Reflection and Open Implementations

Éric Tanter

DCC, University of Chile
Avenida Blanco Encalada 2120
Santiago, Chile
etanter@dcc.uchile.cl

Abstract. We review the state-of-the-art of reflection and metaprogramming, prior to our work on partial behavioral reflection and Reflex, and open implementations. The first four sections are dedicated to reflection. Section 1 introduces the concept of reflection and its application to programming languages. Section 2 discusses reflection in the particular context of object-oriented programming languages. Then, since we are interested in addressing issues in the concrete applicability of reflection, we dedicate Section 3 to the structuring and engineering of metalevel architectures, while implementation considerations are dealt with in Section 4. After this comprehensive review of reflection, the last section discusses the related area of open implementations (Section 5).

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."* [82]

This section is structured as follows. Section 1.1 gives a brief historical introduction to the difference between programs and data and concludes with the appealing idea of conceiving programs as data for other programs [1]. Then, the notions of *metaprogramming* and *reflection* are defined (Section 1.2). Section 1.3 exposes the seminal experiments in reflection in programming languages, based on the idea of *reflective towers*. Finally, Section 1.4 discusses characteristics of reflective languages as well as some of these languages.

[1] Credits for this historical introduction to the distinction between programs and data go to Julien Vayssière [111].

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 [11]. 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* [112].

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 [4], both programs and data are represented by higher-order functions. Similarly, a special kind of Turing machines [109], called the *universal Turing machine* [110], 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.

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 [62], and illustrated in Fig. 1:

Computational system

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

Steyaert makes clear the difference between a program and a computational system [91]: 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. 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*:

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. 1(b)):

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. 1(c)):

Reflective system

A metasystem causally connected to itself.

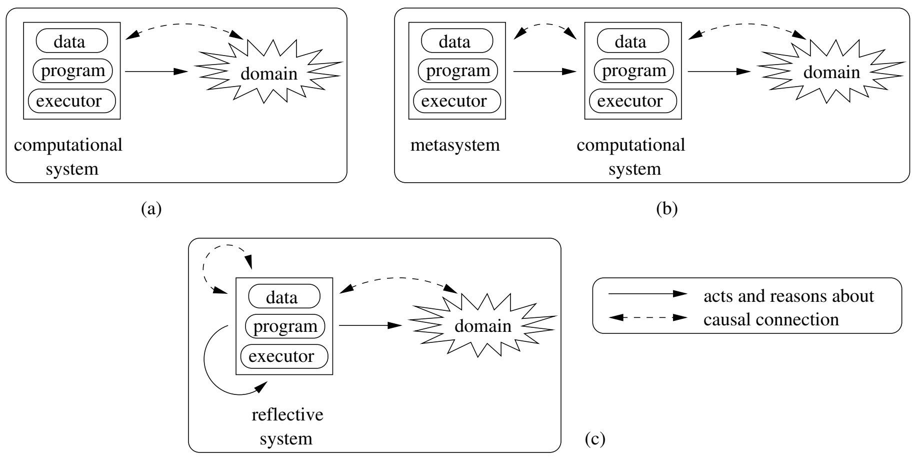

Fig. 1. Computational system (a), metasystem (b) and reflective system (c).

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 [45]:

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.

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 [89, 90, 28].

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 [28]. 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 illustrates the levels of processing in the infinite reflective tower.

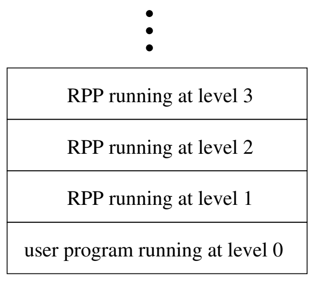

Fig. 2. Processing levels in the reflective tower.

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.

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. 3).

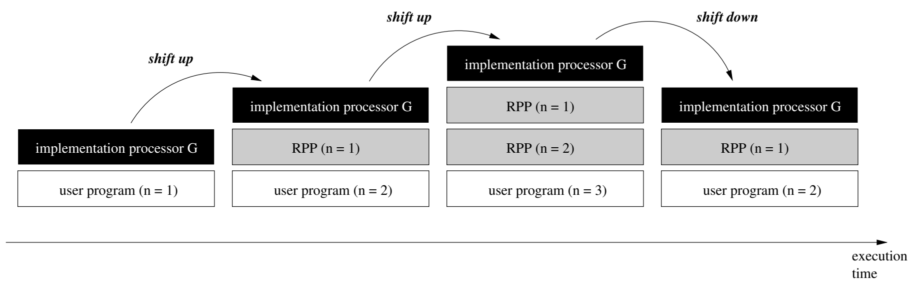

Fig. 3. The level-shifting processor.

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 [36] 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:

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*.

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* [28, 91, 24] or *deification* [126, 31].

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* [113].

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 [90]. 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 [64, 66]. 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 [113], and in an experimental one in [31]. Therefore, the precise point at which a language with reflective mechanisms becomes a reflective language is not well defined [85]. This distinction is however useful to contrast reflective languages with programming languages that only provide (or are extended with) *some* reflective mechanisms.

Reflective mechanisms A reflective mechanism is defined in [66] 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 [91] 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*:

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.

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:

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.

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).

Some reflective languages As defined by Pattie Maes [63], *"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 [89]. Another reflective variant of Lisp is Brown, which served as the basis for the formal work of Friedman and Wand [36]. 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 [120] and Meta-Prolog [7]), rule-based languages (*e.g.* SOAR [57]). 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 [63, 62], Agora [91, 24], ObjVLisp [20], Smalltalk [85], Classtalk [9] and CLOS [53].

Smalltalk is in fact not fully reflective due the pragmatic reason of efficiency [39], 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 [35]. 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) [85]. Conversely, other industrial languages like C++ [95] have no reflective or metalevel features[2]. The more recent Java programming language [96] 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 4.2.

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 [30], 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

[2] Actually C++ *templates* are a textual compile-time metaprogramming facility [23].

in the code. We shall come back on this in Section 4.4 when discussing partial reflection.

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 [45]. We will discuss this match in Section 2.1. In Section 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 2.3

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"* [63]. 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."* [53]

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 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."* [53]

This notion is refined later, in the context of the CLOS MOP, a metaobject protocol for CLOS [6] (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."* [53]

Maes observed that reflective computation may be caused either by an object itself or by the interpreter [63]. 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* [64]. This observation leads to the distinction of different types of MOPs, depending on whether they are *explicit*, implicit or related to communications between metaobjects [128]:

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).

2.3 Reflective Models for Object-Oriented Languages

The first object-oriented languages to incorporate some reflective facilities, such as Smalltalk-72 [38] and Flavors [116], did so in ad hoc ways. A first step towards a cleaner handling of reflective facilities was the introduction of *metaclasses* by Smalltalk-80 [39], further studied by Cointe in ObjVLisp [20] and Classtalk [9]. 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 [62, 63], 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 [67], which does not support the notion of classes. Ferber [34] 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.

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"* [34].

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

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

[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.

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 [14], which is particularly useful in a distributed setting [2, 72]. 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.

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 [53] is rather based around the generic function model, where the application of a generic function is reified as a generic function [66]. As discussed by Malenfant *et al.*, reflective towers appear in both kinds of approaches. Des Rivières [27] 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 [65], 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 [31], 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 [31], 3-KRS [62] and Agora [24] are semantics-based, following Smith's seminal work on 3-Lisp. According to [66], 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.

2.4 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 [45]. 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) [29, 81]. 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 [5, 84]. These strengths of reflection have been exercised in a wide range of domains, including distribution [92, 13, 59, 72], mobile objects [5, 60, 106], concurrency [68], fault-tolerance [33] and atomicity [93].

In [94, 10], 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 [88], which provides persistence and atomicity by inheritance, and distribution transparency via a preprocessor; PC++ [122], which provides atomic data types via a combined use of inheritance and preprocessing; or SOS [87], 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* [94]. 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.

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 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 3.2.

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 [63]. 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 [114], a reflective version of the object-oriented concurrent system ABCL/1 [127], 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* [70] since a single object is the unit of computation at the base level (from a metalevel point of view). The 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

[4] A distinction is introduced in [70] 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.

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 [115]. 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* [70], 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 [42]. 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 [73], as discussed hereafter.

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* [73]. His observations are the result of his work on a metalevel architecture for distributed object systems, CodA [71, 72]. As he states:

[5] The issue of bringing reflection to a compiled language such as C++ will be discussed later in Section 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."* [73]

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.

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 [89], 3-KRS [62] and MetaJ [31], where metaobjects match the structure of the interpreter, and the language-centric approaches, such as the CLOS MOP [53], ObjVLisp [20] and Classtalk [9], 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 [93], concurrency [46], distribution [75, 76] and operating systems [126]. 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"* [73].

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 [42]. This proposal was motivated by the will to make metaobject protocols practical for operating systems [41], 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 [76, 75].

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* [42]. 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 [83, 84].

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 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 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 4.3, attempts to make reflection efficient by anticipating reflective computation; the other approach, explored in Section 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.

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 [126]), 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 [53], 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 [53], Appendix C, for the CLOS MOP, or [15], for the OpenC++ MOP).

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 4.2, we will discuss the notion of *binding time*, which is helpful to understand and characterize the various approaches to implement reflection.

Compiled languages With compilers, the metalevel information that is constructed at compile time is usually not kept beyond the compilation phase. In the

[6] We use the term *metalevel information* as introduced in [42]: *"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".*

generated code, the object model of the language is completely implicit and can not be accessed[7]. Therefore, as discussed in [42], 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) [128].

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 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 4.3.

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 [31]. 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

[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 [86].

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 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 [84]: it was implemented as a native dynamic library integrated very closely with the interpreter, via the Java Just-In-Time (JIT) compiler interface [97]. At that time, Sun started its integrated HotSpot technology [104], and stopped supporting the JIT compiler interface.

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 [43, 48, 44]. Binding basically means associating a value to a name. The following definition, taken from [43], expresses binding from a lower-level point of view:

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."* [66]

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* [23]. 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 [84].

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 [102] 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 [98], remote method invocation [99] and component architectures [103]. 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 [117], Kava [119], Jinline [105], Javassist [18, 19]). Transforming bytecode presents several advantages over source code transformation, as done by OpenJava [107] and Reflective Java [123] 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 [105].

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 [77] and MetaXa [56, 40] are other examples. Some approaches also use the fact that Java features Just-In-Time (JIT) compilation to operate at this level (*e.g.* [69, 84]). Finally, it is also possible to use the debugging interface of Java [101], 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 [12], and the standard *dynamic proxies* introduced with the JDK 1.3 [100]. 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 [61]. Other disadvantages of this approach are discussed in [118], where Welch and Stroud motivate the evolution of Dalang, based on interceptor objects, to Kava, based on bytecode rewriting.

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 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 [16]: 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 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:

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

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:

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

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.

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 [53]. 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):

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

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).

Partial Evaluation Partial evaluation [22, 47] 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, [3] 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 [8], 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.

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++ [15] (version 2). This technique was later on transposed in the Java world with OpenJava [108] and Javassist [18, 19]. 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 [66], 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 [49], *"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 [69], 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 [18]. 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 [17].

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 [45]. 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 4.4). The issue of how selectivity is defined is the subject of Section 4.4, while Section 4.4 examines the problem of specifying the actual protocol between base objects and metaobjects, including the shape of reifications.

Execution model We hereby formulate an attempt to describe the approach of partial reflection with execution expressions, as in Section 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:

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

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 4.2). 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.

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 $\{s \in \mathit{Struct}\}$ and a set of execution points $\{ep \in \mathit{ExecPoints}\}$, ρ is determined by the function:

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

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 [42] –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 [117], Reflective Java [123], the ProActive MOP [12], MetaXa [56, 40] and Guaraná [77] (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.

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) 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 [13] only gives access to the implicit MOP via the explicit MOP. In Iguana [42], the task of (metaobject) *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 [107] also relies on a kind of annotations placed in the source code.

A less intrusive, more declarative approach is used in, *e.g.* Kava [119], Reflective Java [123], and Iguana/J [84], 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 [124, 125, 25] (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 **??**.

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 [42] 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 [84] 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 [119], 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 MOPs*: 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 [123]: 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 [19] actually makes it possible to extract whatever piece of information from a base-level program in a convenient manner.

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 [30], 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 5.1 is an introduction to the concept, Section 5.2 explains why open implementations are important, and finally, Section 5.3 surveys techniques and approaches for providing open implementations.

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 [82]. 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*:

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 [82]. 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 1.4). Recall that a language with a reflective architecture allows much more open-ended access to the implementation of a language [64], 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* [82]:

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 [45], 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.

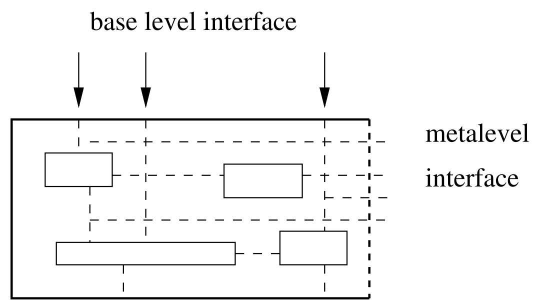

Fig. 4. A system with an open implementation. (From [Rao, 1991].)

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. 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.* [53, 31]), and systems providing compile-time MOPs are open implementations of compilers (*e.g.* [58, 15]).

Patrick Steyaert has proposed a very interesting account of reflection [91], 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 [26], 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* [46].

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.

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 [50], 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 [52]. 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."."* [121]

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 [53]. 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 [50]. 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 [82].

The open implementation approach therefore advocates to open up the implementation, but to do so in a *principled*, *disciplined* way [82, 50]. 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 [82]. An open implementation actually makes it possible to "re-make" some of the tradeoffs in the system to better suit their needs [52], 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* [50]:

> *"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" [53], hence solving the dilemma they were facing.

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.

Designing the metalevel interface In [52], 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.

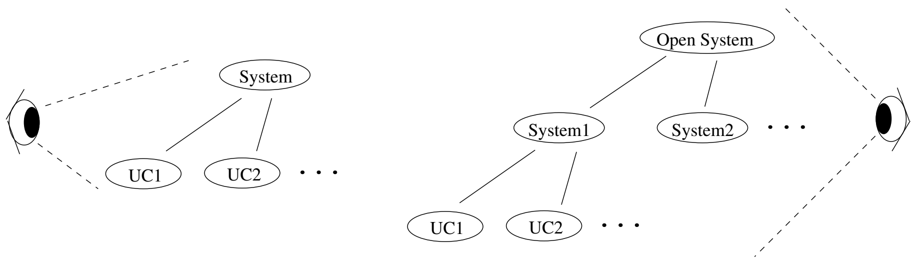

Fig. 5. Contrasting traditional system design (left) and open implementation design (right). (Adapted from [Kiczales *et al.*, 1993].)

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. 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. 5.

Therefore, the first question that pops up when designing an open implementation is *what range* of implementations users should be able to specify [58]. 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"* [50]. 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 [53, 52].

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 3). For instance, meta-objects 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 [50]. We shall come back on this issue at the end of this section on open implementations.

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 [54]. 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.

[54] 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.

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 [37] 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 meta-level interface must not prevent efficient and effective implementation of the base level [82]. 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 [42]. 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 3.2), 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.

Mastering Locality: Towards Aspect-Oriented Programming The issue of locality has been recurrently mentioned since the beginning of this chapter. In [52], 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 [50]. 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.*"*
> [50]

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 [55] was published at ECOOP.

References

[1] P. 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.

[2] M. Ancona, W. Cazzola, G. Dodero, and V. 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, Feb. 1998.

[3] K. Asai, S. Matsuoka, and A. Yonezawa. Duplication and partial evaluation – for a better understanding of reflective languages. *Lisp and Symbolic Computation*, 9(2/3):203–241, 1996.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[5] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. 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.

[6] D. G. Bobrow, R. P. Gabriel, and J. L. White. CLOS in context – the shape of the design space. In Paepcke [80], pages 29–61.

[7] K. A. Bowen. Meta-level techniques in logic programming. In *Proceedings of the International Conference on Artificial Intelligence and its Applications*, Singapore, 1986.

[8] M. Braux and J. 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, Jan. 2000. ACM Press. ACM SIGPLAN Notices, 34(11).

[9] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in SmallTalk-80. In OOPSLA 89 [79], pages 419–431. ACM SIGPLAN Notices, 24(10).

[10] J.-P. Briot, R. Guerraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, Sept. 1998.

[11] A. G. Bromley. The evolution of Babbage's calculating engines. *Annals of the History of Computing*, 9(2):113–136, April-June 1987.

[12] D. Caromel, F. Huet, and J. Vayssière. A simple security-aware MOP for Java. In A. Yonezawa and S. 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*, pages 118–125, Kyoto, Japan, Sept. 2001. Springer-Verlag.

[13] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, Sept. 1998.

[14] W. 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.

[15] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 285–299, Austin, Texas, USA, Oct. 1995. ACM Press. ACM SIGPLAN Notices, 30(10).

[16] S. Chiba. Implementation techniques for efficient reflective languages. Technical Report 97-06, Department of Information Science, University of Tokyo, 1997.

[17] S. 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.

[18] S. Chiba. Load-time structural reflection in Java. In E. 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.

[19] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In F. Pfenning and Y. 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, Sept. 2003. Springer-Verlag.

[20] P. Cointe. Metaclasses are first class: the ObjVLisp model. In Meyrowitz [74], pages 156–162. ACM SIGPLAN Notices, 22(12).

[21] P. Cointe, editor. *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection 99)*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, July 1999. Springer-Verlag.

[22] C. Consel and O. 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, Jan. 1993. ACM Press.

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

[24] W. De Meuter. Agora: The scheme of object-orientation, or, the simplest MOP in the world. In I. Moore, J. Noble, and A. Taivalsaari, editors, *Prototype-based Programming: Concepts, Languages and Applications*, pages 247–272. Springer-Verlag, 1999.

[25] K. De Volder and T. D'Hondt. Aspect-oriented logic meta-programming. In Cointe [21], pages 250–272.

[26] K. de Volder and P. Steyaert. Construction of the reflective tower based on open implementations. Technical Report VUB-PROG-TR-95-01, Vrije Universiteit Brussels, Belgium, Jan. 1995.

[27] J. des Rivières. The secret tower of CLOS. In *Proceedings of the OOPSLA/ECOOP 90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, Oct. 1990.

[28] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347, Aug. 1984.

[29] E. W. Dijkstra. The structure of THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[30] R. Douence and M. 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.

[31] R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1):7–34, 2001.

[32] J.-C. Fabre and S. Chiba, editors. *Proceedings of the ACM OOPSLA 98 Workshop on Reflective Programming in Java and C++*, Oct. 1998.

[33] J.-C. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud, and Z. 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.

[34] J. Ferber. Computational reflection in class based object oriented languages. In OOPSLA 89 [79], pages 317–326. ACM SIGPLAN Notices, 24(10).

[35] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In OOPSLA 89 [79], pages 327–335. ACM SIGPLAN Notices, 24(10).

[36] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355, Aug. 1984.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, October 1994.

[38] A. Goldberg and A. Kay. Smalltalk-72 instruction manual. Technical Report SSL-76-6, Xerox Palo Alto Research Center, Palo Alto, California, 1976.

[39] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[40] M. Golm and J. Kleinöder. Jumping to the meta level, behavioral reflection can be fast and flexible. In Cointe [21], pages 22–39.

[41] B. Gowing and V. 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.

[42] B. Gowing and V. Cahill. Meta-object protocols for C++: The Iguana approach. In Kiczales [51], pages 137–152.

[43] M. Halpern. Binding. In *Encyclopedia of Computer Science*, page 125. Chapman & Hall, 1993.

[44] B. Hayes. The post-OOP paradigm. *American Scientist*, 91(2):106–110, March-April 2003.

[45] M. H. Ibrahim. Report of the workshop on reflection and metalevel architectures in object-oriented programming. In N. Meyrowitz, editor, *Proceedings of the 5th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA/ECOOP 90)*, Ottawa, Canada, Oct. 1990. ACM Press. ACM SIGPLAN Notices, 25(10).

[46] Y. Ichisugi, S. Matsuoka, and A. 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, Nov. 1992.

[47] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, 1993.

[48] A. Kay. Software: Art, engineering, mathematics, or science? Foreword in the book "Squeak: Object-Oriented Design with Multimedia Applications", by Mark Guzdial, Prentice Hall, 2001.

[49] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, Nov. 1991.

[50] G. 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.

[51] G. Kiczales, editor. *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, CA, USA, Apr. 1996.

[52] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat, and D. G. Bobrow. Meta-object protocols: Why we want them and what else they can do. In Paepcke [80], pages 101–118.

[53] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[54] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. 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.

[55] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. 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.

[56] J. Kleinöder and M. 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.

[57] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Intelligence*, 1(1):11–46, 1986.

[58] J. Lamping, G. Kiczales, L. H. R. Jr., and E. 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.

[59] T. Ledoux. OpenCorba: a reflective open broker. In Cointe [21], pages 197–214.

[60] T. Ledoux and N. Bouraqadi-Saâdani. Adaptability in Mobile Agent Systems using Reflection. RM 2000, Workshop on Reflective Middleware, Apr. 2000.

[61] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Proceedings of the 1st International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86)*, pages 214–223, Portland, Oregon, USA, Oct. 1986. ACM Press. ACM SIGPLAN Notices, 21(11).

[62] P. Maes. *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[63] P. Maes. Concepts and experiments in computational reflection. In Meyrowitz [74], pages 147–155. ACM SIGPLAN Notices, 22(12).

[64] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, Alghero, Sardinia, Oct. 1988.

[65] J. Malenfant, C. Dony, and P. Cointe. A semantics of introspection in a reflective prototype-based language. *Lisp and Symbolic Computation*, 9(2,3):153–180, 1996.

[66] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In Kiczales [51], pages 1–20.

[67] K. V. Marke. *The Use and Implementation of the Representation Language KRS*. PhD thesis, Vrije Universiteit Brussels, Belgium, Apr. 1988.

[68] H. Masuhara, S. Matsuoka, and A. Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, 1994.

[69] S. Matsuoka, H. Ogawa, K. Shimura, and H. T. Y. Kimura, K. Hotta. OpenJIT — a reflective Java JIT compiler. In Fabre and Chiba [32], pages 16–20.

[70] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In America [1], pages 231–250.

[71] J. McAffer. Meta-level architecture support for distributed objects. In *International Workshop on Object-Orientation in Operating Systems (IWOOS 95)*, 1995.

[72] J. McAffer. Meta-level programming with CodA. In Olthoff [78], pages 190–214.
[73] J. McAffer. Engineering the meta-level. In Kiczales [51], pages 39–61.
[74] N. Meyrowitz, editor. *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, Orlando, Florida, USA, Oct. 1987. ACM Press. ACM SIGPLAN Notices, 22(12).
[75] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In M. Tokoro and R. 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.
[76] H. Okamura, Y. Ishikawa, and M. 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, Nov. 1992.
[77] A. Oliva and L. E. 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.
[78] W. 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, Aug. 1995. Springer-Verlag.
[79] *Proceedings of the 4th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 89)*, New Orleans, Louisiana, USA, Oct. 1989. ACM Press. ACM SIGPLAN Notices, 24(10).
[80] A. Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.
[81] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
[82] R. Rao. Implementational reflection in Silica. In America [1], pages 251–266.
[83] B. Redmond and V. Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, June 2000.
[84] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, pages 205–230, Málaga, Spain, June 2002. Springer-Verlag.
[85] F. Rivard. Smalltalk: a reflective language. In Kiczales [51], pages 21–38.
[86] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. pages 110–119.
[87] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In S. 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.
[88] S. K. Shrivastava, G. N. Nixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan. 1991.
[89] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science, 1982.
[90] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, Jan. 1984.

[91] P. Steyaert. *Open Design of Object-Oriented Languages – A Foundation for Specializable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussels, Belgium, 1994.

[92] R. J. Stroud. Transparency and reflection in distributed systems. *ACM Operating System Review*, 22(2):99–103, Apr. 1993.

[93] R. J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. In Olthoff [78], pages 168–189.

[94] R. J. Stroud and Z. Wu. *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter Using Metaobject Protocols to Satisfy Non-Functional Requirements, pages 31–52. CRC Press, 1996.

[95] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, third edition, 1997.

[96] SUN Microsystems. *The Java Language Specification*, 1996.

[97] SUN Microsystems. *The Java Native Code API*, 1996.

[98] SUN Microsystems. *Object Serialization*, 1998.

[99] SUN Microsystems. *Remote Method Invocation*, 1998.

[100] SUN Microsystems. *Dynamic Proxy Classes*, 1999.

[101] SUN Microsystems. *Java Platform Debugger Architecture*, 1999.

[102] SUN Microsystems. *Reflection API Documentation*, 1999.

[103] SUN Microsystems. *Enterprise JavaBeans Technology*, 2000.

[104] SUN Microsystems. *Java HotSpot Technology*, 2004.

[105] É. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In D. Batory, C. Consel, and W. 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*, pages 283–298, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[106] É. Tanter, M. Vernaillen, and J. Piquer. Towards transparent adaptation of migration policies. In *8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002)*, Málaga, Spain, June 2002.

[107] M. Tatsubori. An extension mechanism for the Java language. Master's thesis, University of Tsukuba, Japan, 1999.

[108] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. 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.

[109] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[110] A. M. Turing. Correction to: On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 43(2):544–546, 1937.

[111] J. Vayssière. *Une architecture de sécurité pour les applications réflexives – Application à Java.* PhD thesis, Université de Nice Sophia Antipolis, 2002.

[112] J. von Neumann. *The Computer and the Brain.* Yale University Press, June 1958.

[113] M. Wand and D. P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.

[114] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In N. 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, Sept. 1988. ACM Press. ACM SIGPLAN Notices, 23(11).

[115] T. Watanabe and A. Yonezawa. An actor-based metalevel 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.

[116] D. Weinreb and D. Moon. Lisp machine manual. Symbolics, Inc., 1981.

[117] I. Welch and R. J. Stroud. Dalang - a reflective Java extension. In *Proceedings of the OOPSLA 99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, Oct. 1998.

[118] I. Welch and R. J. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Cointe [21], pages 2–21.

[119] I. Welch and R. 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, Jan. 2001.

[120] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1,2), 1980.

[121] N. Wirth. On the design of programming languages. *Information Processing 74*, 1974.

[122] Z. Wu. *A New Approach to Implementing Atomic Data Types.* PhD thesis, Cambridge University, 1994.

[123] Z. Wu. Reflective Java and a reflective component-based transaction architecture. In Fabre and Chiba [32].

[124] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 98*, page 112, 1998.

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

[126] Y. 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, Oct. 1992. ACM Press. ACM SIGPLAN Notices, 27(10).

[127] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*, Computer Systems Series. The MIT Press, 1990.

[128] C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.