otherwise, the tree represents a composite hookset: all nodes are union operators and leaves contain the different primitive hooksets with their associated dynamic restrictions and context exposure requirements. To finally define the pointcut in Reflex, the resulting (composite) hookset is created and defined. When defining the link, for each sub-hookset, the hookset restrictions are specified and the MOP descriptor is configured to reify the required parameters.

Illustration

Let us explain how the pointcut `movePoint`, presented previously, is translated. First, it is represented by a tree with one node (the logical &&) and two leaves: one for the `call` PCD, which embeds its signature restriction and one for the `target` PCD, with its type restriction.

The `call` PCD is replaced by the quadruple *({ph}, {}, {}, {})* where *ph* is the primitive hookset described in Section 5.1.2.3 for the pointcut `move`. The `target` PCD expresses a restriction that is partially matched statically (the target type must be either `Point` or a subtype or supertype of `Point`). It must be completed by a dynamic restriction (`instanceOf(Point)`). Hence, this PCD is replaced by a quadruple *({}, {sr}, {dr}, {})*, where *sr* and *dr* represent the static and dynamic restrictions respectively. Note that since this PCD does not impose any kind restriction, the PH component of the quadruple is empty.

This tree is simply reduced by composing both quadruples, resulting in a quadruple *({ph}, {sr}, {dr}, {})*. To finish, *sr* is composed with *ph*.

Now, consider the pointcut `moveSinglePoint`, which adds a control flow restriction to `movePoint`. The `cflowbelow` PCD is replaced by a quadruple *({}, {}, {dr'}, {})*, where *dr'* expresses the control flow condition to be dynamically checked. In addition, the translation process triggers the translation of the internal pointcut `move`, in order to generate the required control flow information (via an event collector, recall Section 5.1.2.3). The reduction implies negating the `cflowbelow` quadruple (thus negating *dr'*) and composing it with the `movePoint` quadruple. The resulting dynamic restriction would then be *dr && !dr'*, which would be implemented by the following hookset restriction:

```
public static boolean accept(Shape s){
  return (s instanceof Point) && !(MoveCflow.getCounter() > 1);
```

```
}
```

Evaluation

Using hookset restrictions and MOP descriptors, we do not need anymore intermediate metaobjects between the cut and the aspect behavior: metaobjects can simply be objects that implement advices.

Indeed, using hookset restrictions to implement residues avoids the need for checking residues at the intermediate metaobject level. Using MOP descriptors to specify which information should be reified frees the intermediate metaobject of the task of collecting parameters before calling the associated behavior. Furthermore, since a MOP descriptor also specifies which method to call on the metaobject, we can directly invoke the appropriate advice method.

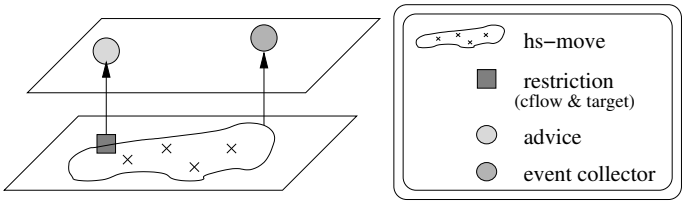

Figure 5.3 – Revised mapping of the aspect `MovingPoint`

Hence, the two extensions to the model of partial behavioral reflection not only enhance performance significantly, they also greatly simplify the overall picture of the mapping: pointcuts are mapped to hooksets and restrictions, advices to metaobject methods, and binding specification is handled in the link (context exposure, advice kind, etc.). Fig. 5.3 shows the revised mapping of the aspect `MovingPoint` (to be compared with Fig. 5.2.

5.1.3.2 Advices

We are now able to simply map an advice to a metaobject method that receives the advice parameters. Apart from this simplification, the mapping principle remains the same as exposed in the previous section. Nevertheless, we now discuss some advanced issues with advices that we ignored in the previous section. Advice bodies in AspectJ have two special features that make them different from simple Java methods: `proceed` statements, and access to reflective information about a join point.

Proceed statement

An around advice traps the execution of a join point and runs instead of it. The original computation of the join point can be invoked through a special `proceed` statement, which acts as a method call. It accepts the same exposed parameters of the original join point, and returns what

the original join point returns. The arguments of the `proceed` may actually replace the original values (Section 5.1.1).

In Reflex, a standard supported operation has a corresponding *dynamic operation* class (Section 4.3.1.2). A dynamic operation encapsulates runtime information describing an operation occurrence, and also provides a `perform` method that actually executes the intercepted operation occurrence.

As seen in Section 5.1.3.1, MOP descriptors can be configured to specify the parameters required by each quadruple. In case of an around advice, we specify an extra implicit parameter which is a command object [Gamma *et al.*, 1994] wrapping a dynamic operation. The command object embeds operation-specific logic related to mapping `proceed` parameters to that of the dynamic operation. All `proceed` statements in an advice body are subsequently replaced by execution of the command object.

Join point reflective information

AspectJ provides three implicit references that provide reflective information of the current join point. They can be used from the body of any advice or from an `if` PCD. These three implicit references are:

- `thisJoinPoint`: provides reflective information about the static part (*i.e.* the signature of the join point) and the dynamic part (*e.g.* the values of the arguments) of the current join point.
- `thisJoinPointStaticPart` is a subset of the `thisJoinPoint`, which only provides reflective information about the static part.
- `thisEnclosingJoinPointStaticPart` provides reflective information about the static part of the *enclosing* join point, that is to say, the method (or constructor, or static initializer) in which the current join point occurred.

These instances essentially collect all the current join point context information, plus the reflective static information describing the actual signature of the join point and expose it by implementing a set of interfaces defined by AspectJ.

As seen in Section 5.1.2.5, parameters specified in MOP descriptors give access to context information of a hook, thus easily handling the dynamic part of `thisJoinPoint`. In addition to this, when parameters are resolved at generation time, they have access to the static information of operation occurrences, a direct counterpart of `thisJoinPointStaticPart`. Furthermore, this information includes information about where the occurrence is located, thereby making it possible to compute `thisEnclosingJoinPointStaticPart`.

Therefore, providing access to these variables in the context of the mapping simply implies providing (1) a set of classes that implement the interfaces defined by AspectJ, and (2) three parameters (one for each instance) that instantiate such classes with the necessary information. These parameters are passed as implicit parameters to the advice body, exactly as in done for AspectJ as described in [Hilsdale & Hugunin, 2004], without having to transform advice bodies.

Features	*Scenario*	*AJC*	*AJT*	Δ
before	**(w/o context)**			
no residue		1542	1705	10%
instanceOf	match	1185	1305	10%
	no match	868	894	2%
cflow	match	841	951	13%
	no match	998	1218	22%
before	**(w/ context)**			
no residue		4533	4616	1%
instanceOf	match	3241	3034	-6%
	no match	884	904	2%
cflow	match	10295	11102	7%
	no match	697	647	-7%
around	**(w/ context)**			
around	always proceed	3404	5721	68%
	half proceed	4416	6349	43%
	never proceed	5711	6990	22%

Table 5.1 – Execution time (in ms) and overhead of the AspectJ translator (AJT) vs. standard AspectJ (AJC).

5.1.4 Benchmarks

The AspectJ translator for Reflex translates AspectJ source code off-line. Interestingly, the translator is implemented by using Reflex to transparently change the behavior of the standard AspectJ compiler so that the intermediate representation is transformed to Reflex configuration.

We report on the performance of programs compiled using the AspectJ compiler (AJC) `ajc` 1.2 and programs translated by the AspectJ Translator (AJT). The benchmarks were run on a Pentium III 1.1GHz with 512MB of memory, running Windows XP and Java 1.4.2_05 (HotSpot client VM). We allocated a large heap size in order to limit the number of garbage collections. We benchmarked specific features of AspectJ's dynamic crosscutting in programs based on the simple shape editor example used in this chapter, applying a logging aspect in different scenarios (Table 5.1). The first two sets of test cases consist in applying a before advice with and without context information, respectively. For each set, three scenarios are considered: there is no residue, there is an instanceOf residue, there is a cflow residue. When using residues, scenarios where residues match and do not match are measured. The last set of test cases is based on the use of an around advice, to which a composite shape and two integers are passed as parameters, in three different scenarios: *1)* the advice always calls `proceed`; *2)* the advice calls `proceed` half of the time, otherwise it reorders the shapes contained in the given composite shape; *3)* `proceed` is never called, the advice always performs the reordering.

The results were obtained by performing five measurements for each scenario, discarding the best and worst cases and taking the average of the remaining three measurements. The third and fourth columns in Table 5.1 show the average execution time using AJC and AJT, respectively. The last column shows the relative overhead of AJT.

The test cases related to the before advice show a very reasonable overhead of AJT (less than 10%), in particular when considering that our implementation is more prototypical than production quality. Test cases with context exposure demonstrate particularly good performance of AJT, even slightly better than AJC in some cases. The most important overhead (up to 22%) in the case of cflow without context seems to be due to better HotSpot optimizations for AJC, since AJT shows much better performance when running these scenarios in interpreted mode (up to 29% better than AJC).

Execution of the around advice however shows a significant (though still acceptable) overhead. This overhead is not mainly due to advice inlining done by AJC, since it is not significantly reduced when running the scenarios without inlining. This strongly suggests that the major cause of inefficiency is the use of standard reflective method invocations to implement `proceed`, whereas AJC generates specific stubs for each join point shadow [Hilsdale & Hugunin, 2004]. These results hence strongly motivate us to develop a similar implementation strategy for `proceed`.

5.1.5 Conclusion

Not surprisingly, partial behavioral reflection is *conceptually* expressive enough to support the kind of dynamic crosscutting offered by AspectJ. This is not surprising because, conceptually, any form of behavioral reflection can cover dynamic crosscutting. The real challenge is to do so practically, in a natural and direct manner. Partial behavioral reflection as presented in Chapter 4 was already a progress in this sense over more classical approaches to behavioral reflection. Still, from the critical study of a first experiment, we have identified and validated two small extensions to our model (hookset restrictions and MOP descriptors) that allow for a clearer and more effective mapping. Benchmarks validate the interest of our approach.

5.2 Illustration: Observer Pattern Revisited and Extended

We now illustrate the interest of MOP descriptors and restrictions through examples based on the observer pattern.

5.2.1 Simplifying Observers with MOP Descriptors

In Section 4.4, we presented how to implement an observer pattern using Reflex. Recall that, in particular, an `Observer` metaobject represents an observation relation between a subject object and observer objects. The pattern was abstracted to some extent in the `Observer` metaobject class. Concrete observers that depend on a particular observation relation, like `ColorObserver` (Fig. 4.13), need to define the following:

- the places where the `updateObservers` method of `Observer` is called, for instance:

```
public void afterFieldWrite(Object[] data){
  updateObservers(data[0]);
}
```

- the update action to perform for each observer, for instance:

```
protected void updateObserver(Object aObserver, Object a Subject){
  ((Screen) aObserver).display("Screen updated - color changed.");
}
```

The definition of the action to undertake in case of an update is something concrete observers should really define: this is the specific part that cannot be abstracted. Conversely, defining the way `updateObservers` is called is definitely not very convenient. Indeed, the fact that updates are triggered upon write access to certain fields is defined in the link (more precisely, in the hookset bound by the link). The method `afterFieldWrite` above just plays the role of an adaptor between the standard MOP and the specific MOP we would like: it takes the generic set of parameters as defined in the standard MOP (`data`), extracts the information from it (`data[0]`), and then calls the required method (`updateObservers`). The observer metaobject is hence completely coupled to the way standard reification is done and implemented.

Using MOP descriptors, we can get rid of this dependency in a very straightforward manner. When defining the link, one can specify the metaobject method to call along with the desired parameters. In the case of an observation relation:

```
link.setMOCall("Observer", "updateObservers", Parameter.CONTEXT);
```

This simple line specifies that for the observation link, the metaobject method to be called should be `updateObservers` (declared by type `Observer`), passing it only one parameter: the currently executing object (`CONTEXT` is a predefined parameter that resolves either to `this` if reification occurs within an object, or to a class object if it does within a class).

Therefore, thanks to MOP descriptors, defining concrete subclasses of `Observer` is reduced to the task of defining the action that should be undertaken, by overriding `updateObserver`. No explicit knowledge of the underlying reification is necessary. Apart from this design advantage, efficiency is improved since arrays are not created anymore at each update: only the reference to the executing object is given upon reification. The flexibility of having MOP descriptors at the link level (or even more, at the hookset level) makes it possible to avoid redefining a whole new operation support for this case: the standard library of Reflex suffices.

5.2.2 Restrictions and Activation Conditions

We now illustrate the use of restrictions and then discuss the difference with activation conditions. To simplify the discussion, we consider a variant of the observer pattern as described above that only describes relations with one observer. This pattern is abstracted in the metaobject class `SingleObserver`. An instance of single observer is initialized with a reference to both its subject and observer, therefore the method to call upon update (defined as abstract) is: `update()`. The description of the metaobject call is therefore:

```
link.setMOCall("SingleObserver", "update", Parameter.NONE);
```

In the present scenario, we define a relation observer between two *valued* objects. A valued object is an object that has a value property (accessed through `getValue` and `setValue`). For

instance, a Swing slider is a valued object: it is a typical GUI component that users can grab and move along a scale (for instance, to control the volume of speakers). The `ValueObserver` maintains a copy (`cpy`) with the same value as a source (`src`):

```
protected void update(){
  cpy.setValue(src.getValue());
}
```

When the value of `src` is changed, the value of `cpy` is updated. Using sliders, this will result in the copy slider following the source slider: the `update` method is called upon each refresh of the display. Considering another example, if the copy object is a persistent object, each time the source changes, the new value is made persistent. In many cases, one would actually like the update not to take place if the value of the source is being *adjusted*. This is why sliders in Swing have a method `isSliderValueAdjusting` that makes it possible to discriminate when the value is to be considered so transient that updating should not occur. We therefore consider that valued objects also have a method `isAdjusting`, and explain how to avoid updates to be done when the source value is adjusting.

There are three means of doing so in Reflex. First, the metaobject itself can check if the value is adjusting in the `update` method:

```
protected void update(){
  if(!src.isAdjusting()) cpy.setValue(src.getValue());
}
```

This alternative has two disadvantages: it complicates the update code and does not avoid reification when unnecessary (although in this particular example, no information is reified). An alternative possibility in Reflex is to use an activation condition bound to the link:

```
link.setActive(new Active(){
  public boolean evaluate(Object o){
    return !((ValuedObject) o).isAdjusting();
  }});
```

Finally, this condition can be defined as a restriction (we use an alternative means of specifying restrictions, with a plain string, rather than a real class):

```
link.addRestriction(
     "(ValuedObject src){ return ! src.isAdjusting(); }",
     Parameter.CONTEXT);
```

The difference between a restriction and an activation condition is that a restriction is embedded within base code, and therefore cannot be manipulated at runtime. Conversely, an activation condition is an object referenced by a base object, that can be accessed dynamically. Actually, in the version of Reflex described in Chapter 4, activation conditions have a well-defined interface, which solely takes one single parameter: the currently executing object. This is because activation conditions were first designed to support per-instance entity selection (see Chapter 4, Section 4.3.1.4). Hence their goal was to select among instances of a reflective class which instances

are effectively reflective. We have evolved this design choice so that, by default, activation conditions are still considered to accept only one parameter, but they are now optionally also subject to the same parameter exposure mechanism than MOP descriptors, like restrictions (first discussed in Section 5.1.2.5). In the restriction code above, a parameter object is used to specify how `src` should be obtained. Recall that the parameter mechanism of Reflex is open: a parameter object simply evaluates at load time to source code that, at runtime, evaluates to the desired reference.

Therefore, the main difference between a restriction and an activation condition is a matter of *binding time*: while both are dynamically evaluated, a restriction is statically bound, while an activation condition is dynamically bound. As a consequence, if a condition is not meant to be manipulated nor changed dynamically, it is best defined as a restriction. Conversely, if it needs to be accessed, then an activation condition is needed. For instance, in order to illustrate the present discussion, we have built a graphical example application with sliders as valued objects. In this application, users can select whether the copy slider should follow the source slider along, or move only when the source slider is released by the user. To support this kind of runtime interaction, an activation condition is used. We shall come back to this comparison in Chapter 7.

Summary

In this chapter, we have exposed how partial behavioral reflection can support dynamic crosscutting of AOP by studying the mapping of AspectJ's dynamic crosscutting mechanism in Reflex. As a result of this study, we have introduced two small extensions to our model that make it possible to support AOP effectively. We have then illustrated the interest of these extensions by revisiting the implementation of the observer design pattern. Our model is now complete enough to support general-purpose AOP in an open manner, at least for the behavioral part.

From this work, we can identify directions to further explore. First, although Reflex is highly expressive, open, and customizable, day-to-day programming with its APIs is not really user-friendly. As a matter of fact, until now, the only efforts focused on the configuration of Reflex have resulted in the introduction of configuration classes –a means to interact with Reflex APIs at application startup– and XML configuration files –a straightforward mapping of the APIs–. Generalizing, one can wonder why we only support these two configuration means. We indeed explore in the next chapters the use of Reflex as a common back end for multiple aspect languages. Moreover, since most general-purpose AOP languages offer some structural features, we must also extend our approach to support structural reflection.

Our intuition is now refined to the idea that partial reflection may be an appropriate substrate to allow *hybrid* AOP, in which different aspect languages and models can be used together. Apart from providing (domain-specific) aspect language designers with a generic and adequate transformational engine, we believe that such a substrate could address the challenging issue of *composition* between aspects defined in various languages. In this line, Chapter 6 presents a language-independent analysis of *versatile kernels* for AOP. The evolution of Reflex towards such a kernel for Java is then presented in Chapter 7.

Chapter 6

Versatile Kernels for Aspect-Oriented Programming

In this chapter, we motivate the need for a *versatile kernel* for AOP in a general setting and analyse its requirements. Such a kernel alleviates the task of implementing an aspect-oriented approach by taking care of basic program alterations. It also lets several approaches coexist without breaking each other by automatically detecting interactions among aspects and offering expressive composition means.

We first motivate our approach in Section 6.1. Then, from a review of the main features of Aspect-Oriented Programming in Section 6.2, we present the main issues that the design of such an AOP kernel should address in Section 6.3: open support for aspect languages taking care of both behavior and structure, base language compliance, and aspect composition. Chapter 7 presents the evolution of Reflex in an AOP kernel for Java. This chapter is based on [Tanter & Noyé, 2004a; Tanter & Noyé, 2004b].

6.1 Introduction

The variety of toolkits and proposals for Aspect-Oriented Programming (AOP) [Elrad *et al.*, 2001; Kiczales *et al.*, 1997b] and related modularization technologies for separation of concerns (SOC) [Parnas, 1972] illustrates the range of possibilities for aspect-oriented programming, either in terms of specification language, binding time, expressiveness, etc. The design space of AOP is under exploration, and each proposal is a fixed point or a restricted region in this space. There are also low-level toolkits [Chiba, 2000; Chiba & Nishizawa, 2003; Dahm, 2001; Kniesel *et al.*, 2005] that can be used to create ad hoc AOP systems and experiment with the design space, but they require redeveloping an ad hoc software layer to bridge the gap with a proper high-level interface.

This work is motivated by the fact that there is a wide variety of models for AO-related programming, either general or domain specific, that are worth experimenting with, and that, in general, several approaches cannot be combined simply because they have been designed with a closed world assumption in mind. We propose a versatile kernel for AOP that makes it possible to use,

and experiment with, various approaches, while guaranteeing that these approaches do not break each other.

6.1.1 Variety of Models

There are different conceptual models for programming with aspects. For instance, AspectJ [Kiczales *et al.*, 2001] relies on the notions of join points, pointcuts, and advices; Event-based AOP (EAOP) [Douence *et al.*, 2001; EAOP, 2001] uses concepts such as crosscuts, monitors, events, and aspects; models from the reflection community rather talk in terms of hooks and metaobjects [Redmond & Cahill, 2002; Tanter *et al.*, 2001; Welch & Stroud, 1999], while the composition filter approach is based on composable method filtering [Bergmans & Akşit, 2001]. Interestingly, there are strong links between these conceptual models, as studied by Kojarski *et al.* [Kojarski *et al.*, 2003] in the case of reflection and aspect orientation. This comes from the fact that, in the end, they all boil down to semantic alterations of applications written in a base language. A model that has some convincing history in describing semantic alterations is the reflective model for structural and behavioral alterations. However research in aspect orientation has exhibited important behavioral notions related to sequences and nesting of events, as exemplified by control flow and pattern matching of events.

Some approaches adopt general-purpose aspect languages [Bergmans *et al.*, 2001; Kiczales *et al.*, 2001; Ossher & Tarr, 2001], while others rely on Aspect-Specific Languages (ASLs, *a.k.a.*, DSALs, Domain-Specific Aspect Languages). Aspect-specific languages present various advantages, in particular due to the fact that aspects are defined more concisely and more intentionally, since the language is close to a particular problem area. Domain-specific approaches present many benefits: declarative representation, simpler analysis and reasoning, domain-level error checking and optimizations [Czarnecki & Eisenecker, 2000]. Several aspect-specific languages were actually proposed in the "early" ages of AOP [Irwin *et al.*, 1997; Lopes, 1997; Mendhekar *et al.*, 1997], as well as recently, for instance DJCutter [Nishizawa *et al.*, 2004] for distributed systems. Wand argued that AOP should refocus again on domain-specific languages [Wand, 2003].

The key idea is that the most adequate conceptual model and level of genericity for a given application domain actually depends on the situation: there is no definitive, omnipotent approach that best suits all needs. Furthermore, when several aspects are to be handled in the same piece of software, combining several AO approaches often reveals fruitful [Rashid, 2001; Shonle *et al.*, 2003].

6.1.2 Compatibility Between Approaches

Combining several AO approaches seems promising. A positive feedback on a hybrid approach to separation of concerns was reported in [Rashid, 2001]. However, in this experiment, specific tools were developed from scratch to fit the experiment. This confirms that combining several AO approaches is hardly feasible with today's tools, since the tools are not meant to be compatible with each other: each tool eventually affects the base code directly, with a "closed world assumption".

If several aspects happen to affect the same program points, they *interact* [Douence *et al.*, 2005]. If they are implemented through different tools that directly transform the program, the

resulting semantics is very likely to depend on the order in which the tools are applied. Interactions among aspects will be silently and blindly handled (Fig. 6.1a). If the resulting semantics appears to be incorrect, then identifying the interaction and resolving it has to be done manually, if at all possible. This issue presents similarities with the issue of data races in concurrent programming, acknowledged to be one of the hardest errors to debug in the area of concurrency, mainly because the incorrectness of the program is not always visible to programmers and it is very hard to track down its cause.

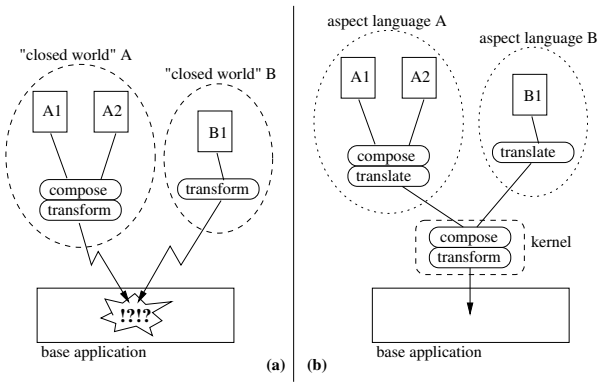

Figure 6.1 – The compatibility issue between AOP approaches.
(a) Different AOP approaches, making a closed world assumption, are applied together: aspect interactions are blindly treated, jeopardizing the resulting semantics.
(b) A common AOP kernel is used as a mediator to detect and resolve interactions: each AOP system only needs to talk to the kernel.

6.1.3 A Versatile AOP Kernel

To sum up the situation, on the one hand, there are many approaches to AOP that are worth exploring and experimenting with, and on the other hand, there is the issue that each AOP language is generally bound to its implementation, typically done from scratch with a closed world assumption. This gives rise to the compatibility issue mentioned above. In order to allow AOP to mature, it seems crucial that several approaches can be applied to a wide range of systems and situations, at various scales. Furthermore, efforts should be better focused, without having to "reinvent the wheel" for each new AOP system.

Our claim is that, since the transformation work done by AOP systems is very similar, it can be factored out in a versatile AOP kernel. Each AOP system then talks to the kernel, instead of attacking directly the base application. Only the kernel effectively affects the base application, after having ensured that aspects are properly composed (Fig. 6.1b).

An AOP kernel enables a wide range of approaches, from well-established to experimental, to work together without breaking each other. Such a kernel provides core semantics, through proper structural and behavioral models, generic enough to support all needed notions (*e.g.* cflow, aspects

of aspects) in an extensible manner. Designers of aspect languages can then experiment more comfortably and rapidly with this kernel as a back-end, focusing on the best ways for programmers to express aspects, may they be domain specific or generic.

In this work, we are concerned with the study of a versatile AOP kernel for a unique base language: we do not aim at multi-language support, or even more ambitiously at any software representation like IBM's CME [Harrison *et al.*, 2002b; CME, 2002], which attempts to address similarly UML diagrams for instance. We first want to get valuable feedback from an AOP kernel dealing with a single base language. When it comes to illustrating our argumentation, we refer to the language with which we are concretely experimenting, Java.

In the following section, we identify the different features of an AOP system. From these features, Section 6.3 draws a list of requirements for a versatile AOP kernel.

6.2 Features of AOP

AOP proposals are characterized by several features. One feature relates to the basic implementation technique (dedicated runtime environment, code transformation, etc.), but this is a non-functional concern for AOP systems, which an AOP kernel takes care of. We will come back to this point in section 6.3.

Another feature is the *symmetry* of the approach, as discussed in [Harrison *et al.*, 2002a]. In *asymmetric* approaches to separation of concerns, there is the notion of a base application to which aspects are applied, conversely to *symmetric* approaches to separation of concerns, like hyperspaces [Ossher & Tarr, 2001] or subject-oriented programming [Harrison & Ossher, 1993] where such a distinction is not done. Still, both kinds of approaches use some specific language to glue pieces together. We are here interested in asymmetric approaches: in such a setting, the relation between aspects and the base application can be characterized by *aspect obliviousness* and the *binding* between the base application and the aspects.

Aspect obliviousness has sometimes been identified as a key property of AOP [Elrad *et al.*, 2001]. It refers to the fact that the base application remains unaware of the aspects that are applied to it. However it has since been considerably softened[1]. First, base code needs to be structured in a sense that makes it possible for aspects to intervene, implying some "awareness" of aspects. This was indeed already noticed in [Kiczales *et al.*, 1997b], and was a basic idea of the work on open implementations [Rao, 1991]. As Wand puts it, aspects reason about the *ontology* of a base program: this shared ontology is knowledge that is held in common between the base program and the aspects [Wand, 2003]. Second, industrial applications of AOP have clearly highlighted that explicitly *annotating* base code with semantic meta-information can be of great value, as an explicit "interface" exposed to aspects: in Wand's terminology, annotations can be seen as a way to define a joint ontology that is more abstract, domain-specific, than the general-purpose, language-based, join point model of typical AOP languages like AspectJ. Finally, we can add that, similarly to the difference between metasystems (systems acting upon other systems) and reflective systems (systems acting upon themselves), it seems too restrictive to prohibit the explicit manipulation of

[1] A discussion on the AOSD mailing list confirms this remark: the idea of "non-invasiveness" is now put forward.

an aspect layer by parts of an application itself subject to aspects.

The binding between the base application and the aspects can be characterized along two lines [Redmond & Cahill, 2002]: the *binding time* and the *binding mode* (see Chapter 2, Section 2.4.2.3).

Finally, aspect languages can address two types of alterations: behavioral and structural ones. Most work on AOP has been around behavioral alterations. But an aspect language may include a part dedicated to structurally altering the base program, *e.g.* adding new members or interfaces to classes. This is known as *introductions* or *inter-type declarations*. Aspect languages like AspectJ [Kiczales *et al.*, 2001] or Josh [Chiba & Nakagawa, 2004] include both behavioral and structural aspects. A difference is that, as argued in [Chiba & Nakagawa, 2004], AspectJ does not deal uniformly with inter-type declarations, while Josh does.

6.2.1 Main Features

An aspect language can be described according to several features. This section focuses on the most common ones: the cut, action and binding languages; and mechanisms for aspect parameterization, instantiation, and scope.

6.2.1.1 Cut language

The cut language is the language provided to specify the places where aspects affect the base application[2].

A *behavioral cut* denotes a set of *execution points* in a program. Such a *dynamic* behavioral cut can be projected (non-injectively) in the program text to a *static* behavioral cut, which denotes a set of *program points* corresponding to expressions in the program called the *shadows* [Masuhara *et al.*, 2003] of the execution points.

Whereas a behavioral cut denotes points in the code space, a *structural cut* denotes points in the data space where data structures reside. In other words, a structural cut denotes program points that correspond to structure *definitions*, not expressions. In the case of structures, there is indeed a bijection between execution points and program points, so it is not necessary to distinguish between them: structural shadows do not make sense[3].

Cut languages may include the possibility of referring to points in aspect programs (to apply aspects to aspects), may allow complex algorithmic cut to be specified [Chiba & Nakagawa, 2004], and may be tailored to a particular domain [Nishizawa *et al.*, 2004].

For instance, AspectJ cuts (pointcuts for behavior and type patterns for structure) are generic (*i.e.* not domain-specific) and can neither refer to aspect programs nor describe algorithmic cuts. Specific features of behavioral cut languages include the expressiveness to refer to both program

[2]Depending on the proposal, the cut language is either referred to as a *crosscut language* [Brichau *et al.*, 2002; Douence *et al.*, 2002] or as a *pointcut language* [Chiba & Nakagawa, 2004; Kiczales *et al.*, 1997b].

[3]Our analysis of structural cut is actually restricted to the class level: we do not consider approaches where only some particular instances of a class are affected by an introduction.

and execution points (for instance related to control flow). For instance, Soul/Aop as presented in [Brichau *et al.*, 2002] only affects program points while AspectJ pointcuts can refer to both program and execution points. But AspectJ pointcuts are limited with respect to control flow, compared to what has been proposed by others, for instance [Douence & Teboul, 2004].

6.2.1.2 Action language

The action language is used to implement the aspect semantics. Both *structural actions* and *behavioral actions* may be supported. The action language may allow the definition of stateful aspects.

Behavioral actions basically consist in extending and/or modifying the behavior of the base application. The expressiveness of this language may be restricted or designed to fit a particular domain, or may be complete.

A structural action language provides means to alter the data structures of a program, for instance by adding new members to a class or making a class implement an interface. In a language like Smalltalk [Goldberg & Robson, 1983], such alterations can be done dynamically, while in Java they can only be done statically.

6.2.1.3 Binding language

The binding language is used to specify which action should be bound to which cut. Many aspect languages do not decouple this language from one of the two above. For instance, in AspectJ, the binding specification is tied to the advice definition: the binding language is merged with the action language.

Interestingly, the four binding combinations are valid (see Fig. 6.2). The definitely most-used combination is to bind a behavioral action to a behavioral cut. This is the usual perception of *dynamic crosscutting*. However, considering a runtime environment that supports runtime structural modifications of classes, a structural action can very well be associated to a behavioral cut. Binding a structural action to a structural cut is the classical case of introductions. Less frequent but yet worthwhile is the binding of a behavioral action to a structural cut: for instance, checking invariants or coding rules, like AspectJ's compile-time warnings and errors, can be seen as behavioral actions associated to a structural cut. To sum up, the different possible bindings between structural and behavioral cut and action are shown in Fig. 6.2, with their typical usage.

	behavioral cut	structural cut
behavioral action	*dynamic crosscutting*	*invariants, AspectJ's warning/error*
structural action		*introductions a la AspectJ*

Figure 6.2 – Summary of the possible bindings between structural and behavioral action and cut, and their typical usage.

Usually, in the case of behavioral actions, the binding language also makes it possible to specify the kind of control the aspect has over the considered execution points (before, after, instead of, etc.), although such a notion becomes irrelevant for structural cuts, simplifying the binding specification.

6.2.1.4 Aspect parameterization

Aspect parameterization refers to the possibility of passing context information from cuts to actions. Providing context information enhances the expressiveness of the action language, and allows for more reusable aspects through genericity. However, for behavioral cuts, this has a cost at runtime, especially if the context information is passed as a whole. AspectJ addresses this issue with selective parameter exposition in pointcuts, similarly to the context exposure mechanism of Josh. For structural cuts, the issue of aspect parameterization is simpler: the necessary information is usually reduced to the considered data structure (in a class-based language, the class subject to modification).

6.2.1.5 Aspect instantiation and scope

Aspect instantiation and scope are features of the aspect language that specify how aspects are instantiated and what their scope is. For actions bound to behavioral cuts, AspectJ supports the common aspect scopes: instance, class and global, while Josh does not support per-object aspects. The possibility of discriminating aspect scope with respect to threads may also be provided. Conversely, an action bound to a structural cut is usually applied statically, hence a single global instance suffices.

6.2.2 Composition of Aspects

Few aspect languages explicitly support aspect composition. Josh does not provide composition support at all, while AspectJ provides a limited aspect composition language that can only state precedence between aspects. More expressive approaches to composition have been justified [Brichau *et al.*, 2002; Douence *et al.*, 2002; Douence *et al.*, 2004]. These proposals focus on behavioral aspects. Composition of structural actions has been poorly addressed in the AOP community, but still, very related work exists in the language community: work on metaclass composition [Bouraqadi-Saâdani *et al.*, 1998], mixins [Bracha & Cook, 1990], traits [Schärli *et al.*, 2003], etc. Finally some approaches aim at automatic conflict *resolution* [Costanza *et al.*, 2001], while others argue for automatic *detection* and explicit resolution [Douence *et al.*, 2002; Schärli *et al.*, 2003].

6.2.3 Interactions Application/Aspects

Interactions between a base application and some applied aspects need to be characterized in both ways: interactions from the aspects to the application, and interactions from the application to the

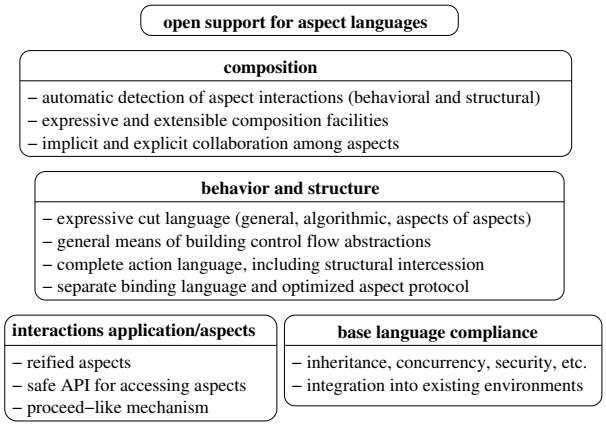

Figure 6.3 – Summary of identified requirements for a versatile AOP kernel.

aspects. The action language determines the possible interactions between the aspect and the base application.

Explicit interactions between the application and aspects –in the line of explicit metaobject protocols (Chapter 2, Section 2.2.2)– may be provided in systems where aspects are runtime entities as such, *i.e.* aspects are *reified*. If aspects are inlined within application code, the application cannot explicitly interact with them. Conversely, reified aspects are made accessible through an API for explicit access by the base application. For instance, in AspectJ, the object representing an aspect `Foo` can be accessed with `Foo.aspectOf()`. An access API may be limited to read access, or may make it possible to dynamically change aspects. Changing aspects may relate to changing actions or cuts, depending on which parts are reified. For instance, in AspectJ, only advices are reified, and are not changeable. Conversely, Steamloom [Bockish *et al.*, 2004] fully reifies aspects, making it possible to access cuts at runtime.

6.3 Requirements for a Versatile AOP Kernel

All the features exposed in the previous section represent the main *variabilities* among the family of aspect languages and systems. The objective of a versatile AOP kernel is to support the largest possible range of aspect approaches by supporting these variabilities. In this section we extract various requirements, summarized in Fig. 6.3, for an AOP kernel in a general setting.

6.3.1 Open Support for Aspect Languages

An AOP kernel makes it possible to use particular AO approaches for handling particular aspects. The family of aspect languages being open-ended –all the more as it includes generic and specific aspect languages–, the kernel must provide *open support* for aspect languages.

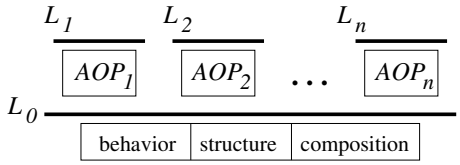

Figure 6.4 – Elements of the AOP kernel approach.
Each AOP system offering an aspect language L_i is implemented as a translator AOP_i to the common kernel language L_0.
The kernel language has 3 main parts: one for behavioral manipulation of the base application, one for its structural manipulation, and one for specifying composition.

The language and underlying conceptual model of an AOP kernel (hereafter L_0) has to be general enough to handle all the variabilities presented in Section 6.2. We have identified three main concerns for L_0, namely behavior, structure, and composition (Fig. 6.4). For behavior and structure, both introspection and intercession should be supported: introspection deals with program *analysis* and is therefore interesting for supporting *cut* languages, while intercession deals with program *transformation*, and hence supports *action* languages. For composition to be manageable by an AOP kernel, each aspect language L_i must not be implemented as a code transformer directly affecting base application code, but rather as a *translator* from L_i to L_0 (AOP_i).

The behavioral and structural parts of L_0 must be an adequate target for any aspect language, while the compositional part has to ensure that *a)* aspect languages can express their composition facilities with it, *b)* interactions among aspects, possibly defined with different languages, are detected and can be resolved, *c)* collaboration between aspects is correctly handled. All these issues are discussed in the remaining of this section.

6.3.2 Behavior and Structure

6.3.2.1 Cut

The kernel language must support expressive cuts, with aspects of aspects and complex algorithmic cuts. With respect to behavioral cut, behavioral notions highlighted by research in aspect-oriented programming (*e.g.* control flow) ought to be supported by the conceptual model at a generic level in order to allow various notions to be implemented in various ways. Generic introspective abilities of both program text and program behavior pave the way for such powerful cut support. For class-based languages, the intuitive representation of program text is based on the class-object model [Tatsubori, 2002], which represents a program by class objects aggregating member objects (fields, methods). This model follows the seminal model for structural reflection developed in Smalltalk-80 [Goldberg & Robson, 1983]. Tools like Javassist have extended this model down to the reification of expressions within method bodies [Chiba & Nishizawa, 2003].

The necessity of introspecting program behavior implies being able to discriminate amongst execution points that result from the same program point (expression). This discrimination can be

done by filtering execution points based on their runtime description, and by looking at control flow relations. As a matter of fact, different variants of control flow can be designed. For instance, control flow can be exposed as a simple call stack depth counter, or as an event stack, offering the various elements of the call stack for introspection. This allows for more expressive control flow conditions in the cut language. Furthermore, control flow, as considered in most proposals (*e.g.* AspectJ and Josh), is only about *nesting* events, not about their *sequences* generally speaking. Research on event-based AOP and trace-based aspects [Douence *et al.*, 2002; Douence *et al.*, 2004; Douence *et al.*, 2005; Douence *et al.*, 2001; Sereni & de Moor, 2003; Filman & Havelund, 2002] has justified the benefits of being able to define aspects that apply depending on the execution history. Douence *et al.* have studied the formalization of such aspects [Douence *et al.*, 2002; Douence *et al.*, 2004; Douence *et al.*, 2005; Douence *et al.*, 2001]. Sereni *et al.* have proposed control flow as regular expressions on the call stack [Sereni & de Moor, 2003]. Filman *et al.* have further extended the design of a language of events in order to fully express relationships among events, such as time frame of occurrence [Filman & Havelund, 2002]. Recently, Douence and Teboul have proposed an expressive cut language for control flow [Douence & Teboul, 2004]. Thus, a generic means to build control flow abstractions is required.

6.3.2.2 Action

To handle all possible action languages, the kernel action language should support stateful aspects, so that they can maintain information across execution. Since this language should also be complete, using the base language is the logical choice. To some extent, a behavioral action can be merged within base code statically, like in Josh. Furthermore, for structural actions, a complete set of structural transformations has to be provided. This consists in offering full structural intercession on the application. Since the base language may not directly support this (like Java), the use of a structural reflection tool (like Javassist) may be required.

6.3.2.3 Binding

The binding language deserves special attention, in particular when considering behavioral cut. As mentioned before, the binding language of AspectJ is merged with the action language: the advice body of an aspect is tied to the binding to a pointcut. This brings performance benefits, since only selected parameters are exposed by a pointcut to an advice. Also, from an ease-of-use perspective, it has the benefit of almost hiding to the programmer the vertigo of writing metaprograms. However, from a software engineering perspective, this limits the possibilities of reusing a given advice in a different context. Looking back at the history of reflection and runtime metaobject protocols, the position is the opposite: the binding language is tied to the cut language. The reified information is both rich and standardized, determining the actual *protocol* of metaobjects (hence the term “metaobject protocol”, MOP). Metaobjects are highly generic and hence reusable, but more costly and complex to write.

Both approaches have their advantages. For building middleware or other kind of infrastructure software, the genericity and high reusability of metaobjects is of great value. For localized AOP, the non-genericity of aspects is not a problem, it is even a plus, simplifying the task of aspect

programming. This duality of approaches with respect to the binding language naturally calls for a versatile AOP kernel that keeps the binding language *separated from both* the cut and action languages, and that provides mechanisms for specializing and optimizing the information bridge between execution points and aspect bodies, which we refer to as the *aspect protocol*.

6.3.3 Composition

Two aspects are said to *interact* if they affect the same program (or execution) point [Douence *et al.*, 2005]. It is true that aspects may interact semantically without affecting the same points. However, it is very hard to capture this kind of "abstract" interactions between aspects. We therefore only focus on the former kind of interactions. In this respect, we concur with Douence *et al.* that the *resolution* of aspect interactions ultimately depends on the application semantics and hence cannot be decided automatically, although such interactions can be *detected* automatically [Douence *et al.*, 2002; Klaeren *et al.*, 2000]. Hence, the AOP kernel should detect interactions and warn the programmer, so that composition strategies can be specified to resolve the interactions.

Composition should be supported in an expressive and flexible manner. A poorly expressive composition language, like in AspectJ, where only aspect precedence can be specified, is not sufficient to handle complex interactions between aspects [Brichau *et al.*, 2002; Douence *et al.*, 2002; Douence *et al.*, 2004]: composing aspects does not solely refer to specifying the order in which they apply, but also, for instance, to possibly condition their application to the presence and application of other aspects. Composition issues relate to both structure and behavior. With respect to aspect precedence as supported by AspectJ, it is in fact a *nesting* of aspects: for aspects that are plugged around a given operation, a `proceed` mechanism is provided in order to dynamically build the appropriate flow of control between nested advices and the base level.

Finally, in AOP it is common practice to introduce structural properties (such as an implemented interface) that may then be visible to other aspects. Conversely, some structural changes may be totally local to a given aspect and should not be exposed to others. This implies that a *collaboration protocol* should be provided to control the visibility of structural changes made to base entities among aspects. If aspects of aspects are supported, the collaboration between behavioral changes is explicitly handled in aspect definitions: an aspect does not see another aspect unless its cut affects it.

6.3.4 Base Language Compliance

An AOP kernel should support the various semantic elements of its base language. For instance, Java is a class-based language, offering inheritance. But is has also been designed for concurrent and distributed programming, and supports security policies. These elements are usually underestimated in the various proposals and ad hoc toolkits, or simply left aside. However, they can have a non-negligible impact on the design of the kernel and its features. We therefore discuss this issue for inheritance, concurrency and security. Persistence and distribution are not addressed here. A discussion of the impact of distribution over the design of a cut language can be found

in [Nishizawa *et al.*, 2004]. Finally, the issue of the implementation approach and integration of the kernel into existing environments is discussed.

6.3.4.1 Inheritance

A Java AOP kernel is expected to behave well with respect to the interaction between inheritance and aspects. The main concern in this regard comes from the fact that aspects are introduced by modifying class definitions, and that inheritance implies that subclasses inherit the structure and behavior of their superclasses. Conceptually, since aspects are dedicated to handle concerns that crosscut the class modularization, their scope does not necessarily follow that of the inheritance hierarchy. Hence an AOP kernel should make it possible to declare whether the cut of an aspect applies to subclasses or not. It should also make it possible to stop downward propagation from a certain class.

6.3.4.2 Concurrency

A Java AOP kernel must be usable in concurrent environments. Aspects can be subject to concurrency, and they may as well be used to control concurrency in an application. This entails that the visibility of an aspect with respect to threads (global or local) should be specifiable, and, in the case of dynamic aspects, their initialization should be thread-safe.

6.3.4.3 Security

Java features a security model based on a "sandbox" customizable with policies [Gong, 1999]. There is also a dual relation, like for concurrency, between aspects and security. Much work has been done on implementing security policies with reflection or aspects [Welch & Stroud, 2002; Ancona *et al.*, 1999; De Win *et al.*, 2001]. But the reverse is indeed crucial: aspectizing an application must not break its security properties. Vayssière *et al.* studied this issue in the case of a simple Java runtime metaobject protocol [Caromel *et al.*, 2001; Caromel & Vayssière, 2001]. A first issue is to ensure that using a metalevel does not tamper with properties of the base application. A second issue is to devise security policies, compatible with the existing Java policy mechanism, to protect the base application by restricting the actions available at the metalevel (such as changing the receiver of a method call or its arguments).

The fundamental point is the necessity of keeping *metacode separated from base code at runtime*. This is required because the security mechanism of Java relies on the call stack to dynamically compute the permissions associated to a call [Gong, 1999]. Hence, if metacode is inlined within base code, it gets exactly the same permissions as base code. The solution is that only infrastructure code should be inserted in the base application. Infrastructure code is assumed to be trusted –and hence can get the same permissions as base code– since it is generated by the reflective (or aspect) system and only delegates to the possibly untrusted metaobject (or aspect) code.

6.3.4.4 Implementation Approach

There are two main implementation approaches to AOP systems, one that consists in extending or modifying the runtime environment of the language, and one that consists in transforming code and leaving the runtime environment intact. To fully support dynamicity, an AOP kernel should be closely integrated into the language environment, in particular into the runtime environment.

In the context of Java, this means that the AOP kernel should be provided by the Java Virtual Machine (JVM) itself. However, the abilities of standard JVMs with respect to behavioral and structural intercession are limited. Hence, experimenting with an AOP kernel at the VM level requires working on a dedicated environment. To be compatible with standard Java environments, we decide to slightly limit the dynamicity supported by our kernel and thus adopt a code transformation approach. We feel that this choice can be beneficial, at least in a first phase, to study an AOP kernel in various settings, since it allows simpler and more widespread experiments to be carried out. If such an AOP kernel turns out to be of practical interest for the Java community, then VM support for AOP kernel services should be considered.

In order to be used as a weaver, an AOP kernel should first be parameterized by a set of aspect languages:

$$(1)\ kernel(\{L_i\}) \Rightarrow kernel_{\{L_i\}}$$
$$(2)\ kernel_{\{L_i\}}(application, aspects) \Rightarrow application_{aspectized}$$

(1) The AOP kernel is parameterized by a set of supported aspect languages ($\{L_i\}$). A language L_i is implemented as a translator from L_i to L_0. Depending on the kernel, parameterization may or may not be available dynamically to incrementally update the set of supported aspect languages. (2) The specialized AOP kernel then acts as a *weaver*, producing the aspectized application from the application and the various aspects (written in any of the aspect languages belonging to $\{L_i\}$). The weaver is conceptually a composition of the translators of the $\{L_i\}$ languages.

For the sake of efficiency, a kernel should be able to use *staging*, fixing some concerns statically in order to enhance performance, in case a particular approach does not require dynamicity. For instance, regarding aspect composition, it can be resolved statically [Brichau *et al.*, 2002; Kiczales *et al.*, 2001] or dynamically [Douence *et al.*, 2002; Tanter *et al.*, 2001]. An AOP kernel may allow composition issues to be resolved statically (at weave time) thanks to an extensible set of composition operators, and also let the possibility of using composition frameworks for dynamic aspect composition. Staging in an AOP kernel can thus be seen as a tradeoff between choices fixed at weaving time and others left open at runtime. Also, an AOP kernel should possibly be used offline (*e.g.* as a post-processor), or online (*e.g.* as a special class loader).

6.3.5 Interactions Application/Aspects

Several requirements identified beforehand indicate that an AOP kernel should not inline aspect code within application code, but rather adopt a model where aspects are runtime entities separated from the objects they influence, *i.e.* aspects should be *reified*: aspects being runtime entities can maintain some state during execution, and reified aspects are compatible with stack-based security

mechanisms. Indeed, the incurred performance penalty is far from obvious in the context of ever-improving dynamic compilers [Hilsdale & Hugunin, 2004].

Moreover, for an AOP kernel to fully support explicit interactions between applications and aspects, reifying aspects is also mandatory: a base application should possibly access reified aspects in order to explicitly interact with them, and also to change them. However, the access API supported by a kernel should be safe: it must be possible to specify that an aspect *cannot* be changed during execution, as well as to impose some restrictions for dynamically replacing aspects. For instance, type restrictions can be used to guarantee that an aspect protocol will not be broken by replacing an aspect action with another one. Finally, as discussed in Section 6.3.3, a proceed-like mechanism is necessary in order to support approaches that allow aspects to wrap operation occurrences.

Summary

In this chapter, we have motivated the need for providing a versatile kernel for AOP, as a means to foster consolidation in both use and exploration of AOP. We have identified the main requirements that an AOP kernel should address: open support for aspect languages taking care of both behavior and structure, base language compliance, and aspect composition. Since this analysis suggests that partial reflection is an appropriate general framework for AOP, we concretize these ideas in the next chapter by evolving Reflex into a Java AOP kernel based on partial reflection, reconciling in this way reflection and aspect orientation.

Chapter 7

A Versatile AOP Kernel for Java

In the previous chapter, we have identified the main requirements for a versatile AOP kernel. A key issue is that the kernel language has to be general enough to support various approaches, and must also support the specificity required by a single approach. Therefore, this thesis suggests that an appropriate model of partial reflection supporting a high degree of selectivity and specialization is an adequate general framework for AOP kernels. In this chapter, we present such a model and the evolution of Reflex to a versatile AOP kernel. We show how to build basic aspect weaving and composition facilities on top of a reflective layer, reconciling the flexibility of reflection and the guidance of AOP.

We first outline why Reflex as presented in the previous chapters cannot be considered as an AOP kernel, although it represents a good starting point. This chapter then shows how we have extended Reflex into a versatile AOP kernel for Java that fulfills the identified requirements. In Section 7.2, we describe the behavioral and structural abilities of Reflex, centered around the notion of *links*. Section 7.3 and Section 7.4 address collaboration and composition issues. Finally, Section 7.5 exposes a simple plugin architecture for open aspect language support, explaining how (possibly domain-specific) aspect languages can be defined on top of the kernel. Section 7.6 discusses related work.

7.1 Introduction

In Chapter 6 we have drawn, from an analysis of the different features of AOP systems, a list of requirements for a versatile AOP kernel in a general setting. These requirements, listed and numbered in Fig. 7.1, are related to open support for aspect languages *(1)*, composition *(2)*, behavior and structure *(3)*, interactions applications/aspects *(4)* and base language compliance *(5)*.

The model of partial behavioral reflection presented in the previous chapters, implemented in Reflex, mostly fulfills the requirements of an AOP kernel related to behavior, interactions application/aspects, and base language compliance. Recall that in this model, an explicit *link* binds a *hookset* to a *metaobject*. In more abstract terms a hookset corresponds to a set of program points (or static cut) and the metaobject to the action to be performed at these program points (or an

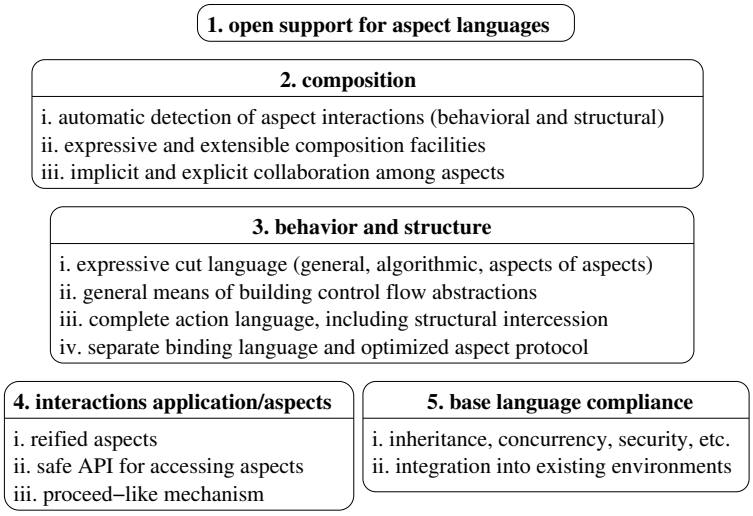

Figure 7.1 – Summary of identified requirements for a versatile AOP kernel.

advice, using AspectJ terminology). The link is characterized by several *attributes*; for instance the scope attribute specifies whether a metaobject is per instance, per class, or global (for the considered link).

7.1.1 Previously Fulfilled Requirements

The class-object model [Tatsubori, 2002], further extended to the reification of expressions within method bodies like in [Chiba & Nishizawa, 2003], provides an intuitive introspectable view of a program. In our model, hooksets are defined from different predicates (class and operation selectors), which precisely use general introspection to support algorithmic behavioral cuts *(3i)*. Yet, this view is restricted to program text. Introspecting program execution requires runtime support. As a matter of fact, a metaobject can be used to track down whether an operation occurs within the control flow of another operation occurrence. Generalizing this idea, we expose execution information uniformly by introducing the concept of event collector, as a general means of building control flow abstractions *(3ii)*. Event collectors gather execution events to expose parts of a program execution (nesting, sequences, etc.), under any structure (counter, stack, tree, DAGs, graphs, etc.), for dynamic introspection. Event collectors make it possible for language designers to experiment with control flow abstractions. As a matter of fact, different variants of control flow can be designed, and have an impact on the expressiveness of the cut language with respect to control flow conditions (see [Douence *et al.*, 2002; Douence *et al.*, 2004; Douence *et al.*, 2001; Douence & Teboul, 2004; Filman & Havelund, 2002; Sereni & de Moor, 2003]). For instance, control flow can be exposed as an event stack, thereby offering the various elements of the call stack for introspection. Sequences of events can also be exposed by appropriate event collectors. Event collectors are integrated uniformly in our model, since they are just metaobjects. They dif-

fer from metaobjects implementing aspect actions in the sense that they only record and expose information about execution, which can be introspected to determine dynamic cuts.

Aspect actions are implemented in metaobjects, which are defined in the same base language. Aspects can therefore be stateful. Aspects of aspects are supported since metaobjects are subject to the cut of other aspects. The expressiveness of the metaobjects, which are defined in the base language, is such that there is no a priori limitations to the behavioral modifications that can be applied to the base program *(3iii)*. Furthermore, a crucial element of our model is that the binding between cuts and actions is explicit and separated from cuts and actions, as an entity called a *link (3iv)*. First, this separation makes it possible to enhance cut and action reuse. In Chapter 6, we argued for the necessity of keeping binding separate from both cut and action in a perspective of being able to support any kind of metaprogramming (MOP-like or AOP-like), since it makes it possible to specialize and optimize the *aspect protocol –i.e.* the communication protocol– between cuts and actions *(3iv)*. Furthermore, as discussed in [Mezini & Ostermann, 2003], providing higher-level modular structures for aspects also requires to separate bindings. Finally, we will illustrate that explicit links prove useful to reason about composition.

With respect to explicit interactions between an application and aspects, our model assumes that aspects are reified *(4i)*. Reflex, as being an implementation in standard Java *(5ii)* based on load-time bytecode transformation using Javassist [Chiba, 2000; Chiba & Nishizawa, 2003], however does not fully reify aspects: only actions are fully reified (as metaobjects), while cuts are only partially reified, since a runtime modification of a static cut will only affect new loaded classes. A VM-based approach, like Steamloom [Bockish *et al.*, 2004], would remove this limitation, but would then no longer be compatible with standard environments. Aspect actions can be changed dynamically, in a safe way *(4ii)*: it is possible to specify type constraints on metaobjects. Such a change can also be forbidden. Finally, a proceed-like mechanism is available, via the standard Java reflection API *(4iii)*.

7.1.2 Newly Fulfilled Requirements

With respect to base language compliance, our previous work did not address issues like concurrency and inheritance properly. We have added some link attributes in order to overcome these limitations. The thread visibility attribute now makes it possible to specify whether a metaobject reference is shared among threads, or if it is thread-local. An initialization attribute specifies if a metaobject should be initialized eagerly or lazily, taking into account synchronization or not. Finally, the inheritance attribute has been added to specify whether the cut of an aspect applies to subclasses or not (the link is said to be *private* or *public*). In the case it does, it is also possible to specify a condition that stops this propagation.

With respect to security, discussed in Section 6.3.4.3, we have highlighted that metacode needs to be separated from base code at runtime in order to support stack-based security mechanisms, like in Java. Although the integration of a security mechanism in the line of Vayssière's work [Caromel *et al.*, 2001; Caromel & Vayssière, 2001; Vayssière, 2002] is not yet done, the compatibility of Reflex with such mechanisms is guaranteed since metaobjects are runtime entities separated from the base application, and only trusted code generated by Reflex is inserted into

application classes.

But most importantly, in order to serve as the basis of a versatile AOP kernel, our approach lacked support for structural aspects *(3)*, aspect composition *(2)*, and definition of (possibly domain-specific) aspect languages *(1)*. With respect to structure, we have integrated both structural cut and actions (including intercession) in our model, following the same concept of links. This is presented in Section 7.2, along with a summary of elements complementing behavioral AOP support. As regards aspect composition, we distinguish between inter-aspect collaboration making it possible to expose or hide their changes to others (Section 7.3), and detection/resolution of aspect interactions (Section 7.4). Finally, Section 7.5 overviews a lightweight plugin architecture for defining (possibly domain-specific) aspect languages.

7.2 Behavioral and Structural Abilities

Considering structural abilities of an AOP kernel, we have discussed in Chapter 6 that any combination of structural and behavioral cut and actions makes sense. However, the standard Java environment brings a number of limitations. In particular, Java supports dynamic class loading and forbids runtime modification of class definitions. This leads us, in the context of Java, to distinguish between two types of links: structural links and behavioral links.

		S–link	**B–link**
cut		structural	behavioral
applied at		load time	runtime
action	S	*adapt structure*	*create classes* *define new S–links*
	B	*check invariants* *warning/error*	*adapt behavior*

Figure 7.2 – Model of partial reflection for Java: the two types of links and their corresponding cut, application time, and actions.

Dynamic loading in Java implies a superposition of load time and runtime. From an execution point of view, load time corresponds to the interception of the runtime event *"a class is being defined"*. Yet, since classes cannot be altered at runtime, we consider that a structural cut applies at load time. Therefore, a *structural link*, termed S-link, binds a structural cut to an action, which can be either structural or behavioral (Fig. 7.2). A behavioral action bound to a structural cut can be used to ensure invariants or coding rules, like AspectJ's warning/error mechanism. An S-link is *applied*, *i.e.*its associated action is performed, at load time.

A *behavioral link*, called B-link, binds a behavioral cut to an action. Our previous model of partial behavioral reflection was restricted to this kind of links. Although actions that can be performed at runtime in Java are mainly behavioral, we may consider the following actions as being structural actions that can be bound to B-links in Java: definition of new classes (including, for instance, implicit reflective subclasses as in [Tanter *et al.*, 2001] and definition of new S-links

(thereby indirectly affecting structure). A B-link is applied at runtime (Fig. 7.2).

We now review in more details how we have evolved Reflex into an AOP kernel based on the proposed model of partial reflection. Fig. 7.3 summarizes how the main concepts of AOP are supported in Reflex. First, in Section 7.2.1 we explain how Reflex operates at load time with respect to the different types of links. Then, Section 7.2.2 presents S-links. In Section 7.2.3, we summarize how B-links are represented and characterized, including some new attributes. We also review the selection mechanisms supported by Reflex to enhance partiality of reflection with respect to B-links, as well as the context exposure mechanism.

AO concept	**support in Reflex**	
	load time	***runtime***
action	LT−metaobject	metaobject
binding	S−link	B−link
static cut	class set	hookset
dynamic cut	–	restrictions activation
aspect protocol	*fixed:* class object	MOP descriptors
instantiation	*fixed:* lazy thread−global	initialization thread visibility
scope	*fixed:* global	scope

Figure 7.3 – Support provided by the Reflex AOP kernel for the main AOP concepts.

7.2.1 Process Overview

In order not to modify the standard Java execution environment, behavioral links are *set up* at load time: during the *B-link setup* phase (BLS), hooks, along with necessary infrastructure, are installed in base code at the places indicated by the hookset definitions (in Chapter 4), only this phase existed). Conversely, structural links are *applied* at load time. Since they can influence B-link setup, for instance by inserting a method whose execution is subject to a behavioral cut, the *S-link application* phase (SLA) is carried out before the BLS phase.

This two-phase process is illustrated in Fig. 7.4. Both phases follow a similar scheme: when a class is loaded, a selection step (a diamond in Fig. 7.4) determines the set of links that potentially apply. In SLA, links that select the loaded class are determined, while in BLS, selection goes down to operation occurrences in the class definition. If more than one link selects an operation occurrence, a detection-resolution-composition step (DRC in Fig. 7.4) occurs. This step, which was absent in our previous work, basically consists of *1)* informing the user that an interaction has been detected, *2)* letting the user resolve it, and *3)* appropriately composing links. DRC is

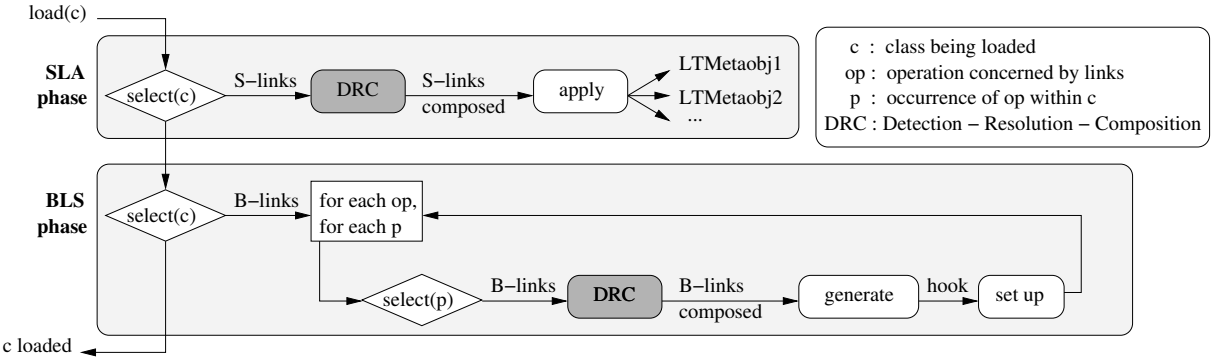

Figure 7.4 – Reflex operates in two phases at load time.
(1) S-link application (SLA). (2) B-link setup (BLS).

presented in Section 7.4. Finally, S-links are applied (in SLA), and B-links are set up (in BLS) after generating hook code.

During this process, both the application of S-links and the set up of B-links effectively introspect and modify code. Hence the issue of what can actually be *seen* in the definition of a class at a given point in time becomes crucial. We propose a general-purpose *collaboration protocol* for dealing with this issue. This protocol is presented in Section 7.3.

7.2.2 Structural Links

A structural link binds a structural cut to some action (either structural or behavioral). In Reflex, a structural cut is a *class set*, defined intentionally by a *class selector*:

```
interface ClassSelector {
  boolean accept(RClass aClass);
}
```

A class selector can base its decision on any introspectable characteristics of a reified class object, *down to the constituents of method bodies* (expressions in a method body are reified if needed).

In evolving Reflex into an AOP kernel, we entirely wrapped the object model of Javassist into our own in order to support the collaboration protocol discussed in Section 7.3. Therefore, our model basically mimics the one of Javassist for both introspection and intercession: a unique class pool object (of type `RPool`) gives access to `RClass` objects, which in turn give access to their defined `RField`s, `RMethod`s and `RConstructor`s (all `RMember`s); both methods and fields give access to their bodies as a sequence of `RExpr` objects.

An action bound to a structural cut is implemented in a *load-time metaobject*, instance of a class implementing the following interface:

```
interface LTMetaobject {
  void handleClass(RClass aClass);
}
```

Since such metaobjects are part of load-time execution, their interface is fixed. The context is defined by the class being loaded. Moreover, since the loading of a class can be nested within the loading of another class, Reflex exposes the stack of classes being loaded. Scope and instantiation concerns are straightforward and also fixed in this case: a load-time metaobject has a global scope and is created when the link is defined. A structural link is represented by an `SLink` object, for instance:

```
SLink l = new SLink(new MyCS(), new InvariantChecker());
API.links().addSLink(l);
```

This is an example of a behavioral action bound to a structural cut: the `InvariantChecker` (a load-time metaobject) checks each class accepted by `MyCS` and throws a warning or an error in case the invariant is violated.

7.2.3 Behavioral Links

A behavioral link is by essence more complex than a structural link: it is *set up* at load time and *applied* at runtime. Runtime application implies that great care should be taken in order to limit runtime cost to the minimum. Our work on partial behavioral reflection and Reflex in the previous chapters has addressed these issues to some extent.

First of all, in Reflex, a B-link has a dual representation as a `BLink` object at load time, and as an `RTLink` object at runtime. The term `Link`, rather than `BLink`, was used in Chapter 4, but we now have to distinguish between behavioral and structural links. A `BLink` object represents the specification of a B-link, and is used for driving the BLS phase (Fig. 7.4). An `RTLink` object represents a B-link during execution: it makes it possible to access metaobjects and activation conditions associated to a link at the appropriate level (object, class, or link). An `RTLink` object is therefore a link-specific runtime API for localized metaprogramming.

In Chapter 5, we carried a case study in using Reflex to support AspectJ's dynamic crosscutting. This study motivated some enhancements for both efficiency and flexibility: *restrictions* and *MOP descriptors*. Restrictions are the Reflex equivalent to residues, *i.e.* conditions that need to be evaluated at runtime to determine whether the runtime occurrence of a given operation actually is a join point [Masuhara *et al.*, 2003; Hilsdale & Hugunin, 2004]. Therefore, Reflex supports several mechanisms for selecting execution points of interest, which differ in terms of the time at which they are bound and evaluated (Fig. 7.5). Typically, the later a mechanism is bound and evaluated, the more flexible and the less efficient it is.

Static selection, typically used to define joint point (JP) *shadows* [Masuhara *et al.*, 2003] (*i.e.* the counterpart of dynamic join points in program text), is done with class and operation selectors, which are both statically bound and evaluated. Restrictions (JP residues) are statically-bound but dynamically-evaluated conditions. Note that in contrast to AspectJ where residues can only express static (global) conditions, restrictions can be associated to instances. Reflex also sup-

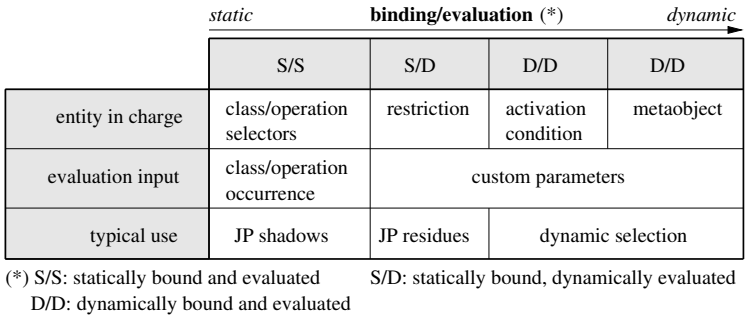

Figure 7.5 – Selection mechanisms classified according to their binding and evaluation times.

ports activation conditions (introduced in Chapter 4), which are both dynamically bound and evaluated. Finally, the same kind of dynamic selection can be done in the metaobject itself. Although metaobjects and activation conditions have the same binding and evaluation times, separating them makes sense for different reasons. Activation conditions make it possible to let metaobjects focus only on the action to perform. Decoupling dynamic conditions and actions is also interesting since activation conditions do not necessarily follow the scope of metaobjects. For example, although a unique global metaobject is shared among all classes and instances involved in a given link, such a link can be deactivated for a single instance (this is instance-based entity selection, see Chapter 4).

Restrictions, activation conditions and metaobjects take custom parameters as input. The underlying mechanism is the same: *call descriptors* have been introduced as general-purpose entities describing generation of method calls (the name of the method to invoke, the type declaring this method, and its parameters).

The description of parameter generation relies on the extended Java language supported by the Javassist compiler, which is both expressive and efficient [Chiba & Nishizawa, 2003]. For instance, `argsParam` is a `Parameter` object, which describes an object array packing the arguments to an invocation, using Javassist special syntax (`$args`):

```
Parameter argsParam = new Parameter(){
  public String getCode(Operation aOp){
    return "$args";
  }};
```

A MOP descriptor basically associates call descriptors to the various controls (before, after, replace). MOP descriptors make it possible to specialize the *aspect protocol* between a given cut and an associated action, since the aspect protocol itself is implemented as a metaobject protocol. MOP descriptors can be specified at the global level of operations like in traditional reflective systems where the reification of an operation occurrence is standardized and common. But a MOP descriptor can also be specified more locally, at the link level, and even at the hookset level. This makes it possible to specialize the aspect protocol at a fine-grained level, hence supporting the specificity of aspect orientation. For instance, in the implementation of Sequential Object

Monitors (SOM) [Caromel *et al.*, 2004], a scheduler (metaobject) is given control both before and after a method invocation on its scheduled object: before, its `enter` method just needs to receive the name of the invoked method and its arguments, while after, its `exit` method does not need any information. This is specified as follows:

```
somLink.setMOCall(Control.BEFORE, "som.Scheduler", "enter",
                  new Parameter[]{ nameParam, argsParam });

somLink.setMOCall(Control.AFTER, "som.Scheduler", "exit");
```

7.3 Collaboration Protocol

Aspects may change both the structure and behavior of a program as a consequence of their actions, while relying on the state of that same program, via introspection, as part of their cuts. Reflex adopts a defensive approach in this regard by ensuring that changes made to class definitions in both the SLA and BLS phases are *hidden*. Still, since some aspects actually need to see the structural changes made by other aspects during the SLA phase, Reflex supports a general-purpose *collaboration protocol*.

SLA phase

	introspection	intercession
scope	*body*	*member*
visibility	*custom*	

BLS phase

	introspection	intercession
scope	*body*	*body*
visibility	*custom*	*hidden*

Figure 7.6 – Scope and visibility of introspection and intercession.

Fig. 7.6 shows, for both the SLA and BLS phases of the process, the *scope* and the *visibility* of introspection and intercession. The scope refers to the finest structural element that can be observed or changed. In SLA, member *bodies* (the body of a field refers to its initializer) can be introspected (*body*), but intercession is limited to members (*member*); conversely, in BLS, method bodies can be both introspected and intercessed. The visibility indicates whether the collaboration protocol presented hereafter can be used (*custom*) or not (*hidden*). Intercession in the BLS phase is necessarily hidden. This restriction is motivated by the fact that behavioral changes are conceptually runtime changes: the fact that Reflex operates at load time, by introducing hooks, should be transparent. In order to support collaboration when aspects effectively need to see the structural changes made by other aspects, we propose a general-purpose collaboration protocol, which is both implicit and explicit.

On the one hand, when modifying the program structure, a metaobject can *explicitly* expose some information via a general-purpose key-value property map. Any structural element holds such an open set of properties. Furthermore, the structural elements affected by the application of a link *implicitly* record that link identity. This ensures that it is always possible to determine whether a structural element is original or results from the application of a given link. This feature is easily implemented, since Reflex maintains the information of which link is currently being applied.

Therefore, all intercession methods on structural elements automatically record this information in the property map. Furthermore, it is possible to force a new structural element to be always visible (resp. always hidden) by setting a particular property `forceVisible` (resp. `forceHidden`).

One the other hand, when introspecting the program structure, both metaobjects and class selectors can *explicitly* ask for classes or members that match a user-defined criterion. This criterion can be based on the creator of the structural element, or any property that may have been previously exported. In other words, by default, introspection *implicitly* uses a criterion *"has no associated creator"*.

Objects that give access to classes and members (*i.e.* `RPool` for classes, and `RClass` for members) have overloaded `get` methods that accept a *selector* as a parameter. For classes, class selectors are used. For members, *member selectors* are introduced. They implement the following interface:

```
interface MemberSelector {
  boolean accept(RMember aMember);
}
```

The following piece of code stores in `methods` the set of the declared methods that match a criterion defined by `custom`:

```
MemberSelector custom = ...;
RMethod[] methods = aClass.getDeclaredMethods(custom);
```

For instance, considering an aspect adding history to fields, and another aspect making fields persistent, the issue of whether the field added by the first aspect in order to record history should be made persistent appears. In Reflex, the implementation of each aspect involves a behavioral link whose cut selects field accesses. In addition, a structural link must add a history field to each selected class. Therefore, the history field will only be made persistent if the cut of the persistency behavioral link requires to consider the effect of the history aspect, *i.e.* uses a field selection criterion that accepts members including those added by the history aspect. This makes it possible to avoid *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [Chiba *et al.*, 1996]. If the history aspect sets the `forceVisible` property of the history field, other aspects will by default see it as part of the original class definition.

7.4 Link Composition

Composition of aspects is a challenging and multi-faceted issue. Although some proposals try to achieve automatic composition (see [Costanza *et al.*, 2001]), most assume that aspect composition is inherently impossible to resolve automatically. Some proposal only address means to specify composition [Brichau *et al.*, 2002; Kiczales *et al.*, 2001], others focus on means to detect conflicts [Klaeren *et al.*, 2000], while some aim at both, like work on event-based AOP (EAOP) by Douence *et al.* [Douence *et al.*, 2002; Douence *et al.*, 2004; Douence *et al.*, 2001]. We follow the latter approach.

Part of the solution to resolving aspect interactions (illustrated in [Brichau *et al.*, 2002] for instance) lies in specifying that an aspect should apply whenever another aspect does (implicit cut), or that an aspect should be applied onto another one (aspects of aspects). These kinds of interactions are easily expressible in Reflex: an implicit cut is obtained by defining a link whose cut is the same as the cut of another link, and aspects of aspects are simply aspects whose cut affects the action (metaobject) of another. In this section, we are targeting automatic detection of aspect interactions at the code level and expressive means for the explicit resolution of these interactions [Douence *et al.*, 2002]. By aspect interaction, we mean aspects that affect the same program points (execution or structure). In their work, Douence *et al.* are concerned with the formal definition and interaction analysis of aspect interactions, while we address the actual detection of interactions directly at the implementation level. Our approach therefore follows a detection-resolution-composition (DRC) scheme, as advocated in [Douence *et al.*, 2002]: the kernel ensures that interactions are detected, and possibly reported to users; it provides expressive and extensible means to specify the resolution of these interactions; from such specifications, it composes links appropriately.

7.4.1 Interaction Detection

In the process illustrated in Fig. 7.4, selection steps determine the subset of links that (potentially) apply. If more than one link applies, then there is an interaction. For S-links, there is an interaction when a class being loaded is selected by more than one class set; for B-links, there is an interaction when an operation occurrence in program text is selected by more than one hookset. Actually, the kind of interactions detected by Reflex for S-links is quite coarse-grained. Refining detections for S-links down to the member level is future work.

For B-links, the level of detection is at the level of program points, not program execution. In other words, a link is considered to apply at a given execution point if it does at its shadow. As a matter of fact, an aspect action is not necessarily triggered upon all runtime occurrences of a program point: this fact can either be determined by restrictions, activation conditions, or even the action (metaobject) itself deciding whether it applies or not (as in the all-dynamic EAOP model). We have already started to work on an extension to this, considering a staged notion of link application [Tanter & Noyé, 2004b].

7.4.2 Mechanisms for Interaction Resolution

When the kernel detects an interaction, a set of links that (potentially) apply is known. Recall that for S-links, the element upon which links interact is a class, while for B-links, it is an occurrence of an operation in program text. Resolving an interaction is carried out in two steps: *1)* selecting, within the current interaction, the subset of links that should actually be applied, and *2)* ordering the application within the subset.

7.4.2.1 Interaction selectors

A first kind of resolution consists in defining inter-link conditions, like *"do not apply l_1 if l_2 does"*. This kind of mutual exclusion or dependencies between aspects is mentioned in [Brichau *et al.*, 2002; Douence *et al.*, 2002; Klaeren *et al.*, 2000], but not supported by AspectJ. As we have said, for the time being, link application is statically determined, during the load-time process. Therefore, Reflex supports a simple general-purpose *link interaction selector* mechanism. An interaction selector can be attached to a link, and will be queried whenever the link is involved in an interaction, in order to determine whether it actually applies or not, depending on the other links present in the interaction. A link interaction selector is an object implementing the following interface:

```
interface InteractionSelector {
  boolean accept(LinkInteraction li);
}
```

If an interaction selector returns `true`, then the link actually applies, otherwise it is withdrawn from the link interaction. For instance, the following specifies that l_2 should not be applied if l_3 does, and that l_1 should be applied only if l_2 does:

```
l2.setInteractionSelector(
  new InteractionSelector(){
    boolean accept(LinkInteraction li){
      return !li.contains(l3);
    }});

l1.setInteractionSelector(
  ... return li.contains(l2) ...);
```

The first part of this code attaches an interaction selector to l_2. This interaction selector returns `true` only if l_3 is not present in the link interaction. Then, an interaction selector that only accepts an interaction if it does contain l_2 is attached to l_1. An issue here is that the resulting link interaction should not depend on the order in which interaction selectors are applied. A solution consists of repetitively applying interaction selectors until a fixed point is reached, as discussed in [Costanza *et al.*, 2001]. In the present example, when l_1, l_2 and l_3 are initially present in a link interaction, the correct result is that only l_3 should be applied.

7.4.2.2 Ordering and nesting

Once a set of interacting links has been cleaned up by the application of interaction selectors, the issue of the *order* in which links will be applied remains. For S-links, simple sequencing is enough. However, since B-links can apply both *before* and *after* a given operation occurrence, more expressiveness is needed. As discussed in both [Brichau *et al.*, 2002] and [Douence *et al.*, 2002], considering the interaction between two aspects can be resolved using composition operators, *seq* and *wrap*, dealing with sequencing (not supported by AspectJ) and wrapping. Reflex uses predicates to specify link composition. The rule $seq(l_1, l_2)$ means that l_1 must be applied

before l_2, both before and after the considered operation occurrence. The rule $wrap(l_1, l_2)$ means that l_2 must be applied within l_1 (where is clarified in the following).

These predicates are defined in terms of lower-level kernel predicates not dealing with links but with *link elements*: a link element is a pair $(link, control)$, where *control* is one of the control attributes[1]. For instance, b_1 (resp. a_1) is the link element of l_1 for before (resp. after) control. The first of this predicates is *ord*, which expresses sequencing between link elements, and makes it possible to define sequencing and wrapping with respect to before and after control (the rules $ord(b_i, a_i)$ are directly implemented by the kernel):

$$seq(l_1, l_2) = ord(b_1, b_2), ord(a_1, a_2)$$
$$wrap(l_1, l_2) = ord(b_1, b_2), ord(a_2, a_1)$$

So far, following [Brichau *et al.*, 2002] and [Douence *et al.*, 2002], we have only dealt with before and after control. Still, Reflex, like AspectJ and most AOP proposals, supports the possibility for an aspect to act around an execution point. In Reflex, this is *replace* control. Aspect precedence in AspectJ implies a *nesting* of advices: in the precedence chain, a nested advice is only executed if its parent around advice invokes `proceed`. Around advices cannot be simply sequenced in AspectJ: they always imply nesting, and hence their execution always depends on the upper-level around advice [Wand *et al.*, 2004]. For versatility with respect to the possible ways of composing around actions, an AOP kernel should make it possible to express any combination of nesting and sequencing. To this end, the Reflex kernel supports another kernel-level predicates, *nest*, which only applies to *replace* link elements. The rule $nest(r, e)$ means that the application of the link element e is nested in the application of the replace element r. The place of the nesting is defined by the occurrences of `proceed` within r. We can complete both *seq* and *wrap* with their semantics for replace control:

$$seq(l_1, l_2) = ord(b_1, b_2), ord(r_1, r_2), ord(a_1, a_2)$$
$$wrap(l_1, l_2) = ord(b_1, b_2), ord(a_2, a_1), nest(r_1, b_2), nest(r_1, r_2), nest(r_1, a_2)$$

Since S-links do not have a control attribute, link elements do not make sense for them. As a consequence, only *ord* is meaningful: *ord* and *seq* are equivalent, and *wrap* does not exist.

7.4.3 Specifying Resolution

The kernel predicates *ord* and *nest* make it possible to flexibly express advanced composition facilities. In Reflex, rules are defined using higher-level composition predicates, like *seq* and *wrap*, discussed hereafter.

7.4.3.1 Composition predicates

With *ord* and *nest*, a handful of user predicates for composition can be defined. Reflex makes it possible to define composition predicates on top of these kernel predicates. For instance, `Seq`

[1] This needs to be extended in order to deal with different variants of after control, such as `after throwing` as provided by AspectJ.

and `Wrap` are binary predicates that implement the *seq* and *wrap* predicates exactly as defined previously, for instance:

```
class Wrap extends CompositionPredicate {
  void expand(Link l1, Link l2){
    ord(b(l1), b(l2)); ord(a(l2), a(l1));
    nest(r(l1), b(l2)); nest(r(l1), r(l2)); nest(r(l1), a(l2));
} }
```

The methods `b`, `r`, `a`, `ord`, and `nest` are provided by the `CompositionPredicate` abstract class. The method `expand` is the place where user-defined predicates are defined in terms of kernel predicates.

Higher-level composition predicates can even mix link interaction restrictions. For instance, in EAOP, binary operators like *fst* (resp. *snd*) are proposed, expressing that if the left child applies, then the right child does not apply (resp. applies) [Douence *et al.*, 2004]. In the Reflex kernel, a predicate *snd* can be defined as a specialization of the `Seq` predicate that further uses a link interaction selector:

```
class Snd extends Seq {
  void expand(Link l1, Link l2){
    super.expand(l1, l2);
    l2.setInteractionRestriction(new DoesApply(l1));
} }
```

Similarly, it is possible to implement a wrapping variant of *snd* by extending `Wrap` instead of `Seq`.

7.4.3.2 Declaring composition rules

User composition rules are represented as abstract syntax trees consisting of predicates as nodes, and links as leaves. Reflex offers a new API for declaring composition rule. Composition need not be specified by a unique global composition rule: it can be partially specified with multiple smaller rules. When defining rules via the API, Reflex checks consistency and reports errors, such as cyclic definitions (*e.g.* b_1 then b_2 ... then b_1).

Of course, at this level no language support is provided to define rules more conveniently: they need to be manually instantiated, node by node. Language support for Reflex configuration is the subject of the next section. This support is provided for defining (domain-specific) aspect languages that make it possible to use Reflex more conveniently, at the desired level of abstraction. It is obviously possible to define languages dedicated to composition, or to define more full-fledged aspect languages that include some kind of support for composition. For instance, in supporting the AspectJ language, the notion of precedence of AspectJ is expressed with composition rule using the *wrap* predicate.

7.4.4 Composition

When detecting link interactions, the composition algorithm of Reflex generates a hook skeleton based on the specified composition rules. During this generation Reflex issues warnings whenever composition is under-specified. Users are free to ignore them and let Reflex arbitrarily compose the non-specified parts. The hook skeleton, which expresses the result of ordering and nesting relations between the links involved in an interaction, is then used for driving the hook generation process.

For instance, let us consider two links l_1 and l_2 with before-after control, both activatable. If the rule $wrap(l_1, l_2)$ is defined and an interaction between l_1 and l_2 is detected, the following hook skeleton is generated:

`if(`*determine activation for* l_1`){` *before invocation on metaobject for* l_1`}`
`if(`*determine activation for* l_2`){` *before invocation on metaobject for* l_2`}`
original code
`if(`*determine activation for* l_2`){` *after invocation on metaobject for* l_2`}`
`if(`*determine activation for* l_1`){` *after invocation on metaobject for* l_1`}`

Nesting of *replace* link elements is provided via a proceed-like mechanism similar to that of AspectJ (described in [Hilsdale & Hugunin, 2004]). This mechanism relies on the generation of closure objects, which make it possible to trigger the application of nested aspects upon invocation of a `proceed` statement.

7.5 Plugin Architecture for Open Language Support

A versatile AOP kernel makes it possible to modularly define aspect languages, either general-purpose or domain-specific, so that programmers can implement aspects at the level of abstraction that most suits their needs. Implementations of aspect languages make use of the kernel facilities, rather than creating their own from scratch. In the case of the Reflex AOP kernel, an aspect language is implemented by a translator to Reflex configuration. Translators can be easily added to Reflex as *plugins*. A plugin takes as input an aspect program written in a given language and outputs, either on-line or off-line, the adequate Reflex configuration: metaobject classes, selectors and other entities, together with calls to the kernel API.

7.5.1 Bridging the Abstraction Gap

A Reflex plugin is typically expected to bridge the abstraction gap between the aspect level and the kernel level. At the kernel level, the main conceptual handle is the notion of *links*. Though making it possible to abstract from low-level details, links are lower-level abstractions than *aspects*. As a result, an aspect is typically implemented by several links (including several hooksets and metaobjects).

For instance, consider an AspectJ aspect `Foo` that performs an introduction, and has a pointcut, associated to a simple advice, that depends on the control flow of a nested pointcut. As explained

in Chapter 5, the pointcut-advice part of the aspect results in the definition of two B-links: one for the main pointcut (bound to the advice), and one for exposing control-flow information of the nested pointcut (bound to a simple event collector). Furthermore, the introduction results in the definition of an S-link. Therefore, three links are necessary for implementing the single aspect `Foo`.

The major issue with this abstraction gap is that the composition mechanisms of Reflex (including interaction notification, composition rules, resolution and generation) work with links, not aspects. The aspect programmer does not care about links. In order to support traceability of a link back to its associated aspect-level entity, we introduce the notion of *linksets*. A linkset is, as its name suggests, a set of links that can be considered as part of the same higher-level conceptual entity. A linkset is therefore the counterpart, in the kernel world, of an entity in the aspect world. The mapping is defined by the plugin. For instance, in the case of the aspect `Foo`, an AspectJ plugin will typically embed the three links in a linkset named `Foo`. Furthermore, composition rules can deal with linksets: a linkset in a composition rule represents all the links in that linkset.

The plugin architecture supports the definition of linksets rather than stand-alone links. In this way, a link defined by a plugin can always be traced back to its linkset, which can be traced back to its plugin. Plugins implement some methods like `getSource(Linkset)` and `getSource(Rule)`, which return descriptions of the source of a linkset or rule. For instance, in the case of the aspect `Foo`, `getSource` may indicate that this linkset corresponds to an aspect defined in the file named `Foo.aj`. All errors and warnings make use of this traceability to report errors at the appropriate level of abstraction.

7.5.2 Illustration

Fig. 7.7 illustrates the overall architecture of a simple example: four aspects A, B, C and D are defined. Aspect A is expressed in an aspect language $L1$ (implemented by plugin $P1$), B and C are expressed in $L2$ (implemented by $P2$), and D is expressed in $L3$ (implemented by $P3$).

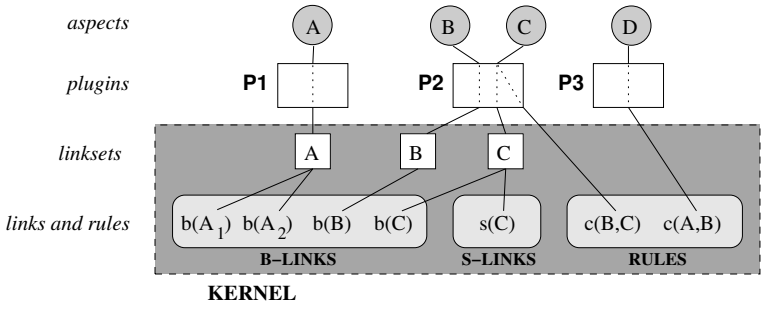

Figure 7.7 – Aspects defined in different aspect languages and their mapping in the Reflex AOP kernel.

Aspect A is implemented by a linkset A which includes two B-links $b(A_1)$ and $b(A_2)$. Aspect B results in the definition of linkset B which includes a single B-link $b(B)$. Aspect C is more

complex, as it results in both the creation of a linkset containing a B-link $b(C)$ and an S-link $s(C)$, and in the definition of a composition rule specifying composition between aspects B and C, $c(B, C)$.

Finally, when running this configuration, consider that an unexpected interaction between aspects A and B is reported by the kernel: a composition language $L3$ is then used to define, in aspect D, a composition rule between A and B (two aspects defined in different aspect languages), $c(A, B)$. Note that lines in Fig. 7.7 denote a bidirectional path.

7.5.3 Defining and Registering Plugins

Starting Reflex with command line arguments, as shown in Section 4.3.3.1, has been extended to support plugins. From the kernel point of view, a plugin is a simple *configuration handler*: an object that can understand some command line arguments and interact with the kernel API to defines links, rules, etc. Reflex supports configuration handlers in an open manner: they have to implement a very basic interface that makes it possible, in particular, to know which command line tag they handle. Configuration handlers can be either located in a specific directory (packaged as jar files), or be specified via a special command line argument. At startup, Reflex first looks up all available configuration handlers, instantiates them, and then delegates to each handler the string that was adjacent to its tag.

Still, because of the necessary support for linksets, plugins are requested[2] to use a special `PluginAPI` to define their configuration. This is to enforce them to effectively define linksets, not stand-alone links. In order to generate appropriate error messages, plugins must implement some methods like `getSource(Linkset)` and `getSource(Rule)`: these methods return string descriptions of the source of a linkset or rule definition. As an illustration, in the case of the aspect `Foo`, `getSource` may return the fact that this linkset corresponds to an aspect defined in the file named `Foo.aj`.

Finally, plugins are free to offer a runtime API via specific classes with static methods. This can either make it possible for plugins to interact together, but it can also serve the purpose of applications dynamically defining new aspects in the desired language, or dynamically interacting with existing aspects. In the latter case, the runtime API class will typically be responsible for bridging the abstraction gap between aspect-level runtime manipulation and lower-level entities such as links and metaobjects.

7.6 Discussion

Several tools for domain-specific software engineering have been proposed in the literature (*e.g.* [Batory *et al.*, 1998; Gray *et al.*, 2001]). They share our motivation with respect to the advantages of supporting multiple domain-specific languages, but do not directly deal with aspect-oriented programming. Several approaches directly relate to our work: XAspects [Shonle *et al.*, 2003], Aspect-Oriented Logic MetaProgramming (AOLMP) [Brichau *et al.*, 2002], the Concern Manip-

[2] A simple code analysis of plugins could ensure this is respected.

ulation Environment (CME) [CME, 2002; Harrison *et al.*, 2002b], and Josh [Chiba & Nakagawa, 2004].

XAspects is a plug-in mechanism for domain-specific aspect languages [Shonle *et al.*, 2003]. In XAspects, each domain-specific solution communicates with a common language, which is the AspectJ language. The authors motivate their choice of AspectJ by the possibility of better expressing crosscutting abstractions, conversely to what could be done with Intentional Programming [Simonyi, 1995] or macro systems such as Maya [Baker & Hsieh, 2002]. We share the viewpoint that an AOP kernel should intrinsically support crosscutting, but to us AspectJ is not a candidate for an AOP kernel. AspectJ is a mature, production-quality proposal of a general-purpose aspect language, but its practitioner perspective entails that too many design choices are fixed, which results in limited versatility. For instance, semantics of aspect composition is fixed; complex algorithmic cuts cannot be specified; runtime manipulation and evolution of aspects are not fully supported (additional code has to be developed each time such features are needed [David *et al.*, 2001]).

The work on XAspects clearly identifies and separates structural and behavioral transformations. XAspects proposes a 6-phase compilation process. However, it is unclear whether their crosscutting analysis is applicable to a scenario with dynamic class loading. Transformers in XAspects are provided with plain bytecode, whereas Reflex provides an object-oriented and high-level API for load-time structural introspection and intercession among all phases, based on an extended version of the object model of Javassist. Furthermore, XAspects only supports implicit collaboration between plugins, whereas Reflex offers a more flexible protocol allowing both implicit and explicit collaboration. Concerning the phase dealing with "generation of semantics" (*i.e.* behavioral changes), XAspects imposes the constraint that a domain-specific aspect can only be turned into *one* AspectJ aspect. Conversely, in our approach, the number of links involved in the implementation of one domain-specific aspect is not limited: links are embedded within linksets. Finally, being based on AspectJ, XAspects cannot provide expressive and extensible composition semantics of domain-specific aspects, nor can it provide detection of aspect interactions.

[Brichau *et al.*, 2002] presents an approach to building composable aspect-specific languages with logic metaprogramming. Aspect-specific languages are uniformly defined and composed using the same Prolog-like base language: an aspect language is implemented as a set of logic rules in a logic module. This approach is very expressive, in particular with respect to composition: it actually provides means not only to compose aspects written in different aspect languages (as in Reflex), but also to actually compose *aspect languages*. This is a very valuable feature that we would like to further explore in the context of Reflex. A drawback of this approach is that aspect languages by themselves do not really shield the programmer from the inherent power of the logic metaprogramming approach. This comes from the fact that aspects are defined in the same logic framework as the languages: no aspect-specific syntax is provided. Finally, this approach does not address detection of aspect interactions.

The Concern Manipulation Environment (CME) developed at IBM [CME, 2002] is a large-scale project aiming to support aspect-oriented software development at any level (analysis, design, implementation, etc.), with respect to any computing environment (programs in various languages, UML diagrams, etc.). The motivation for developing a flexible infrastructure with advanced build-

ing blocks to experiment with various AOSD approaches is definitely shared with our work. However, there are fundamental differences. First of all, the CME targets both asymmetric and symmetric approaches to separation of concerns [Harrison *et al.*, 2002a], while the present work only addresses asymmetric approaches. To be more precise, following the classification of symmetries presented in [Harrison *et al.*, 2002a], our proposal exhibits element asymmetry with support for aspect-aspect composition, join-point asymmetry, and partial relationship symmetry (asymmetric placement, but symmetric scope with respect to aspect). Also, the Concern Assembly Toolkit (CAT) [Harrison *et al.*, 2002b], part of the CME, aims at many different representations of software, such as Java source code, Java class files, and other representations, like UML diagrams. Though restricted to object-orientation, the great variety of target representations for CAT has a serious impact on the concern assembly language (the equivalent of the kernel language in our approach): assembly directives are usually specified open-endedly as strings. This has to be contrasted with our higher-level conceptual model used to reason about behavioral, structural and compositional issues. Moreover, composition is only considered at the level of composing class hierarchies [Snelting & Tip, 2002] or at the level of method bodies through method combination graphs. Detection and resolution of aspect interactions is not considered.

Finally, Josh [Chiba & Nakagawa, 2004] is an open AspectJ-like language, which makes it possible to experiment with new cut languages and means of describing advices. Josh is based on Javassist [Chiba, 2000; Chiba & Nishizawa, 2003], like Reflex, and hence has similar expressive capabilities. Nevertheless, Josh weaves aspects by inlining advice bodies, whereas Reflex keeps aspect behavior in separate objects. Keeping aspect behavior in separate objects is necessary in particular to support stateful aspects, and per-instance aspects (not supported by Josh). Finally, Josh lacks support for both aspect composition and collaboration. Conversely, Reflex provides automatic detection of aspect interactions and expressive means for explicitly resolving interactions among aspects, and offers a general-purpose protocol for implicit and explicit collaboration among aspects.

Summary

In this chapter, we have presented the evolution of Reflex to a Java versatile AOP kernel that relies on partial reflection, which supports expressive cuts and actions, with a reified and highly-configurable binding between both. We have discussed how the kernel fulfills the requirements related to structure and behavior, and exposed composition support in the Reflex kernel: implicit and explicit collaboration between aspects, automatic detection of interactions between aspects and expressive, extensible means for their explicit resolution. A plugin architecture makes it possible to modularly define aspect languages, and ensures backward traceability from kernel-level entities such as links to aspects.

We have therefore concretized, in an AOP infrastructure, the connection between the power of reflection and the guidance, at the language level, of aspect-oriented programming.

Chapter 8

Case Study: Sequential Object Monitors

In this chapter, we present a case study of the Reflex AOP kernel, based on the design and implementation of *Sequential Object Monitors* (SOM) for Java. Reifying monitor method calls as requests, and providing full access to the pending request queue, gives rise to fully sequential monitors: the SOM programmer gets away from any code interleaving. Moreover, useless context switches are avoided. Finally, from a software engineering point of view, SOM promotes separation of concerns, by untangling the synchronization concern from the application logic. SOM is implemented on top of Reflex, as a plugin supporting a lightweight domain-specific aspect language.

Section 8.1 presents SOM, its motivation, concepts, and design. In Section 8.2 we discuss the implementation of SOM on top of Reflex, as a plugin supporting a domain-specific aspect language. Finally, Section 8.3 shows an example of interaction between aspects defined in different languages.

8.1 Sequential Object Monitors

Programming with Java monitors is recognized to be difficult, and potentially inefficient due to many useless context switches induced by the `notifyAll` primitive. This work presents SOM, *Sequential Object Monitors*, an alternative to programming with Java monitors, a joint work with Denis Caromel and Luis Mateu. The SOM paper [Caromel *et al.*, 2004] does not really discuss the implementation of SOM, it just focuses on the contribution of SOM by itself. This section can be seen as a short version of this paper, while the following sections present the implementation of SOM with Reflex in more details.

This section starts with background information on concurrency abstractions and motivates our proposal. Then we present the main ideas of SOM (Section 8.1.2) as well as some canonical examples to illustrate the approach (Section 8.1.4). [Caromel *et al.*, 2004] includes more examples, and also illustrates SOM expressiveness with high-level abstractions like *guards* and *chords*.

8.1.1 Background and Motivation

In this section, we first quickly review the classical synchronization mechanisms, as well as the way synchronization is handled in Java. From this, we identify several issues that motivate our work on SOM.

8.1.1.1 Classical Synchronization Mechanisms

Monitors. A monitor is a language-level construct similar to a class declaration. In a monitor, private variables and public operations are declared. The semantics of the monitor ensures that concurrent invocations of operations are executed in mutual exclusion, hence avoiding data races. Monitors were invented by Brinch Hansen [Brinch Hansen, 1974] and Hoare [Hoare, 1974]. These monitors avoid thread context switches by introducing condition variables (thread queues) to explicitly resume only one thread instead of all threads. However, Brinch Hansen states in [Brinch Hansen, 1993] that such monitors are *baroque and lack the elegance that comes from utter simplicity only*.

Guards. Guards are a simple concept, easy to understand and reason about. The idea of associating a boolean expression to indicate when a given operation may be executed was first introduced for the critical region construct [Brinch Hansen, 1972]. These boolean expressions evolved to become the guarded commands of [Dijkstra, 1975] and [Hoare, 1978]. The main problem with guards is to implement them efficiently, that is, without requiring lots of thread context switches.

Schedulers. The scheduler approach relies on having an entity, called a *scheduler*, that is responsible for determining the order in which concurrent requests to a shared object are performed, similarly to the way an operating system scheduler manages the access to the CPU by concurrent processes. The scheduler approach relates to the *actor* and *active object* models[1], which focus on the separation of coordination and computation (see for instance [Agha, 1986; Frolund & Agha, 1993; Atkinson *et al.*, 1990]). They introduce the concept of a *body*: *"a distinguished centralized operation, which explicitly describes the types and the sequence of requests that the object might accept during its activity"* [Briot *et al.*, 1998]. This approach originated in Simula-67 [Birtwistle *et al.*, 1973], and has been used in several distributed object systems like POOL [America & van der Linden, 1990], Eiffel// [Caromel, 1993], and, in Java, ProActive [Caromel *et al.*, 1998].

Scheduler approaches are usually in the framework of active entities, which implies at least an extra thread for synchronization and extra context switches: a scheduler runs in its own thread of control in an infinite loop. The cost of context switches is not really an issue for systems aiming at parallel programming of distributed memory systems, since the overhead of thread context switches is hidden by network latency. On the other hand such an overhead is a concern for concurrent programming of shared memory multiprocessors.

[1] The actor model is in a functional setting, while the active object model is rather in imperative object languages.

```
(1) public synchronized Object get()
       throws InterruptedException {
(2)   while (!bufarray.size() > 0)
(3)     wait();
(4)   Object o = bufarray.get();
(5)   notifyAll();
(6)   return o;
}
```

Figure 8.1 – Guard-like code for a bounded buffer in Java.

8.1.1.2 Java Monitors

After Ada [Ichbiah *et al.*, 1991], Java is one of the first massively-used languages that includes multi-threaded programming as an integral part of the language. For synchronization, Java offers a flavor of monitors which we will refer to as *Java monitors*. They are inspired from the *critical region* concept invented by Brinch Hansen [Brinch Hansen, 1972]. The main idea behind the original critical regions is to support the *guard* programming pattern: each operation has an associated guard, a boolean expression which must be true before executing the operation. If the guard is false, the critical region transparently delays the operation until the guard becomes true.

Java monitors are somehow lower level than critical regions because the programmer must *explicitly* test the guard condition before each operation, and must *explicitly* notify waiting threads when guards must be evaluated. Fig. 8.1 shows the typical code of a guard-like implementation of the `get` method of a bounded buffer.

In Java, a monitor is a normal class definition that includes methods declared with the modifier `synchronized` (1). Concurrent invocations of synchronized methods are executed in mutual exclusion. Conversely, a guard does not have a special syntax construct in Java. It is implemented by a `while` statement where the boolean expression is the (negated) guard condition (2). The programmer must *explicitly* call `wait` (3) to suspend a thread until `notifyAll` is invoked by another thread (5).

This simple example clearly highlights the main disadvantages of the standard Java synchronization mechanism:

- Java monitors are a low-level abstraction and therefore, programmers are prone to introduce many bugs in their programs: *e.g.* forgetting to specify the `synchronized` modifier, using an `if` statement instead of a `while` for evaluating guards, wrongly using `notify` instead of `notifyAll`[2] or not invoking `notifyAll` when needed, etc.
- The application functional code (4-6) is tangled with the synchronization concern (1-2-3-5). Tangling non-functional concerns with application code is a violation of the Separation of

[2]Recall that `notify` only awakes a single thread, but using it is not recommended because in most cases it introduces subtle race conditions which are very hard to track down.

Concerns (SOC) principle [Dijkstra, 1968], and leads to less understandable, reusable and maintainable code.

- From an efficiency point of view, calling `notifyAll` is inefficient because it awakes all waiting threads. When many threads are waiting on the same lock, this entails a lot of useless, expensive, thread context switches. For instance, in the bounded buffer problem, if many consumers are waiting for an item to be produced, putting a single item in the buffer will awake *all* consumers, although only one of them can get the item (see Sect. 4.3.4.3 for benchmarks).
- Programmers frequently disregard multi-threading when defining classes. This entails that there are plenty of useful libraries including classes that are not thread-safe. Making such classes thread-safe, if at all possible, is hard and error prone.

8.1.1.3 Motivation

Overall, this work proposes an alternative for programming concurrency which aims at solving the problems mentioned above:

- **easy**: it should be a high-level and easy-to-use concurrency mechanism, ensuring thread safety;
- **powerful**: it should be expressive enough to support any concurrency abstraction;
- **efficient**: it should avoid useless threads and useless thread context switches;
- **modular**: synchronization code should be specified separately from application code, in order to achieve a clean separation of concerns to and make it easy to "plug" synchronization onto existing, not thread-safe, classes;
- **portable**: the system should be a standard Java library, not requiring a specific virtual machine.

8.1.2 Main Ideas

A sequential object monitor, SOM, is a standard object to which a low-cost, threadless, *scheduler* is attached (Fig. 8.2a). A SOM does not have its own thread of control, *i.e.* it is a *passive object*. The functional code of a SOM is a standard Java class in which synchronization does not need to be considered at all. The synchronization code is localized in a separate entity, a scheduler, which implements a *scheduling method* responsible for specifying how concurrent requests should be scheduled. A SOM system makes it possible to define schedulers and to specify which schedulers should be attached to which objects in an application.

When a thread invokes a method on a monitor, this invocation is reified and turned into a *request* object (Fig. 8.2b(1)). Requests are then queued in a *pending queue* until they get scheduled by the scheduling method (Fig. 8.2b(2)). The scheduling method can mark several requests for

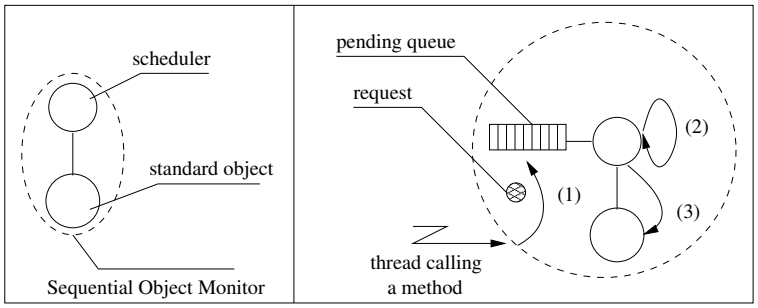

Figure 8.2 – Structure and operational sketch of a Sequential Object Monitor.

scheduling. A scheduled request is safely executed (Fig. 8.2b(3)), in mutual exclusion with other scheduled requests (and the scheduling method).

A scheduling method simply states how requests should be scheduled. Fig. 8.3 is an example, in pseudo-code, of a scheduling method specifying a classic strategy for a bounded buffer.

```
schedule method:
  if buffer empty then schedule oldest put
  elseif buffer full then schedule oldest get
  else schedule oldest
```

Figure 8.3 – Pseudo-code of a fair scheduling strategy for a bounded buffer.
(The equivalent code in SOM is shown later, in Fig. 8.8)

A SOM is a *sequential* monitor since considering thread interleaving is not necessary when writing the functional code; a method body is always executed atomically from begin to end with respect to other invocations. Conversely, in a *quasi-parallel* monitor [Kaubisch *et al.*, 1976; Briot *et al.*, 1998] (*e.g.* Hoare's, Java monitors) although only one thread can be active at a time, several method activations may coexist. Reasoning about the program is much more complex. Fig. 8.4 summarizes the main principles of SOM.

SOM provides certain guarantees, as listed in Fig. 8.5. Some of these guarantees are functional, like monitor reentrancy and execution order of scheduled requests. Others have more to do with the thread management strategy of SOM, presented in detail in Section 8.2.3.1.

The SOM model is indeed close to the active object model. The fundamental difference is that a sequential object monitor is a *passive object*, *i.e.* it has no autonomous behavior, no additional scheduling thread: a SOM is much more lightweight than an active object. Nevertheless, the synchronization mechanism of both entities are similar. Compared to existing work carried out in the context of actors and active objects, the specific contribution of SOM rather relates to two

1. *Any method invocation on a SOM is reified as a* request *and delayed in a pending queue until scheduled.*
2. *The scheduling method is guaranteed to be executed if a request may be scheduled.*
3. *A request is executed atomically with respect to other requests.*

Figure 8.4 – Main SOM principles.

1. *A SOM is reentrant, meaning that any self send on a monitor is executed immediately, without calling the scheduling method.*
2. *Given that the scheduling method can schedule several requests at a time:*
   - *After execution of the scheduling method, the scheduled requests are executed by their calling thread, in the scheduling order.*
   - *The scheduling method will not be called again before* all *scheduled requests complete.*
3. *There is no infinite busy execution of the scheduling method.*
4. *The scheduling method is executed by caller threads, in mutual exclusion. The exact thread executing this method is unspecified.*
5. *After a caller thread has executed its request, it is guaranteed to return at most after one execution of the scheduling method.*
6. *Whenever a SOM is free, if a request arrives and is scheduled by the scheduling method, the request is executed without any context switch.*

Figure 8.5 – Main SOM guarantees.

specific original points: the sequential nature of synchronizations, for simplicity, and the absence of a synchronization thread, for efficiency.

8.1.3 Main Entities and API

We now present some elements concerning the entities and the API of the SOM library, in order to go through concrete examples afterwards. In SOM, a scheduler is defined in a class extending the base abstract class `Scheduler` (Fig. 8.6). A scheduler must simply define the no-arg

`schedule()` method. This method defines the scheduling strategy. The basic idea is that a scheduler can *mark for execution* one or more pending requests, stored in a pending queue. This is called *scheduling (a) request(s)*. Such a scheduling decision may be based on request characteristics as well as the state of the associated base object (passed to the scheduler as a constructor parameter) or any other external criteria.

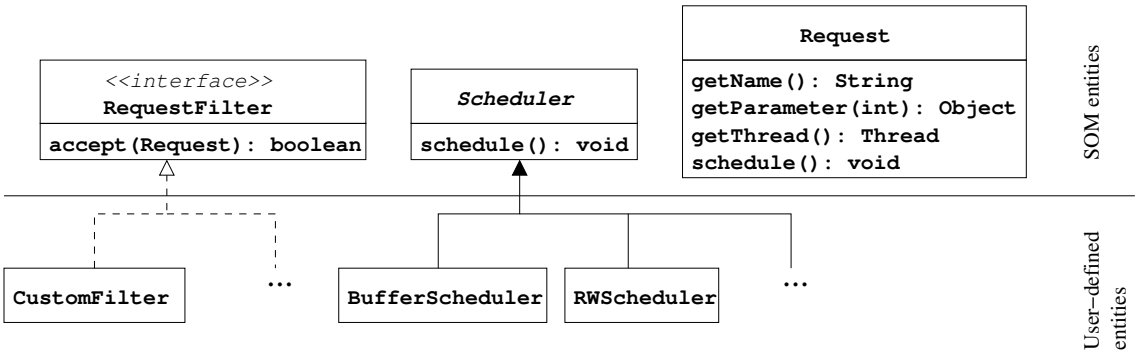

Figure 8.6 – Main entities provided by SOM, and some potential user-defined extensions.

Various methods are provided for the scheduler to express its scheduling strategies (Fig. 8.7). For instance, `scheduleOldest("put")` schedules the oldest pending request on method `put`, if any (otherwise it does nothing). Requests that are not scheduled remain in the queue to be scheduled later, on a future invocation of `schedule`. Requests are represented as `Request` objects. A scheduler can obtain an iterator on the pending queue using the `iterator()` method, and can then introspect request objects, in arrival order, to determine which one(s) to schedule. Once a request is scheduled, it is removed from the pending queue. Request objects encapsulate the name of the requested method, its actual parameters, and a reference to the calling thread (Fig. 8.6).

To express elaborated selection schemes, most scheduling methods accept *filters* as an alternative to simple request names. A request filter implements the `RequestFilter` interface (Fig. 8.6), defining the `accept()` method. For instance, `scheduleAll(rf)` will schedule *all* requests in the queue that are accepted by the `rf` filter, while `scheduleOldest(rf)` will only schedule *one* request, the oldest accepted by `rf` (if any).

Recall that scheduled requests are executed in the scheduling order. To execute requests in the original arriving order, they should simply be scheduled in that order. For instance, ensuring FIFO mutual exclusion with SOM is trivial: it is enough to attach a scheduler whose scheduling method simply calls `scheduleAll()`.

scheduling	queue management
`void schedule(Request)`	`Iterator iterator()`
`void scheduleAll`[1]	`boolean hasRequest`[1]
`void scheduleOldest`[1]	`int requestCount`[1]
`void scheduleYoungest`[1]	
`void scheduleOlderThan`[2]	
`void scheduleAllOlderThan`[2]	
`void scheduleYoungerThan`[2]	
`void scheduleAllYoungerThan`[2]	

- [1] Method available in various overloaded versions:
  - `()`: apply to all requests in the queue
  - `(String)/(String[])`: apply to request(s) with given name(s)
  - `(RequestFilter)`: apply to request(s) accepted by filter
- [2] Method available in 2 overloaded versions, taking either two `String` or two `RequestFilter` parameters.

Figure 8.7 – Scheduler API for scheduling and queue management.

8.1.4 Canonical Examples

We now briefly present SOM solutions to some classical concurrency problems: bounded buffer and readers and writers.

8.1.4.1 Bounded buffer

Fig. 8.8 presents the implementation in SOM of a scheduler for the bounded buffer example. It is a straightforward mapping of the pseudo-code shown previously in Fig. 8.3. Class `Buffer` is a trivial, unsynchronized, implementation of a buffer (not presented). In this implementation, when the buffer is neither full nor empty, the oldest request is scheduled (`scheduleOldest`). Now, imagine we use the following schedule method instead:

```
public void schedule() {
  if (!buffer.isEmpty()) scheduleOldest("get");
  if (!buffer.isFull())  scheduleOldest("put");
}
```

In this case, when the buffer is neither full nor empty, it alternates serving `get` and `put` requests, not respecting the order. This calls for the following comment. The SOM abstraction provides the user with the ability to finely control and tune the synchronization if needed. Of course, higher-level abstractions, potentially with good non-determinism, are also needed. They will be expressed on top of the basic SOM primitives (see [Caromel *et al.*, 2004] for guards and chords).

```
public class BufferScheduler extends Scheduler {
  Buffer buffer;
  public BufferScheduler(Buffer b) {
    super(b);
    buffer = b;
  }
  public void schedule() {
    if (buffer.isEmpty()) scheduleOldest("put");
    else if (buffer.isFull()) scheduleOldest("get");
    else scheduleOldest();
} }
```

Figure 8.8 – Scheduler for the bounded buffer example.

8.1.4.2 Readers and writers

The readers and writers problem is another classical problem of concurrent programming. Readers are threads that query a given data structure and writers are threads that modify it. A coordinator object `c` is responsible for granting access to the data structure. Readers request access by calling `c.enterRead()` and notify when they stop accessing data with `c.exitRead()`, while writers use `c.enterWrite()` and `c.exitWrite()` respectively. This problem is easily solved by making the coordinator a sequential object monitor. The code of the solution is presen-

```
public class RWCoordinator {
  int readers = 0;
  boolean write = false;

  void enterRead(){readers++;}
  void exitRead(){readers--;}

  void enterWrite(){write = true;}
  void exitWrite(){write = false;}

  int getReaders(){return readers;}
  boolean isWriting(){return write;}
}
```

```
public FairRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    scheduleAll(new String[]{
       "exitRead", "exitWrite",
       "getReaders", "isWriting"});

    if(!c.isWriting()) {
      if(c.getReaders() > 0)
        scheduleOlderThan(
         "enterRead","enterWrite");
      else scheduleOldest();
} } }
```

Figure 8.9 – The coordinator and its associated scheduler.

ted in Fig. 8.9. The functional part is the coordinator implementation, which is self-explaining.

The code of the scheduler specifies the following strategy. First, `exitRead` and `exitWrite` requests are scheduled immediately and unconditionally, because they are just notifications, not requests for access – similarly for `getReaders` and `isWriting` requests.

If a writer is currently modifying the data structure, the scheduler does not grant other permissions for access. If there are readers accessing the data structure, it grants permission to another `enterRead`, if any. Finally, if there is currently no access to the data structure, it schedules the oldest request.

Note that readers are scheduled by calling `scheduleOlderThan`, not `scheduleOldest`. This is to ensure that writers may not starve: an `enterRead` request is scheduled *only if it is older* than the first `enterWrite` in the pending queue.

Also, `schedule` only schedules one pending `enterRead` at a time (call to `scheduleOlderThan`). This does not mean that two or more readers cannot work in parallel. Indeed, when finishing the execution of `enterRead`, `schedule` will be reinvoked and another `enterRead` may be scheduled for execution, even if current readers have not called `exitRead`.

8.1.5 Modularity and Reuse of Synchronization Policies

SOM makes it easy to define various synchronization policies, thanks to the full access given in the scheduling method to the queue of pending requests. For instance, in the case of the readers and writers problem, several fairness policies can be devised. We already exposed (Fig. 8.9) a fair policy, where both writers and readers are ensured not to starve. Alternative policies can easily be provided, for instance giving priority to readers or writers (Fig. 8.10).

```
public WriterPriorityRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    // schedule all notifications

    if(!c.isWriting()) {
     if(hasRequest("enterWrite")){
      if(c.getReaders() == 0)
       scheduleOldest("enterWrite");
     }
     else scheduleAll();
} } }
```

```
public ReaderPriorityRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    // schedule all notifications

    if(!c.isWriting()) {
     if(hasRequest("enterRead"))
       scheduleAll("enterRead");
     else if(c.getReaders()==0)
       scheduleOldest();
} } }
```

Figure 8.10 – Alternative policies for the readers and writers problem.

Reuse of synchronization policies in different contexts depends on their genericity. The schedulers we have exposed so far all depend on string names (*e.g.* `"put"`). A scheduler class can be

made independent from actual method names through configuration. For instance, considering buffer-like containers, reusable policies just need to be configured in order to know which methods are to be considered *put* methods and which ones are *get* methods. Then, the `schedule` method can be made independent of method names, for instance (`putMethod` is an instance variable configured to hold the name of the *put* method):

```
void schedule() { if (buffer.isEmpty()) scheduleOldest(putMethod); ... }
```

Determining emptiness and fullness of the synchronized data structure can also be made generic: the reflection API can be used to invoke emptiness and fullness methods according to configuration. Apart from being reusable, generic synchronization classes are more robust with respect to changes.

8.2 SOM with Reflex

The standard implementation of SOM takes the form of a plugin for the Reflex AOP kernel. The plugin supports a very simple domain-specific aspect language for configuring SOM without bothering about its implementation at the kernel level. The plugin also offers a runtime API for configuring SOM dynamically. In this section, we present the DSAL for SOM, the runtime API, and then discuss the implementation of the SOM plugin. Finally we present benchmarks of the implementation, which confirm the expected efficiency of SOM.

8.2.1 A Domain-Specific Aspect Language for SOM

SOM programmers surely do not want to directly configure Reflex to use SOM. The SOM DSAL makes it possible to express a SOM configuration easily. In fact, the basic configuration needed for SOM simply consists in specifying which class should be scheduled by which scheduler. Hence, programmers should not bother with link definitions, metaobject instantiation and so on. Hence we designed a lightweight DSAL for SOM, which simply consists of statements of the form:

`schedule:` $class\_name$ `with:` $scheduler\_name$`;`

This statement specifies that instances of the class named $class\_name$ should be sequential monitors scheduled by instances of the scheduler class named $scheduler\_name$. The language also makes it possible to directly specify the scheduling method of the scheduler, rather than a scheduler class as such:

`schedule:` $class\_name$ `withBody:` $schedule\_body$`;`

Using such a declaration, an anonymous scheduler class, subclass of `Scheduler`, is generated with $schedule\_body$ as the body of its scheduling method.

8.2.2 SOM Runtime API

A runtime API is also exposed by the plugin, via the `SOM` class. This API wraps Reflex functionalities that are useful in the case of SOM. Most importantly, this API makes it possible to explicitly create a particular instance that behaves as a monitor:

```
Vector v = (Vector) SOM.newMonitor(Vector.class, aScheduler);
```

This uses the ability of Reflex to dynamically create an implicit reflective subclass (with the classical restrictions associated to this approach). The SOM API can also be used to dynamically change the scheduler associated to a monitor:

```
SOM.setScheduler(v, anotherScheduler);
```

In addition, the API includes a method that corresponds to the `schedule:With:` statement of the SOM DSAL:

```
SOM.scheduleClassWith(aClassName, aSchedulerClass);
```

Again, due to the implementation approach of Reflex, if the specified class has already been loaded, this configuration statement will not affect it.

8.2.3 SOM Plugin

The SOM plugin is structured as follows:

- A set of classes implements the SOM library as such: in particular, the class `Scheduler` is the abstract class for user-defined schedulers; other classes include the `Request` and `RequestQueue` classes for explicit request management by schedulers.
- The class `SOMPlugin` implements the `Plugin` interface from the plugin architecture of Reflex. In addition, it offers the configuration methods needed for performing link definitions and the like.
- The classes `SOMLexer` and `SOMParser` are automatically generated from the simple SOM DSAL grammar using ANTLR[3]. Actions of the parser are invocations of the `SOMPlugin` configuration methods.
- The class `SOM` implements the runtime API as a set of static methods that simply delegate to the appropriate methods of `SOMPlugin`.

8.2.3.1 Scheduler class

In SOM, the base objects that should be turned into sequential monitors have their method invocations controlled by a metaobject, which plays the role of the scheduler. The `Scheduler`

[3] `www.antlr.org`

class is an abstract metaobject class that serves as the base class for schedulers in SOM. This base class is a thread-safe and efficient implementation of a scheduler, which reifies invocations as requests, manages request queues, and delegates to subclasses the precise scheduling strategy (implemented in the scheduling method of subclasses). A SOM metaobject has an `enter` and an `exit` methods, which are called, respectively, before and after invocations of public methods of the monitor. We now explain how the scheduler manages threads efficiently, avoiding unnecessary context switches.

The scheduler metaobject handles two queues:

- The *wait queue* holds pending requests that have not been scheduled for execution.
- The *ready queue* keeps pending requests that have just been scheduled (during the last execution of the scheduling method), but have not been executed yet because the monitor is busy, either still executing the scheduling method, or executing another request.

A sequential object monitor M works as follows. Let T be a thread that has begun the execution of a request R on a method of M. Suppose the ready queue already contains requests made by other threads. T is said to *own* M and the other threads *wait*. M is in a *busy* state. When T finishes the execution of the method, the scheduler takes control and extracts the oldest request R' in the ready queue. Thread T thereby passes the ownership of M directly to thread T', the thread requesting R'. Finally, T wakes T' up and returns to the caller of M. T' starts the execution of its own request, R'.

When the ready queue is empty, the scheduler makes thread T automatically invoke its method `schedule` (the body of which is defined in a user-specific subclass of `Scheduler`). Recall that this method will schedule one or several requests; these requests will be transferred from the wait queue to the ready queue. Making T invoke the scheduling method implies that T spends some time scheduling requests for other threads. Thus programmers should preferably write simple scheduling methods. If after invoking the scheduling method, the ready queue is still empty, the sequential object monitor is said to be *free*.

Let us now consider a thread T requesting a method of M. First, the request is put in the wait queue. If M is busy, T is blocked, provoking a context switch. If M is free, the scheduler makes T invoke its scheduling method. If the request is scheduled, T takes ownership of M and executes the method immediately, *i.e.* no context switch occurs. If the request is not scheduled, M remains free and T is blocked.

8.2.3.2 Plugin class

We now present how `SOMPlugin` configures Reflex appropriately. The implementation of the method `scheduleClassWith` of the plugin is given in Fig. 8.11. Stand-alone capitalized variables (such as `SCHEDULER_TYPE`, `ENTER`, etc.) are constants defined in the plugin class. In (1), a hookset `hs` is created, matching execution of the public methods of the class named `aClass`. Then, link `l` is created, binding instances of the scheduler class named `aScheduler` to the hookset `hs` (2). The argument `true` specifies that the constructor of the scheduler will be given a

```
    void scheduleClassWith(String aClass, String aScheduler){
(1)   Hookset hs =
         new PrimitiveHookset(MsgReceive.class, new NameCS(aClass),
                              new PublicOS());

(2)   BLink l =
         PluginAPI.links().getBLink(hs,
                                    new MODefinition.Class(aScheduler,
                                                           true));

(3)   l.setScope(Scope.OBJECT);
      l.setControl(Control.BEFORE_AFTER);
      l.setInitialization(Initialization.EAGER);
      l.setDeclaredType(DeclaredType.get(SCHEDULER_TYPE));

(4)   l.setMOCall(Control.BEFORE,
                  new CallDescriptor(SCHEDULER_TYPE, ENTER,
                                     new Parameter[]{ PAR_NAME,
                                                      PAR_ARGS }));

      l.setMOCall(Control.AFTER,
                  new CallDescriptor(SCHEDULER_TYPE, EXIT));

(5)   LINKS.put(aClass, l.getRTLink());

(6)   Linkset ls = new Linkset(this, "SOM_"+ aClass);
      ls.add(l);
      PluginAPI.links().addLinkset(ls);
    }
```

Figure 8.11 – Implementation of the `scheduleClassWith` method.

reference to the base object. In (3), link attributes are set: a link in SOM is per object, acts before and after, is initialized eagerly, and the metaobject is always of the scheduler type.

In (4), a specific MOP descriptor is specified. Indeed, in order to retain one of the main advantages of SOM –efficiency thanks to no useless context switches–, the SOM metaobject protocol must be efficient: only needed information should be reified. We therefore use a specific MOP descriptor that ensures that only required information is reified and passed to metaobjects: the `enter` method only requires as a parameter the name of the called method and its parameters, while the `exit` method does not take any parameter.

Then, in order to support dynamic configuration of the link, (through `setScheduler`, presented hereafter), the runtime link is kept in a map `LINKS` (5). Finally, a linkset containing `l` is defined and registered via the `PluginAPI` (6).

8.2.3.3 Runtime API

We now explain how a runtime API service is implemented. During runtime, it is possible via the `setScheduler` method to change the scheduler associated with a given monitor. Recall that this simply means changing the metaobject associated with a base object for a given link, considering that the base object is the sequential object monitor, its metaobject the scheduler, and the link the SOM link set up via configuration. The API class `SOM` simply delegates to the `SOMPlugin` class, which hence implements the `setScheduler` method, as shown in Fig. 8.12.

```
    void setScheduler(Object aTarget, Scheduler aSched){
(1)   RTLink link = LINKS.get(aTarget.getClass().getName());
(2)   if(link == null)
        throw new RuntimeException(aTarget + " is not a SOM");

(3)    link.setMetaobject(aTarget, aSched);
    }
```

Figure 8.12 – Implementation of the `setScheduler` method.

First, the map of SOM links (filled by `scheduleClassWith` as shown in Fig. 8.11(5)) is queried to obtain the runtime link object corresponding to the class of the object passed as a parameter (1). If no such link is found, this is because the given object is not a sequential object monitor, therefore an exception is thrown (2). Finally, the runtime link object (which acts as a localized runtime API for metaprogramming, recall Section 7.2.3) is used to change the metaobject of the given base object (3).

8.2.3.4 Registration

The SOM plugin, packed in a `som.jar` archive, is automatically registered with Reflex if placed in the appropriate directory (otherwise, a command line argument is used). Once registered, it is possible to apply a SOM configuration using Reflex via the `som` tag:

```
(1) java reflex.Run -som som.conf App
(2) java reflex.Run -som "schedule: A with: S" App
```

This will run `App` with Reflex, applying SOM as specified. In (1), the file `som.conf` is passed to the SOM plugin, and should contain valid SOM DSAL statements. In (2), such a statement is directly passed in the command line argument. Plugins typically support both means of input. A straightforward future work includes the possibility to have statements for different languages in a common file, as can be done in XAspects [Shonle *et al.*, 2003], where a common syntax makes it possible to specify which plugin should handle which part of the file.

8.2.4 Micro-Benchmarks

This section presents measurements of the execution time of three different buffer implementations; the typical solution using legacy Java monitors, as advised in the Sun Java tutorial [SUN, 2003], a "smart" solution using condition variables and mutexes as presented in [Lea, 1999], and the solution using SOM (as in Fig. 8.8).

The measurements are given for a buffer of one slot, with one producer, and with different number of consumers. As the interest is measuring the cost of the synchronization, the time to produce and consume items in the tests is marginal. The results (Table 8.1) were obtained by performing five measurements –each of which consists in the production of 100,000 items– discarding the best and worst cases and taking the average of the remaining three measurements. The benchmarks were executed on a single processor Athlon XP 2600+ machine with 512 MB of memory, with Java 1.4.2 with native threads. We allocated a large heap size to the JVM in order to limit the number of garbage collections. We run the benchmarks under Linux, kernel 2.4 (Table 8.1)[4]. The micro-benchmarks presented here do not take into account the cost of load-time transformation done by Reflex for introducing hooks for SOM: classes were transformed statically before the benchmarks.

number of consumers	*Java monitors*	*Condition Variables*	*SOM*
1	1006	1905	1656
2	1225	2018	1708
4	1918	2276	1891
8	5723	2412	2125
16	16005	2451	2435
32	49767	2659	3156
64	133612	2946	4407
128	358218	3049	6653

Table 8.1 – Benchmark results under Linux 2.4 with JDK 1.4.2 (time in ms).

The case with one consumer is a best case for the Java monitors solution, because when the producer puts an item in the buffer, `notifyAll` always wakes up one thread only, and this thread will get the item. Hence no useless context switches occur. The SOM solution and the solution based on condition variables are slower in this case because they are implemented with multiple Java monitors, and therefore there is an associated overhead.

Increasing the number of consumers while keeping a single producer is greatly disadvantageous for the Java monitors solution, because when the producer puts an item in the buffer, several consumers must be waken up, although only one will get the item. The others will be put to wait again. Each failed wake up is a useless context switch, which is expensive in terms of execution time. We can easily see that the SOM solution scales much better with respect to the number of consumers, because *(i)* only one thread is waken up, and *(ii)* the evaluation of which thread to

[4]The ECOOP paper [Caromel *et al.*, 2004] contains more benchmarks, under different platforms.

wake up is done by the thread leaving the monitor: no useless context switches occur.

The solution based on condition variables scales similarly well, and performs slightly better. With SOM, increasing the number of consumers lowers performance because the evaluation of the scheduling method depends on the size of the pending request queue (due to iterations over the queue). In contrast, the condition variables implementation is independent from the number of consumers, but still its performance slightly decreases because context switches seem to cost more when more threads are running.

An interesting point is that a straightforward implementation of the buffer with SOM (recall the simplicity of Fig. 8.8) brings better performance than the standard Java monitors solution, and comparable performance to the one with condition variables, which is more complex to correctly design and program. Hence, the micro-benchmarks presented here validate the interest of the SOM approach: although SOM implies an overhead at start, it scales very well.

It has to be highlighted that benchmarks were first carried out using the standard MOP of Reflex, which is far from efficient since it reifies all information in a generic manner. The bad resulting efficiency compared to hand-written modifications were actually one of the strong validation of the interest of MOP specialization from an efficiency viewpoint. With MOP descriptors (and the optimization related to the declared type of the metaobject), Reflex generates the same bytecode as the hand-written version.

8.3 Interaction Example

Providing SOM on top of the Reflex AOP kernel presents several advantages. First, once the metaobject (abstract `Scheduler`) class is defined, it is really quick to obtain the complete plugin, which is then easy to use. But also, doing so implies that, when SOM is used in conjunction with other kernel-supported approaches, interactions among aspects are detected and can be resolved adequately. This section illustrates such an interaction between a SOM aspect (with the identifier `SOM`) and a simple profiling aspect (with the identifier `Profiler`).

The profiling aspect is supported by a profiler plugin for Reflex that simply makes it possible to specify methods in a class that should be timed. The result is that the execution time of such methods is logged to the console. The profiler metaobject used to time methods is bound to the specified cut (methods) by a before-after link.

If both aspects happen to apply at the same point, Reflex detects such an interaction and tries to solve it. When it determines that no rules have been specified that could help resolve such interaction, it issues a warning:

```
[WARNING] unspecified composition for: SOM - Profiler
[WARNING] composing arbitrarily (sequence)
```

In this case, Reflex arbitrarily uses sequence between both links. Unfortunately, sequence as provided by the seq operator does not yield any meaningful result: since both aspects are implemented with before-after links, the result of seq is an interleaving of both links (Fig. 8.13). As a consequence, the time measured by the profiling aspect does not correspond to any interesting

measurement since it is a interleaving of different partial actions.

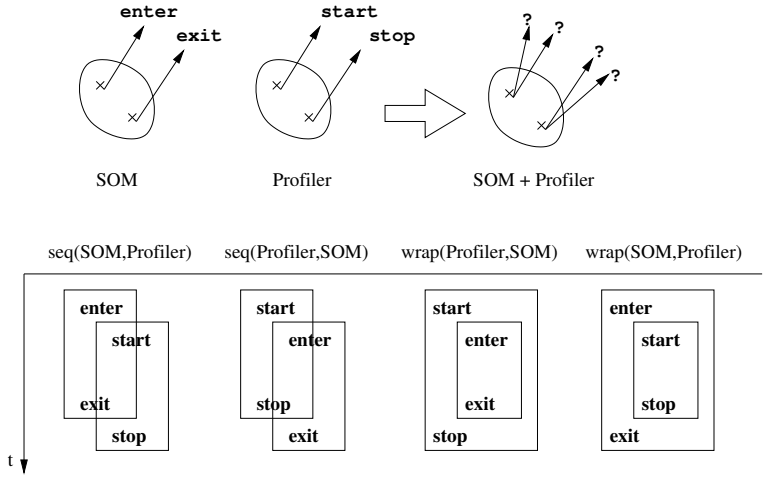

Figure 8.13 – Interaction between a SOM aspect and a profiling aspect.
The two aspects act before and after a method: SOM calls `enter` and `exit`, while Profiler calls `start` and `stop`. There are 4 ways of resolving this interaction in order for both aspects to apply: only the two last are meaningful.

As a matter of fact, in this interaction, two interesting profiling semantics can be considered: *(a)* measuring the whole execution time of a method, including scheduling and suspension time, or *(b)* measuring only method execution (hence not measuring SOM's cost at all). The composition semantics of *(a)* can be obtained by specifying $wrap(Profiler, SOM)$, while that of *(b)* can be obtained with $wrap(SOM, Profiler)$ (Fig. 8.13). Such a specification can be done directly using the Reflex API:

```
API.rules().addRule(new Wrap("Profiler","SOM"));
```

When running Reflex again with this supplementary specification, the interaction between corresponding links is automatically resolved. Hook generation consequently inserts hooks in the proper order, leading to the expected meaningful semantics.

Summary

In this chapter, we have exposed the case study of providing SOM on top of the Reflex AOP kernel. First, we have presented SOM, the principles and main design elements, as well as few illustrative examples of its use and expressiveness. Then, we have explained how SOM is implemented in Reflex, as a plugin offering both a runtime API and a lightweight domain-specific aspect language. We have shown that the resulting implementation of SOM is efficient: this efficiency was made possible by the particular thread management policy of SOM, and not compromised by the

use of Reflex thanks to a specific MOP descriptor. Finally, we have illustrated a simple interaction between two aspects defined in different aspect languages, showing how Reflex detects such interaction and makes it possible to resolve it.

Basing our implementation of SOM on the Reflex AOP kernel was a first positive indicator for our approach: it was quick to implement, compared to the range of features that could directly benefit to SOM, and exposes satisfying efficiency and scalability. Finally, relying on the AOP kernel makes it possible to use SOM with other kernel-supported approaches, being sure that interactions are detected and that resolving them is possible.

Part III

Conclusions

Table of Contents

Chapter 9

Contributions

This chapter reviews the main contributions of this thesis. Our contributions can be divided in four different topics:

- **reflection**: we have proposed a comprehensive model of partial reflection, integrating structural and behavioral reflection and addressing the performance and flexibility issues of the standard approaches (Section 9.1),
- **aspect-oriented programming**: we have shown how the previous model can be used as a basis for a versatile AOP kernel (Section 9.2),
- **open implementation**: we have concretized the previous ideas into a reflective extension for Java built according to open implementation principles (Section 9.3),
- **applications**: we have validated our ideas and our prototype on various case studies in different domains (Section 9.4).

9.1 Model of Partial Reflection

As reviewed in Chapter 2, traditional reflective models adopt a restricted view on the metalink, usually by coupling metaobjects to individual base entities or kind of operations. In general, this limited flexibility of the metalink prevents the metalevel from adopting a well-adapted locality with respect to the base-level concern it is tackling. The model of partial reflection we have presented in this thesis completely decouples the metalink from base entities and operations [Tanter *et al.*, 2003]. Although such a coupling can still be obtained, more flexible solutions can be designed. Base entities and operations may be seen as orthogonal axes of a plane, and a metalink can cover any subregions of this plane. Our model introduces links as first-class entities representing the binding between base programs and metaobjects. The precise definition of links addresses efficiency by applying reflection only where and when needed, implementing a form of partial reflection. It also tackles complexity by making it possible to flexibly structure the metalevel. Links allow our model to adequately support aspect-oriented programming, since they provide the

basis for proper metalevel engineering. As a matter of fact, metalevel engineering requires total control over locality and granularity of metaobjects. Moreover, links make it possible for various metalevel configurations to coexist.

We have followed an iterative path to reach easy and fine-grained specialization of metaobject protocols. We have finally arrived to a point where MOPs can be locally specialized on a per link (even on a per nested hookset) basis, thereby further extending the work on fine-grained MOPs reviewed in Chapter 2. Apart from allowing more efficient behavioral reflection by reifying only necessary information, MOP specialization complements our model of partial reflection in the perspective of enhanced metalevel engineering. As a matter of fact, MOP specialization makes it possible to find the right tradeoff between genericity (and complexity) of metaobjects and their specificity (and simplicity). We believe that issues arise when there is a mismatch between the desired level of genericity/specificity and the one provided (or enforced) by a particular technology or approach. Fine-tuning MOPs makes it possible to adjust genericity/specificity as required and is therefore a crucial property in order to support both the genericity of reflective systems and the specificity of AOP.

Finally, we have integrated structural reflection to our initial model of partial behavioral reflection. This general model of partial reflection has then been instantiated in the standard environment of the Java programming language. Particularities of Java, such as the restrictions concerning the time at which changes to class definitions can be done, have led us to establish a distinction between structural links applied at load time and behavioral link applied at runtime.

9.2 AOP Kernels

The second major contribution of this thesis lies in the identification of the need for and the analysis of versatile kernels for aspect-oriented programming. A systematic review of AOP approaches has led us to motivate the need for such substrates and to identify key requirements that should be met [Tanter & Noyé, 2004a; Tanter & Noyé, 2004b]. We have argued that our model of partial reflection is appropriate as an underlying model of an AOP kernel due to its support for flexible metalevel engineering. Based on our requirement analysis, open support for aspect languages and composition have been studied in the framework of our reflective model.

This has led us to propose basic building blocks for a versatile AOP kernel. In order to bridge the abstraction gap between aspect-level artifacts and kernel-level ones, we have developed a lightweight plugin architecture for defining aspect languages, either general purpose or domain specific. With respect to composition, our proposal addresses both collaboration between aspects at the structural level and composition upon weaving. Our general-purpose collaboration protocol allows for controlled visibility of structural changes to programs. Concerning composition at the weaving level, links are useful low-level units for the detection of interactions and their resolution. With respect to resolution, we have identified three elementary composition relations, supported by the kernel, from which higher-level composition operators can be defined. Finally, thanks to the traceable refinement relation between aspects and reflective artifacts such as links, our approach makes it possible to compose aspects written in different aspect languages, at the appropriate level of abstraction.

This part of our work suggests a particular view on the relation between reflection and AOP, in which the power and generality of reflection is combined with the guidance of (domain-specific) aspect languages.

9.3 Open Implementation

From the point of view of open implementations, this thesis brings insights on how to open reflective systems. Our quest for opening reflective systems has been reflected in the iterative development process that we have followed, progressively decoupling more and more concepts within the system, thereby refining customization interfaces. For instance, in the first version of our architecture [Tanter *et al.*, 2001], operation support was not explicitly decoupled from infrastructure and low-level transformations; this was solved in the second version where links were introduced, but still, MOP specialization was bound to operation support [Tanter *et al.*, 2003]; finally, operation support and MOP specialization have been decoupled, offering finer-grained interfaces, and making it possible to control MOP specialization at the link level [Rodríguez *et al.*, 2004]. It is interesting to notice that it is precisely through this iterative decoupling and refinement that connections to AOP have been clarified.

This work has resulted in the implementation of Reflex, a portable and open reflective system for Java, which progressively evolved into an AOP kernel. At the time this thesis work ends, we have reached a reasonably open implementation of Reflex, whereby only few functionalities are hardwired. Most features are provided via proper interfaces to the system, based on well-known design patterns, constituting open doors through which the system can be fine-tuned and extended. Various open implementation interface designs are present, since some decisions can tolerate user code to be specified, while others require openness to be limited to a selection of predefined strategies. Finally, Reflex is distributed under an open source license in order to foster feedback and improvements.

9.4 Applications

Implementing Reflex for Java has not been the only concrete validation of this thesis. Several case studies and applications have been carried out, in different domains. Applications in distributed systems have first exercized our approach with respect to expressiveness and flexibility. In particular, we have worked on the adaptation of migration policies in mobile object systems [Tanter & Piquer, 2001; Tanter, 2002; Tanter *et al.*, 2002b] and proposed a flexible solution to the Java implementation of transparent futures in distributed object systems with asynchronous messaging [Tanter *et al.*, 2003]. We also studied the application of our approach to runtime inspection and software evolution [Tanter & Ebraert, 2003; Ebraert & Tanter, 2004].

A case study in supporting dynamic crosscutting as provided by a general-purpose aspect language like AspectJ has helped in enhancing our model of partial behavioral reflection for supporting AOP [Rodríguez *et al.*, 2004]. Furthermore, in order to experiment with domain-specific AOP, we have significantly contributed to both the design and implementation of the proposal of

Sequential Object Monitors (SOM) [Caromel *et al.*, 2004]. The demand of SOM in terms of efficiency has made it possible for us to confirm the value of MOP specialization in this perspective. Also, providing a lightweight domain-specific aspect language for SOM has given us initial feedback on our AOP kernel with respect to both domain-specific aspect languages and composition of aspects written in different languages.

Chapter 10

Perspectives

Several aspects of the contributions of this thesis directly open perspectives for future work. This chapter briefly discusses the most important ones. Apart from the perspectives discussed here, which originate from our contributions, a number of related research directions are worth exploring. In particular, reflective and aspect-oriented analysis and design, as well as a formal treatment of the subject, are promising perspectives.

The structure of this chapter follows the structure of the previous chapter: perspectives regarding the model of reflection and AOP kernels are first discussed in Sections 10.1 and 10.2, while Sections 10.3 and 10.4 summarize perspectives with respect to the open implementation of Reflex and applications, respectively. Some of the issues mentioned here represent on-going work.

10.1 Model of Partial Reflection

The model of partial reflection we have proposed is independent of the (OO) programming language. But still, we have instantiated it in the standard Java world: several compromises had to be found, and therefore the implementation and the various refinements we have proposed are dependent on the language and its execution environment. Instantiating our model in the C# language would certainly be very similar, since this language follows the same fundamental design rationale as Java. Instantiating it in Smalltalk would be similar as well, because of the fact that Smalltalk is also based on an intermediate bytecode language. Nevertheless, this would result in a much more flexible platform, since Smalltalk does not have static typing and allows for dynamic recompilation of code fragments.

Today, we feel that it would be worthwhile studying a minimal version of our model in a pure, uniform, and dynamic context. An envisioned setting is to work on a Scheme interpreter for the model, in order to make its core semantics clearly independent of the concern of efficiency and the particularities of the Java environment. We can postulate that such an experiment will not affect in a fundamental manner our model. Similarly to the experiment of Masuhara *et al.* on the Scheme interpreter for the Pointcut-Advice model [Masuhara *et al.*, 2003], core elements of our model like links and hooksets will be handled explicitly in the interpreter. This will bring a lot of flexibility

to explore dynamicity very conveniently. Furthermore, in this work we have been concerned with efficiency and have addressed issues related to staging in various places, for instance by providing different selection mechanisms with different binding and evaluation times. Since staging has been dealt with in an ad hoc manner, the resulting design (and implementation) is still fairly complex. We could rather address efficiency and staging by resorting to specialization techniques like partial evaluation, similarly to what is done in [Masuhara *et al.*, 2003].

Finally, the projection of our model in a completely static environment also represents a challenge. In some sense, our approach is "in between" the fully static and the dynamic world, taking advantage of dynamic class loading in Java. In a pure compile-time approach, the dependencies that we introduce, via links, between otherwise unrelated classes should be properly tackled. In particular, classes involved in a link have a reference to the runtime representation of that link: such a reference should be maintained in the static class definition, and properly bootstrapped. Moreover, successive off-line transformations also raise particular issues. Since in such a context there is not a single point in time at which links are applied or installed anymore, it becomes necessary to maintain more information across transformations if appropriate detection and feedback is to be provided for collaboration and composition.

10.2 AOP Kernels

With respect to AOP kernels, a line of study lies in the detection of interactions at a finer-grained level than what we have been considering in this thesis. For instance, structural links are said to interact when they apply to the same class: however, they could very well perform orthogonal changes to that class. A finer-grained detection would require considering interactions at the member level rather than at the class level. Subtle interactions, such as member renaming, should also be detected and reported.

A major area of future work related to AOP kernels is the design of domain-specific aspect languages (DSALs). In this regard, the existing body of work on domain-specific language design and implementation represents a valuable source of know-how. Studying various DSALs also opens the door to further experimentation with respect to interactions among aspects written in different languages. This would surely make it possible to gain insight on composition and interaction issues that naturally appear in large-scale applications.

Also, the simple plugin architecture we have proposed only allows for composition of aspects written in different languages: it does not make it possible to compose plugins together, in order to build higher-level aspect languages from lower-level ones. Although basic plugin composition may be relatively straightforward to achieve, more advanced composition will probably imply exploring composition of language grammars.

In the perspective of an interpreter-based implementation of our proposal, an AOP kernel would be an integral part of the interpreter. We can conjecture at this stage that the fundamental elements related to composition and aspect language support would be preserved.

10.3 Open Implementation

The implementation of Reflex also calls for further enhancement. First of all, the current implementation has several minor limitations (*e.g.* the lack of support for the *after throwing* control), most of which have been mentioned in the previous chapters. Futhermore, we believe the openness of Reflex can still be improved by refining the design and pushing open implementation guidelines to their limit. From our experience, we are convinced that an interesting side effect of this effort would be the reduction in size of the Reflex core. As a matter of fact, continuous refactoring and enhancement of Reflex should be carried out in parallel with future applications and experiments.

As a new version of Java (1.5) is coming, Reflex should be upgraded to support its new features. The new code instrumentation support is of particular interest. Most importantly, Java will now support annotations on packages, classes, and members. Annotations will provide an explicit means for developers to qualify certain elements of their programs. In particular, specifying the cut of an aspect will possibly be done using annotations. Supporting annotations in Reflex is fairly straightforward: after all, annotations are just introspectable characteristics of structural elements. Therefore, class selectors will remain: they will just have the additional possibility of basing their criteria on annotations. Only the object model of Reflex shall be updated, in order to expose getters and setters on annotations. The underlying bytecode manipulation tool used by Reflex, Javassist, is already being extended to support this.

Finally, this work on the open implementation of an AOP kernel in which staging is considered raises issues with regard to means of describing complex architectures. On the one hand, openness introduces different levels at which an architecture can be adapted (the customization interfaces). On the other hand, staging also provides several levels, this time referring to the possibility for some decisions to be bound and evaluated at different times. These two views actually *crosscut* each other as well as the basic functional description of the architecture, complexifying the task of describing the system. At the time being, there seems to be a lack of support for untangling architectural descriptions of systems.

10.4 Applications

With respect to applications, two major tracks should be pursued.

First, as the kernel represents a potentially useful tool for hybrid AOP, applying it intensively in complex situations is a must. In this line, we plan to go back to the area of code mobility, and more generally, distributed systems. We want to devise a distributed kernel infrastructure to support distributed AOP. Research projects are being prepared in the line of grid computing and web applications, which we believe are complex enough to exercize the kernel in interesting ways, and to test the scalability of our approach. In the perspective of applicability, profiling and benchmarking should be carried out in order to fine-tune the implementation of the kernel.

Second, fully supporting a general-purpose aspect language (GPAL) like AspectJ should be considered as a necessary validation of the expressiveness of the kernel. As of now, only a core subset of AspectJ's dynamic crosscutting has been implemented on top of the kernel. Further-

more, AspectJ represents one kind of AO model: other models, like superimposition offered by hyperspaces [Ossher & Tarr, 2001] should be studied. This should make it possible to further validate our claim that partial reflection is a good starting point for an AOP kernel. A challenging perspective beyond the uniform integration of different AO models is the study of a unified model between components and aspects, and its self-application, for instance in the context of Reflex. Work on the Caesar model [Mezini & Ostermann, 2003] represents a good starting point in this direction.

Bibliography

[Agha, 1986] Gul Agha. *ACTORS: a model of concurrent computation in distributed systems*. The MIT Press: Cambridge, MA, 1986.

[Akşit, 2003] Mehmet Akşit, editor. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, Boston, MA, USA, March 2003. ACM Press.

[America & van der Linden, 1990] Pierre H. M. America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In Meyrowitz [1990], pages 161–168. ACM SIGPLAN Notices, 25(10).

[America, 1991] Pierre America, editor. *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP 91)*, volume 512 of *Lecture Notes in Computer Science*, Geneva, Switzerland, July 1991. Springer-Verlag.

[Ancona *et al.*, 1998] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel reification: a reflective model for distributed computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC 98)*, pages 32–36. IEEE Computer Society Press, February 1998.

[Ancona *et al.*, 1999] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective authorization systems: possibilities, benefits, and drawbacks. In *Secure Internet programming: security issues for mobile and distributed objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 35–49. Springer-Verlag, 1999.

[Asai *et al.*, 1996] Kenishi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation – for a better understanding of reflective languages. *Lisp and Symbolic Computation*, 9(2/3):203–241. Kluwer Academic Publishers, 1996.

[AspectJ Website, 2002] The AspectJ website, 2002. http://www.eclipse.org/aspectj.

[AspectWerkz, 2002] Aspectwerkz website, 2002. http://aspectwerkz.codehaus.org/.

[Atkinson *et al.*, 1990] Colin Atkinson, Andrea Di Maio, and Rami Bayan. Dragoon: An object-oriented notation supporting the reuse and distribution of Ada software. In *Proceedings of the 4th International Workshop on Real-Time Ada Issues*, pages 50–59, Pitlochry, Perthshir, Scotland, 1990.

[Baker & Hsieh, 2002] Jason Baker and Wilson C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.

[Barendregt, 1984] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[Batory *et al.*, 1998] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the Fifth Internation Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

[Batory *et al.*, 2002] Don Batory, Charles Consel, and Walid Taha, editors. *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[Bergmans & Akşit, 2001] Lodewijk Bergmans and Mehmet Akşit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[Bergmans *et al.*, 2001] Lodewijk Bergmans, Mehmet Akşit, and Bedir Tekinerdogan. Aspect composition using composition filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 357–382. Kluwer Academic Publishers, 2001.

[Birtwistle *et al.*, 1973] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Petrocelli Charter, 1973.

[Blair *et al.*, 2000] Gordon Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Mike Clarke, Fabio Costa, Hector Duran, Nikos Parlavantzas, and Katia Saikoski. A principled approach to supporting adaptation in distributed mobile environments. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 3–12, Limerick, Ireland, 2000.

[Bobrow *et al.*, 1993] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in context – the shape of the design space. In Paepcke [1993], pages 29–61.

[Bockish *et al.*, 2004] Christoph Bockish, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Lieberherr [2004], pages 83–92.

[Bouraqadi-Saâdani *et al.*, 1998] Noury Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In OOPSLA 98 [1998], pages 84–96. ACM SIGPLAN Notices, 33(10).

[Bowen, 1986] Kenneth A. Bowen. Meta-level techniques in logic programming. In *Proceedings of the International Conference on Artificial Intelligence and its Applications*, Singapore, 1986.

[Bracha & Cook, 1990] Gilad Bracha and William Cook. Mixin-based inheritance. In Meyrowitz [1990], pages 303–311. ACM SIGPLAN Notices, 25(10).

[Braux & Noyé, 2000] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 2–11, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 34(11).

[Brichau *et al.*, 2002] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [2002], pages 110–127.

[Brinch Hansen, 1972] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[Brinch Hansen, 1974] Per Brinch Hansen. A programming methodology for operating system design. In *Proceedings of the IFIP Congress 74*, pages 394–397, Amsterdam, Holland, August 1974. North-Holland.

[Brinch Hansen, 1993] Per Brinch Hansen. Monitors and concurrent Pascal, a personal history. *ACM SIGPLAN Notices*, 28(3):1–35, March 1993.

[Briot & Cointe, 1989] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in SmallTalk-80. In OOPSLA 89 [1989], pages 419–431. ACM SIGPLAN Notices, 24(10).

[Briot *et al.*, 1998] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329. ACM Press, September 1998.

[Bromley, 1987] Allan G. Bromley. The evolution of Babbage's calculating engines. *Annals of the History of Computing*, 9(2):113–136, April-June 1987.

[Cardelli, 1997] Luca Cardelli. Mobile computation. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 3–6. Springer-Verlag, 1997.

[Cardelli, 2003] Luca Cardelli, editor. *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, Darmstadt, Germany, July 2003. Springer-Verlag.

[Caromel & Vayssière, 2001] Denis Caromel and Julien Vayssière. Reflections on MOPs, components, and Java security. In Knudsen [2001], pages 256–274.

[Caromel *et al.*, 1998] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061. Wiley & Sons, September 1998.

[Caromel *et al.*, 2001] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple security-aware MOP for Java. In Yonezawa and Matsuoka [2001], pages 118–125.

[Caromel *et al.*, 2004] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[Caromel, 1993] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[Cazzola, 1998] Walter Cazzola. Evaluation of object-oriented reflective models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOP 98), 12th European Conference on Object-Oriented Programming (ECOOP 98)*, 1998.

[Cheverst *et al.*, 2000] Keith Cheverst, Christos Efstratiou, Nigel Davies, and Adrian Friday. Architectural ideas for the support of adaptive context-aware applications. In *Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality*, Bristol, UK, September 2000.

[Chiba & Nakagawa, 2004] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An open AspectJ-like language. In Lieberherr [2004], pages 102–111.

[Chiba & Nishizawa, 2003] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In Frank Pfenning and Yannis Smaragdakis, editors, *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376, Erfurt, Germany, September 2003. Springer-Verlag.

[Chiba *et al.*, 1996] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[Chiba, 1995] Shigeru Chiba. A metaobject protocol for C++. In OOPSLA 95 [1995], pages 285–299. ACM SIGPLAN Notices, 30(10).

[Chiba, 1997] Shigeru Chiba. Implementation techniques for efficient reflective languages. Technical report 97-06, Department of Information Science, University of Tokyo, 1997.

[Chiba, 1998] Shigeru Chiba. Macro processing in object-oriented languages. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98)*, pages 113–126, Australia, November 1998. IEEE Computer Society Press.

[Chiba, 2000] Shigeru Chiba. Load-time structural reflection in Java. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[Clifton *et al.*, 2000] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch in java. In *Proceedings of*

the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), pages 130–145, Minneapolis, Minnesota, USA, October 2000. ACM Press. ACM SIGPLAN Notices, 35(11).

[CME, 2002] The Concern Manipulation Environment website, 2002. http://www.research.ibm.com/cme.

[Cointe, 1987] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In Meyrowitz [1987], pages 156–162. ACM SIGPLAN Notices, 22(12).

[Cointe, 1999] Pierre Cointe, editor. *Proceedings of the 2nd International Conference on Meta-level Architectures and Reflection (Reflection 99)*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, July 1999. Springer-Verlag.

[Consel & Danvy, 1993] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[Costanza *et al.*, 2001] Pascal Costanza, Günter Kniesel, and Michael Austermann. Independent extensibility for aspect-oriented systems. In *Workshop on Advanced Separation of Concerns at ECOOP 2001*, 2001.

[Czarnecki & Eisenecker, 2000] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[Dahm, 1999] Markus Dahm. Byte code engineering. In Clemens Cap, editor, *Proceedings of JIT 99, Berlin*, pages 267–277, 1999.

[Dahm, 2001] Markus Dahm. Byte code engineering with the BCEL API. Technical report B-17-98, University of Berlin, 2001.

[David *et al.*, 2001] Pierre-Charles David, Thomas Ledoux, and Noury M. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[De Meuter, 1999] Wolfgang De Meuter. Agora: The scheme of object-orientation, or, the simplest MOP in the world. In Ivan Moore, James Noble, and Antero Taivalsaari, editors, *Prototype-based Programming: Concepts, Languages and Applications*, pages 247–272. Springer-Verlag, 1999.

[De Volder & D'Hondt, 1999] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta-programming. In Cointe [1999], pages 250–272.

[de Volder & Steyaert, 1995] Kris de Volder and Patrick Steyaert. Construction of the reflective tower based on open implementations. Technical report VUB-PROG-TR-95-01, Vrije Universiteit Brussels, Belgium, January 1995.

[De Win *et al.*, 2001] Bart De Win, Bart Vanhaute, and Bart De Decker. Security through aspect-oriented programming. In Bart De Decker, Frank Piessens, Jos Smits, and Els Van Herreweghen, editors, *Advances in Network and Distributed Systems Security*, pages 125–138. Kluwer Academic Publishers, 2001.

[des Rivières & Smith, 1984] Jim des Rivières and Brian C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[des Rivières, 1990] Jim des Rivières. The secret tower of CLOS. In *Proceedings of the OOPSLA/ECOOP 90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[Dijkstra, 1968] Edsger W. Dijkstra. The structure of THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dijkstra, 1975] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Dittrich *et al.*, 1995] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer-Verlag, 1995.

[Douence & Südholt, 2000] Rémi Douence and Mario Südholt. On the lightweight and selective introduction of reflective capabilities in applications. In *Proceedings of the ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, 2000.

[Douence & Südholt, 2001] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1):7–34. Kluwer Academic Publishers, 2001.

[Douence & Teboul, 2004] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114, Vancouver, Canada, October 2004. Springer-Verlag.

[Douence *et al.*, 2001] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [2001], pages 170–186.

[Douence *et al.*, 2002] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [2002], pages 173–188.

[Douence *et al.*, 2004] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [2004], pages 141–150.

[Douence *et al.*, 2005] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Filman et al. [2005], pages 201–217.

[Ducasse, 1999] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming*, 12(6):39–44. SIGS Press, 1999.

[Dynamo 2000, 2000] *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 00)*, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 35(7).

[EAOP, 2001] The EAOP tool homepage, 2001. http://www.emn.fr/x-info/eaop/tool.html.

[Ebraert & Tanter, 2004] Peter Ebraert and Éric Tanter. A concern-based approach to software evolution. In *Proceedings of the AOSD Workshop on Dynamic Aspects (DAW 2004)*, Lancaster, UK, March 2004.

[Ebraert, 2003] Peter Ebraert. Tool support for partial behavioral reflection. Master's thesis, Universidad de Chile, Chile – Vrije Universiteit Brussel, Belgium, 2003.

[Elrad *et al.*, 2001] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[Fabre & Chiba, 1998] Jean-Charles Fabre and Shigeru Chiba, editors. *Proceedings of the ACM OOPSLA 98 Workshop on Reflective Programming in Java and C++*, October 1998.

[Fabre *et al.*, 1995] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert J. Stroud, and Zhixue Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 489–498, Pasadena, CA, USA, June 1995. IEEE Computer Society Press.

[Ferber, 1989] Jacques Ferber. Computational reflection in class based object oriented languages. In OOPSLA 89 [1989], pages 317–326. ACM SIGPLAN Notices, 24(10).

[Filman & Havelund, 2002] Robert E. Filman and Klaus Havelund. Source-code instrumentation and quantification of events. In *Workshop on Foundations of Aspect-Oriented Languages (FOOL) at AOSD 2002*, Twente, Netherlands, April 2002.

[Filman *et al.*, 2005] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[Foote & Johnson, 1989] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In OOPSLA 89 [1989], pages 327–335. ACM SIGPLAN Notices, 24(10).

[Friedman & Wand, 1984] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355, August 1984.

[Frolund & Agha, 1993] Svend Frolund and Gul Agha. A language framework for multi-object coordination. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 952 of *Lecture Notes in Computer Science*, pages 346–360, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Fuggetta *et al.*, 1998] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *Transactions on Software Engineering*, 24(5):342–361. IEEE Computer Society Press, May 1998.

[Gamma *et al.*, 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[Goldberg & Kay, 1976] Adele Goldberg and Alan Kay. Smalltalk-72 instruction manual. Technical report SSL-76-6, Xerox Palo Alto Research Center, Palo Alto, California, 1976.

[Goldberg & Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Golm & Kleinöder, 1999] Michael Golm and Jurgen Kleinöder. Jumping to the meta level, behavioral reflection can be fast and flexible. In Cointe [1999], pages 22–39.

[Gong, 1999] Li Gong. *Inside Java 2 Platform Security: Architecture, API design, and implementation*. Addison-Wesley, 1999.

[Gowing & Cahill, 1995] Brendan Gowing and Vinny Cahill. Making meta-object protocols practical for operating systems. In *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems*, pages 52–55, 1995.

[Gowing & Cahill, 1996] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. In Kiczales [1996], pages 137–152.

[Gray *et al.*, 2001] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, October 2001.

[Halpern, 1993] Mark Halpern. Binding. In *Encyclopedia of Computer Science*, page 125. Chapman & Hall, 1993.

[Halstead, Jr., 1985] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538. ACM Press, 1985.

[Hanneman & Kiczales, 2002] Jan Hanneman and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 161–173, Seattle, Washington, USA, November 2002. ACM Press. ACM SIGPLAN Notices, 37(11).

[Harrison & Ossher, 1993] William H. Harrison and Harold L. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 411–428, Washington, D.C., USA, October 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[Harrison *et al.*, 2002a] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report RC22685, IBM Research, 2002.

[Harrison *et al.*, 2002b] William H. Harrison, Harold L. Ossher, Peri L. Tarr, Vincent Kruskal, and Frank Tip. CAT: A toolkit for assembling concerns. Technical report RC22686, IBM Research, 2002.

[Hayes, 2003] Brian Hayes. The post-OOP paradigm. *American Scientist*, 91(2):106–110, March-April 2003.

[Hilsdale & Hugunin, 2004] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [2004], pages 26–35.

[Hirschfeld, 2002] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In Mehmet Akşit, Mira Mezini, and R. Unland, editors, *International Conference NetObjectDays on Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2002.

[Hoare, 1974] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–577, October 1974.

[Hoare, 1978] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Ibrahim, 1990] Mamdouh H. Ibrahim. Report of the workshop on reflection and metalevel architectures in object-oriented programming. In Meyrowitz [1990]. ACM SIGPLAN Notices, 25(10).

[Ichbiah *et al.*, 1991] Jean D. Ichbiah, John G. P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, April 1991.

[Ichisugi *et al.*, 1992] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 24–35, Tokyo, Japan, November 1992.

[Irwin *et al.*, 1997] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *Lecture Notes in Computer Science*, pages 249–256. Springer-Verlag, 1997.

[JBoss.AOP, 2004] JBoss AOP website, 2004. http://www.jboss.org/developers/projects/jboss/aop.

[Jones *et al.*, 1993] Neil D. Jones, Charles K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[Katz, 1994] Randy H. Katz. Adaptation and Mobility in Wireless Information Systems. *Personal Communications*, 1:6–17. IEEE Computer Society Press, First quarter 1994.

[Kaubisch *et al.*, 1976] W.H. Kaubisch, Ronald H. Perrott, and Charles A.R. Hoare. Quasi-parallel programming. *Software: Practice and Experience*, 6(3):341–356. Wiley & Sons, 1976.

[Kay, 2001] Alan Kay. Software: Art, engineering, mathematics, or science? Foreword in the book "Squeak: Object-Oriented Design with Multimedia Applications", by Mark Guzdial, Prentice Hall, 2001.

[Keppel *et al.*, 1991] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[Kiczales *et al.*, 1991] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kiczales *et al.*, 1993] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In Paepcke [1993], pages 101–118.

[Kiczales *et al.*, 1997a] Gregor Kiczales, John Lamping, Cristina V. Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA, 1997. ACM Press.

[Kiczales *et al.*, 1997b] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Knudsen [2001], pages 327–353.

[Kiczales, 1992] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures*. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[Kiczales, 1996] Gregor Kiczales, editor. *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, CA, USA, April 1996.

[Kiczales, 2001] Gregor Kiczales. The future of reflection. Invited talk at the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001), September 2001.

[Klaeren *et al.*, 2000] Herbert Klaeren, Elke Pulvermüller, Awais Rashid, and Andreas Speck. Aspect composition applying the design by contract principle. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 57–69. Springer-Verlag, 2000.

[Kleinöder & Golm, 1996] Jurgen Kleinöder and Michael Golm. MetaJava: An efficient run-time meta architecture for Java. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS 96)*. IEEE Computer Society Press, 1996.

[Kniesel *et al.*, 2005] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - a powerful back-end for aspect-oriented programming. In Filman et al. [2005], pages 311–342.

[Knudsen, 2001] Jorgen L. Knudsen, editor. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, Budapest, Hungary, June 2001. Springer-Verlag.

[Kojarski *et al.*, 2003] Sergei Kojarski, Karl Lieberherr, David H. Lorenz, and Robert Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Intelligence*, 1(1):11–46. Kluwer Academic Publishers, 1986.

[Lamping *et al.*, 1992] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA 92 Workshop on Reflection and Meta-Level Architectures*, pages 95–106. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[Lange, 1998] Danny B. Lange. Mobile objects and mobile agents: The future of distributed computing? In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP 98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 1–12, Brussels, Belgium, July 1998. Springer-Verlag.

[Lea, 1999] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison Wesley, second edition, 1999.

[Ledoux & Bouraqadi-Saâdani, 2000] Thomas Ledoux and Noury Bouraqadi-Saâdani. Adaptability in Mobile Agent Systems using Reflection. RM 2000, Workshop on Reflective Middleware, April 2000.

[Ledoux, 1999] Thomas Ledoux. OpenCorba: a reflective open broker. In Cointe [1999], pages 197–214.

[Liang & Bracha, 1998] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In OOPSLA 98 [1998], pages 36–44. ACM SIGPLAN Notices, 33(10).

[Lieberherr & Silva-Lepe, 1994] Karl Lieberherr and Ignacio Silva-Lepe. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, 1994.

[Lieberherr, 2004] Karl Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.

[Lieberman, 1986] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the 1st International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86)*, pages 214–223, Portland, Oregon, USA, October 1986. ACM Press. ACM SIGPLAN Notices, 21(11).

[Lopes & Lieberherr, 1996] Cristina V. Lopes and Karl Lieberherr. AP/S++: Case-study of a MOP for purposes of software evolution. In Kiczales [1996], pages 167–184.

[Lopes, 1997] Cristina V. Lopes. *D: A Language Framework for Distributed Programming*. Ph.D. thesis, College of Computer Science, Northeastern University, 1997.

[Maes & Nardi, 1988] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, Alghero, Sardinia, October 1988.

[Maes, 1987a] Pattie Maes. *Computional reflection*. Ph.D. thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[Maes, 1987b] Pattie Maes. Concepts and experiments in computational reflection. In Meyrowitz [1987], pages 147–155. ACM SIGPLAN Notices, 22(12).

[Magnusson, 2002] Boris Magnusson, editor. *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, Málaga, Spain, June 2002. Springer-Verlag.

[Malenfant *et al.*, 1996a] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantics of introspection in a reflective prototype-based language. *Lisp and Symbolic Computation*, 9(2,3):153–180. Kluwer Academic Publishers, 1996.

[Malenfant *et al.*, 1996b] Jacques Malenfant, Michel Jacques, and François-Nicolas Demers. A tutorial on behavioral reflection and its implementation. In Kiczales [1996], pages 1–20.

[Marke, 1988] Kris Van Marke. *The Use and Implementation of the Representation Language KRS*. Ph.D. thesis, Vrije Universiteit Brussels, Belgium, April 1988.

[Masuhara & Kawauchi, 2003] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, November 2003.

[Masuhara & Kiczales, 2003] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Cardelli [2003], pages 2–28.

[Masuhara *et al.*, 1994] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, 1994.

[Masuhara *et al.*, 2003] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[Matsuoka *et al.*, 1991] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In America [1991], pages 231–250.

[Matsuoka *et al.*, 1998] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, and H. Takagi Y. Kimura, K. Hotta. OpenJIT — a reflective Java JIT compiler. In Fabre and Chiba [1998], pages 16–20.

[McAffer, 1995a] Jeff McAffer. Meta-level architecture support for distributed objects. In *International Workshop on Object-Orientation in Operating Systems (IWOOS 95)*, 1995.

[McAffer, 1995b] Jeff McAffer. Meta-level programming with CodA. In Olthoff [1995], pages 190–214.

[McAffer, 1996] Jeff McAffer. Engineering the meta-level. In Kiczales [1996], pages 39–61.

[Mehner & Rashid, 2002] Katharina Mehner and Awais Rashid. Towards a Standard Interface for Runtime Inspection in AOP Environments. In *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA 2002*, November 2002.

[Mendhekar *et al.*, 1997] Anurag Mendhekar, Gregor Kiczales, and Jim Lamping. RG: A case-study for aspect-oriented programming. Technical report SPL97-009P9710044, Xerox PARC, February 1997.

[Meyrowitz, 1987] Norman Meyrowitz, editor. *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, Orlando, Florida, USA, October 1987. ACM Press. ACM SIGPLAN Notices, 22(12).

[Meyrowitz, 1990] Norman Meyrowitz, editor. *Proceedings of the 5th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA/ECOOP 90)*, Ottawa, Canada, October 1990. ACM Press. ACM SIGPLAN Notices, 25(10).

[Mezini & Ostermann, 2003] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In Akşit [2003], pages 90–99.

[Mulet *et al.*, 1995] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In OOPSLA 95 [1995], pages 316–330. ACM SIGPLAN Notices, 30(10).

[Nishizawa *et al.*, 2004] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [2004], pages 7–15.

[Okamura & Ishikawa, 1994] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP 94)*, volume 821 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, July 1994.

[Okamura *et al.*, 1992] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 36–47, Tokyo, Japan, November 1992.

[Oliva & Buzato, 1999] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems (COOTS 99)*, pages 203–216, San Diego, CA, USA, May 1999.

[Olthoff, 1995] Walter G. Olthoff, editor. *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP 95)*, volume 952 of *Lecture Notes in Computer Science*, Åarhus, Denmark, August 1995. Springer-Verlag.

[OOPSLA 89, 1989] *Proceedings of the 4th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 89)*, New Orleans, Louisiana, USA, October 1989. ACM Press. ACM SIGPLAN Notices, 24(10).

[OOPSLA 95, 1995] *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, Austin, Texas, USA, October 1995. ACM Press. ACM SIGPLAN Notices, 30(10).

[OOPSLA 98, 1998] *Proceedings of the 13th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 98)*, Vancouver, British Columbia, Canada, October 1998. ACM Press. ACM SIGPLAN Notices, 33(10).

[Ossher & Tarr, 2001] Harold L. Ossher and Peri L. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In Mehmet Akşit, editor, *Software Architectures and Component Technology*, volume 648 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 2001.

[Paepcke, 1993] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Parnas, 1972] David Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pawlak *et al.*, 2001] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka [2001], pages 1–24.

[Popovici *et al.*, 2003] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: Efficient dynamic weaving for Java. In Akşit [2003], pages 100–109.

[Rao, 1991] Ramana Rao. Implementational reflection in Silica. In America [1991], pages 251–266.

[Rashid, 2001] Awais Rashid. A hybrid approach to separation of concerns: The story of SADES. In Yonezawa and Matsuoka [2001], pages 231–249.

[Redmond & Cahill, 2000] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, June 2000.

[Redmond & Cahill, 2002] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behavior. In Magnusson [2002], pages 205–230.

[Rivard, 1996] Fred Rivard. Smalltalk: a reflective language. In Kiczales [1996], pages 21–38.

[Rodríguez *et al.*, 2004] Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, November 2004. IEEE Computer Society Press.

[Schärli *et al.*, 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In Cardelli [2003], pages 248–274.

[Ségura-Devillechaise *et al.*, 2003] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In Akşit [2003], pages 110–119.

[Sereni & de Moor, 2003] Damien Sereni and Oege de Moor. Static analysis of aspects. In Akşit [2003], pages 30–39.

[Shapiro *et al.*, 1989] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP 89)*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[Shonle *et al.*, 2003] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 Domain-Driven Development Track*, October 2003.

[Shrivastava *et al.*, 1991] Santosh K. Shrivastava, Graeme N. Nixon, and Graham D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, January 1991.

[Simonyi, 1995] Charles Simonyi. The death of computer languages. Technical report TR-95-52, Microsoft Research, 1995.

[Smith, 1982] Brian C. Smith. Reflection and semantics in a procedural language. Technical report 272, MIT Laboratory of Computer Science, 1982.

[Smith, 1984] Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, January 1984.

[Snelting & Tip, 2002] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In Magnusson [2002], pages 562–584.

[Stasko *et al.*, 1998] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization*. The MIT Press, 1998.

[Steyaert, 1994] Patrick Steyaert. *Open Design of Object-Oriented Languages – A Foundation for Specializable Reflective Language Frameworks*. Ph.D. thesis, Vrije Universiteit Brussels, Belgium, 1994.

[Stroud & Wu, 1995] Robert J. Stroud and Zhixue Wu. Using metaobject protocols to implement atomic data types. In Olthoff [1995], pages 168–189.

[Stroud & Wu, 1996] Robert J. Stroud and Zhixue Wu. *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter Using Metaobject Protocols to Satisfy Non-Functional Requirements, pages 31–52. CRC Press, 1996.

[Stroud, 1993] Robert J. Stroud. Transparency and reflection in distributed systems. *ACM Operating System Review*, 22(2):99–103, April 1993.

[Stroustrup, 1997] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[SUN, 1996a] SUN Microsystems. *The Java Language Specification*, 1996.

[SUN, 1996b] SUN Microsystems. *The Java Native Code API*, 1996.

[SUN, 1998a] SUN Microsystems. *Object Serialization*, 1998.

[SUN, 1998b] SUN Microsystems. *Remote Method Invocation*, 1998.

[SUN, 1999a] SUN Microsystems. *Dynamic Proxy Classes*, 1999.

[SUN, 1999b] SUN Microsystems. *Java Platform Debugger Architecture*, 1999.

[SUN, 1999c] SUN Microsystems. *Reflection API Documentation*, 1999.

[SUN, 2000] SUN Microsystems. *Enterprise JavaBeans Technology*, 2000.

[SUN, 2003] SUN Microsystems. *The Producer/Consumer Example, from Java tutorials*, 2003.

[SUN, 2004] SUN Microsystems. *Java HotSpot Technology*, 2004.

[Tanter & Ebraert, 2003] Éric Tanter and Peter Ebraert. A flexible approach to interactive runtime inspection. In *Proceedings of the ECOOP Workshop on Advancing the State-of-the-Art in Runtime Inspection*, Darmstadt, Germany, July 2003.

[Tanter & Noyé, 2004a] Éric Tanter and Jacques Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, September 2004.

[Tanter & Noyé, 2004b] Éric Tanter and Jacques Noyé. Versatile kernels for aspect-oriented programming. Research Report RR-5275, INRIA, July 2004.

[Tanter & Piquer, 2001] Éric Tanter and José Piquer. Managing references upon object migration: Applying separation of concerns. In *Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001)*, pages 264–272, Punta Arenas, Chile, November 2001. IEEE Computer Society Press.

[Tanter *et al.*, 2001] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex – towards an open reflective extension of Java. In Yonezawa and Matsuoka [2001], pages 25–43.

[Tanter *et al.*, 2002a] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java semantics via bytecode manipulation. In Batory et al. [2002], pages 283–298.

[Tanter *et al.*, 2002b] Éric Tanter, Michaël Vernaillen, and José Piquer. Towards transparent adaptation of migration policies. In *8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002)*, Málaga, Spain, June 2002.

[Tanter *et al.*, 2003] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[Tanter, 2000] Éric Tanter. Reflex, a reflective system for Java — application to flexible resource management in Java mobile object systems. Master's thesis, Universidad de Chile, Chile – Vrije Universiteit Brussel, Belgium, 2000.

[Tanter, 2002] Éric Tanter. Runtime metaobject protocols: the quest for their holy application. In *12th ECOOP Workshop of PhD Students in Object-Oriented Systems (PhDOOS 2002)*, Málaga, Spain, June 2002.

[Tatsubori *et al.*, 2000] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In *1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE 99)*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133, Denver, USA, 2000. Springer-Verlag.

[Tatsubori, 1999] Michiaki Tatsubori. An extension mechanism for the Java language. Master's thesis, University of Tsukuba, Japan, 1999.

[Tatsubori, 2002] Michiaki Tatsubori. *A Class-Object Model for Program Transformations*. Ph.D. thesis, Graduate School of Engineering, University of Tsukuba, Japan, January 2002.

[Turing, 1936] Alan M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[Turing, 1937] Alan M. Turing. Correction to: On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 43(2):544–546, 1937.

[Vayssière, 2002] Julien Vayssière. *Une architecture de sécurité pour les applications réflexives – Application à Java*. Ph.D. thesis, Université de Nice Sophia Antipolis, 2002.

[Vernaillen, 2002] Michaël Vernaillen. Transparent adaptation of migration policies in mobile object systems. Master's thesis, Universidad de Chile, Chile – Vrije Universiteit Brussel, Belgium, 2002.

[von Neumann, 1958] John von Neumann. *The Computer and the Brain*. Yale University Press, June 1958.

[Wand & Friedman, 1988] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37. Kluwer Academic Publishers, 1988.

[Wand *et al.*, 2004] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[Wand, 2003] Mitchell Wand. Understanding aspects (extended abstract). In *Proceedings of the International Conference on Functional Programming (ICFP 2003)*, pages 299–300. ACM Press, 2003.

[Watanabe & Yonezawa, 1988] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In Norman Meyrowitz, editor, *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88)*, pages 306–315, San Diego, California, USA, September 1988. ACM Press. ACM SIGPLAN Notices, 23(11).

[Watanabe & Yonezawa, 1990] Takuo Watanabe and Akinori Yonezawa. An actor-based meta-level architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL)*, Lecture Notes in Computer Science, pages 405–425, Noordwijkerhout, the Netherlands, May 1990. Springer-Verlag.

[Weinreb & Moon, 1981] Daniel Weinreb and David Moon. Lisp machine manual. Symbolics, Inc., 1981.

[Weiser, 1993] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[Welch & Stroud, 1998] Ian Welch and Robert J. Stroud. Dalang - a reflective Java extension. In *Proceedings of the OOPSLA 99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[Welch & Stroud, 1999] Ian Welch and Robert J. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Cointe [1999], pages 2–21.

[Welch & Stroud, 2001] Ian Welch and Robert J. Stroud. Kava - using bytecode rewriting to add behavioral reflection to Java. In *Proceedings of USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, USA, January 2001.

[Welch & Stroud, 2002] Ian Welch and Robert J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432. IOS Press, 2002.

[Weyhrauch, 1980] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1,2). North-Holland, 1980.

[Wirth, 1974] Niklaus Wirth. On the design of programming languages. *Information Processing 74*, 1974.

[Wu, 1994] Zhixue Wu. *A New Approach to Implementing Atomic Data Types*. Ph.D. thesis, Cambridge University, 1994.

[Wu, 1998] Zhixue Wu. Reflective Java and a reflective component-based transaction architecture. In Fabre and Chiba [1998].

[Wuyts, 1998] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 98*, page 112, 1998.

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

[Yokote, 1992] Yasuhiko Yokote. The ApertOS reflective operating system: The concept and its implementation. In *Proceedings of the 7th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 92)*, pages 414–434, Vancouver, British Columbia, Canada, October 1992. ACM Press. ACM SIGPLAN Notices, 27(10).

[Yonezawa & Matsuoka, 2001] Akinori Yonezawa and Satoshi Matsuoka, editors. *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, September 2001. Springer-Verlag.

[Yonezawa, 1990] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*, Computer Systems Series. The MIT Press, 1990.

[Zimmermann, 1996] Chris Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

Lists

List of definitions

List of Figures

This report has been written with GNU EMACS v21.3.1, LGRIND v3.6, TEX v3.14159 (WEB2C v7.4.5), on a LINUX MANDRAKE distribution.
Version 0.0.1a, compiled 11th January 2005 at 16 h 14.

From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming

The advent of complex software systems calls for enhanced support for modularization and adaptation. This thesis deals with two related language-based approaches, reflection and aspect-oriented programming (AOP). Reflection is a very general approach, originally focused on adaptation, which suffers from complexity and cost issues. Moreover, reflection is accessed via metaobject protocols, which are paradoxically hardwired and hence not adequate in all situations. These issues have limited the acceptance of reflection as a modularization technology, and consequently, AOP has been proposed as an alternative whose specificity contrasts with the generality of reflection. Although AOP provides more adequate support for modularization by advocating the use of dedicated aspect languages, most AOP proposals do so by sacrificing flexibility and extensibility.

This thesis reconciles both approaches by proposing a model of reflection that supports both the generality of reflective systems and the specificity of AOP. The model includes links as first-class entities representing the binding between base programs and metaobjects. The precise definition of links addresses efficiency by applying reflection only at appropriate places, implementing a form of partial reflection. It also tackles complexity by supporting fine-grained specialization of metaobject protocols and flexible structuring of the metalevel. In order to combine the power and generality of reflection and the guidance of aspect languages, we propose a versatile substrate for AOP based on partial reflection, extended with compositional means. An open implementation of our model of partial reflection, tailorable and extensible, is developed for Java. It is then evolved into a versatile kernel for AOP supporting (possibly domain-specific) aspect languages and composition of aspects written in different languages. Our proposal therefore bridges the gap between metaobject protocols and AOP in a way that is fruitful to both. Significant applications of our prototype in different contexts, such as distributed and concurrent programming, empirically validate our argument.

Keywords: metaobject protocols, aspect-oriented programming, reflection, open implementations.

Des Protocoles de Métaobjets aux Noyaux Versatiles pour la Programmation par Aspects

L'avènement des systèmes logiciels complexes nécessite une amélioration de la modularisation et de l'adaptation des logiciels. Cette thèse traite de deux approches axées sur les langages de programmation, la réflexion et la programmation par aspects (PPA). La réflexion est une approche très générale, initialement centrée sur l'adaptation, qui souffre de problèmes de coûts et de complexité. Par ailleurs, la réflexion est rendue accessible par les protocoles de métaobjets qui, paradoxalement, sont rigides et donc plus ou moins adaptés à une situation donnée. Face à ces problèmes, la PPA a été proposée comme une alternative dont la spécificité contraste avec la généralité de la réflexion. Bien que la PPA fournisse un support plus adéquat de modularisation, par le biais de l'utilisation de langages d'aspects dédiés, la plupart des propositions compromettent la flexibilité et l'extensibilité.

Cette thèse réconcilie les deux approches en proposant un modèle de réflexion qui supporte à la fois la généralité des systèmes réflexifs et la spécificité de la PPA. Le modèle promeut au statut d'entités de première classe les *liens* associant les programmes de bases et les métaobjets. La définition précise des liens améliore l'efficacité en appliquant la réflexion seulement aux endroits appropriés, implémentant ainsi une forme de réflexion partielle. Elle résout également le problème de complexité en permettant la spécialisation détaillée des protocoles de métaobjets et une structuration souple du métaniveau. Nous proposons ensuite un substrat versatile pour la PPA basé sur la réflexion partielle, étendue avec des capacités de composition. Une implémentation ouverte, ajustable et flexible, est développée pour Java. Elle est ensuite étendue en un noyau versatile pour la PPA supportant les langages d'aspects (éventuellement dédiés), et la composition d'aspects définis dans des langages différents. Par conséquent, nos travaux comblent le vide entre les protocoles de métaobjets et la PPA d'une manière mutuellement profitable. Des applications significatives de notre prototype dans différents contextes, comme la programmation de systèmes distribués et concurrents, valident de manière empirique notre proposition.

Mots clés: protocoles de métaobjets, programmation par aspects, réflexion, implémentations ouvertes.