Lecture Notes in Computer Science 1616

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
*Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo*

Pierre Cointe (Ed.)

# Meta-Level Architectures and Reflection

Second International Conference, Reflection'99
Saint-Malo, France, July 19-21, 1999
Proceedings

Springer

Volume Editor

Pierre Cointe
Ecole des Mines de Nantes, Computer Science Department
La Chantrerie, 4, rue Alfred Kastler, BP 20722
F-44307 Nantes Cedex 03, France
E-mail: Pierre.Cointe@emn.fr

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

**Meta level architectures and reflection** : second international conference, reflection '99, Saint-Malo, France, July 19 - 21, 1999 / Pierre Cointe (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1616)
ISBN 3-540-66280-4

CR Subject Classification (1998): D.2, D.3, F.3, D.1

ISSN 0302-9743
ISBN 3-540-66280-4 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10705212 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

# Preface

This volume constitutes the proceedings of the second conference on Meta-Level Architectures and Reflection (Reflection'99), which will take place in Saint-Malo, France, from 19-21 July 1999.

Upon Reflection'96 held in San Francisco, several workshops organized in cooperation with OOPSLA, ECOOP and IJCAI, the IMSA'92 conference in Tokyo and the founding workshop in Alghero (1986), this year's conference intends to provide a new opportunity for researchers with a broad range of interests to discuss recent developments in the field. Our main goal is to address the issues arising in the definition and construction of reflective systems and to demonstrate their practical applications.

The Reflection'99 programme committee had the hard task of selecting 13 full papers (both research and experiential reports) and 6 short papers out of 44 submissions from all over the world. The selection of papers was carried out during a 1-day meeting at Xerox PARC. The only selection criteria were the quality and originality of the papers themselves. These papers, of both practical and more theoretical bent, cover a wide range of topics including programming languages, systems and applications, software engineering and foundations.
Apart from the presentation of the selected papers, the conference also welcomes John Stankovic, Jean-Bernard Stefani and Yasuhiko Yokote as three keynote speakers.

The success of such an international conference is due to the dedication of many people. I would like to thank the authors for submitting a sufficient number of papers, and the programme committee members and the additional referees for reviewing and selecting the papers contained herein. I would also like to thank Tina Silva and Christine Violeau for organizing the programme committee. Special appreciation goes to Jacques Malenfant, the conference chair, and to Mario Südholt, who was in charge of the electronic paper submissions.

The programme committee members also wish to thank their employers and sponsors for support of these conference activities. Finally, we would like to thank AITO and ACM Sigplan for having provided essential support in organizing Reflection'99.

Ultimately, the success of any conference is measured by the number of attendees who feel they spent their time well. So enjoy the conference and help us to disseminate the idea of open architectures in the software industry.

May 1999

P. Cointe
Reflection'99 Programme Chair

## Organization

Reflection'99 is organized by the Université de Bretagne Sud and École des Mines de Nantes under the auspices of AITO (Association Internationale pour les Technologies Objets).

## Executive Committee

| | |
|---|---|
| Conference Chair: | Jacques Malenfant (U. de Bretagne Sud, F) |
| Programme Chair: | Pierre Cointe (École des Mines de Nantes, F) |
| US Coordinator: | Gregor Kiczales (Xerox PARC, USA) |
| Asian Coordinator: | Satoschi Matsuoka (Tokyo Institute of Techology, J) |

## Programme Committee

Gordon Blair (Lancaster University, UK)
Gilad Bracha (Sun Microsystems, USA)
Vinny Cahill (Trinity College, IRL)
Shigeru Chiba (University of Tsukuba, J)
Pierre Cointe (EMNantes, F) Chair
Charles Consel (Irisa, F)
Jean-Charles Fabre (Laas, F)
Jacques Ferber (Lirm, F)
Dan Friedman (Indiana University, USA)
Lee Gasser (University of Illinois, USA)
Rachid Guerraoui (EPFL, CH)
Yutuka Ishikawa (Tsukuba ResearchLab, J)
Gregor Kiczales (Xerox PARC, USA)
John Lamping (Xerox PARC, USA)
Ole Lehrmann Madsen (Aarhus Uinversity, DK)
Jacques Malenfant (Université de Bretagne sud, F)
Satoshi Matsuoka (Tokyo Institute of Techonlogy, J)
José Meseguer (SRI International, USA)
Wolfgang Pree (University of Constance, D)
Jim des Rivières (OTI Inc., CDN)
John A. Stankovic (University of Virginia, USA)
Patrick Steyaert (MediaGeniX, B)
Robert J. Stroud (University of Newcastle, UK)
Carolyn Talcott (Stanford University, USA)
Yasuhiko Yokote (Sony Corporation, J)

## Additional Referees

Jim Dowling
Steven E Ganz
Mayer Goldberg
Julia Lawall
Shinn-Der Lee
Eric Maarsden
Anurag Mendhekar
Gilles Muller
Christian Queinnec
Barry Redmond
Jonathan Rossie
Juan Carlos Ruiz-Garcia
Tilman Schaefer
Kris Van Marcke
Johan Vanwelkenhuysen.
Wilfried Verachtert

## Cooperating Institutions of Reflection'99

AITO (Association Internationale pour les Technologies Objets)
http://www.aito.org/

ACM SIGPLAN (Association for Computing Machinery, Special Interest Group in Programming Languages)
http://www.acm.org/sigplan/

## Sponsoring Institutions

Conseil Régional de Bretagne
Université de Bretagne Sud
École des Mines de Nantes

# Contents

## Invited Talk 1

## Programming Languages

## Meta Object Protocols (MOPs)

## Invited Talk 2

**Middleware/Multi Media**

**Work in Progress (Short Papers)**

**Invited Talk 3**

## Applications (Experience Papers)

## Meta-Programming

# Invited Talk 1
# Reflection in Real-Time Systems

*John A. Stankovic*
*Department of Computer Science, University of Virginia*
*Charlottesville, VA 22903.*
*E-mail : stankovic@cs.virginia.ed*

Reflection was introduced as a main principle in real-time systems in the Spring real-time kernel. Its use coincided with and complemented the notions of admission control and planning-based scheduling for real-time systems. Taken together, these three principles establish a new paradigm for real-time systems. Recently, admission control is being used for soft real-time systems such as distributed multimedia and distributed real-time databases. Once again, reflection can be used as a central organizing and fundamental principle of these system.

This talk first discusses how reflection is used in the more classical real-time systems using examples from the Spring kernel and agile manufacturing. In particular, the synergistic properties that emerge from combining reflection, admission control (for guarantees), and planning will be stressed. I will also discuss how reflection helps resolve a key real-time systems dilemma: the need for abstraction to deal with the complexity versus the need for very specific, low level details to deal with the real-time requirements.

The second part of the talk discusses the more recent use of these principles in global, multimedia real-time databases using examples from the BeeHive system being built at the University of Virginia. In this system, objects are enhance to contain significant amounts of reflective information and algorithms and protocols are developed to make use of this reflective information to improve the performance of the systems along real-time, fault tolerance, security, and quality of service for audio and video dimensions.

*Professor John A. Stankovic is the BP America Professor and Chair of the Computer Science Department at the University of Virginia. He is a Fellow of the IEEE and a Fellow of the ACM. Professor Stankovic also serves on the Board of Directors of the Computer Research Association. Before joining the University of Virginia, Professor Stankovic taught at the University of Massachusetts. He has also held visiting positions in the Computer Science Department at Carnegie-Mellon University, at INRIA in France, and Scuola Superiore S. Anna in Pisa, Italy. He has served as the Chair of the IEEE Technical Committee on Real-Time Systems. Prof. Stankovic has also served as an IEEE Computer Society Distinguished Visitor, has given Distinguished Lectures at various Universities, and has been a Keynote Speaker at various conferences. His research interests are in distributed computing, real-time systems, operating systems, distributed multimedia database systems, and global virtual computing.*

P. Cointe (Ed.): Reflection'99, LNCS 1616, p. 1, 1999

# From *Dalang* to *Kava* - the Evolution of a Reflective Java Extension

Ian Welch and Robert Stroud

University of Newcastle-upon-Tyne, United Kingdom NE1 7RU
{I.S.Welch, R.J.Stroud}@ncl.ac.uk,
WWW home page:http://www.cs.ncl.ac.uk/people/
{I.S.Welch, R.J.Stroud}

**Abstract.** Current implementations of reflective Java extensions typically either require access to source code, or require a modified Java platform. This makes them unsuitable for applying reflection to Commercial-off-the-Shelf (COTS) systems. In order to address this we developed a prototype Java extension *Dalang* based on class wrapping that worked with compiled code, and was implemented using a standard Java platform. In this paper we evaluate the class wrapper approach, and discuss issues that relate to the transparent application of reflection to COTS systems. This has informed our design of a new version of *Dalang* called *Kava* that implements a metaobject protocol through the application of standard byte code transformations. *Kava* leverages the capabilities of byte code transformation toolkits whilst presenting a high-level abstraction for specifying behavioural changes to Java components.

## 1 Introduction

We are interested in the problems of applying non-functional requirements to Commercial Off-the-Shelf (COTS) software components. In an environment such as Java components are usually supplied in a compiled form without source code, and can be integrated into a system at runtime.

Metaobject protocols [20] offer a principled way of extending the behaviour of these components. Metaobjects can encapsulate the behavioural adaptations necessary to satisfy desirable non-functional requirements (also referred to as NFRs) such as fault tolerance or application level security [1][30][31] transparently at the metalevel. Ideally we want to apply these metaobjects to compiled code that executes on a standard Java platform.

We reviewed available implementations of reflective Java that could be used to implement these metaobject protocols and found that none of them met our requirements. They either relied upon customised Java platforms or required access to source code. Accordingly we developed a prototype reflective Java extension called *Dalang* based on the standard technique of using class wrappers to intercept method invocation. The advantage of *Dalang* was that it did not require access to source code and was implemented using a standard Java platform.

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 2-21, 1999

Our subsequent experiences with *Dalang*[1] have highlighted a number of problems with implementing reflection that arise from the approach of using class wrappers to implement reflection, and more generally with attempting to apply reflection transparently to existing COTS software built from components.

We have applied these lessons to the design of the successor to *Dalang* called *Kava*[2] which is based on wrapping not at the class level, but at the byte code level. *Kava* implements a runtime behavioural metaobject protocol through the application of standard byte code transformations at load time. Metalevel interceptions are implemented using standard byte code transformations, and behavioural adaptation is implemented using Java metaobject classes. Although neither byte code transformation or metaobject protocols are new, what is novel is the implementation of metaobject protocols using byte code transformation in order to provide a higher level view of component adaptation than current byte code transformation toolkits currently provide.

The rest of the paper is organised as follows. In section two we provide a review of different approaches to implementing reflection in Java. Section three gives an overview of the class wrapper approach to implementing reflection. Section four presents an evaluation of the class wrapper approach. Section five discusses some of the problems of applying reflection transparently to existing applications. In section six we discuss the advantages of using byte code transformation to unify class wrappers and wrapped objects. In section seven we outline our design for a new version of *Dalang* called *Kava* that addresses a number of the problems raised in the two previous sections. In section eight we discuss the application of *Kava*. Finally in section nine we present our conclusions.

A prototype implementation of *Dalang* has been completed and is available from *http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/dalang*. We are currently developing an implementation of *Kava*, for further information see the project page at *http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/kava*.

## 2 Review of Reflective Java Implementations

In this section we briefly review a number of reflective Java implementations and attempt to categorise them according to the point in the Java class lifecycle where metalevel interceptions (MLIs)[37] are added. Metalevel interceptions cause the switch during execution from the baselevel to the metalevel thereby bringing the base level object under control of the associated metaobject. Table 1 summarises the features of the different reflective Java implementations.

The Java class lifecycle is as follows. A Java class starts as source code that is compiled into byte code, it is then loaded by a class loader into the Java Virtual Machine (JVM) for execution, where the byte code is further compiled by a Just-in-time compiler into platform specific machine code for efficient execution.

All these implementations have drawbacks that make them unsuitable for use with compiled components or in a standard Java environment. Either they

[1] *Dalang* is the puppetmaster in Javanese *wayang kulit* or shadow-puppet plays.

[2] *Kava* is a traditional South Pacific beverage with calming properties.

**Table 1.** Comparison of Reflective Java Implementations

| Point in Lifecycle | Reflective Java | Description | Capabilities | Restrictions |
|---|---|---|---|---|
| Source Code | Reflective Java [36] | Preprocessor. | Dynamic switching of metaobjects. Intercept method invocations. | Can't make a compiled class reflective, requires access to source code. |
| Compile Time | OpenJava [32] | Compile-time metaobject protocol. | Can intercept wide range of operations, and extends language syntax. | Requires access to source code. |
| Byte Code | Bean Extender [17] | Byte code preprocessor. | No need to have access to source code. | Restricted to Java Beans, requires offline preprocessing. |
| Runtime | MetaXa [14] | Reflective JVM. | Can intercept wide range of operations, Can be dynamically applied. | Custom JVM. |
| | Rjava [15] | Wrapper based reflection allowing dynamic binding. | Intercepts method invocations, and allows dynamic extension of classes. | Custom JVM - addition of new byte code. |
| | Guaran [28] | Reflective kernel supported by modified JVM. | Interception of message sends, state access and supports metaobject composition. | Custom JVM. |
| Just-in-time Compilation | OpenJIT [25] | Compile-time metaobject protocol for compilation to machine language. | Can take advantage of facilities present in the native platform. No need for access to source code. Dynamic adaptation. | Custom Just-in-time compiler. |

require access to source code, or they are non-standard because they make use of a modified Java platform. In order to address these drawbacks we implemented a reflective extension called *Dalang* that required only access to compiled classes and use the standard Java platform. It was based upon standard class wrapper approach which we discuss in the next section.

## 3 Overview of Implementing Reflection Using Wrappers

In this section we give an overview of how class wrappers can be used to implement reflection in statically typed languages such as Java. We use *Dalang* as an example. For a detailed description of *Dalang* refer to [35].

### 3.1 The Basic Idea

The use of class wrappers or proxy classes in order to implement metalevel interception for method invocation is a common approach for adding reflection into a statically typed language such as C++[5] or Java [15][36]. This approach is similar to the Wrapper or Decorator pattern [13]. Figure 1 shows the collaboration diagram for the general case where a base level object has its method invocations intercepted and handled at a metalevel. Each role in the diagram is detailed below:

*BaseObject.* The class in this role has the behaviour of method invocations sent to it by a client adapted by the associated `Wrapper MetaObject` and any `Concrete MetaObjects`.

*MetaObject.* This abstract class takes the responsibility of defining the binding between a specific `Wrapper MetaObject` and the `BaseObject`, and implementing a general method for invoking methods of the `BaseObject`.

*Wrapper MetaObject.* This class extends the behaviour of each public method of the `BaseObject` class by redefining each method to invoke the respective method in the `BaseObject` class through a call to the `invokeMethod` method. Since each invocation is now handled by the `invokeMethod` method redefining this method will redefine the behaviour of each method invocation sent to the `BaseObject`.

*Concrete MetaObject.* There may be any number of `Concrete MetaObject` classes that are invoked by the `Wrapper MetaObject` in order to extend the behaviour of the `BaseObject` class.

The main differences between class wrapper implementations of reflection are the way that class wrappers are generated and how they are substituted for the base object class.

*Dalang* exploits the reflective hook [22] into the Java class loading mechanism provided by user-defined class loaders. User-defined class loaders can be created by subclassing `java.lang.ClassLoader` and implementing a custom `loadClass()` method. Classes are then loaded by the user-defined class loader either explicitly by application code invoking `loadClass()`, or by the

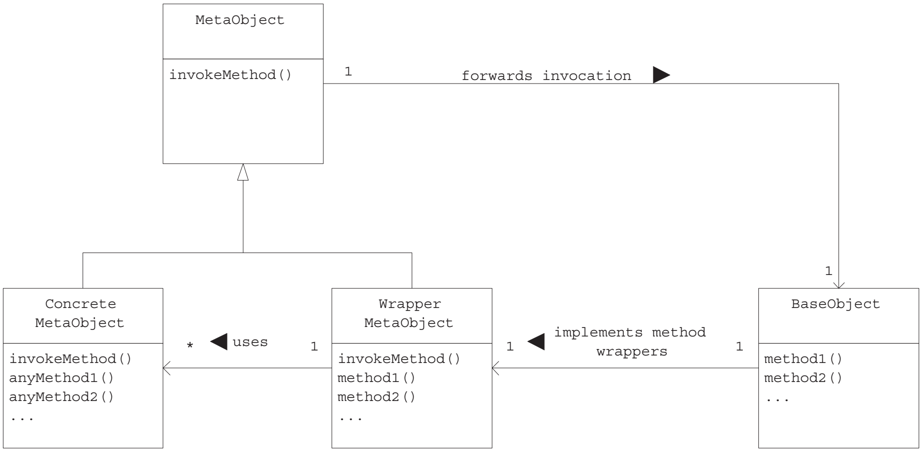


**Fig. 1.** Class collaboration diagram for class wrapper approach to implementing reflection.

JVM implicitly invoking `loadClass()` to load a class referenced by a class previously loaded by the user-defined class loader. *Dalang* uses a special reflective class loader (`dalang.ReflectiveClassloader`) that transforms classes as they are loaded into the JVM. At load time *Dalang* uses the Java Core Reflection API (JCR) in order to discover the public interface of the *Base Object*. It then generates source code for the `Wrapper MetaObject` which is dynamically compiled. The generated `Wrapper MetaObject` is then switched for the `BaseObject` through class renaming at the byte code level.

Figure 2 shows an overview of the *Dalang* architecture. In this figure the reflective class loader is acting as a bootstrapping class loader. This means that since it loads the root class of the application any class subsequently referred to will also be loaded by the reflective class loader. Therefore all classes are available to be transformed into reflective classes. Note that the metaconfiguration file controls which metaobject class binding to which base level class.

## 4 Evaluation of Class Wrapper Approach

In this section we evaluate and discuss implementation of reflection using the class wrapper approach. We focus on the following areas:

1. Wrappers
2. Inheritance
3. Security Considerations of applying Metaobjects
4. Class loaders

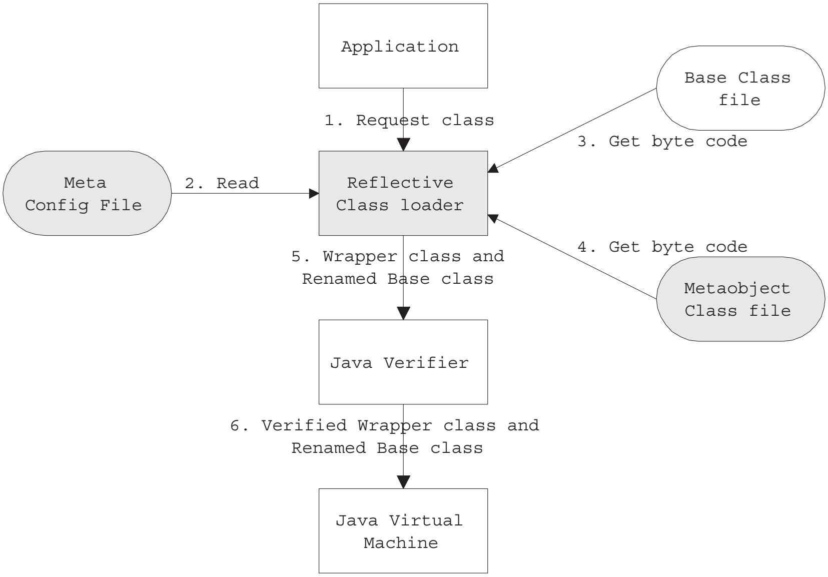


**Fig. 2.** Overview of Architecture. Shaded parts indicate *Dalang*. Non-shaded parts are the standard Java runtime system.

### 4.1 Wrappers

In a paper on using class wrappers for adapting independently developed components [16], Holzle makes the point that problems occur when the wrapper object and the object being wrapped are implemented separately as in this approach. He explains that such an approach to wrapping suffers from two major problems: the "self problem", and the encapsulation problem. The identification of these problems is not new but they have not previously been discussed in relation to the implementation of reflective systems. The "self problem" was first described by Lieberman [23]. Lieberman asserts that you cannot use inheritance based languages such as Java to implement delegation. The problem is with the rebinding of the self[3] variable that takes place when the method invocation is delegated. It is possible to work around this by passing the self variable as an additional argument in all method invocations and making all self invocations use this passed variable instead of the default. This requires that the classes follow a particular programming convention. However, since the classes being wrapped were constructed independently of their use with class wrappers it is unlikely that they would support such a convention unless we could transform the compiled classes' methods.

[3] In Java the *self* variable is represented by the *this* keyword.

Figure 3 illustrates the problem. In the left hand case where true delegation takes place all method invocations are intercepted by the wrapper for $A$ whilst in the right hand case only the first method invocation is intercepted.

This is a problem for a reflective implementation because all invocations should be intercepted. Whether the semantics of invocation are redefined is being implemented at the metalevel. For example, a metaobject implementing access control might need to redefine the semantics of only those invocations that come from clients of the class not from the class itself whereas a metaobject implementing resource controls would need to intercept and control every invocation regardless of source.

The second problem, an encapsulation problem, is based on the fact that the wrapping implemented above is only a "logical" form of wrapping. The unwrapped class is normally only hidden through a change of name. If the new name is known then it can be used by a programmer and this would allow access to the unwrapped class and new instances of it to be constructed. Another encapsulation problem arises if a method in the new class returns a pointer to the wrapped class. Once a client receives this pointer it can bypass the wrapping by sending method invocations directly to the wrapped class, bypassing the new class that logically wraps it. In order to solve these two problems Holzle proposes removing the separate class that acts as the logical wrapper by unifying the wrapper and object at binary level. If there is no separation then self will refer to the wrapper and the object, and there is no way to break encapsulation. We will discuss how this approach relates to the implementation of reflection in Java in section 6.

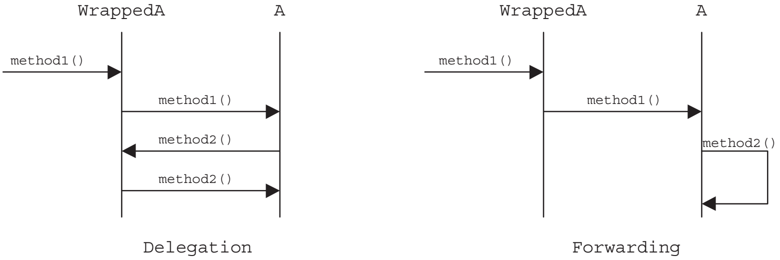


**Fig. 3.** Interaction diagrams illustrating what happens when a method invocation is sent to *WrapperA*. In the delegation scenario *WrapperA* receives *method1* invocation and invokes *method1* on $A$, $A$ then invokes *method2* on itself which as *self* is still bound to *WrapperA* results in the invocation going to *WrapperA* where it is then delegated to $A$. In the forwarding scenario *WrapperA* receives *method1* and forwards the invocations to $A$. $A$ then invokes *method2* and since *self* as rebound to $A$ it is invoked on $A$ directly bypassing *WrapperA*.

### 4.2 Inheritance

There are two different aspects of inheritance that cause us concern. The first relates to the type hierarchy imposed by the use of a class wrapper, and the second relates to inheritance of metaobject classes. We feel that such implications of inheritance are often neglected in implementations of reflective languages, with the notable exception of *CLOS* [20] and *SOM* [9]. We discuss each problem below. The first problem is that the class wrapper that we create to logically wrap the base class does not share the same type hierarchy as the baselevel class. Instead it extends the metalevel type hierarchy in order to inherit the metaobject implementation. In order that it can be used interchangeably with the baselevel class the interface of the baselevel class (including its superclasses) is replicated in the new class and the class must be tagged as implementing all the interfaces that the baselevel class implemented. Otherwise the new class wrapping the baselevel could not be downcast to any of the interfaces. However, any downcast to one of the supertypes of the baselevel class will fail as, although the interface is present, Java does not see that the class wrapper is a subtype. This could lead to problems with existing client code. Ideally the type hierarchy of the class wrapper should reflect the baselevel application level type hierarchy.

The second problem is the well known meta constraint problem [9]. If a class that is bound to a metaobject class is extended by another class and that class is bound to a different metaobject class then there should be some way of sensibly resolving the effect of both metaobject classes on the resultant class. In *Dalang* the methods inherited from the superclass are controlled by both metaobject classes, whilst those methods introduced by the extending class are only controlled by the later metaclass. In contrast, in SOM a new metaclass would be synthesised that solved the constraints on the two metaobject classes and this new metaclass would control the methods of the extended class. The *Dalang* result appears inconsistent as we would expect on a logical level that if metaobject classes are thought of as adjectives then extending a *red plane* to make a *jetplane* and then applying the adjective *fast* should result in *red* and *fast* being applied to every method of the *jetplane* not just the methods inherited from *plane*. Alternatively only the last applied metaobject class should control the behaviour of the methods of the extended class. An in-between situation seems unsatisfactory.

### 4.3 Security Considerations of Applying Metaobjects

How do we know that a base class is not bound to a malicious metaobject? This could occur through either the metaobject binding specification being changed, or the metaobject itself being substituted with a malicious version.

In the current *Dalang* security model we assume that both the metaobject classes and the binding specification represented by a metaconfiguration file are trusted. However, if the metaobject has been replaced or corrupted then this introduces a major vulnerability into the system as the malicious metaobject could completely redefine or interfere with the normal operation of the application object it is bound to.

Ideally the JDK1.2 security infrastructure for securing ordinary classes should be also used for metaobject classes. This supports the digital signing of classes using private keys and the checking of the signatures at loadtime by a `SecureClassLoader`. When a class is loaded it has its signature checked automatically and fine-grained controls are placed on its access to system resources. The `ReflectiveClassLoader` should extend the `SecureClassLoader` class and throw an exception if the class is not signed with the expected key.

Similarly, some form of protection must be implemented for the metaconfiguration file. Again a digital signature would be appropriate for the integrity check, and the structure of the file should include ownership and change information. If the metaconfiguration file fails its integrity check a runtime exception should be thrown. These two countermeasures will ensure that metaobjects and metaconfiguration file can be trusted (assuming that the Java platform, and *Dalang* implementation have not been corrupted - however, the *Dalang* implementation can also be secured using a `SecureClassLoader`).

### 4.4 Class loaders

Several authors [2][8][19] make use of a user-defined class loader to support their extensions to Java as this allows transparent interception and redefinition of class loading. The class loader is used to bootstrap the application which means that the class loader will load all classes referenced in the first class it loads. Therefore it can apply appropriate transformations to all classes. However, if another user-defined class loader is loaded then any class loaded by the new class loader will bypass the bootstrapping class loader[4]. In the case of *Dalang* this means that these classes cannot be made reflective.

This could be avoided if we changed the Java platform either by modifying the standard class loader or by changing the native code that instantiates a class. However, this would reduce the portability of *Dalang* and would not make sense if a change was required for every extension to Java. Arguably the JVM should provide some kind of facility for injecting before and after wrappers around the primordial classloader. As the result of class transformations are always verified by the JVM and don't bypass type checks then this should be a safe approach.

## 5 Transparency and Reflection

A number of authors, including ourselves, have argued that reflection can be used to transparently implement non-functional requirements (NFRs) such as security [8], fault tolerance [11], distribution [29], atomicity [31], and concurrency [33]. The idea is that components or classes developed independently can be modified

[4] There is a related problem with composing class loaders using the JDK1.2 delegation model but there are no current plans to fix it [3]

in a principled way in order to support new properties that are orthogonal to their functionality. The approach provides reusable implementations of NFRs. Our experiences with *Dalang* suggest that there are the following problems with a completely transparent application of reflection:

1. Recursive Reflection
2. Exceptions
3. Composability

The problems discussed in this section are not handled by the current *Dalang* metaobject protocol but have influenced the design of its successor *Kava*.

### 5.1 Recursive Reflection

In *Dalang* the reflective class loader loads all classes referenced by a reflective class. This means that if necessary these classes can be made reflective. Currently a referenced class will not be made reflective unless specifically bound to a metaobject class in the metaconfiguration file. However, what if the binding is not known ahead of time and the real requirement is that the binding between the initial class and metaclass be recursive in applicability? Thus if the initial reflective class references another class, that class should also be bound to the same metaclass. An example would be persistence, should all classes referenced by a root class be made persistent and a deep copy made when the class is stored? A counterexample might be Java-style serialisation, where it is not appropriate to make a deep copy and a shallow copy only is required. An example of this problem can be seen in the FRIENDS scheme proposal for tracing state changes for checkpointing using reflection [21]. Which changes of state should be tracked is application dependent - in some cases the object referenced by the object being checkpointed should also be checkpointed but in other cases it is a volatile object that should not be checkpointed. There must be some form of application control over recursive reflection at the metalevel either through some policy file such as metaconfiguration file or a metalevel program.

### 5.2 Exception Handling

Exceptions are an integral part of most modern object-oriented languages. However, it seems to us that they are rarely explicitly addressed by designers of reflective languages with the exceptions of the *Lore* and *Smalltalk* languages [10] where exceptions are implemented reflectively. Recently, our colleagues at York University [4] have argued that using reflection to implement exceptions allows adaptive runtime redefinition of exceptional behaviour. We argue that the behaviour of exception handling should not be redefinable at runtime as it makes validation intractable but nonetheless there is a need to deal with exceptions in the metalevel. Exceptions should be viewed as another possible outcome of a method call. If the behaviour of method calls is reflected upon, then in the same way as the return value of a method call is reified and available to the metalevel, so the raising of an exception by a method should be as well. This need finds expression in two ways. The first is what control and representation is appropriate for exceptions generated at the application level in the metalevel? The second is how do we deal sensibly with exceptions that occur at the metalevel that were never considered when the reflected component was constructed?

Firstly, although we do not want to redefine exception handling there are situations where being able to detect the raising of an exception at the application level, and to switch to the metalevel, is desirable. Consider the intention of taking components and distributing them through the use of reflection. Co-operating components may need to participate in a distributed exception handling routine. This requires the interception of exception raising and switching to the metalevel to propagate the exception to associated distributed components before resuming.

Secondly, when using reflection to add a non-functional characteristic new types of failure become possible that couldn't occur previously. For example, if a component is bound to a client metaobject that allows distributed invocation then a failure such as the loss of a method invocation will generate an exception at the metalevel that the component at the application level was never designed to raise. One approach to this problem would be to modify the application level component on the fly to support the new exception types that can be raised in the metalevel. However, this simply leads to propagation of the problem as any client of the component would also need to be able to handle the new exceptions. Perhaps the only sensible approach is to mask the exception and raise a runtime exception and hope that the existing application handles runtime exceptions in a sensible way. This is not an ideal solution but appears to be a necessary consequence of trying to reuse transparently an independently developed metalayer in a transparent way with an independently developed application layer.

### 5.3 Composability

In the same way that independently developed metaobject classes are combined with independently developed application components, independently developed metaobject classes may be combined with each other. With wrapper based approaches each object has one statically bound metaobject wrapper. However, this does not mean that only one behaviour can be enforced on an object. This is because chaining can be used to compose metaobjects together at the same metalevel to create a complex meta architecture (similar to the approach described in Mulet et al. [27], or Oliva [28]). We believe that using chaining to achieve composition of metaobjects is more appropriate than using reflection recursively to build a tower of metaobjects (the approach adopted by *FRIENDS* [11]). The advantage of using chaining is that each metaobject may introspect on the baselevel object whereas with a metaobject tower only the base metaobject may introspect on the baselevel object. In addition, with chaining the cost of interception and reification of method calls is only incurred once as we do not have to transform the metaobject classes involved in the tower. The main problem with the tower approach is that it confuses the meta interception mechanism with the metalevel structuring mechanism, which we feel should be kept separate. However, the tower approach is appropriate only when going to a new level of abstraction such as in the *ABCL/R* [24] approach.

## 6 Byte Code Transformation

In this section we introduce a new metaobject protocol based on byte code transformation that we are implementing as the *Kava* system. As stated in section four the separation of wrapper and wrapped class introduces problems for method call interception, encapsulation and typing. The solution as proposed by Holzle [16] is to unify the wrapper and wrapped class at the binary level. In order to achieve this at load time in Java we need to transform classes at the byte code level using byte code transformation techniques. In this section we discuss the general approach, introduce the two types of standard transformations used to implement MLIs, provide an example of the application of a standard transformation, highlight which aspects of the Java object model can be changed and examine some tricky issues.

### 6.1 General Approach

Byte code transformation has become an established technique for extending Java both syntactically and behaviourally. For example it has been used to support parametric types [2] and add resource consumption controls to classes [8]. Generic frameworks for transforming byte code such as *JOIE* [7] and *Binary Component Adaptation* [19] have been developed to make coding byte code transformations easier. However, as pointed out by the authors of *JOIE*, most of these frameworks lack a high-level abstraction for describing the behavioural adaptations. This makes coding adaptations difficult as it requires a detailed knowledge of byte code instructions and of the structure of class files. Binary Component Adaptation does support a form of a higher-level abstraction in that it has the concept of *deltaClasses* that describe structural changes to a class file in terms of methods for renaming methods, mixin type methods, etc. However, the purpose of the framework is to support software evolution rather than behavioural adaptation. This means that the focus is on adding, renaming or removing methods and manipulating the type hierarchy in order to adapt ill-fitting components to work together rather than describing behavioural adaptation.

Our contribution is to provide a high-level abstraction for adaptation of component behaviour that specifies the change to behaviour in terms of the Java object model and is implemented using byte code transformation. We exploit the *JOIE* framework to simplify the implementation of this metaobject protocol. The framework frees us from dealing with technical details such as maintaining relative addressing when new byte codes are inserted into a method, or determining the number of arguments a method supports before it has been instantiated as part of a class.

As byte code instructions and the structure of the class file preserve most of the semantics of the source code we can use byte code transformation to implement MLIs for a wide range of aspects of the Java Object model such as caller and receiver method invocation, state access, object finalisation, object initialisation and some aspects of exception handling. Like compile-time reflection we reflect upon the structure of the code in order to implement reflection. However, we work at a level much closer to the Java machine than most compile-time approaches which deal with the higher-level language. Although this means we cannot extend the syntax of the higher-level language it does mean that we can implement some kinds of reflection more easily than in a traditional compile-time MOP. For example, in the application of *OpenC++ version 2* to adding fault tolerance in the form of checkpointing *CORBA* applications [21] data flow analysis is performed on the source code to determine when the state of the object is updated. With *Kava* no such analysis would be necessary; all that would be required is to intercept the update of state of an object by reflecting upon the behaviour of the update field byte code instruction. When an update was done a flag could be set indicating that the current state should be checkpointed.

By transforming the class itself we also address the problems introduced by the separation of baseclass and class wrapper. Instead, standard transformations of byte code are used to wrap individual methods and even bytecode instructions. These micro-wrappers will switch control from the baselevel to metalevel when the methods or byte code instructions are executed at runtime. As with *Dalang* the metalevel will be programmed using standard Java classes. The metalevel will allow the customisation of the Java object model at runtime. The scope of the customisation will be determined by which methods and byte code instructions are wrapped at load time, but the exact nature of the customisation will be adjustable at runtime.

Figure 4 shows the class collaboration diagram for *Kava*. Note that there is no separate class wrapper. This is because wrappers have been applied *within* the base level class. Whenever the `BaseObject` has a method invoked or it performs state access the appropriate method of the bound `Concrete MetaObject` is invoked before and after the operation. For example, when a method invocation is received the associated `Concrete MetaObject`'s method `beforeReceiveMethod()` is called with parameters representing the source, arguments, and method. Subsequently. when the invocation takes place the method `afterReceiveMethod()` is called with a parameter representing the result of the invocation. The `Concrete MetaObject` can cooperate with any number of other `Concrete MetaObjects` to implement the overall behavioural adaptation. All `Concrete MetaObjects` extend the abstract `MetaObject` which defines the default functionality for a `MetaObject`.

### 6.2 Standard Transformations

There are two types of standard transformation applied by *Kava* to add MLIs to classes.

The first standard transformation makes use of the structure of a class file to identify blocks of byte code representing class methods, initialisation methods, and finalisation methods and then adds wrappers around them. This allows the method invocations sent to the base level to be intercepted and control handed to a metalayer. This is similar to the type of MLIs implemented in *Dalang*. In order to make the metaobject aware of self directed invocations all self invocations in the base level class are rewritten to pass an extra parameter indicating the source of the invocation.

The second standard transformation is applied to byte code instructions such as those dealing with invocation, access to state and wraps these individual instructions. Here fine-grained wrappers are applied around individual instructions. This allows the interception and switch to the metalayer when the class itself makes an invocation or accesses state. These types of interceptions would not be possible if byte code transformation had not been implemented.

In order to safely insert new byte code instructions into a class, some difficult technical issues must be solved. For example, how to handle the effects of inserting instructions on relative branches on other branch instructions and the exception table. Fortunately, we can rely upon the *JOIE* framework to take care of these low-level issues.

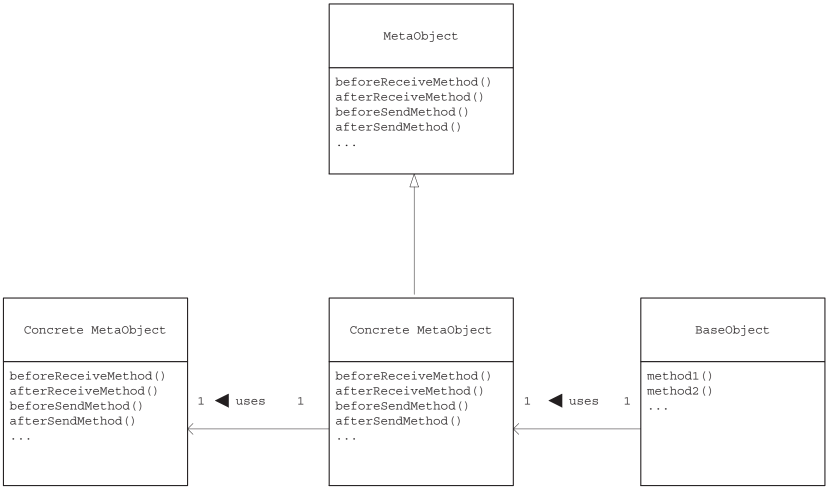


**Fig. 4.** Class collaboration diagram for *Kava*. Shows the relationship between classes.

## 6.3 Example

To provide a flavour of this approach we provide an example of the wrapping of access to a field of a base level class. Due to space constraints we present this at a high level using source code instead of byte code. Consider the following field access:

```
helloWorld = "Hello " + name;
```

At the byte code level this is rewritten to:

```
Value fieldValue = Value.newValue("Hello " + name);
meta.beforePutField("helloWorld", fieldValue);
try
{
  helloWorld = (String)fieldValue.getObject();
}
catch (MetaSuppressUpdateException e)
{
}
meta.afterPutField("helloWorld", fieldValue);
```

In this example the first line of code marshalls the value that the field is going to be updated with and then, the `beforePutField` method of the associated metaobject is invoked with parameters representing the name of the field (”helloWorld”), and the marshalled value.

At the metalevel the value may be modified in order to adjust the final value that is stored in the field. Alternatively the update of field could be suppressed by the throwing of a `MetaSuppressUpdateException`. This is caught at the base level and causes the update of the base level field not to take place.

The last line calls the `afterPutField` method of the associated metaobject with the same parameters as the initial call to the metalevel.

### 6.4 Reflective Aspects of Java Object Model

In this section we highlight the aspects of the Java object model that can be reflected upon by *Kava*.

As object-oriented operations such as object creation, state access and invocation are represented directly as byte codes it is relatively straightforward to identify them within class methods. Once identified the *JOIE* framework can be used to determine the arguments for these operations which in turn allows us to dynamically construct bytecode that marshalls their values. We can then place *before* and *after* calls around the individual operations that can manipulate the arguments and even suppress the operations. We term this *caller side reflection* because it allows us to capture behaviour such as the calling of another method by a class.

As the structure of the class file is known we can easily identify blocks of byte code representing method calls, object initialisation and finalisation methods. Again we can use the *JOIE* framework to determine the arguments for these methods and dynamically generate marshalling code. We can then insert calls to the metaobject’s *before* method appropriate to the type of method being instrumented at the beginning of the method's byte code instruction block. Then we insert calls to the metaobject's appropriate *after* method before every return instruction in the method. This allows us to intercept received invocations and represents what we term *receiver side reflection*.

Note that with both *caller side reflection* and *receiver side reflection* we add MLIs into the class' byte code. Each MLI added using byte code transformation switches control from the baselevel to the metalevel which is implemented using metaobject. This means that at runtime we can perform dynamic reflection to adjust the runtime behaviour of the baselevel class by either changing the implementation of the before and after methods of the metaobject associated with the baselevel object or by changing the binding to another metaobject. With such a facility we can easily support per-class instance metaobjects or per-instance metaobjects.

Once a class has been instantiated in the JVM no more MLIs can be added to it. This means that if a MLI has not been added at load time it cannot be exploited at any later stage. For example, if a MLI is not added to a method of a class at load time that method cannot be brought under the control of a metaobject at a later stage. One way of addressing this problem would be to add all possible MLIs by default. However, this would have considerable performance considerations. Ideally we would wish to see some form of lazy reflection taking place with MLIs being switchable on and off once the class is executing in the JVM. There is some lazy evaluation that takes place in a standard JVM where byte codes are replaced at runtime if required and perhaps this could be exploited to provide a form of lazy reflection where a class can be made reflective only as required. Another approach would be to use an *OpenJIT* to add interceptions as required when the Java byte code is compiled to native machine code.

## 7 Current Work

We are currently implementing the *Kava* system. *Kava* provides the portability benefits of a class wrapper approach without many of the problems. It also provides the opportunity to address some of the more general problems faced by reflective languages. *Kava* will have the following characteristics:

*Encapsulation.* The MLIs will be non-bypassable as class wrapper and base class are no longer separate. This is important for supporting applications such as security.

*Self Problem.* It will be possible to specify at the metalayer whether self rebinds or not as this is dependent on the metalevel functionality being provided.

*Reflective Capabilities.* Unification will allow both reification and reflection upon the receipt of method invocations and also the sending of method invocations. In addition, reification and reflection upon construction and finalisation will be possible. State access will also be under control of the metalayer. There will be some representation of exception handling at the metalayer.

*Transparency and Security.* We envisage a security model such that a trusted *Kava* kernel exists on the target machine and secure class loaders are used to download application code, metaobject code and the metaobject binding configuration. The use of a secure class loader is necessary so that we can trust the identity of the code and check for tampering. Alternatively where the client machine is totally untrusted we propose that *Kava* be applied before delivery of code to the client either in a web server or through a proxy server.

*Inheritance.* Due to unification the wrapper and wrapped class will have identical superclasses and the same implementations of application methods. This means that the class wrapper will be indistinguishable from the baselevel wrapped class. Also a subclass of the class wrapper will also inherit any reflective behaviour.

*Metaclass constraints.* Given that we can construct classes on-the-fly at runtime it will be possible to implement metaclass constraint solving [9].

*Recursive Reflection.* Like the self problem, whether reflection is applied recursively should be under the control of the metalayer, because the correct behaviour depends on the functionality provided by the metalayer. The recursive class loader should support a metaobject protocol that allows the metalevel programmer to customise its behaviour and devise sophisticated policies for handling recursive reflection.

*Metaclass Combination. Kava* will provide default metaobject classes that support metaclass combination and dynamic rebinding of metaobjects at runtime. We will provide metaobject classes that can be subclassed from that support co-operation between metaobjects, and also the ability to dynamically switch the binding between base and metalevel.

*Exception Handling.* Although we cannot redefine how the JVM handles the throwing of exceptions we can provide the ability to detect the throwing of exceptions and switch to the metalayer. This should provide the ability to support features such as distributed exception handling.

*Visibility of the Metalevel.* One aspect that should be configurable by the metalevel programmer is the visibility of the metalevel. Certainly in some cases we want to be able to send method invocations to the metalevel in order to tweak an aspect of the behaviour of the metaobject. For example, we suggest that timeout values of a metaobject that handles distributed communication could be controlled by sending messages to the metalevel.

## 8 Example Applications of *Kava*

*Kava* has wide application potential and should prove well suited to the customising the behaviour of COTS applications that are built from dynamically loaded components. As it provides a high-level abstraction for implementing bytecode transformation it could be used to implement behavioural adaptations that have already been implemented through byte code transformation e.g. enforcing resource controls on applications, instrumentation of applications, visualisation support etc. The advantage of using *Kava* would be that these adaptations could be combined as required since they have been built using a common abstraction.

## 9 Conclusions

Ideally, a reflective extension for Java that is intended to be used to adapt the runtime behaviour of COTS components should not require access to source code, or modifications to the Java platform. Unfortunately, the reflective extensions that we have reviewed do not meet these requirements.

Using class wrappers appears a promising way to implement a reflective extension that does meet the requirements. However, as our experience with *Dalang* has shown there are serious drawbacks to implementing reflection using this general approach. Aside from these problems there are other general issues that must be addressed when attempting to apply reflection transparently to COTS software. Some of the problems identified and discussed were problems associated with inheritance, encapsulation, exception handling etc.

Having identified these problems we discussed how we plan to address a number of these issues by applying reflection to bytecode transformation techniques in a new version of *Dalang* called *Kava*. Although neither reflection or byte code transformation are new concepts, what is new is the implementation of metaobject protocols using byte code transformation in order to provide a high-level abstraction for controlling component adaptation.

We are currently working on the implementation of *Kava*, and intend to use it to implement a reflective security metalayer.

## 10 Acknowledgements

We would like to thank DERA for their financial support for this work, Brian Randell for helping review an initial version of this paper, and Jim Whitman for his reviewing of the final version of the paper.

## References

[1] Benantar M., Blakley B., and Nadain A. J. : Approach to Object Security in Distributed SOM. IBM Systems Journal, Vol35, No.2, (1996).

[2] Agesen, O., Freund S., and Mitchell J. C.: Adding Type Parameterization. In Proceedings of OOPSLA'97, Atlanta,Georgia (1997).

[3] Bracha, Gilad : personal communication (1999).

[4] Burns A., Mitchell S., and Wellings A. J. : Mopping up Exceptions. In ECOOP98 Workshop Reader : Workshop on Reflective Object-Oriented Programming and Systems (1998).

[5] Chiba S and Masuda T. : Designing an Extensible Distributed Language with a Meta-Level Architecture. In proceedings of ECOOP93 (1993).

[6] Chiba S. : A Metaobject Protocol for C++. In proceedings of ECOOP95 (1995).

[7] Cohen, Geoff A., and Chase, Jeffery S. : Automatic Program Transformation with JOIE. Proceedings of USENIX Annual Technical Symposium (1998).

[8] Czaijkowski, Grzegorz and von Eicken, Thorsten. JRes: A Resource Accounting Interface for Java. Proceedings of OOPSLA'98 (1998).

[9] Danforth, S. H. and Forman, I. R. : Reflections on Metaclass Programming in SOM. In proceedings of OOPLSA'94(1994).

[10] Dony, Christophe : Exception Handling and Object Oriented Programming: towards a synthesis. Proceedings ofECOOP/OOPSLA'90. Ottawa, Canada (1990).

[11] Fabre, J.-C. and Perennou, T. : FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications. Proceedings of the Second European Dependable Computing Conference (EDCC-2) (1996).

[12] Fabre, J.-C. and Nicomette, V. and Perennou, T., Stroud, R. J. and Wu, Z.: Implementing Fault-Tolerant Applications using Reflective Object-Oriented Programming. Proceedings 25th International Symposium on Fault-Tolerant Computing(FTCS-25), (1995).

[13] Gamma E., Helm R., Johnson R., and Vlissides J. : Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley, (1995).

[14] Golm, M. : Design and Implementation of a Meta Architecture for Java. MSc Thesis (1997). University of Erlangen.

[15] Guimares, Jos de Oliveira : Reflection for Statically Typed Languages. Proceedings of ECOOP98 (1998).

[16] Holzle, U. : Integrating Independently-Developed Components in Object-Oriented Languages. Proceedings of ECOOP'93(1993).

[17] IBM. Bean Extender Documentation, version 2.0 (1997).

[18] JavaSoft, Sun Microsystems, Inc. : Reflection, 1997. JDK 1.1 documentation, available athttp://java.sun.com/products/jdk/1.1/docs/guide/reflection.

[19] Keller, R. and Holzle, U. : Binary Component Adaptation. Proceedings of ECOOP'98 (1998).

[20] Kiczales G., des Rivieres J. : The Art of the Metaobject Protocol. MIT Press (1991).

[21] Killijian Marc-Olivier, Fabre Jean-Charles, Ruiz-Garcia Juan-Carlos, and Chiba Shigeru: A Metaobject Protocol for Fault-Tolerant CORBA Applications. Proceedings of SRDS'98 (1998).

[22] Liang, Sheng and Bracha, Gilad : Dynamic Class Loading in the Java(tm) Virtual Machine. Proceedings of OOPSLA'98(1998).

[23] Lieberman, Henry : Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. Proceedings of OOPSLA'86 (1986).

[24] Masuhara H., Matsuoka S., Watanabe T. and Yonezawa A. : Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In proceedings of OOPSLA'92 (1992).

[25] Matsuoka S., Ogawa H., Shimura K., Kimura, Y., and Takagi H. : OpenJIT - A Reflective Java JIT Compiler. Proceedings of Workshop on Reflective Programming in C++ and Java, UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan ISSN 1344-3135, (1998).

[26] McAffer, J. : Meta-Level Programming with CodA. In proceedings of ECOOP95 (1995).

[27] Mulet, P. and Malenfant, J. and Cointe, P. : Towards a Methodology for Explicit Composition of MetaObjects. Proceedings of OOPSLA'95 (1995).

[28] Oliva, Alexandre; Buzato, Luiz Eduardo; Garcia, Islene Calciolari : The Reflective Architecture of Guaran. Available from: http://www.dcc.unicamp.br/ oliva.

[29] Stroud, Robert : Transparency and Reflection in Distributed Systems. Operating Systems Review 27(2): 99-103 (1993)

[30] Stroud, R. J. and Z. Wu : Using Metaobject Protocols to Satisfy Non-Functional Requirements. Chapter 3 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.

[31] Stroud, R.J. and Wu, Z. : Using Metaobject Protocols to Implement Atomic Data Types, Proceedings of ECOOP'95(LNCS-925), Aarhus, Denmark, (1995).

[32] Tatsubori, Michiaki and Chiba, Shigeru : Programming Support of Design Patterns with Compile-time Reflection. Proceedings of Workshop on Reflective Programming in C++ and Java, UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan ISSN 1344-3135, (1998).

[33] Van Oeyen Johan, Bijnens Stijn, Joosen Wouter, Robben Bert, Matthijs Frank and Verbaeten Pierre : A Flexible Object Support System as a Runtime for Concurrent Object-Oriented Languages. Chapter 13 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.

[34] Welch, Ian and Stroud Robert : Adaptation of Connectors in Software Architectures. ECOOP'98 Workshop on Component-Oriented Programming. Extended abstract published in workshop reader, and full paper published by Turku Centre for Computer Science, (1998).

[35] Welch, Ian and Stroud, Robert : Dalang - A Reflective Extension for Java. Computing Science Technical Report CS-TR-672, University of Newcastle upon Tyne, (1999).

[36] Wu, Z. and Schwiderski, S. : Reflective Java - Making Java Even More Flexible. FTP: Architecture Projects Management Limited (apm@ansa.co.uk), Cambridge, UK (1997).

[37] Zimmerman, Chris : Metalevels, MOPs and What all the Fuzz is All about. Chapter 1 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.

# Jumping to the Meta Level

## Behavioral Reflection can be fast and flexible

Michael Golm, Jürgen Kleinöder

University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Operating Systems)
Martensstr. 1, D-91058 Erlangen, Germany
{golm, kleinoeder}@informatik.uni-erlangen.de

**Abstract.** Fully reflective systems have the notion of a control transfer from base-level code to meta-level code in order to change the behavior of the base-level system. There exist various opinions on how the programming model of a meta architecture has to look like. A common necessity of all models and systems is the need to intercept messages and operations, such as the creation of objects. We analyze the trade-offs of various message interception mechanisms for Java. We show their impact on the meta-level programming model and performance. We demonstrate that it is beneficial to integrate the interception mechanism with the virtual machine and the just-in-time compiler.

## 1 Introduction

An important aspect that determines the applicability and performance of a meta architecture is the way the connection between the base-level code and the meta-level code is established and how control and information flows between the two levels.

The modification of the control flow of base-level programs - called *behavioral reflection* - is a very important part of a fully reflective architecture. Most often the meta-level code is hooked into the message passing mechanism. The process of modifying the message passing mechanism is called *reification of message passing*. The importance of this kind of reflection for “real-live problems” can be observed by looking at the huge amount of design patterns whose sole purpose is the interception of messages. The most obvious examples are the proxy and adapter patterns [10].

Although we focus our attention on Java systems, we also consider systems that are based on languages, such as Smalltalk and C++, provided that the features we are interested in are not language-dependent.

This paper is structured as follows. In Section 2 and 3 we evaluate techniques for message interception. In the remaining sections of the paper we describe a technique that requires modifications of the Java Virtual Machine (JVM). Therefore we describe the relevant structures and mechanisms of the JVM in section 4. Section 5 discusses special problems of information transfer between base and meta level. Section 6 describes the changes to the JVM that where necessary to

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 22-39, 1999

support reflection in our own reflective Java system - *metaXa*. Section 7 describes experiments that we performed to measure the overhead of behavioral reflection when a JIT compiler is used.

## 2 Implementation of behavioral reflection

There exist several systems with fairly different approaches to reifying message passing. We evaluate message interception techniques, that are used in these systems, according to the criteria *cost*, *functionality*, and *transparency*.

### 2.1 Costs

The cost for method reification must be paid as an overhead for a certain mechanism or as overhead for the whole system. The cost is paid somewhere between the time the program is developed and the time the mechanism is used. We take a closer look at the following costs:

**Virtual method call.** The invocation of a (virtual) method is a very frequently executed operation that determines the performance of the whole system. The cost of such a method call in a reflective system should be the same as in a non-reflective system.

**Reflective method call.** The cost of a reflective method call will naturally be higher than that of a normal method call. However, it should not be too high and it should scale with the complexity of the meta-level interaction. Short and fast meta-level operations should only produce a small call overhead, while complicated operations, that require a lot of information about the base level, could create a larger overhead.

**Installation.** Installation costs are paid when a method is made reflective. This cost is paid at compile time, load time, or run time.

**Memory.** Reflective systems consume additional memory for additional classes, an enlarged class structure, enlarged objects, etc.

### 2.2 Functionality

The implementation technique for the meta-level jump also affects the functionality of the whole meta architecture. We will examine which functionality each of the implementation techniques is able to deliver, according to the following questions:

- Can behavioral reflection be used for single objects or only for object groups, such as all instances of a class?
- What context information is available at the meta-level method? Is it possible to obtain parameters of the intercepted call? Is it possible to obtain information about local variables of the call site?
- Is it possible to build libraries of reusable metaobjects?

### 2.3 Transparency

As base-level code and meta-level code should be developed and maintained independently from each other, it is essential that the actions of the meta level are transparent to the base-level program. Programmers should not be forced to follow certain coding rules to exploit the services of the meta level. Forcing the programmer to follow certain coding rules would impede resuse of class libraries and applications - one of the most compelling reasons for extending an *existing* systems. If we extend an existing language or runtime system we should take care to guarantee the downward compatibility of the system.

A change in the behavior of a program should be actuated by the meta-level programmer and should not be a side effect of the implementation of the reflective mechanism.

When we consider transparency in the Java environment we must ensure that:

- the identity of objects is preserved (otherwise the locking mechanism would be broken and data inconsistencies can result),
- inheritance relations between classes are not visibly changed (such a change can become visible if the instanceof operator or an invocation of a superclass method or constructor differ from their normal behavior)

## 3 Implementation techniques

### 3.1 The preprocessor approach

With the availability of JavaCC [23] and a Java grammar for this compiler generator it seems to be easy to build Java preprocessors. However, we encountered several difficulties while building a preprocessor that inserts code at the begin and end of a method (reification at the callee site) and at each place where a method is called (caller site reification) [2]. While the caller site reification required only an additional try-finally block enclosing the whole method, insertion of code at the caller site was more difficult. If the method has arguments and an expression is passed as argument, this expression has to be evaluated first. This requires introducing new local variables to which the results of these expressions are bound. This must be done confoming to the Java evaluation order.

The preprocessor approach has the disadvantage that the preprocessor must be changed when the base level language is changed. A system that only operates at the bytecode or virtual-machine level must only be modified if the JVM specification is changed. In the past, the Java language was changed more frequently than the JVM specification.

OpenC++ [4] and OpenJava [5] are examples of preprocessors that insert interceptor methods in the base class.

Preprocessor based systems need full information about the used classes at compile time and can not handle dynamic class loading. This can only be done by paying the installation cost at runtime, as done in the classloader and proxy approach.

### 3.2 The classloader approach

Generating stubs at compile time means that the stub generator has no information about which stubs are really needed, because it is not known which classes are loaded during a program run. To cope with this problem we can move the stub generation from compile time to load time.

The Java system makes the class loading mechanism available to application programs. This reification of class loading is used by systems like JRes [8] and Dalang [25] to modify classes when they are loaded. The original class is renamed and a new class is generated that has the name of the loaded class but contains only stub code to invoke the metaobject. The rest of the program that uses the original class now gets the created class.

By its very nature a classloader approach can only be used for class-based reflection.

Szyperski [24] describes problems that appear when the wrapped object invokes methods at its self reference. The two presented choices are to invoke the method at the wrapper object or at the wrapped object. In our opinion an invocation at the self reference is only a special case of an incoming/outgoing message. So it would be natural not to bypass the wrapper but invoke these methods also at the wrapper object.

### 3.3 The proxy approach

To overcome the restriction of being class-based, one needs to modify the object code at the time the object is created or even later in the object's life. Therefore the proxy class must be created and compiled at run time. A separate compiler process must be created. This means that there is a high installation cost that has to be paid at runtime.

Identity transparency in the proxy approach means that the proxy and the original object are indistinguishable. This is only ensured if the proxy is the only object that has a reference the original object.

Inheritance transparency can not be accomplished with the proxy approach, because the proxy must be a subclass of the original class [13] . Being a subclass of the wrapped object has the consequence, that all fields of the original object are present in the proxy thus duplicating the memory cost. These problems can only be avoided, if the original object is exclusively used via an interface [22]. Then the proxy implements the same interface as the original class.

The proxy approach is also used in reflective systems that are based on Smalltalk [11], [19]. Smalltalk has a very weak type system which makes it easily possible to replace an object with a proxy. The object and the proxy need not even provide the same interface. The proxy object only defines the "doesNotUnderstand" method, which is called by the Smalltalk runtime if the called method is not implemented by the receiver. Smalltalk systems implement the method invocation mechanism in a way that adds only little overhead when using the "doesNotUnderstand" method.

### 3.4 The VM approach

Being able to extend the JVM provides complete functionality while preserving transparency.

There are two places where a method call can be intercepted: at the caller or at the callee. At first sight this seems to be no difference at all. But when considering virtual methods one immediately sees the difference. If we want to intercept all methods that are invoked at a specific object X of class C, we have to intercept the method at the callee. Otherwise we are forced to modify every caller site that invokes a method at class C or its superclasses. On the other side, if we are interested in the invocations that are performed by a specific object we have to modify the caller site.

When extending a JVM one has several alternatives to implement the interception mechanism:

**New bytecode.** All interesting bytecodes (invokevirtual and invokespecial) are replaced by reifying pedants (e.g. invokevirtual_reflective). This is a clean way to create reflective objects but requires an extension of the bytecode set. Replacing the original bytecode by a new one has only a small installation cost. This new bytecode must be understood by the interpreter and the JIT compiler. Additionally, if the system allows the structural reification of a method's bytecodes the new bytecode should not become visible, because this bytecode is only an implementation detail of the message interception mechanism.

**Extended bytecode semantics.** The interpretation process of interesting bytecodes is extended. Guaraná [21] uses this approach. Creating additional cost for each virtual method invocation degrades the performance of the whole system. Using the kaffee JIT compiler, Oliva et al. [21] measured the invokevirtual overhead on different Pentium and Sparc processors .The overhead they measured on the Pentium Pro (158%) conforms with our measurement on the Pentium 2 (see Section 7.2 and 7.3).

**Code injection.** Method bytecodes of C.m() are replaced by a stub code which jumps to the meta space. This approach is compatible to JIT compilation. Because normal Java bytecode is inserted, the code can be compiled and optimized by a normal JIT compiler.

Code modification is a powerful technique, that can be used in a variety of ways.

- *Kind of injected code*: The usual way is to execute a virtual method, but we can imagine a static method or just code fragments. Static methods and code fragments can be inlined in the base-level code.
- *Granularity and Location*: Aspect Oriented Programming [14] is such a powerful approach because it provides very fine-graned code injection. This, however, requires to write a complex injection mechanism - called weaver- that can not be generally reused. Furthermore the combination of different aspects, i.e. different code fragments, leads to complex interferences between the injected code sequences.

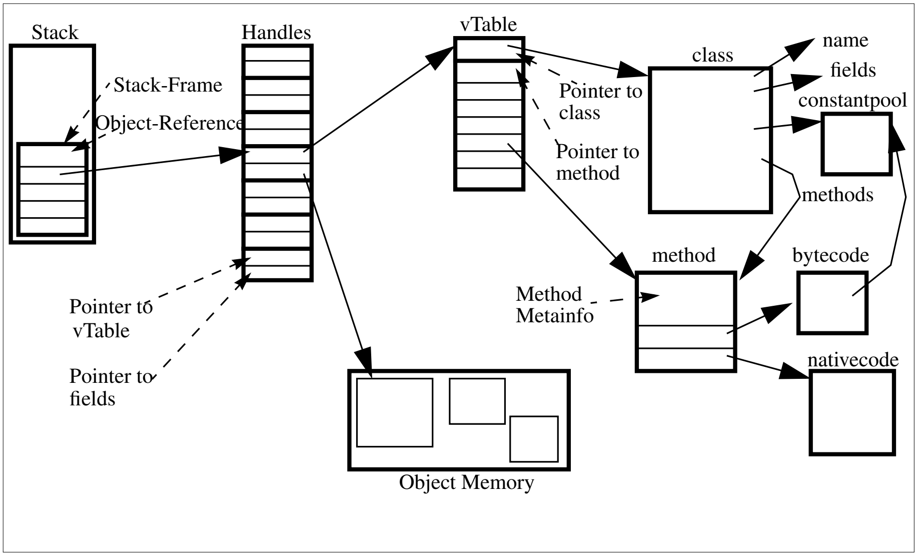


Fig. 1 Classes, Methods, and References in a JVM

Most meta architectures allow the injection of code only at distinct injection points, such as method calls. This makes the job of code injection a lot more easier.

- *Kind of information for the injected code*: This depends on the location of the injected code. Before a method call, the method parameters and local variables of the caller could be interesting; before a new operation, the class of the object is interesting.

# 4 Java VM implementations

To understand the modifications of the virtual machine that are necessary to enable reification of message passing, one needs a overall understanding of the data structures and mechanisms used inside a JVM. While the JVM specification [16] does not enforce a specific implementation we will base our discussion on the JVM structure outlined in [26]. There exist many JVM implementations that differ from this structure in several aspects [20], [6], [7].

Fig. 1 sketches the relationship between the stack, references, objects, classes, and methods. Local variables are stored in a stack frame. If the local variable has a reference type it contains a pointer to a handle. This handle contains a pointer to the object data and to the virtual method table (vTable). The vTable contains a pointer to the object's class and a list of pointers to the object's methods.

When a virtual method is invoked, the callee reference and all method arguments are pushed on the stack. Then the invokevirtual bytecode is executed. The invokevirtual opcode has three arguments: a class name, a method name and a method signature that are all looked up in the constantpool of the class. The constantpool entry must be resolved to a pointer to a class and an index into the vTable of this class. The method is then looked up using the vTable of the receiver object.

## 5 Passing information to the meta level

The cost of the meta-level jump itself is proportional to the information that is passed to the meta level. Different metaobjects have different requirements on the kind of information and the way this information is passed. In the following we assume that meta-level code is represented by a virtual method.

Meta interaction is cheapest if the meta level only needs a trigger, just informing the metaobject about the method call but not passing any information to the metaobject.

A bit more expensive is passing control to a meta-level method that has the same signature as the intercepted base-level call [12]. This interception mechanism has the advantage that it is not necessary to copy arguments.

An interaction that is more useful and is commonly used by many meta-level programs in metaXa is the invocation of a generic meta-level method. All arguments of the base-level call are copied into a generic container and this container is passed to the metaobject. The metaobject can then inspect or modify the arguments or pass them through to the original base-level method.

Some sophisticated metaobjects also need information about the broader context where a method was called. Especially the call frame information is needed in addition to the arguments. This information is also packed into a generic container and passed to the metaobject.

While the simple passing strategies are appropriate for situations where the interaction with the meta level has to be fast, some metaobjects need more complex strategies to fulfill their tasks. Thus, we obviously need a mechanism to configure the kind of interaction. Such a configuration can be regarded as a meta-meta level. This meta-meta level is concerned with the kind of code that is provided by the meta level and how this code is injected into the base-level computation. It is also concerned with the way information is passed to the meta level and the kind of this information. The JIT compiler that is described in Section 7 can be considered as such a meta-meta level.

## 6 How it is done in metaXa

In this section we describe the modifications to the virtual machine necessary to build a system that allows reification of message passing. The modifications allow a level of *transparency* and *functionality* that can not be achieved with the other approaches. In Section 7 we explain how the *cost* can be minimized by using a JIT compiler.

### 6.1 Shadow classes

Class-based meta architectures associate one metaobject with one base-level class. The reference to the metaobject can be stored in the class structure. Instance-based meta architectures can associate a metaobject with a specific base-level object. The metaobject reference must be stored in an object-specific location. One could store it together with the object fields or in the object handle. Storing it in the handle has the advantage to be reference-specific. Another interesting idea is to generate the slot only when needed, as in MetaBeta with dynamic slots [3]. We have not investigated yet if this mechanism can be implemented efficiently in a JVM. Storing the metaobject reference in the handle means an additional memory consumption in the handle space of 50%. This also applies for VMs where references point directly to the object, without going through a handle. In such an architecture the size of each object must be increased by one word to store the metaobject link.

In metaXa we use a different mechanism to associate the meta object with the base-level object. When a metaobject is attached to an object, a *shadow class* is created and the class link of the object is redirected to this shadow class. The shadow class contains a reference to the associated meta object.

A metaobject can be used to control a number of base-level objects of the same class in a similar manner. It is not necessary to create a shadow class for every object. Shadow classes are thus installed in two steps. In the first step, a shadow class is created. In the second step, the shadow class is installed as the class of the object. Once a shadow class is created, it can be used for several objects.

A shadow class C' is an exact copy of class C with the following properties:

- C and C' are undistinguishable at the base level
- C' is identical to C except for modifications done by a meta-level program
- Static fields and methods are shared between C and C'.

The base-level system can not differentiate between a class and its shadow classes. This makes the shadowing transparent to the base system. Fig. 2 shows, how the object-class relation is modified when a shadow class is created. Objects A and B are instances of class C. A shadow class C' of C is created and installed as class of object B. Base-level applications can not differentiate between the type of A and B, because the type link points to the same class. However, A and B may behave differently. Several problems had to be solved to ensure the transparency of this modification.

**Class data consistency.** The consistency between C and C' must be maintained. All class-related non-constant data must be shared between C and C', because shadow classes are targeted at changing object-related, not class-related properties. Our virtual machine solves this problem by sharing class data (static fields and methods) between shadow classes and the original class.

**Class identity.** Whenever the virtual machine compares classes, it uses the original class pointed to by the *type* link ( Fig. 2). This class is used for all type checks, for example

- checking assignment compatibility

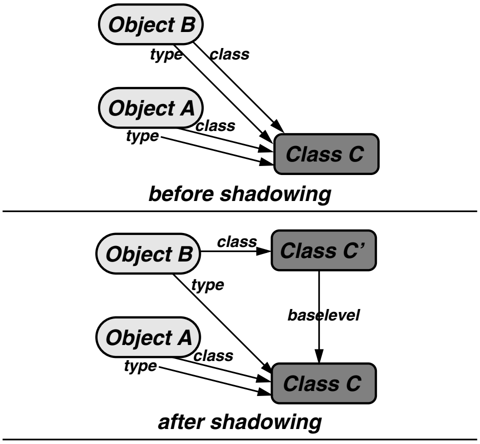


Fig. 2 The creation of a shadow class

- checking if a reference can be cast to another class or interface (checkcast opcode)
- protection checks

**Class objects.** In Java, classes are first-class objects. Testing the class objects of C and C' for identity must yield true. In the metaXa virtual machine every class structure contains a pointer to the Java object that represents this class. These objects are different for C and C', because the class objects must contain a link back to the class structure. Thus, when comparing objects of type class, the virtual machine actually compares the type links.

Because classes are first class objects it is possible to use them as mutual exclusion lock. This happens when the monitorenter/monitorexit bytecodes are executed or a synchronized static method of the class is called. We stated above that shadowing must be transparent to the base level. Therefore the locks of C and C' must be identical. So, metaXa uses the class object of the class structure pointed to by the type link for locking.

**Garbage Collection.** Shadow classes must be garbage collected if they are no longer used, i.e. when the metaobject is detached. The garbage collector follows the *baselevel* link (Fig. 2) to mark classes in the tower of shadow classes (tower of metaobjects). Shadowed superclasses are marked as usual following the superclass link in the shadow class.

**Code consistency.** Some thread may execute in a base-level object of class C when shadowing and modification of the shadow takes place. The system then guarantees that the old code is kept and used as long as the method is executing. If the method returns and is called the next time, the new code is used. This guarantee can be given, because during execution of a method all necessary information, such as the current constantpool or a pointer to the method structure, are kept on the Java stack as part of the execution environment.

**Memory consumption.** A shadow class C' is a shallow copy of C. Only method blocks are deep copied. A shallow class has a size of 80 bytes. A method has a size of 52 bytes. An entry in the methodtable is a pointer to a method and has a size of 4 bytes. Hence, the cost of shadowing is a memory consumption of about 400 bytes for a shadow class with five methods. The bytecodes of a method are shared with the original class until new bytecodes are installed for this method.

**Inheritance.** During shadow class creation a shadow of the superclass is created recursively. When the object refers to its superclass with a call to super the shadowed superclass is used. All shadowed classes in the inheritance path use the same metaobject. Fig. 3 shows what happens if a metaobject is attached to a class that has several superclasses. The shaded area marks the original configuration. Object A and B are of class $C_1$ which is a subclass of $C_0$. When the metaobject M' is attached to object B, a shadow class $C_1$' of $C_1$ is created. Later, the metaobject M" is attached to B. This causes the creation of shadow class $C_1$".

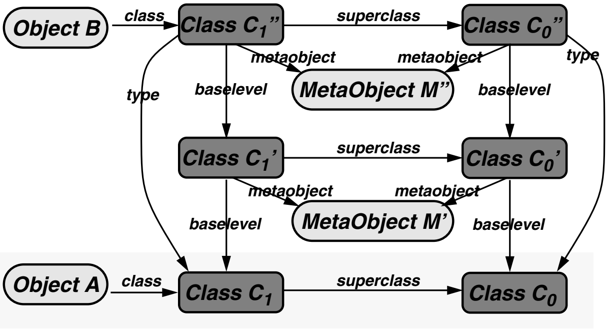


Fig. 3 Metaobjects and inheritance

**Original behavior.** In addition to the metaobject link a shadow class in metaXa also needs a link baselevel to the original class. It is used when resorting to the original behavior of the class. In all non-shadow classes the baselevel link is null. The base-level link is used to delegate work to the original class of the object.

### 6.2 Attaching metaobjects to references

In metaXa it is also possible to associate a meta object with a reference to an object. If a metaobject is attached to a reference, method invocations via this *reference* must be intercepted. A reference is a pointer, which is stored on the stack or in objects and points to a handle (Fig. 1). If the reference is copied - because it is passed as a method parameter or written into an object - only the pointer to the handle is copied, still pointing to the same handle as before. To make a reference behave differently, the handle is copied and the class pointer in the handle is changed to point to a shadow class. After copying the handle it is no longer sufficient to compare handles when checking the identity of objects. Instead, the data pointers of the handles are compared. This requires that data pointers are unique, i.e. that every object has at least a size of one byte. The metaXa object store guarantees this. Fig. 4 shows the handle-class relation after a shadow class was installed at a reference. A reference is represented by a handle, which contains a link to the object's data and a link to the object's type (class or method-table). If a shadow class must be attached to the handle (original handle), the handle is cloned (reflective handle) and the class link is set to the shadow class.

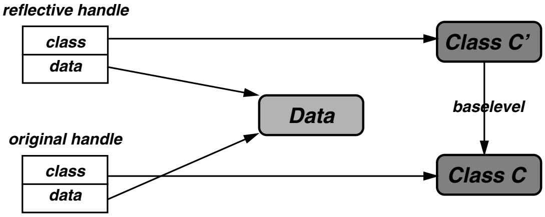


Fig. 4 Attaching to a reference

### 6.3 Creating jump code using a bytecode generator

We will now explain how the code to jump to the metaobject is created using a bytecode generator. The problems when directly generating native code are similar. When inserting code into an existing method the jump targets and the exception tables must be corrected. This is done automatically by our very flexible bytecode generator. This generator first translates a method's bytecodes to objects. One can then replace and modify these bytecode objects, and insert new ones. This flexible generator allowed us to easily implement different stub generators but it also leads to high installation costs.

We describe three different mechanisms for which stubs are created:

- a method is sent by the base-level object (*outcall*),
- an object is created by the base-level object (*new*),
- a method is received by the base-level object (*incall*).

**outcall.** An outgoing messages is caused by the invokevirtual opcode. The other opcodes that invoke methods are invokespecial to call constructors or superclass methods and invokestatic to call class methods. They are not interesting for our current considerations. The invokevirtual opcode has three arguments: a class name, a method name, and a method signature, that are all looked up in the constantpool of the class. The code generator creates the following code and replaces the invokevirtual opcode by a code sequence that performs the following operations:

(1) Create an object that contains information about the method call. This object contains the class name, method name, and signature of the called method. It also contains the arguments of the method call.

(2) At the metaobject invoke a method that is responsible to handle this interception.

**new.** The new bytecode is replaced by a stub that looks similar to the outcall stub.

**incall.** While it is rather obvious how the interception code has to look like in the outcall and new interceptions, there are alternatives for the interception of incoming invocations.

- One solution would be to insert calls to the meta level at the beginning of the method. But then it is not possible to avoid the execution of the original method other than throwing an exception. Furthermore, the return from the method can not be controlled by simply replacing the return bytecode. Instead, all exceptions have to be caught by inserting a new entry into the exception table.
- A simpler way of executing code before and after the original method is to completely replace the original method by the stub code that invokes the metaobject. The metaobject then is free to call the original method. This call executes the bytecodes of the original method by doing a lookup in the original class.

## 7 Integration into the JIT compiler

A central part of metaXa is the bytecode rewriting facility that was explained in the previous section. After we have implemented a just-in-time compiler (JIT) for the metaXa VM [15] it seems more appropriate to directly create instrumented native code instead of instrumented bytecode. Such an integration has several advantages:

- The installation cost is contained in and dominated by the JIT compilation cost, which must be paid even if no meta interaction is required.
- The JIT can inline a call to a meta object, which makes the interaction with meta objects very fast.

Another project that also uses a JIT for meta-level programming is OpenJIT [18]. OpenJIT is a JIT compiler that is designed to be configurable and extensible by metaprograms.

There are other systems that use partial evaluation to remove the overhead of the tower of meta objects, e.g. ABCL/R3 [17]. Because we use the OpenC++ [4] model of method interception, rather than the more general architecture of a meta-circular interpreter, we are faced with a different set of problems.

### 7.1 Inlining of meta-level methods

The overhead of a meta interaction should grow with the amount of work that has to be done at the meta level. This means especially that simple meta-level operations, such as updating a method invocation counter, should execute with only a minimal overhead. On the other side, it should be possible to perform more complex operations, such as forwarding the method call to a remote machine. In this case a user accepts a larger overhead, because the cost of the meta-level jump is not relevant compared to the cost of the meta-level operation, e.g. a network communication.

The reduce the meta-jump overhead in case of a small meta-level method we use method inlining. Inlining methods results in speedup, because method-call overhead is eliminated and inter-method optimizations can be applied. These optimizations could perform a better register allocation and elimination of unnecessary null-reference tests. A method can be inlined as long as the code that has to be executed is known at compile time. Therefore, one can generally inline only non-virtual (private, static, or final) methods. Fortunately, it is also possible to inline a method call, if the method receiver is known at (JIT-) compile time. When inlining meta-level methods, the metaobject usually already exists and is known to the JIT compiler.

The principle of inlining is very simple. The code of the method call (e.g. invokestatic) is replaced by the method code. The stack frame of the inlined method must be merged into the stack frame of the caller method. The following example illustrates this merging. If a method test() contains the method call a = addInt(a, 2), this call is compiled to

```
aload 0
iload 1
iconst_2
invokenonvirtual addInt(II)I
istore 1
```

The method int addInt(int x, int y) { return x+y; } is compiled to

```
iload 1
iload 2
iadd
ret
```

The merged stack frame has the following layout:

| merged stack | stack of test() | stack of addInt() |
|---|---|---|
| 0 | local 0 | |
| 1 | local 1 | |
| 2 | operand 0 | local 0 |
| 3 | operand 1 | local 1 |
| 4 | operand 2 | local 2 |
| 5 | | operand 0 |
| 6 | | operand 1 |

The created code would look as follows (bytecodes have stack positions as operands):

```
aload 0 -> 2
iload 1 -> 3
iconst #2 -> 4
checkreference 2
iload 3 -> 5
iload 4-> 6
iadd 5, 6 -> 5                    addInt()
istore 5 -> 2
istore 2 -> 1
```

Several optimizations can now be applied to the code. Provided that the method test() only contains the addInt() method call, the code can be transformed using copy propagation and dead code elimination [1] to

```
checkreference 0
iadd 1, #2 -> 5
istore 5 -> 2
istore 5 -> 1
```

Virtual methods can be inlined, if no subclasses override the method. If the JVM loads a subclass that overrides the method, the inlining must be undone. Such code modifications must be done very carefully, because a different thread could be executing this code.

### 7.2 Experiments

We measured the execution times for the following operations:

**ccall and c++call.** To have an idea how good or bad the performance of the meta-level interaction actually is, we measured the time for a function call in C and for a virtual method call in C++ (using the gcc compiler with -O2 optimization).

**vcall.** Invocation of a virtual method through a method table, doing the required null reference check before the method lookup. The called method immediately returns. x instructions are executed on the code path. x memory locations are read.

**intercept.** A call to a virtual method of a different object is inserted before the original method invocation. The object to be called is known at compile time. This allows inlining of the object reference into the code. It is necessary to register this reference with the garbage collector and unregister the reference if the compiled method is garbage collected. The injected method is called *meta method* in the following. The meta method has the same signature as the original method. The meta method returns immediately.

**counter.** A very common application for meta methods is counting of base-level method invocations. The meta method increments the counter variable. If this counter reaches a certain value (e.g. 5000), a different method (e.g. action()) is called to perform a meta-level computation, e.g. to make a strategic decision. The bytecode for this meta method is shown below:

```
Method void count()
  0 aload_0
  1 getfield #8 <Field int counter>
  4 sipush 5000
  7 if_icmple 14
 10 aload_0
 11 invokevirtual #7 <Method void action()>
 14 return
```

We measured the times for the following operations:

- not taken: The threshold is never reached and the action method is never called.
- taken1: The threshold is set to 1. This means that the action method is invoked before each invocation of the base-level method.
- taken10 and taken100: The threshold is set to 10 (100). The action method is invoked before every 10th (100th) call of the base-level method.

**argobj.** The method arguments are copied into a parameter container. This container is then passed to the meta method. The elements of the container are very special because they must contain reference types as well as primitive types, which is not possible when a Java array is used as the container. We rejected the solution to use wrapper classes for the primitive types (the Reflection API does it this way), because this means that additional objects must be created for the primitive parameters. Instead we use a container whose element types can be primitive types and reference types at the same time. As this is not allowed by the Java type system, we have to use native methods to access the container elements. Furthermore, the garbage collector must be changed to cope with this kind of container objects. The test performs the following operations:

- the parameters are read from the stack and written into the container object
- a virtual method is called with this object as parameter
- this method uses a native method to invoke the original method with the original parameters

This test is an optimized version of the metaXa method interception.

**inlinedobj.** We now instruct the JIT to inline the call. Because the meta method passes the call through to the original method, the result of the inlining should be the original method call.

A requirement for inlining is, that the implementation, that is used to execute the method call, is known at JIT-compile time. Therefore the meta method must either be static, final, or private or the class of the metaobject is known.

The JIT has some further difficulties inlining the method, because the parameter container is accessed using native calls and native methods cannot generally be inlined. So we either have to avoid native methods or enable the JIT to inline them. In the discussion of the argobj test we explained why avoiding the native methods is not possible. In order to inline the native method, the JIT now queries the VM for a native code fragment that can be used as the inlined method. In most

cases the VM will not provide such a code segment. For the native methods that access the parameter container such a native code equivalent is provided.
To eliminate the object allocation the JIT must deduce that the parameter container object is only used in this method so that the object can be created on the stack. Our JIT does not do such clever reasoning but it can be implemented using the algorithms described in [9]. In our current implementation we give the JIT a hint that the object is used only locally. Using its copy propagator the JIT can now remove the unnecessary parameter passing through the parameter container.

### 7.3 Performance results

We were interested in the performance of each of the discussed code injection techniques when using a native code compiler. We did not measure the installation cost (which would include the compilation time) because our JIT compiler is not optimized yet. It operates rather slowly and generates unoptimized code. We used a 350 MHz Pentium 2 with 128 MB main memory and 512 kByte L2 cache. The operating system is Linux 2.0.34. The system was idle.

We first measured the time for an empty loop of 1,000,000 iterations. Then we inserted the operation, that we were interested in, into the loop body and measured again. The difference between the empty loop and the loop that executes the operation is the time needed for 1,000,000 operations. Times for an empty loop of 1,000,000 iterations in C and compiled Java code are 15 milliseconds.

| **Operation** | **execution time (in nanoseconds)** |
|---|---|
| **ccall** | 17 |
| **c++call** | 35 |
| **vcall** | 65 |
| **vcallchecked** | 164 |
| **intercept** | 190 |
| **counter/not taken** | 145 |
| **counter/taken1** | 223 |
| **counter/taken10** | 195 |
| **counter/taken100** | 151 |
| **argobj** | 14000 |
| **inlinedobj** | 250 |

### 7.4 Analysis

The argobj operation is so extremely slow compared to the other operations, because it is the only operation that jumps into the JVM, executes JVM code, and allocates a new object. All other operations execute only very few JIT-generated machine instructions. However, a general pupose interception mechanism must allow the interceptor to access the method parameters in a uniform way and independent from the method signature. So there seems to be no alternative to using an parameter container. To improve the performance of the interception the compiler must try to eliminate the object allocation overhead, which is possible - as the inlinedobj test shows. We expect that the performance of this operation can be further improved when our JIT generates code that is more optimized.

## 8 Summary

We have studied several approaches to implement reification of message passing. It became apparent that only a system that integrates the meta architecture with the JVM and the JIT compiler delivers the necessary performance, functionality, and transparency to make behavioral reflection usable. The method interception mechanism can aggressively be optimized by certain techniques, such as inlining of methods and inlining of object allocations. While the JIT compiler can automatically perform such micro-level optimizations it is necessary to have a meta-meta level that controls the configuration of the meta level and selects the appropriate mechanism for the interaction between base level and meta level. For optimal performance this meta-meta level should be part of the just-in-time compiler, or should at least be able to influence the code generation process.

## 9 References

1. V. A. Aho, R. Sethi, D. J. Ullman: *Compilers: Principles, Techniques, and Tools* , Addison-Wesley 1985
2. M. Bickel: Realisierung eines Prototyps für die Koordinierungs-Metaarchitektur SMArT in Java. , March 1999
3. S. Brandt, R. W. Schmidt: The Design of a Metalevel Architecture for the BETA Language. In *Advances in Object-Oriented Metalevel Architectures and Reflection* , CRC Press, Boca Raton, Florida 1996, pp. 153-179
4. S. Chiba, T. Masuda: Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of the European Conference on Object-Oriented Programming '93*Lecture Notes in Computer Science 707, Springer-Verlag 1993, pp. 482–501
5. S. Chiba, M. Tatsubori: Yet Another java.lang.Class . In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems* 1998
6. T. Cramer: Compiling Java Just-in-time. In *IEEE Micro* , May 1997
7. R. Crelier: *Interview about the Borland JBuilder JIT Compiler.*
8. G. Czajkowski, T. von Eicken: JRes: A Resource Accounting Interface for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, ACM Press 1998, pp. 21-35
9. J. Dolby, A. A. Chien: An Evaluation of Object Inline Allocation Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, ACM Press 1998, pp. 1-20

10. E. Gamma, R. Helm, Johnson R., J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* , Addison-Wesley, Reading, MA 1994
11. B. Garbinato, R. Guerraoui, K. Mazouni: Implementation of the GARF Replicated Objects Platform. In *Distributed Systems Engineering Journal* , March 1995
12. J. d. O. Guimaraes: Reflection for Statically Typed Languages. In *Proceedings of the European Conference on Object-Oriented Programming '98* 1998, pp. 440-461
13. U. Hölzle: Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the European Conference on Object-Oriented Programming '93*Lecture Notes in Computer Science 512, July 1993
14. G. Kiczales, J. Lamping, C. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming '97*Lecture Notes in Computer Science 1357, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo 1997
15. H. Kopp: *Design und Implementierung eines maschinenunabhängigen Just-in-time-Compilers für Java. Diplomarbeit (Masters Thesis)* , Sep. 1998
16. T. Lindholm, F. Yellin: *The Java Virtual Machine Specification.* , Addison Wesley 1996
17. H. Masuhara, A. Yonezawa: Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of the European Conference on Object-Oriented Programming '98* 1998, pp. 418-439
18. S. Matsuoka: OpenJIT - A Reflective Java JIT Compiler. In *OOPSLA '98 Workshop on Reflective Programming in C++ and Java*UTCCP Report , Center for Computational Physics, University of Tsukuba, Tsukuba, Oct. 1998
19. J. McAffer: Engineering the meta-level. In *Proceedings of the Reflection '96* 1996
20. Microsoft: *Microsoft Web Page. http://www.microsoft.com/java/sdk/20/vm/Jit_Compiler.htm* 1998
21. A. Oliva, L. E. Buzato: The Design and Implementation of Guaraná. In *COOTS '99* 1999
22. T. Riechmann, F. J. Hauck, J. Kleinöder: Transitiver Schutz in Java durch Sicherheitsmetaobjekte. In *JIT - Java Informations-Tage 1998*, Nov. 1998
23. SunTest: *JavaCC - A Java Parser Generator* 1997
24. Z. Szyperski: *Component Software - Beyound Object-Oriented Programming* , ACM Press 1998
25. I. Welch, R. Stroud: Dalang - A Reflective Java Extension. In *OOPSLA '98 Workshop on Reflective Programming in C++ and Java*UTCCP Report , Center for Computational Physics, University of Tsukuba, Tsukuba, Oct. 1998
26. F. Yellin, T. Lindholm: Java Internals. In *JavaOne '97* 1997

# The Oberon-2 Reflection Model and its Applications

Hanspeter Mössenböck, Christoph Steindl

Johannes Kepler University Linz
Institute for Practical Computer Science
Altenbergerstraße 69, A-4040 Linz
{moessenboeck,steindl}@ssw.uni-linz.ac.at

**Abstract.** We describe the reflection model of Oberon-2, a language in the tradition of Pascal and Modula-2. It provides run-time information about the structure of variables, types and procedures and allows the programmer to manipulate the values of variables. The special aspect of the Oberon-2 reflection model is that metainformation is not obtained via metaclasses. It is rather organized as structured sequences of elements stored on a disk, which can be enumerated by an iterator. This results in a simple and uniform access mechanism and keeps the memory overhead to a minimum. We also show a number of challenging applications that have been implemented with this reflection model.

## 1. Introduction

Metaprogramming, i.e. the observation and manipulation of running programs, has become an important instrument in the toolbox of today's software engineer. Pioneered by languages such as Lisp and Smalltalk, metaprogramming is now part of many modern programming languages such as Java [ArG96], CLOS [Att89] or Beta [LMN93]. If metaprogramming is not only applied to *other* programs, but also to the program that uses it, it is called *reflection*. A reflective program can obtain and manipulate information about itself.

In this paper we describe the reflection model of Oberon-2 [MöW91], a language in the tradition of Pascal and Modula-2. Oberon-2 is a hybrid object-oriented language. It provides classes with single inheritance that are declared within modules. Oberon [WiG89] is not only a programming language but also a run time environment, providing garbage collection, dynamic module loading, and so-called *commands*, which are procedures that can be invoked interactively from the user interface, thus providing multiple entry points into a system.

Commands and dynamic loading already constitute a kind of metaprogramming. True reflection, however, was added to Oberon-2 by the work of Josef Templ [Tem94]. We adapted and refined his ideas into a reflection model that allows us to answer questions like:

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 40-53, 1999

- What are the components of a record type *T* declared in a module *M*?
- What procedures are currently active? What are the names, types and values of their local variables?
- Does the caller of the currently executing procedure have a variable named *x*, and if so, what is its type and value?

Questions like these allow us to do a number of interesting things, which would not be possible with an ordinary programming language. We will show examples of such applications in Section 3 of this paper.

The rest of this paper is organized as follows. In Section 2 we describe the reflection model of Oberon-2 and its general usage. Section 3 shows a number of useful applications that we implemented on top of this reflection model, i.e. a generic output function, an object inspector, an embedded SQL facility, and an exception handling mechanism. In Section 4 we discuss security and performance issues. Section 5 summarizes the results.

## 2. The Oberon-2 Reflection Model

The reflection model of a programming language is characterized by two aspects:

- What metainformation is available about programs at run time?
- How can this information be accessed in order to observe and manipulate programs?

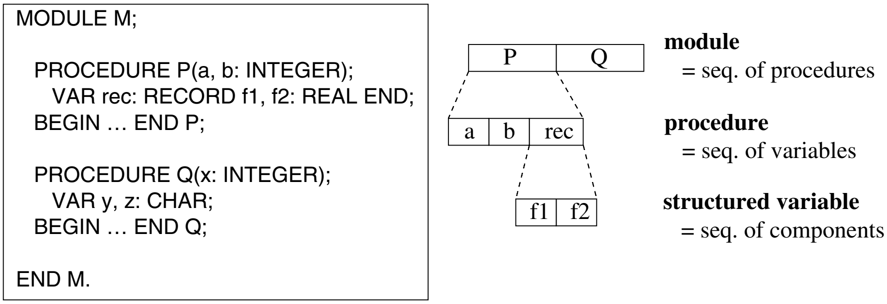
We will describe these two aspects for Oberon-2. The special thing about our approach is that all metainformation resides on disk instead rather than in main memory and that it is accessed by so-called *riders*, which iterate over the structure and parse it as required. This technique is space-efficient since no metaobjects have to be kept in memory. All access to the metainformation follows the *Iterator* pattern [GHJV95], which guarantees simple and uniform access.

Our reflection mechanism is encapsulated in a library module called *Ref* [StM96]. It defines a *Rider* type for iterating over the metainformation as well as procedures for placing riders on various kinds of metainformation sequences.

In the following sections we will first describe the structure of the metainformation and then explain how to navigate through it.

### 2.1 Metainformation

As a simple abstraction, a program can be organized as a set of hierarchical sequences. For example, a program consists of a sequence of modules. Every module is a sequence of variables, types and procedures. A procedure, in turn, is a sequence of variables, and so on. Fig. 1 shows an example of such a decomposition.



```
MODULE M;

  PROCEDURE P(a, b: INTEGER);
    VAR rec: RECORD f1, f2: REAL END;
  BEGIN ... END P;

  PROCEDURE Q(x: INTEGER);
    VAR y, z: CHAR;
  BEGIN ... END Q;

END M.
```

**Fig. 1.** Metainformation of module *M* in form of hierarchical sequences

Information about such program sequences is called *metainformation* and can be described by the following grammar in EBNF notation (curly brackets denote zero or more repetitions):

```
ProgramStruct = {Module}.
Module        = {Variable} {RecordType} {Procedure}.
RecordType    = {Variable} {Procedure}.
Procedure     = {Variable}.
Variable      = SimpleVar | RecordVar | ArrayVar.
RecordVar     = {Variable}.
ArrayVar      = {Variable}.

ProgramData   = GlobalVars | LocalVars | DynamicVars.
GlobalVars    = {Variable}.
LocalVars     = {Frame}.
Frame         = {Variable}.
DynamicVars   = {HeapObject}.
HeapObject    = {Variable}.
```

The information about the structure of a module is created by the compiler when the module is compiled. It is appended to the module's object file so that it is always copied and moved together with the object file. This avoids inconsistencies between a module and its metainformation.

The structural information is also used to interpret the program's data, which would otherwise be just a sequence of bytes without any interpretation.

## 2.2 Navigation

The metainformation is accessed by so-called *riders*, which are iterators that allow us to traverse a sequence of elements and to zoom into structured elements. The class *Rider* declared in module *Ref* looks as follows:

```
TYPE
  Rider = RECORD
    name: ARRAY 32 OF CHAR;
    mode: SHORTINT;    (*Var, Type, Proc, …, End*)
    form: SHORTINT;    (*Int, Char, Bool, Record, …*)
    ...
    PROCEDURE (VAR r: Rider) Next;
    PROCEDURE (VAR r: Rider) Zoom (VAR r1: Rider);
    ...
  END;
```

When a rider *r* is placed on an element of a sequence, *r.name* holds the name of this element; *r.mode* tells if the element is a variable, a type, a procedure, etc.; *r.form* encodes the type of the element (structured types are denoted by a special code; their components can be inspected by zooming into the element; see below).

**Iterating.** The following example shows how to traverse the global variables of a module *M* and print their names:

```
Ref.OpenVars("M", r);
WHILE r.mode # Ref.End DO
  Out.String(r.name);
  r.Next
END
```

A rider *r* is opened on the global variables of module *M*. While it is not moved beyond the last variable (*r.mode = Ref.End*) the name of the current variable is printed and the rider is advanced to the next variable by the operation *r.Next*.

**Zooming.** If a rider is placed on a structured element, it is possible to zoom into this element and to iterate over its components. For example, to access the local variables of the current procedure's caller we can zoom into the second frame on the activation stack, using the following statements:

```
Ref.OpenStack(r);
r.Next;
r.Zoom(r1)
```

*Ref.OpenStack(r)* opens a rider *r* on the frame of the currently active procedure. *r.Next* moves it to the caller's frame. *r.Zoom(r1)* zooms into that frame and sets a new rider *r1* to the first local variable in that frame. The variables of this frame can then be traversed as above using *r1*.

**Placing a rider.** Riders can be opened on various kinds of metainformation sequences as shown by the following table:

| | |
|---|---|
| *OpenVars(module, r)* | sets *r* to the first global variable of the specified module |
| *OpenStack(r)* | sets *r* to the topmost stack frame |
| *OpenPtr(p, r)* | sets *r* to the first component of the object pointed to by *p* |
| *OpenProcs(module, r)* | sets *r* to the first procedure of the specified module |
| *OpenTypes(module, r)* | sets *r* to the first record type of the specified module |

If a rider is opened on data (using *OpenVars*, *OpenStack* or *OpenPtr*), and if it is positioned on a non-structured variable, the value of this variable can be read or written using operations such as *r.ReadInt(n)* or *r.WriteInt(n)*. In this case the rider serves as a link between the data (e.g. a stack frame) and the metainformation that is used to interpret that data (Fig.2).

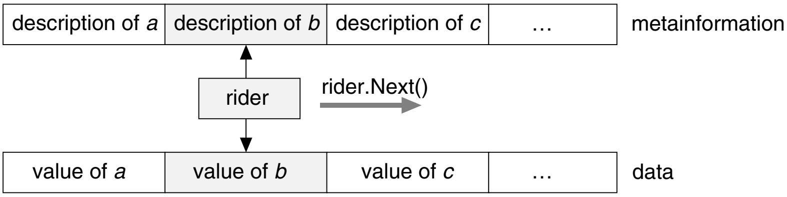


**Fig. 2.** A rider as a link between data and its metainformation

If a rider is opened on structural information (using *OpenProcs* or *OpenTypes*), there is no data to be read or written. Such riders can only be used to explore the structure of procedures and types.

Details about the *Rider* class, its fields and its operations are described in [StM96]. The difference between our implementation and the one in [Tem94] is mainly that we use a single rider type to iterate over all kinds of metainformation while [Tem94] uses special rider types for variables, procedures, types, etc.

## 2.3 Examples

The following examples should give you a rough impression of what you can do with the module *Ref* and its riders.

Assume that we want to print the names of all currently active procedures together with the names of their local variables. The following code fragment does the job:

```
VAR r, r1: Ref.Rider;
...
Ref.OpenStack(r);      (*r is on the most recent frame*)
WHILE r.mode # Ref.End DO
  Out.String(r.mod);   (*name of this frame’s module*)
  Out.String(".");
  Out.String(r.name); (*name of this frame’s proc.*)
  Out.Ln;
  r.Zoom(r1);          (*r1 is on first var. of frame*)
  WHILE r1.mode # Ref.End DO
    Out.String(r1.name); Out.Ln;
    r1.Next
  END;
  r.Next               (*move to the caller's frame*)
END
```

Of course we could do any processing with the traversed variables or procedures. For example, we could print their values and types (this was actually used for the implementation of the Oberon debugger). We could also look for all occurrences of a certain value within the variable sequence and report them to a client.

The next example looks for a global record variable named *varName* declared in a module named *modName*. Note that the names of the variable and the module need not be statically known. They could have been obtained at run time.

```
VAR
  r: Ref.Rider;
  varName, modName: ARRAY 32 OF CHAR;
...
Ref.OpenVars(modName, r);
WHILE (r.mode # Ref.End) & (r.name # varName) DO
  r.Next
END;
IF r.form = Ref.Record THEN (*found*)
  ...
END
```

# 3. Applications

In this section we show how the Oberon-2 reflection model can be used to implement a number of interesting system services. In other programming systems, such services are often part of the programming language. Reflection, however, allows us to implement them outside the language in separate library modules so that they don't increase the size and complexity of the compiler. If a user does not need a service, he does not have to pay for it. Reflection also gives the system programmer a chance to adapt these services to special needs.

## 3.1 A Generic Output Function

In most modern programming languages input/output is not part of the language but is implemented in form of library functions. The problem with this approach is that it requires a separate function for every data type that is to be read or written. A typical output sequence could look as follows:

```
Out.String("Point (");
Out.Int(p.x);
Out.String(", ");
Out.Int(p.y);
Out.String(") is inside the search area");
```

Function overloading alleviates this problem, but still requires multiple function calls to print the above message. It would be nice to have a single generic function which is able to print any sequence of variables of a program with a single call. Reflection allows us to do that.

The following output function takes a string argument with the names of the variables to be printed. For example, the call

```
Put.S("Point (#p.x, #p.y) is inside the search area")
```

prints the argument string, but before printing it, it replaces every variable that is preceded by a # with the value of this variable.

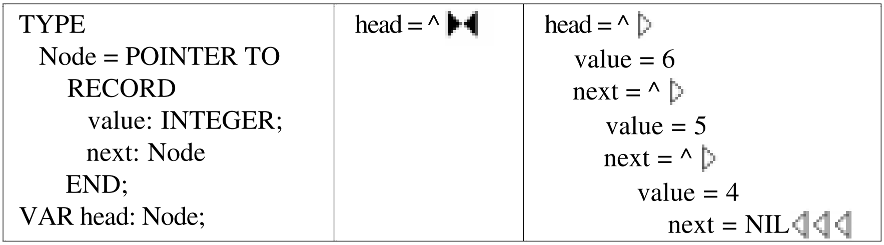
The function *Put.S* is implemented with module *Ref.* It looks up the marked variables of the argument string in various scopes. For example, a variable *rec.arr[i]* is looked up as follows:

- *rec* is first searched in the local scope of the currently active procedure (using *OpenStack*) and—if not found—in the global scope of the current module (using *OpenVars*).
- If *rec* is found and turns out to be a record variable, a rider *r* is positioned on *rec. Put.S* zooms into *rec* (using *r.Zoom(r)*) and looks for a field *arr*. If it is found, the rider *r* is positioned on it.
- Since *arr* turns out to be an array, *Put.S* zooms into this array. It starts a new search for the variable *i* (using first *OpenStack*, then *OpenVars* as above). The value of *i* is read, the rider is positioned on the *i*-th element of *arr*, and the value of this element is read. This is the value of *rec.arr[i]*. It is inserted into the output string.

## 3.2 An Object Inspector

An object inspector is a debugging tool that can be used to inspect the values of the object fields. It can be conveniently implemented with Module *Ref.* To inspect an object that is referenced by a pointer *p*, one opens a rider *r* on the object's fields using *Ref.OpenPtr(p, r)* and iterates over them. Fields with a basic data type are simply shown with their values, whereas structured variables (arrays and records) are first represented in a collapsed form that that can be expanded on demand. When a collapsed variable is expanded, a new rider is placed on the inner elements (using *r.Zoom(r1)*) and is used to traverse the inner sequence.

Fig. 3 shows an example of a variable *head* that points to a list of three nodes. The middle part of the picture shows the list in collapsed state, the right part in expanded state. The triangles are so-called *active text elements* [MöK96] that hide the inner structure of an object. If the user clicks on a filled triangle, the object structure is expanded and shown between hollow triangles. A click on the hollow triangles collapses the structure again.



| | | |
|---|---|---|
| TYPE<br>Node = POINTER TO<br>RECORD<br>value: INTEGER;<br>next: Node<br>END;<br>VAR head: Node; | head = ^ ▶◀ | head = ^ ▷<br>value = 6<br>next = ^ ▷<br>value = 5<br>next = ^ ▷<br>value = 4<br>next = NIL◁◁◁ |

**Fig. 3.** Object inspector view (collapsed and expanded)

A textual view like the one in Fig.3 is sufficient to represent data structures such as linear lists or trees, but it is less suitable for circular lists or graphs. For such purposes we have implemented a graphical tool that can also show circular data structures. This tool uses the same reflection mechanism as explained above.

### 3.3 Embedded SQL

SQL (structured query language) is a widely used standard for a database query language. It is normally used interactively from a dialog window. If a programmer wants to issue an SQL query from within a programming language (e.g. C++), however, he has to use an extended form of the language. For C++ there exists such an extension which is called *embedded SQL* [ESQL89]. It adds database query statements that are translated into library calls by a preprocessor.

We used a different implementation for embedded SQL that does not need any language extensions but is based on reflection [Ste96]. SQL queries can be specified as strings, which are passed to a function *Prepare* that analyzes and prepares them for execution. For example, one could write

```
stat := conn.Prepare("CREATE TABLE Persons FOR Person")
```

A prepared SQL statement can be executed with *stat.Execute*. Thus the statement can be executed several times without rebuilding internal data structures every time.

The SQL query can contain names of variables or types, which are then looked up in the calling program and are converted to appropriate data structures of the SQL libraries. For example, *Person* could be a type declared as follows:

```
TYPE
  Person = RECORD
    firstName, lastName: ARRAY 32 OF CHAR;
    age: INTEGER
  END ;
```

Its structure is used by the above SQL statement to create a table with the columns *firstName, lastName* and *age*. To distinguish program variables from database names (e.g. for tables and columns), variables are preceded by a colon in a query. In the following code fragement

```
VAR
  minAge: INTEGER;
  name: ARRAY 32 OF CHAR;
...
stat := conn.Prepare("SELECT firstName FROM Persons
WHERE age > :minAge INTO :name");
```

*minAge* and *name* are Oberon variables. They are looked up by reflection. The value of *minAge* is used to evaluate the WHERE clause. As a result, the database value *firstName* is transferred to the Oberon variable *name*.

Variables can be either scalar or of a record type. When record variables are specified, they are implicitly expanded to their fields. The statement

```
"SELECT * FROM Persons INTO :person"
```

is therefore equivalent to

```
"SELECT * FROM Persons INTO :person.firstName,
:person.lastName, :person.age".
```

## 3.4 An Exception Handling Mechanism

Exception handling is often part of a programming language, but it can also be implemented outside of the language, i.e. in a library module [Mil88]. Library-based exception handling is often implemented with the Unix functions *setjmp* and *longjmp*, which save and restore the machine state. We have followed a different approach based on reflection [HMP97]. Our technique has the advantage that it does not slow down programs as long as they do not raise exceptions.

Our exception handling mechanism is based on three concepts: a *guarded block* of statements which is protected against exceptions, one or more *exception handlers*, and a mechanism to *raise* exceptions (Fig. 4).

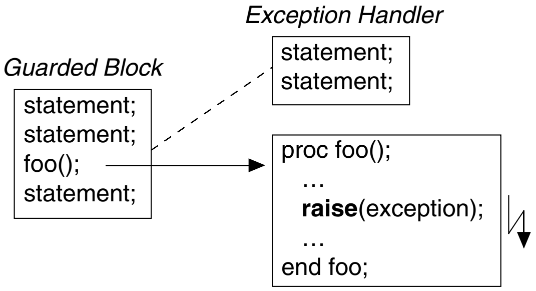


**Fig. 4.** Exception handling concepts

If an exception is raised in the guarded block or in one of the functions called from it, a suitable exception handler is called. After executing the handler, the program continues with the first statement after the guarded block. Exceptions are classes derived from a common exception class.

In our implementation, the concepts of Fig. 4 are mapped to the Oberon-2 language as follows:

- The guarded block is represented by an arbitrary procedure *P*.
- An exception handler is represented by a local procedure of *P*. It must have a single parameter whose type is a subclass of *Exceptions.Exception*.
- An exception is raised by the call of the library procedure *Exceptions.Raise(e)* where *e* is an object of an exception class.

The following code fragment shows an example (The classes *Overflow* and *Underflow* are subclasses of *Exceptions.Exception*).

```
PROCEDURE GuardedBlock;
  VAR ofl: Overflow; ufl: Underflow;

  PROCEDURE HandleOfl (VAR e: Overflow); …
  END HandleOfl;

  PROCEDURE HandleUfl (VAR e: Underflow); …
  END HandleUfl;

BEGIN
  …
  IF … overflow … THEN
    … fill the ofl object with error information …
    Exceptions.Raise(ofl)
  END;
  …
END GuardedBlock;
```

In this example, *GuardedBlock* raises an exception by calling *Exceptions.Raise(ofl)*. Procedure *Raise* is implemented with reflection. It determines the type of *ofl* and searches through the local procedures of *GuardedBlock* to find a procedure with a matching parameter type. This is the exception handler (in this example *HandleOfl*). If such a handler is found, it is called. Finally, the activation stack is unrolled so that the control is returned to the caller of *GuardedBlock*.

If no matching exception handler is found in *GuardedBlock*, the lookup continues in the caller of *GuardedBlock*. If this caller *P* contains a local procedure *H* with a matching parameter type, *H* is called as an exception handler, and then the program continues with the first statement after *P*.

If no exception handler is found in any of the currently active procedures, the program is aborted with a standard error message. The following pseudocode fragment shows the implementation of *Raise*:

```
PROCEDURE Raise (VAR e: Exceptions.Exception);
  E := dynamic type of e;
  FOR all stack frames in reverse order DO
    P := procedure of this frame;
    FOR all local procedures H of P DO
      IF H has a parameter of type E or a subtype THEN
        Invoke H(e);
        Return to the caller of P
      END
    END
  END
END Raise;
```

Except of the underlined parts, all actions of *Raise* are implemented with the reflection model described in Section 2. In particular, riders are used to traverse the stack frames, the procedures and the parameters to perform the handler lookup. The underlined actions involve stack manipulation. They are implemented using low-level facilities of Oberon-2 and are not show here (see [HMP97]). The dynamic type of an object can be obtained with an Oberon system function.

The exception handling mechanism is encapsulated in a library module with the following simple interface:

```
DEFINITION Exceptions;
  TYPE Exception = RECORD END; (*abstract base class*)
  PROCEDURE Raise (VAR e: Exception);
END Exceptions.
```

# 4. Discussion

In this section we discuss some consequences and tradeoffs of the Oberon reflection model, namely security and performance issues as well as the interference of reflection with optimizing compilers.

## 4.1 Security

The Oberon reflection model gives the systems programmer full access to all variables, types and procedures in a program, even to the private objects that are not exported from a module. This of course raises the question of security. The visibility rules of the language can be circumvented by reflection, however, this is necessary to implement system software such as debuggers or inspectors.

Although the whole structure of a program is visible to reflection, it is not possible to access it in an undisciplined way. The Oberon reflection model is strongly typed. All metainformation is read and written according to their types. It is never possible, for example, to access a pointer as an integer or vice versa.

The most critical fact about the Oberon reflection model is that data can not only be read but also written. This is sometimes necessary as for example in the Embedded SQL facility described in Section 3.3. One should use this feature very carefully. According to our experience most reflective applications don't make use of it.

Clearly, reflection allows a programmer to do more than what he could do with an ordinary programming language. But this is exactly its advantage. System programmers need sharper tools than application programmers.

## 4.2 Interference with Optimizing Compilers

An optimizing compiler may decide to keep variables in registers rather than in memory, to eliminate variables at all, or to introduce auxiliary variables, which are not in the source program. Some of these optimizations are easy to cope with in the reflection model, others are more difficult to handle. The Oberon-2 compiler does not perform aggressive optimizations. In addition to some code modifications (which do not affect the metainformation) the compiler keeps certain variables in registers. For such variables, their register numbers are stored in the metainformation, so that riders can find them. The Oberon-2 compiler does not eliminate, introduce or reorder variables. But even such cases could be handled if the metainformation carried enough information about the optimizations that the compiler performed.

## 4.3 Performance

The Oberon reflection model leads to small memory overhead at the cost of run time efficiency. The layout of our metainformation is sequential. For example, in order to access the information of the third local variable in the fourth procedure of a module, one has to skip three procedures, zoom into the fourth one and search for the third variable. Table 1 shows the approximate costs for reading (i.e. skipping) various kinds of elements in the metainformation sequence. Of course, the time to skip a procedure or a record type depends on its size.

**Table 1.** Access times for specific program elements (on a Pentium II with 300 MHz)

| *Element to be skipped* | *Cost* |
|---|---|
| Local variable | 2 s |
| Global variable | 2 s |
| Record field | 2 s |
| Record type | 9 s |
| Procedure | 15 s |

The sequential layout of the metainformation as described in Section 2.1 requires us to skip all record types before we get to the first procedure. Furthermore, in our current implementation, the global variables of a module are treated like local variables of the module body, which is considered to be a special procedure. Accessing them requires us to skip to this procedure first. Constraints like these make random access of metainformation elements somewhat inefficient. In practice, however, a typical access pattern involves both random access and sequential access so that the run time was never a serious problem in all the applications described in Section 3. As a task for further research one could try to redesign the metainformation so that it is indexed and random access becomes more efficient.

Our current implementation has the advantage that the metainformation is stored in a very compact form. For example, the metainformation of the whole Oberon compiler (9 modules, 413 procedures, 14 types, 1690 variables and 86 record fields) consumes only 23706 bytes on the disk. When it is accessed it is cached in memory so that it is not necessary to go to the disk for every access. Reading the metainformation of the whole compiler sequentially takes 30 milliseconds.

## 5. Summary

The fundamental difference between the Oberon-2 reflection model and other models is that the metainformation is not accessed via metaobjects. It is rather parsed on demand from a file (which is usually cached in main memory to avoid file operations). One advantage of this approach is the reduced number of objects needed to represent the metainformation and the reduced memory consumption. For example, when metainformation is used in a post-mortem debugger to produce a readable stack dump, it is important not to waste memory since the reason for the trap might be the lack of memory. A disadvantage of our approach is that the information is parsed again and again. But this is not time-critical as the measurements show.

The Oberon-2 reflection model is currently designed for convenient access to the structure and values of program objects. In the future, we plan to extend it with mechanisms for calling and intercepting methods.

### Acknowledgements

The Oberon-2 reflection model was originally designed by Josef Templ [Tem94] in his PhD thesis. We refined his model and adapted it for our purposes. Our colleagues Markus Knasmüller [Kna97], Markus Hof [Hof97], Martin Rammerstorfer and Günter Obiltschnig [Obi98] used the model to implement various applications on top of it. We wish to thank the anonymous referees for their comments that helped us to make the paper more focussed.

## References

[ArG96] Arnold K., Gosling J.: The Java Programming Language. Addison-Wesley, 1996

[Att89] Attardi G., et al.: Metalevel Programming in CLOS. Proceedings of the ECOOP'89 Conference. Cambridge University Press, 1989

[ESQL89] Database Language – Embedded SQL (X3.168-1989). American National Standards Institute, Technical Committee X3H2

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[HMP97] Hof M., Mössenböck H., Pirkelbauer P.: Zero-Overhead Exception Handling Using Metaprogramming. Proceeding of SOFSEM'97, Lecture Notes in Computer Science 1338, 1997

[Hof97] Hof M.: Just-In-Time Stub Generation, Proc. Joint Modular Languages Conference '97, Hagenberg, Lecture Notes in Computer Science 1204, Springer-Verlag, 1997

[Kna97] Knasmüller M.: Oberon-D, On Adding Database Functionality to an Object-Oriented Development Environment, Dissertation, University of Linz, 1997

[LMN93] Lehrmann-Madsen O., Moller-Pedersen B., Nygaard K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993

[Mil88] Miller W.M.: Exception Handling without Language Extensions. Proceedings of the USENIX C++ conference, Denver CO, October 1988

[MöK96] Mössenböck H., Koskimies K.: Active Text for Structuring and Understanding Source Code SOFTWARE - Practice and Experience, 26(7): 833-850, July 1996

[MöW91] Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991

[Obi98] Obiltschnig G.: An Object-Oriented Interpreter Framework for the Oberon-2 Programming Language, Diploma Thesis, University Linz, 1998

[Ste96] Steindl Ch.: Accesing ODBC Databases from Oberon Programs. Report 9, University of Linz, Institute for Practical Computer Science, 1996

[StM96] Steindl Ch., Mössenböck H.: Metaprogramming facilities in Oberon for Windows and Power Macintosh. Report 8, University of Linz, Institute for Practical Computer Science, 1996

[Tem94] Templ J.: Metaprogramming in Oberon. Dissertation, ETH Zurich, 1994

[WiG89] Wirth N., Gutknecht J.: The Oberon System. Software—Practice and Experience, 19(9):857-893, 1989

# Designing Persistence Libraries in Reflective Models with Intercession Property for a Client-Server Environment

Stéphane Demphlous Franck Lebastard

CERMICS / INRIA, Database Team
B.P.93, 2004, route des Lucioles, 06902 Sophia Antipolis Cedex, France
Email: {Stephane.Demphlous, Franck.Lebastard}@sophia.inria.fr
URL: http://www.inria.fr/cermics/dbteam/

**Abstract.** This paper presents an architecture where persistence is added to an object-oriented reflective model in a client-server environment. When the client and the server database management system do not share a common formalism, conversion rules must be set. If the open reflective client has intercession properties and the server does not, for example, when an open reflective language is bound to a relational database management system, we show that the conversion rules may become inadequate when a designer extends the semantics of the client language. An object-oriented reflective system is reified as a metaobject model and intercession is often designed as a metaobject protocol that can be specialized. In this case, we state that the best architecture to bring persistence to a reflective client is to extend the standard metaobject protocol with a fine-grained persistence and conversion protocol. We present such a protocol, and we illustrate it with a binding between Power Classes, an open reflective language, and ObjectDriver, our wrapper to relational databases.

## 1 Introduction

A key to design modern applications is to use the client-server paradigm. When an application needs to create or to manipulate persistent data, it is a good policy to design a persistent client addressing a server database management system (DBMS).

Now, the reflection, defined in [2] as the ability for a program to manipulate as data something representing its state during its own execution, and the reflection paradigm [3, 7, 19] have become a major issue for system definition. Languages and tools [1, 8, 9] have widened the use of reflection. So client persistent applications may be built using a reflective programming language or be themselves reflective.

Moreover, adding reflective properties to DBMSs appears to be very promising: it helps to adapt the DBMS data model, that is, the formalism in which user data and their definition (metadata) are expressed and controlled. Several studies have shown the interest of such adaptations [14]. The intercession property of many reflective systems, defined in [2] as the ability for a program to modify its own execution state or alter its own interpretation or meaning, can be applied to a DBMS, allowing the user

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 54-73, 1999

to modify or to extend the data model. Vodak [14] and Tigukat [23] are fully reflective object-oriented DBMSs (OODBMS).

So we believe that, in a near future, there will be growing needs of persistence management in reflective models with intercession properties. Two architectures may arise: the use of a reflective OODBMS, or the reuse of existing DBMSs as servers and the design of reflective persistent clients. However there are few works specifically related to this last problem [22].

Consequently, our aim, in this paper, is to present a way to design persistent clients in a reflective object-oriented system with explicit metaclasses. The respect of the possible intercession property of this system is an important part of our proposals. If a system has this property, a user can modify the system semantics. Therefore it is important to design any part of an open system in a way that allows easy extensions or alterations. Our thesis is that any extension of an open system – including persistent extensions – should be open in the same way than the system itself. We show that this is a very important requirement when the server DBMS is not reflective and/or not object-oriented. We present an implementation example with the language Ilog Power Classes [11] and ObjectDriver our wrapper to relational databases [17].

This paper is divided in five parts. The first one is this introduction. In the second part, we present why the opening property is a key part of a client-server application when only the client is reflective. Moreover, we present why the existing works are not optimal to solve the problems induced by this configuration. In a third part, we present the principles of our proposal, and the need to design a class and metaclass library with a fine-grained metaobject protocol for the persistence management. In the fourth part, we present in a detailed way the persistence and conversion protocols. Finally, we conclude our works.

## 2 The Opening Property: a Key for Persistence Control

### 2.1 Homogeneous Client and Server Formalisms

In a client-server architecture where the client and the server share a same object-oriented formalism, the persistence property can be added using a “black box” approach.

In such a configuration, all the standard metaobjects of the client have corresponding standard metaobjects in the server. A user can create, in the client, new persistent objects, instances of standard class metaobjects, and can mark them as persistent. These objects need to be replicated in the server when the transaction in which they have been defined is committed. Their replication is done through a distant instanciation message sending to the server.

For example, let us arbitrarily reduce the scope to metaclass definition and let us imagine that we have a client and a server OODBMS sharing the ObjVLisp model [3]. In this model, a standard class named *object* is the superclass of every classes and *object* is instance of a standard metaclass named *class*, which is instance of itself. Let us note that *class* is subclass of *object*. We consider, to simplify the example, that the class *object* allows its instances to be persistent, that is, to simplify, that a message *save* can be sent to any instance of *object*. Now, we define in the client a new metaclass *(MC)*$_{client}$, subclass and instance of *class*, a new class *(car)*$_{client}$, instance of *(MC)*$_{client}$, and, finally a new object *(myVehicle)*$_{client}$, instance of the class *(car)*$_{client}$. If the user wants the objects *(MC)*$_{client}$, *(car)*$_{client}$, and *(myVehicle)*$_{client}$ to be persistent, s/he has to send to each of them the message *save*. Since the server uses the ObjVLisp model, saving the metaclass *(MC)*$_{client}$ means that an instanciation message is sent to the standard class *class* of the server in order to create there a new metaclass *(MC)*$_{server}$ associated to the metaclass *(MC)*$_{client}$. In the same way, saving the *(car)*$_{client}$ implies the instanciation of *(MC)*$_{server}$ to create an associated class *(car)*$_{server}$, and saving *(myVehicle)*$_{client}$ implies the instanciation of *(car)*$_{server}$. The figure 1 presents the chain of behaviors invoked in the client and in the server.

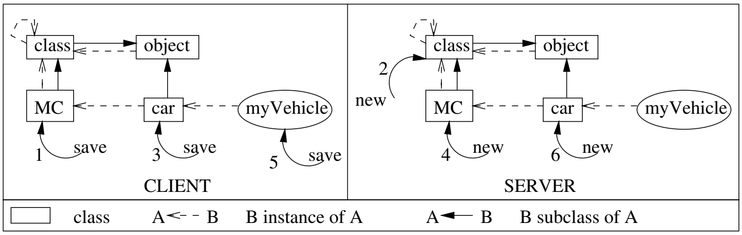


**Fig. 1.** Homogeneous formalisms between a client and a server: object first saving.

Now, many works have shown that metaclasses can be introduced to extend the semantics of a language [13]. In the previous example, the *MC* metaclass could have been introduced to manage different inheritance rules in its instances, to add them documentation, or to extend the set of legal types for slots. So, the set constituted of the classes *object*, *class* and *MC* fixes a new semantics to the language, different than the standard one. The classes *object* and *class* are common to the client and the server. We have seen that the class *MC* can become persistent and be replicated in the server. It is clear that we can have exactly the same metaobjects in the client and the server, that is, the same language semantics.

In that case, the persistence tool can be provided as a “black box”. In a very simplified manner, the persistence tool can be seen as a tool allowing a client to create, in the server, twin objects associated to the client objects. So the server formalism can evolve in the same way than the client formalism. Except for performance or technical reasons, no object or metaobject modeling logically implies to modify the way the saving of objects is done: the only goal is to replicate the objects, or metaobjects, into the server, and this goal is always met.

## 2.2 Heterogeneous Client and Server Formalisms

In this section, we show that a “black box” approach can raise problems when the client and server formalisms are heterogeneous.

**Conversions between Formalisms.** This heterogeneity of formalisms can occur, for example, when an object-oriented application must use a relational database. This architecture is very likely to be found since there is a large number of existing relational DBMSs. A DBMS is a key component in a company information system. Many clients may use a relational DBMS through a network, and there is no reason why the choice of an object-oriented technology in some projects in the company should imply the replacement of the existing DBMS server by an object-oriented one.

Now, binding two heterogeneous formalisms implies to realize conversions. These conversions depend on semantic associations between concepts of the two formalisms.

For example, there are many ways to bind an object-oriented application with a relational database. An option is:

- to convert a class by one or several relational tables, which attributes are associated with the slots of the class, and
- to convert objects by tuples of the relational tables associated with their class [18].

Most of the proposed object-relational bindings are associations between a specific object-oriented formalism and the relational one, for example, the C++ language and a relational database [12]. These tools rely on academic works where it is shown that abstract data types can be converted to relational structures [16].

The interesting point to consider at this stage is that, whatever the two heterogeneous formalisms may be, the designer of a two-formalism binding knows that different concepts in different formalisms can be associated, and how to associate them. But, in most of the systems, the conversion paradigm is presented in the documentation but cannot be easily modified.

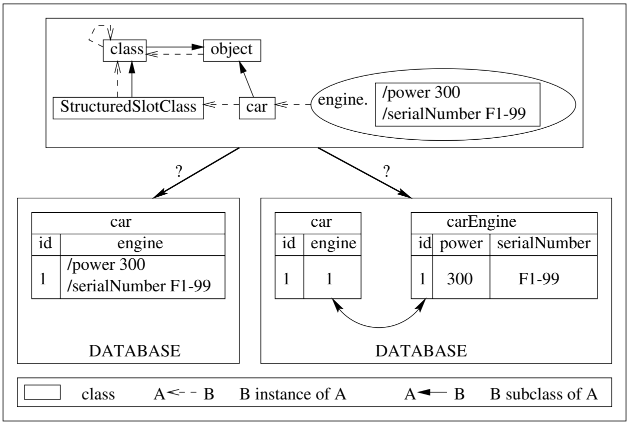
**Need to Keep the System Open.** We have seen that a black box approach can be justifiably chosen when the client and the server share a same reflective model. Now, following such a black box approach when conversions must be done, means that the the persistence would not be open, since complex conversion behaviors would be hidden to the designer. This can raise problems.

Let us consider a simple class-based language based on the ObjVLisp model, and enforcing that the slot types are primitive types like integer, float, character, or reference types. A straightforward mapping can be done to relational databases, associating a unique relational table to a class, and an attribute of this relational table to each slot of the class. The attribute type is a primitive type when the slot type is primitive, and it is a foreign key when the slot type is a reference type. With the model of the persistent ObjVLisp of the previous section, we can set a *save* behavior on the class *object* so that these conversions are done. All models and metamodels defined in persistent ObjVLisp are persistent.

However, the set of slot types provided by our small language is obviously too poor. A user would like, for example, to define structured, that is, aggregate, slots in the same manner s/he could do it with the C++ language. A new metaclass *StructuredSlotClass* can be defined so that it allows its instances to declare structured slots. The *StructuredSlotClass* metaclass extends the language semantics. As any class in the system, the classes instances of *StructuredSlotClass* and their own instances are persistent.

Now, the way these objects are stored in the underlying database may not be satisfactory. For example, suppose that we manage a car factory. We can define a class *car* with a structured slot *engine* which has two attributes *power* and *serial number*. The structured values of this slot must be stored in every instance of the class *car*. One solution is to put them linearly in the actual slot value. The *engine* slot value of a vehicle which engine power is *300* and engine serial number is *F1-99* can be *"/power 300 /serialNumber F1-99"*. The slot accessors must be redefined so that querying the engine power of a car implies a transparent retrieval of the values stored in the *engine* slot value behind the keyword *"/power"*, and querying the engine serial number of a car implies a transparent retrieval of the values stored in the *engine* slot value behind the keyword *"/serialNumber"*. The *engine* actual type and the accessors are set by the instanciation protocol defined on *StructuredSlotClass* and its own class. This option is interesting since standard slots can be used: it does not require that a new class of slots is created. Our example is thus simplified.

Now, with the point of view of the persistence protocol, there are only standard slots in the class *car*. So the relational mapping of the class *car* is straightforward: a unique relational table *car* with a primitive attribute *engine*. As a consequence, the *engine* slot value of an object is stored in the database in a straightforward way. For example, the value *"/power 300 /serialNumber F1-99"* will be the *engine* attribute value of a tuple in the relational table *car*. A schema of this configuration can be found in the left of the figure 2.



| car | |
|---|---|
| id | engine |
| 1 | /power 300<br>/serialNumber F1-99 |

| car | |
|---|---|
| id | engine |
| 1 | 1 |

| carEngine | | |
|---|---|---|
| id | power | serialNumber |
| 1 | 300 | F1-99 |

**Fig. 2.** Inadequate conversions between a client and a server DBMS.

This is not a problem if this client application is the only one to use the underlying database. The objects are saved as described previously, and they are later recreated from relational data without loss of information. If the goal of a binding between an open reflective client and a relational database is to allow the user to extend the language semantics in the client without caring about the way the data are actually stored in the underlying database, there is no need to modify the conversion rules between the client and the server DBMS.

But a DBMS is generally used to share data and schemas. Data can be used by other applications than the ones that created them. So, it is very important, when defining a database, to create clear schemas so that another designer, or a database administrator, can easily understand their semantics.

The relational schema we have described is not clear. It is not obvious that the primitive relational attribute *engine*, in the relational table *car*, represents structured data. Moreover an attribute value like *"/power 300 /serialNumber F1-99"* lacks of expressivity, since it includes keywords like *"/power"* and *"/serialNumber"*: these keywords depend on the implementations of the protocols defined on the class *StructuredSlotClass.* It is almost certain that a designer examining this database would be puzzled and would spend some time to discover its semantics. Moreover it would be difficult to query the power or the serial numbers.

A much better schema would be, for example, to define the *engine* attribute of the relational table *car* as a foreign key to another table *CarEngine* with two attributes *power* and *serialNumber*, as shown on the right of the figure 2.

So we would like to modify the standard conversions so that structured slots are saved in the way we have described. Let us note that we only need to alter slot conversions. The class and the inheritance conversions remain identical.

### 2.3 Choosing an Architecture

We came to the conclusion that, in a reflective system with intercession properties, the persistence related behaviors can be designed as a black box, with no need to open to the user the mechanisms involved, when:

- there is a same reflective formalism in the client and in the server,
- or the client and the server formalisms are different, but the user does not care about the expressiveness of the stored data, because these data will never be shared.

Since there is a large number of existing DBMSs and since cooperation between applications is increasing, we think that these previous cases are less likely to be found than a third one in which:

- the client and the server do not have a common formalism and
- the data are stored to be shared.

In that last case, whenever the user extends the formalism of the client, he needs to adapt the standard conversions. If he does not, most of the database schemas associated with her/his applications will be hard to reuse. With a triplet composed of a client, a server and the conversion rules between the client and the server, when the client formalism evolves, the server formalism or the conversion rules have to evolve. We study the way to realize the latter conversions. In the persistence domain, this is a new problem which is specific to the binding of reflective systems with intercession properties to existing DBMSs.

### 2.4 Limits of Related Works

The problems of formalism heterogeneity with a reflective system have been addressed in existing works with two different points of view. A first approach was to use the metamodeling abilities in order to make easier schema conversions between two different formalisms. A second one was to show that the conversions between formalisms can be realized with metaobjects and metaclasses. But, in the first point of view, systems often lack of uniformity, and data, that is, final instances, are not managed. In the second one, easy conversion alterations have not been considered. We develop this subject in the next lines.

**Schema Converters.** So, a first point of view is to use the metamodeling abilities provided by the reflection paradigm to parameterize schema conversions. This approach has been taken by the TIME [20] and METAGEN [25] systems. The main idea is to let the user describe two formalisms as two object metamodels, and define rules to associate specific data structures of a target metamodel to specific data structures of a origin metamodel.

There are two parts in these tools: a metamodel editor to let the user describe her/his origin and target metamodels, and a rule-based expert system to generate a new schema (model) in the target metamodel from an existing schema in the origin metamodel. Of course, if some metamodels have been already described, they can be used as a pivot metamodel between an origin and a target metamodel in order to reuse existing conversion rules. This point is interesting since it minimizes the work to do when a designer has modified an existing metamodel.

In our previous car factory example, if there are existing rules to link the first version of the object-oriented formalism to the relational formalism, we can define, from scratch, rules to convert slots of the extended object-oriented formalism to the standard one, and be sure that clear relational schemas will be generated.

These tools are very useful to help a designer to create a new object model, or a new database schema, when he needs to migrate an existing application or database, but they do not meet our goals. As a matter of fact,

- they do not take into account the conversions of data and final instances, but only type conversions;
- moreover, they force the designer to understand and use different principles, that is, the object-oriented modeling for metamodel definitions, and a rule-based expert system for conversions between metamodels.

These tools are powerful generic schema converters but can hardly be used in a client-server application.

**Bindings Between Object-Oriented Reflective Systems and Relational DBMSs.** Several works have been done to bind a reflective object-oriented system with a relational DBMS. However the problem we have shown has never been considered. So, the models involved to realize the binding are not adequate enough to allow easy extensions of conversion rules.

A library has been provided to Vodak to use relational databases [15]. It gives the user means to realize explicit rerouting toward the underlying database by modifying its own source code. For example, the user must define new accessors to reroute slot accesses to the underlying database: these accessor creations are not hidden by the meta level. Moreover, no pre-existing framework helps the designer to realize a complete binding with a relational database.

PCLOS [22] binds the CLOS language with a relational DBMS. But it has not been designed to open to the designer the way the conversions are realized. The only way to modify how objects are saved is to change a type conversion function that binds a CLOS type to a database type. However, these functions were intended to manage primary types, and they discriminate on the type of the saved data. This can lead to confusions. In our car factory example, the structured slot values of final instances are actually strings. We could define a type conversion function dealing with strings but every string in the application would be treated as a structured slot: this is obviously a problem. At the meta level, although another design would be to subclass the class of a standard slot, we could define the structured slots as a standard slot with a string type, assuming that a standard slot belonging to a structured class should be managed in a different way than a standard slot belonging to a standard class. It cannot be, since PCLOS manages every standard slot of the application in a same way. But such a mistake can be done since the persistence algorithms are not presented, so the way the slots are managed is not clear. It appears that PCLOS is not adaptable enough to deal with formalism evolution since it does not offer an open framework for conversion.

The limits of these works, and the need for a new architecture, have been discussed in previous works [4, 5].

## 3 Designing a Persistence Metaobject Protocol

We believe that an answer to these problems is to provide persistence abilities to a reflective system by a class and metaclass library extending the standard metaobject protocol of the language. This implies that persistence can be added by object-oriented modeling and there is no need of system programming. In a first part, we justify this statement, then in a second part, we present the principles of an open protocol.

### 3.1 Adding Persistence by Object-Oriented Modeling

**Legitimacy.** Persistence can be defined as the way to allow entities to survive the application in which they have been created. In an object-oriented reflective system, all the entities are objects. Hence, we need only to consider the persistence of objects.

Let us consider a client-server configuration, and let us assume that we want an object to be persistent. Either the object is persistent when it is created, or it becomes persistent when the user wants it to be so. For example, in the first case, a persistent object could actually be, in the client, when created, a proxy linked to the actual object in a server OODBMS. In the second case, a persistent object could be created on the client side, then be marked as persistent by the user and replicated on the server.

If an object must be persistent when created, we have to manage two kinds of instanciation: a transient one and a persistent one. As usual in metaclass-based system, the instanciation is a message passing to a class metaobject. If an object is transient when created, and then becomes persistent, since the persistence is a property added to the object, we propose that it may be done through a message sending to the object itself.

In any case, the persistent property can be stated by a message passing. Hence, according to the chosen persistence paradigm, persistence can be provided to an object-oriented reflective system, or to an application defined in the system, using a class library or a metaclass library.

In the rest of the paper, we arbitrarily choose that an object is first created transient, and is subsequently marked as persistent and replicated in the server.

**Feasibility.** We have seen that providing persistence through a class library is legitimate. However, it is important to evaluate its feasibility. According to the persistence paradigm chosen, all of the persistent objects and metaobjects are instances of a common persistence class (in order to be able to be marked as persistent), or have a common persistence metaclass (in order to be able to be created as persistent). Two problems may arise:

The first one is related to the ability to modify standard class metaobjects. If it cannot be, transient metaobjects are considered as any transient object in a standard ODBMS: they are ignored. So, when these metaobjects are needed on the server side to express and control objects, the persistence protocol can, for example, rely on the introspection ability of the language to create twin persistent metaobjects.

A second problem can arise when the language enforces rules to solve the metaclass compatibility problem [10]. For example, the Power Classes language allows explicit metaclasses but enforces that the metaclass of a class *C* must inherit from the metaclasses of every superclasses of *C*. In a persistent application, an object *o* and its class metaobject *mo* can be both persistent. As a consequence, the class represented by *mo* and its own class will have both to inherit from a same class. This can lead to metaclass compatibility problems. These problems can be solved enforcing metaclass inheritance rules on the persistence library and on the user classes and metaclasses.

The complete answers to these problems, and their formal presentations, are far beyond the scope of this paper, which purpose is to show an open object-oriented protocol and not to consider the design of persistence library in a general case. They can be found in [6]. Our aim is only to show that the feasibility of our proposals in these cases has been considered, and that models have been proposed. So it legitimates the design of persistence as an open library.

### 3.2 Designing Persistence as an Open Library

**Principles.** So we provide a class and metaclass library. Since we want to use heterogeneous databases, we embed the conversion mechanisms in the persistence behaviors related to the saving and retrieval of objects, that are defined on the classes of our persistence library. The goal of the rule-based generic schema converters, that is, minimizing the amount of code needed when a metamodel evolves, and the goal of open reflective OODBMSs, that is, managing metaobjects and final instances, are both legitimate. So, to meet these goals, we use a uniform approach to save objects and metaobjects, we solely rely on object-oriented modeling, and we define conversion behaviors that can easily be parameterized according to the formalism evolution with a minimal amount of code writing.

Our proposal is to embed the conversion mechanisms in the persistence behaviors related with the saving and retrieval of objects, that are defined for the classes of our persistence library. These conversions are structured as a hierarchized protocol.

The main idea is not to offer a unique, "monolithic" conversion message but to propose a whole set of behaviors which is the architecture of the conversion mechanism. So when an object receives a save/convert message, a sequence of message is sent transparently to this object. Each of these messages represents a "semantics stage" of the saving.

For example, to save a class, the system needs to save its superclasses, then its eventual relationships, then to save its slots. So, three messages can be sent to the class. Of course, each of these behaviors could, and should, be divided themselves in several behaviors that may send messages to other metaobjects. If we use a C++ or Java terminology, we can say that the *save* or the *retrieve* behaviors are *"public"* behaviors that invoke several *"protected"* behaviors.

When a designer introduces a new class, s/he can specialize on it one of these behaviors. In our car factory example, the slot conversion protocol is the only part that has to be modified. So only the behaviors managing the conversion of slots have to be specialized on the class metaobjects defined to extend the slot semantics. So, if the protocol is fine-grained the amount of code writing may be minimized.

**A metaobject protocol.** The protocol approach is not new in the field of object-oriented programming. Most of the object-oriented libraries are built with a protocol approach and visibility choices (private, protected, public). But the use of this approach in a domain where it has not be previously used can be an innovation and can widen the prospects in this domain. So was the definition of the CLOS' documented metaobject protocol [13]. The object-oriented protocol approach applied to metaobjects allows easy language extensions.

We take a similar approach in the persistence field: we design the conversion part as a fine-grained object and metaobject protocol. As far as the authors know, there is no previous attempt to describe the persistence mechanisms at the meta level as a fine-grained object-oriented protocol.

In the car factory example, we have saved and restored metaobjects, like class metaobjects and metaclass metaobjects, and final instances, like the car objects, that represent objects from the real world. The latter ones are not metaobjects. But we use a same user interface for the two kinds of objects, that is, we save and retrieve objects. So, this can bring some confusion: are we really designing a metaobject protocol?

Our aim is to allow the user to provide persistent metaobjects. But we do not want to limit our study to metaobject persistence. So we provide a protocol that can be applied to any object, that is, to final instances and to metaobjects. Any behavior that can be invoked on final instances and that can be invoked, as it stands, on metaobjects, is part of our persistence metaobject protocol. There are behaviors specific to metaobjects, or behaviors that must be specialized on some class of metaobjects. These behaviors are obviously part of the metaobject protocol too. As a consequence, we state that we can rightfully use the term of metaobject protocol: we define the part dedicated to persistence management of an existing metaobject protocol, and subparts of this protocol can be applied to final instances.

**Advantages of our architecture.** Our architecture has two main advantages. The object-oriented paradigm is the only concept the user has to learn. And a unique methodology is used to extend a language and to adapt the persistence management.

When a user needs to extend the standard semantics of a language, s/he can, for example, define new metaclasses or new classes. First s/he specializes on them the standard metaobject protocol of the language, to give new behaviors to the language. Then s/he specializes on the same class metaobjects the part of the persistence metaobject protocol that has to be adapted, according to the evolution of the language semantics.

## 4 A Binding between Power Classes and ObjectDriver

We have applied our principles on a real size application binding a reflective object-oriented language with explicit metaclasses to relational DBMSs. In this section, we first shortly present the tools we have used; then we propose an object model; finally we present the protocols.

### 4.1 An Overview of Involved Tools

**Power Classes.** We have used Ilog Power Classes [11] as a reflective object-oriented language with intercession properties. It is a commercial version of EuLisp [21] and uses a reflective model, named Telos, close to the one of CLOS. We can note two interesting properties of Power Classes. First, it enforces metaclass compatibility rules: the metaclass of a class *C* must inherit from, or be equal to, the metaclass of each of the superclass of *C*. Finally, Power Classes manages multiple inheritance.

**ObjectDriver.** ObjectDriver [17] is an object wrapper to relational databases which is developed in our team. Its main interest is to propose a powerful Abstract Data Component (ADC) layer that makes possible to reorganize and to access relational data as complex data through complex views. Of course it also allows to make complex data persistent in a relational database. All classical database operations are possible on complex data through the views, that is, inserting, updating, deleting, retrieving and querying with OQL. A convenient interface allows to connect the ADC layer to many object models.

### 4.2 Object Model

To bind a reflective object-oriented system with a non object-oriented heterogeneous system, we distinguish two important concepts that can be mapped to most of the representation formalisms:

- An object must be converted as data. If we consider the relational formalism, an object should be converted as a tuple in a relation.

- A class must be converted as a data structure. In the relational formalism, a class should be converted as a relation.

Since, in a reflective model, classes are objects, they can also be converted as data in the data structures associated with their own class.

If we consider the ObjVLisp model, we can implement on the class *object* the object conversion protocol associating a tuple to an object, and on the metaclass *class* the class metaobject conversion protocol associating a tuple and relation to a class metaobject.

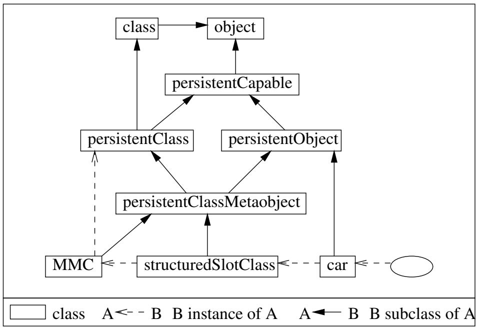


**Fig. 3.** Object model of our persistence library. The classes with no instanciation link are instance of the class *class*.

Now this model implies that every object in the language is persistent, and, consequently, that the underlying database can describe as many metaschemas as there are meta levels. For example, suppose that a class $C_0$ is instance of a class metaobject $C_1$, instance of another class metaobject $C_2$, ..., instance of $C_{n-1}$ instance of *class*. The class *C* must be described as a tuple in the relational structure associated with the class $C_1$ which must be described as a tuple in the relational structure associated with $C_2$, and so on, until *class* is described as a tuple in its own associated relational structure. We see that *n* relational tables must be created and that these relational tables define *n-1* meta-schemas. We have noted that many database administrators do not want such an increase of the number of metaschemas. So we propose to let the user classify its objects and metaobjects in three sets:

- objects that should be converted as data;
- class metaobjects that should be converted as data structures;
- and class metaobjects that should be converted as data and as data structures.

Naturally, these three sets share public persistent behaviors: objects can be marked as persistent, saved, locked, and so on.

So, to express these notions, we define the following classes:

- the abstract class *persistenceCapable* that defines the signature of public persistent behaviors;
- the class *persistentObject*, subclass of *persistenceCapable*, on which is implemented the object conversion protocol;
- the class *persistentClass*, subclass of *persistenceCapable*, on which is implemented the class conversion protocol, generating data structures;

- the class *persistentClassMetaobject* that inherits from *persistentObject* and *persistentClass* and composes their behaviors.

Semantically, the instances of *persistentClass* and *persistentClassMetaobject* are classes. These two classes are metaclasses. As a consequence, we set that *persistentClass* inherits from the class *class*. The figure 3 sums up our object model. Let us note that we have chosen not to modify the language, but to offer a persistence library. So, no persistence protocol is defined on the classes *class* and *object*. We can see that the metaclass compatibility rules of Power Classes are complied.

With this model, the designer can choose the classes that may be handled as objects during the conversions, for example, the classes *car* and *StructuredSlotClass* in the figure 3, and the classes that may not be converted as data in a metaschema, like, for example, the class *MMC* in the same figure.

### 4.3 Protocols

In this part, we use an intuitive graphical representation to present our protocols. We have chosen to use UML sequence diagrams in an instance form [24] to present representative applications. Such diagrams can be found in the figures 4, 5, 6 and 7. In short, the vertical axis represents the time. The labeled vertical dotted lines represent the cycle of life of an object. Message passing is represented by an horizontal arrow. The arrow head is linked to the receiver object and the arrow bottom is linked to the sender object. The behavior execution is represented by a vertical rectangle. An arrow in the opposite direction to the message passing arrow represents the end of the behavior execution. When choices must be made, for example, according to the return value of a behavior, a fork represents the different choices.

In the following comments, we assume that the underlying heterogeneous database can express data structures, and can associate a key to an elementary set of data (e.g. a tuple or an object) to characterize it. We assume that foreign keys are semantically similar to references in an object world.

**Saving objects**. In the sequence diagram of the figure 4, we imagine that an object *anObject* refers to a second object *anObject2*. We state that the object *anObject* is instance of a class metaobject *aClass*. Then we consider that the object *anObject* receives a *save* message which semantics is to convert and store the object in the underlying database, when it is not already persistent or saved. We use this example because it is general enough to show two important principles: the reference links and the instanciation links.

We use the persistence root paradigm. It implies that any potentially persistent object referred by any potentially persistent object must be saved when the latter one is saved. It is important to avoid deadlocks when two potentially persistent objects refer to each other, or belong to a cycle. So we define a protocol entry with the message *beingSaved* to check if this object is being saved.

Let us consider that *anObject* is not being saved. First, its own class must be saved, so the *save* message is sent to *aClass*. Then *anObject* can be saved. We must decide if *anObject* must be saved “as a primitive type” or “as an object”.

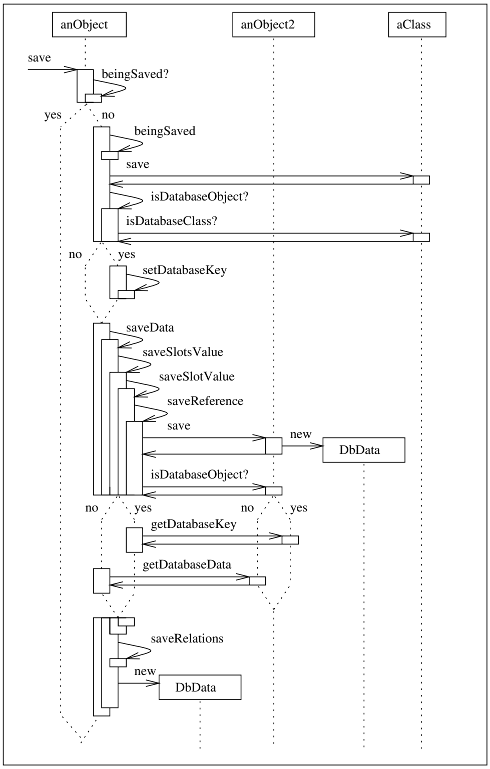


**Fig. 4.** Diagram sequence in a instance form illustrating the object saving protocol.

This distinction may be puzzling if we forget that the client is an environment where everything is an object, and the server is a environment where there may be data that are not object. For example, in the client, integers, characters, strings may be object, but, in the server, we can imagine that perhaps they cannot be objects, even if it is an object-oriented DBMS. A same problem occurs if we use an underlying relational database. We have chosen to associate at least a relational table to each class and at least a tuple to each object. Now a user can define its own persistent string class *MyOwnString*. A string slot in a class would be typed as a reference to *MyOwnString*. String objects instances of *MyOwnString* must be saved but it is obviously not interesting to define a relational table named *MyOwnString* to store every string defined in the application, and to convert any slot which type is a reference to *MyOwnString* to a foreign key attribute indexing the relational table *MyOwnString*. A much better proposal would be to convert these slots to relational string attributes. So it is interesting to decide if an object must be converted "as a primitive type" or "as an object". So we define a protocol entry, that is a behavior, *isDatabaseObject*. In most of the cases, every instance of a same class should be converted "as objects" or "as data". So the message *isDatabaseClass* is sent to *aClass*. Its semantics is identical to the one of *isDatabaseObject* but it manages the properties of a whole extent of a class.

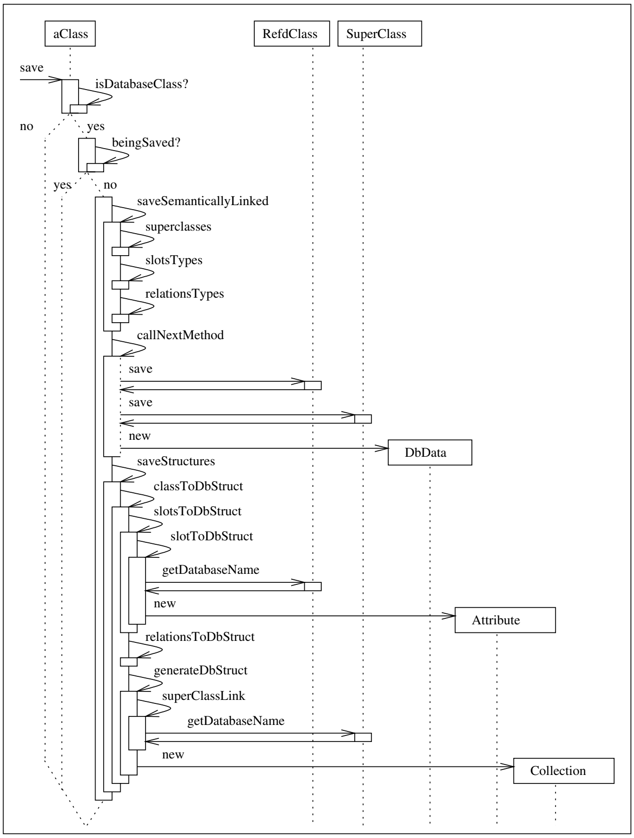


**Fig. 5.** Diagram sequence in an instance form illustrating the class metaobject saving protocol.

If an object must be saved “as an object”, a database key must be associated with it. For example, if we use a relational database, a unique relational key must be set to determine the tuples that represent the object. This is the semantics of the protocol entry *setDatabaseKey*.

Now, the saving of an object is done with two behavior calls: encapsulated data are saved with the behavior *saveSlotsValue* and *saveRelations* saves the objects linked to an object by external relations.

With the behavior *saveSlotValue*, we propose a protocol entry to manage the conversion of slots. If a slot *s* has a primitive type, then primitive conversions are realized. Now, we offer a protocol entry to manage reference slots. If the slot *s* has a reference type then the *saveReference* behavior is invoked. If the referred object *anObject2* that can be reached from *anObject* through *s* must be converted as an object, the *s* slot value of *anObject* has to be converted as a foreign key. So we need to know the database key associated with *anObject2*. This can be done by sending the message *getDatabaseKey* to *anObject2*. But if the referred object *anObject2* must be converted as a literal, we need the whole data representing *anObject2* in order to integrate it in the database structure associated with *s*. So the message *getDatabaseData* is sent.

The behavior *saveRelations* manages the relations between objects that are not modeled by slots in their respective classes. This property can be found in some languages – including Power Classes – where the classes express the internal structures of objects, but not their relations which are expressed outside of the classes and can be added or removed without modifying the class itself. We use the previous conversions and we convert them into foreign keys.

**Saving classes.** Now we present how a class metaobject can be saved. So we state that the object *anObject* of the previous sequence diagram is instance of a class *aClass* which inherits from another class *superClass*, and that the object *anObject2* is instance of a class *RefdClass*.

Saving a class implies that data structures, in the database, are associated with it. So we define a protocol entry (behavior) *saveStructures* on the class *persistentClass*.

But a class cannot be saved alone. It belongs to a semantic graph of classes. For example, it is necessary to save the superclasses of a class, in order to describe the inherited slots. Moreover, we have seen that reference slots can be described as foreign key relational attributes. In most of the existing relational databases, the only way to define a foreign key attribute is to describe two relational tables and explicitly set an attribute of a table as a foreign key related to the key attribute of the second table. So, if we want to convert a reference slot as a foreign key attribute we need to save the class the slot refers to.

Since classes are objects, semantically related classes should be in most of the cases saved. If we use the object protocol presented in the previous part, whenever a class metaobject *classMO* is saved, every potentially persistent objects referred from it are saved. Superclasses metaobjects are likely to be refered from *classMO* and slot types are likely to be refered from the slot metaobjects, which are themselves referenced from *classMO*. However, there are two problems.

- We want to define a general protocol and we cannot assume that all of the semantics links between classes will be modeled, in every language, by references.
- We have proposed a model so that there may be classes that will not be saved as objects: only structures will be associated to them. These classes are direct instances of *persistentClass* but are not instances of *persistentClassMetaobject*. So we cannot rely on the protocol of *persistentObject* since *persistentClass* does not inherit from *persistentObject*.

So we define a second protocol entry point on *persistentClass* called *saveSemanticallyLinked* that forces semantically linked classes to be saved.

Naturally, the save behavior is specialized on *persistentClassMetaobject* so that either *saveSemanticallyLinked* or the saving by exploration of the reference tree may manage semantically linked classes. With Power Classes, we have arbitrarily chosen to privilege the latter one.

In the figure 5, we show, in a diagram sequence, how the three behaviors *saveSemanticallyLinked*, *save* and *saveStructures* are combined. We assume that *aClass* is instance of *persistentClassMetaobject*. First a class must be saved if it is considered, in the underlying database, as a class, that is, if its instances must be saved as “objects” and not as literals. So, the message *isDatabaseClass* is first sent to *aClass*. Then, the semantically related classes are saved, through a call to *saveSemanticallyLinked*, then the class metaobject is treated as a regular object and, finally, data structures are linked to it.

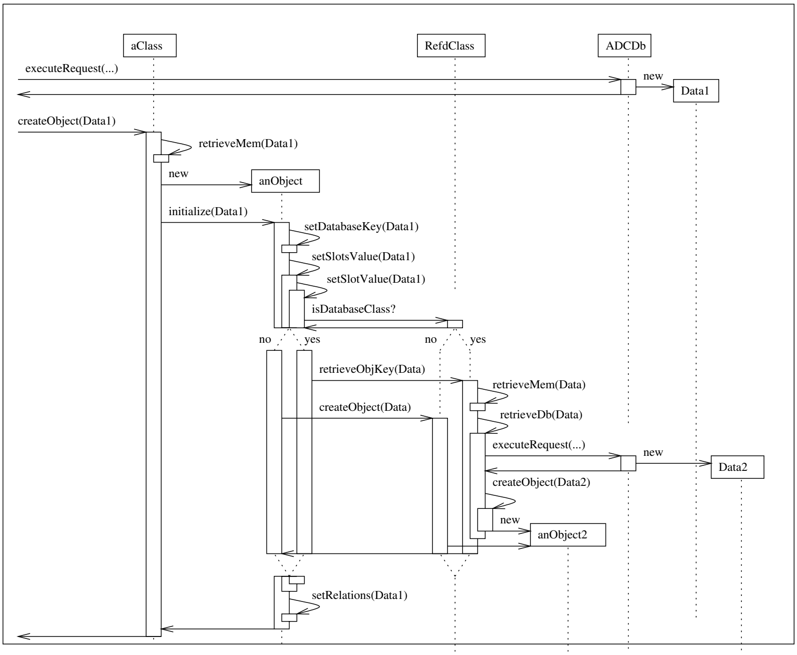


**Fig. 6.** Diagram sequence in an instance form illustrating the object retrieval protocol.

We distinguish three kinds of class metaobjects that can be semantically linked to a class: its superclasses, the types of some of its slots, and the classes it is linked to by an external relationship. So we define three parts in our protocol: *superClasses*, *slotTypes*, and *relationTypes*. Note that these behaviors are implemented for *persistentClass* and are empty for *persistentClassMetaobject*. As a matter of fact, in our example, the class metaobjects *RefdClass* and *SuperClass* are saved when the implementation of *save* defined on *persistentObject* is invoked.

**Retrieving Objects.** Binding an object-oriented system to an underlying heterogeneous database implies that, at any moment, queries can be sent to the database, and, consequently, non-object data can be retrieved from it. Retrieving objects means that new objects can be instanciated encapsulating data retrieved from the database. So designing a protocol for object retrieval means designing a protocol for object creation from non-object data.

Now persistent objects to be retrieved can be instances of a persistent class metaobject that must be also retrieved. In the approach we have chosen, it means that it is up to the responsibility of the user to first retrieve the upper instanciation level, that is, the class metaobject, and then retrieve the lower level, that is the instances of these newly created class metaobjects. We do not provide mechanism to transparently retrieve a whole instanciation branch rooted on a existing metaclass, by querying a leaf of this branch: the user has to explicity retrieve each node of the branch.

In this part, we take an example represented by the sequence diagram of the figure 6. We state that a request has been sent to the underlying database and that a data, named *Data1*, has been retrieved. We assume that *Data1* represents an instance of the class *aClass*. So a message *createObject* is sent to *aClass*.

First, it is important to decide if the object represented by *Data1* has been yet retrieved, and is still in memory. We provide the *retrieveMem* entry point to check if there is, in the extent of a class, an object corresponding to the data retrieved from the underlying database. If the object has not been retrieved yet, a new object must be created, so, in the diagram, *anObject* is created, and then initialized when it receives the *initialize* message. It is initialized with the data contained in *Data1*.

The *initialize* protocol is divided into three parts. First, in order to be later retrieved in another application, a database key is associated to the object when the behavior *setDatabaseKey* is invoked. Then the slot values are set with the behavior *setSlotsValue* and finally, the relationships between objects are set.

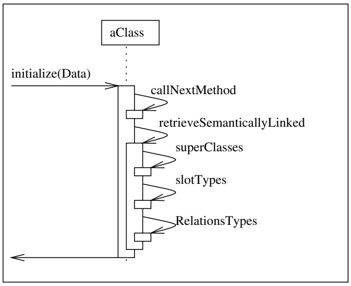


**Fig. 7.** Diagram sequence in an instance form illustrating the initialization protocol of a class.

Let us consider slot management, with the *setSlotValue* entry point. When the slot type is primitive, basic conversions are done. But a problem may occur when a slot type refers to another class, for example, *RefdClass* in our diagram. The object *anObject* must refer to another object, instance of *RefdClass*. But when it has been previously saved, an object could have been converted "into an object", for example converted into a tuple value in a relational database, or it can be converted "into a literal", for example into an attribute value of a tuple. So, if the instances of *RefdClass* are saved "as objects", *Data1* encapsulates a foreign key representing a link to an instance *anObject2* of *RefdClass*. Now, if the instances of *RefdClass* are saved "as litterals", the whole data corresponding to *anObject2* are encapsulated in *Data1*.

So, when a slot type is a reference type, our protocol tests if the referred class *RefdClass* has been converted "into a class" sending to it the message *isDatabaseClass*. If it does not, a new object instance of *RefdClass* and encapsulating the data corresponding to the referred object that are contained in *Data1* must be created. Hence, a message *createObject* is sent to *RefdClass*. But if it does, the appropriate instance of *RefdClass* has to be found according to the foreign key encapsulated by *Data1*. So the message *retrieveObjKey* is sent to *RefdClass*. The object is first sought in the class extent with the behavior *retrieveMem*. If it is not found, it is retrieved from the underlying database: a message *retrieveDb* is sent to *RefdClass*, a query is generated and sent to the database, and finally a new object *anObject2* is created and initialized.

**Retrieving Class Metaobjects:** The retrieval protocol of class metaobjects is similar to the one of objects. We just specialize the behavior *initialize* in order to retrieve classes that are not actually linked by the class metaobject but are semantically related to it. This behavior corresponds to the behavior *SaveSemanticallyLinked* in the class metaobject saving protocol. The figure 7 presents a sequence diagram illustrating the initialization of a class *aClass*.

As an example of protocol specialization, in order to manage the factory example of the section 2.2, we only had to specialize the behaviors *saveSlotValue* (object saving protocol), *slotToDbStruct* (class metaobject saving protocol) and *setSlotValue* (object and class metaobject retrieval protocols) since only the slot management had to be modified.

## 5 Conclusion

In this paper, we have shown that the use of a reflective client with intercession properties coupled with an heterogeneous server can raise problems. Data conversions must be realized between the client and the server. When a user extends the client semantics by introducing new metaobjects, the built-in conversion rules may become inadequate. Adapting the data conversion rules, and the data structures conversion rules with a minimal amount of code to write is necessary.

We propose to bring persistence to a reflective system with a class and metaclass library. We define the persistence protocol, including the conversion protocol as a fine-grained metaobject protocol. This architecture brings three main advantages:

(1) The object-oriented paradigm is the only concept the user must know. We only provide class libraries. For example, there are no external expert system implying to write inference rules in another language than the programming language.

(2) In order to extend the semantics of a language, and to integrate the resulting new language in a persistent environment, the user uses a unique methodology in a unique system. S/he has to extend the standard metaobject protocol to extend the language. To adapt the persistence management s/he needs to extend the part of the metaobject protocol dedicated to persistence. For example, the user can have defined new metaclasses, and have specialized on them the standard metaobject protocol. So s/he will have to specialize on the same metaclasses the persistence protocol.

(3) Finally, this architecture is a convincing example of the interest to define open metaobject protocols. We have seen that persistence can be added in an optimal way extending a standard metaobject protocol. This fact strengthens our thesis. New needs can appear in the persistence field. Since we have designed persistence as a metaobject protocol, we can consider that it could be extended more easily than if it had been designed with another approach.

Moreover we have described a protocol that has been applied to bind the reflective language Ilog Power Classes and relational databases.

## References

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales and D. Moon, *Common Lisp Object System Specification*, SIGPLAN notices 23 (special issue), 1988

[2] D. Bobrow, R. Gabriel and J. White, *CLOS in Context. In Object-Oriented Programming, the CLOS Perspective*, The MIT Press, chap. 2, pp. 29-61, 1995

[3] P. Cointe, *Metaclasses are First Class: the ObjVLisp Model*. Proc. of OOPSLA'89, SIGPLAN notices 24(10), pp. 156-167, 1989.

[4] S. Demphlous, *A Metaobject Protocol for Interoperability*, Proc. of the ISCA Int. Conf. on Parallel and Distributed Computing Systems, pp. 653-657, 1996.

[5] S. Demphlous, *Databases Evolution: An Approach by Metaobjects*, Proc. the third Int. Workshop on Databases and Information Systems, ACM SIGMOD Chapter, pp. 31-37, 1996.

[6] S. Demphlous, *Gestion de la persistance au sein de systèmes réflexifs à objets*, PhD thesis, University of Nice-Sophia Antipolis, France, 1998.

[7] J. Ferber, *Computational Reflection in Class based Object Oriented Languages*, Proc. of OOPSLA'89, SIGPLAN notices 24(10), pp. 317-326, 1989.

[8] I. Forman, *Putting Metaclasses to Work*, Addison-Wesley, 1998.

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

[10] N. Graube, *Metaclass Compatibility*, Proc. of OOPSLA'89, SIGPLAN notices 24(10), pp. 305-315, 1989.

[11] Ilog, *Ilog Talk/Power Classes Reference Manual v.1.3*, Gentilly, France, 1994.

[12] A. Keller, R. Jensen and S. Agarwal, *Persistence Software: Bridging Object-Oriented Programming and Relational Databases*, SIGMOD record, 22(2), pp. 215-230, 1993.

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

[14] W. Klas and M. Schref, *Metaclasses and their Application: Data Model Tailoring and Database Integration*, Lecture Notes in Computer Science n.943, Springer-Verlag, 1995.

[15] W. Klas, G. Fischer and K. Aberer, *Integrating Relational and Object-Oriented Databases using a Metaclass Concept*, Journal of Systems Integration, vol. 4, pp. 341-372, 1994.

[16] F. Lebastard, *DRIVER: une couche objet persistante pour le raisonnement sur les bases de données relationelles*, PhD thesis, INSA de Lyon / INRIA / CERMICS, 1993.

[17] F. Lebastard, S. Demphlous, V. Aguilera and O. Jautzy, *ObjectDriver: Reference Manual*, CERMICS, France, `http://www.inria.fr/cermics/ dbteam/ObjectDriver`, 1999.

[18] F. Lebastard, *Vues objets compatibles ODMG sur base de données relationnelles*, Actes des premières journées Ré-ingénierie des Systèmes d'Information, pp. 16-25, 1998.

[19] P. Maes, Concepts and Experiments in Computational Reflection, *Proc. of OOPSLA'87*, ACM SIGPLAN notices 22(12), pp. 147-155, 1987.

[20] C. Nicolle, D. Benslimane and K. Yetongnon, *Multi-Data Models Translation in Interoperable Information Systems*, Proc. of the 8th Intl Conf. on Advanced Information Systems Engineering (CAiSE'96), pp. 176-192, 1996.

[21] J. Padget, G. Nuyens and H. Bretthauer, *On Overview of EuLisp*, Lisp and Symbolic Computation, vol.6, n.1/2, pp. 9-99, 1993.

[22] A. Paepcke. *PCLOS: A Flexible Implementation of CLOS Persistence*, Proc. of ECOOP, Lecture Note in Computer Science n.322, pp. 374-389, Springer-Verlag, 1988;

[23] R. Peters, *Tigukat: a Uniform Behavioral Objectbase Management System*, PhD thesis, University of Alberta, Canada, 1994.

[24] Rational Software Corporation, *Unified Modeling Language, Notation Guide*, can be found at the URL: `http://www.rational.com/`, 1997.

[25] N. Revault, H. Sahraoui, G. Blain and J.-F. Perrot, *A Metamodeling Technique: the Metagen System*, Proc. of TOOLS Europe'95, pp. 127-139, 1995

# Non-Functional Policies

Bert Robben, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten

Distrinet
Computer Science Department
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven BELGIUM
{ bert | bartvh | wouter | pv }@cs.kuleuven.ac.be

**Abstract.** It is well known that a meta-object protocol (MOP) is a powerful mechanism to control the behavior of an application and to implement non-functional requirements such as fault-tolerance and distributed execution. A key feature of these architectures is the strict separation between the application at the base-level and the non-functional concerns at the meta-level. This makes it possible to develop generic meta-programs that can be reused for many applications. However, an important difficulty with this approach is the expression of application specific policies with respect to non-functional behavior. In this paper, we show a new approach that solves this problem by clearly separating policy from both application and meta-level. In our approach, policies are expressed at a high-level of abstraction as separate strategies. This results in highly reusable meta-programs that take application specific characteristics into account. We have validated our approach in the Correlate prototype.

## 1 Introduction

Meta-level architectures have become an important research topic in object-oriented programming. Such architectures enable the development of highly flexible application programs that can manipulate the state of their own execution [6].

An important topic in this research domain is the study of meta-object protocols (MOPs hereafter). A MOP is a powerful mechanism that can be used to control the behavior of an application and to implement non-functional requirements such as fault-tolerance and distributed execution. This has already been extensively addressed in [1, 2, 8]. One particularly relevant problem is the complexity of the meta-programs that use these MOPs to reflect on the base-level behavior. This complexity results from the inherent complexity of reflective systems and from the non-trivial distributed protocols and algorithms that are used to meet the non-functional requirements. As a result, it is hard for the application programmer, who is typically not an expert in these domains, to specialize such meta-programs to the needs of the application.

Existing research on describing synchronisation constraints[5] has tried to solve this problem. In this work, the application programmer is provided with a

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 74-92, 1999

high-level declarative language that can be used to express application specific synchronisation policies. Similar ideas have been used to specify concurrency semantics in the context of atomic data types [9]. However all these approaches propose domain specific policy languages and are as such not applicable in a general context. Aspect Oriented Programming (AOP) [7] addresses this problem and takes a more general approach. In AOP, multiple aspects can be described, each in their own special-purpose high-level aspect language. However, current state-of-the-art requires a dedicated aspect weaver for each aspect. This constrains the rapid development of new aspect languages. In addition, aspect languages are composed with the base-level component's code at compile-time. Therefore the possibility for run-time composition, a powerful feature of run-time MOPs, has been lost.

Here, a new approach is proposed which brings the ease-of-use of AOP to run-time reflection. Using this approach, meta-programmers use a specialized language to define a template. This template expresses how the meta-program can be configured. Application programmers instantiate these templates to define application specific policies. This way we can combine high-level declarative policies with reusable and flexible meta-programs. An important property of our approach is that it is not specific to a particular non-functional property. This means that it can be applied to any meta-program.

The rest of this paper is structured as follows. First, we describe how our work is related to other ongoing research. We then elaborate our approach in more detail. This is based on two examples: load balancing and secure execution. We briefly show how this work has been implemented in the Correlate prototype. We then discuss the composition of several non-functional requirements and finally conclude.

## 2 Problem Description

Run-time reflection is a powerful technique that can be used to control an application's behaviour. Existing research has shown how this technique is used successfully to implement non-functional requirements such as reliability, security and physical distribution [8]. In this approach, the code that realises the non-functional requirements is expressed as a meta-program. This meta-program consists of a collection of meta-level objects that use the MOP to control the application.

The major benefit of this approach is that all non-functional aspects are captured inside the meta-program. As such, an excellent separation of concerns is achieved where the base-level code is completely free of any non-functional concerns. This separation greatly reduces the overall complexity as both the base and the meta-level can be understood independently of each other. In addition, the reflective approach improves reusability and versatility. The meta-object protocol defines an abstract application-independent view on the base-level objects. Therefore, a meta-program can be reused for many different applications. Because the application code is separated from the meta-program, it is possible to use different meta-programs for a single application.

An important issue with this approach is the specification of the binding between base and meta-level program. One might think that it would be sufficient to simply choose a meta-program that realizes the required non-functional properties of the application. If an application needs for instance to be secure, we simply take the security meta-program and everything is fine. In practice however, things are much more complicated. Non-functional requirements are largely independent of an application but not completely. For example, to achieve optimal performance application-specific requirements have to be taken into account. This means for instance that some application objects are treated differently at the meta-level. In a dynamic environment, it might even mean that the treatment of a single application object depends on the current state. In such cases, a much more expressive specification of the binding is required. An alternative way of looking at this problem is to consider it as a parameterization of the meta-program with respect to the application. Current work on run-time reflection does not address this problem in a satisfactory way.

In OpenC++ Version1 [1], the application's source code is annotated with special comments. With these comments, an application programmer can indicate the binding between base and meta-level classes. In addition, methods of base-level classes can be annotated with a category name to enable their meta-level objects to recognize the role of the methods. Iguana [3] uses the same technique but also enables instances and expressions to be annotated. In our opinion, this approach is not satisfactory because the separation of concerns is violated. Indeed, the application's source code is tangled with references to the meta-program. Each time the binding between application and meta-program changes, the application's source code needs to be changed. In addition, the expressive power of these mechanisms is limited because the annotations only allow a static binding defined at compile-time.

A more modular solution is provided by Dalang [11]. In this system, a separate configuration file is used to specify the binding between the third party classes used by the application and the appropriate meta-object class. This file also specifies if all methods and constructors are to be reflected upon or only a subset of them. This yields a much better separation of concerns than the previous approaches. A change in the configuration files does not require a recompilation of the application classes. This leads to an approach where the programming task can be divided amongst three different roles. The application programmer is concerned with the application functionality. The meta-level programmer provides the meta-program that implements the non-functional behavior such as security and fault-tolerance. Finally, the system integrator configures the system to ensure that the overall system meets the organizational policy. It is the task of the latter person to ensure that application specific requirements are taken into account. However, support for system integration is very limited in Dalang. It is not possible to decide on configuration at run-time[1] or to provide detailed application specific information.

In the rest of this paper, we will present a new approach that provides high-level support for the integration of meta-program and application. We maintain the separation of concerns yet allow complex dependencies to be expressed. In our approach, meta-programmers create policy templates that define declarative languages. Application-specific policies can be specified in these languages at a high level of abstraction.

## 3 Examples

Throughout this paper, two examples of non-functional requirements are used to illustrate our approach. These examples are load balancing and security.

### 3.1 Load balancing

As a first example, consider the physical distribution of an application over a cluster of workstations. This requires the following mechanisms.

- *Location-transparent communication between objects.* Application objects interact with each other. This interaction must be possible independent of the location of the participating objects.
- *Load balancing.* The location of objects is an important factor in the global performance of an application. Because remote communication introduces a non-negligible overhead, objects that interact frequently should be allocated as close to each other as possible. However, to profit from the availability of multiple CPU's, CPU-intensive objects should be allocated on different hosts.

The first mechanism is independent of application specific characteristics[2] and can thus be applied in an application-independent way. The second one however needs application-specific information for optimal performance. Some applications for instance require a very specific object allocation where each object must be allocated on the correct host. Other applications are more dynamic and need reallocation at run-time to deal with ever-changing interaction patterns [4].

### 3.2 Secure execution

A second example is secure execution of an application. The core element here is to constrain the interaction between objects according to some policy. It also includes protection of communication from eavesdropping and tampering. Several mechanisms are used to reach these goals.

[1] This is in the assumption that the configuration files are created before the application starts.

[2] For this example, we don't take the occurrence of failures into account.

- *Principal authentication.* Each object in the application acts on behalf of a principal. This can be a user or some system service. When two objects start interacting, they need to authenticate each other before any security-related decision can be made.
- *Authorization.* Access to services can be restricted to a subset of privileged clients. In an object-oriented setting, these services correspond to methods of objects, and the clients are the objects invoking these methods. It is generally suitable to define roles for principals and define access according to these roles.
- *Encryption and hashing.* In a distributed application, a combination of encryption and hashing can be used to provide secure communication between communicating objects located on different nodes. The algorithms can be selected according to the cryptographic strength needed. Within a local trust domain, this could be weaker than inter-domain communication.
- *Auditing.* Some services may need to log all the incoming requests. In other cases, the requests that some client object makes may need to be logged.

Except maybe for the third, all of these mechanisms will normally be dependent on the application. Encryption and hashing could depend on the environment only. However, for performance reasons the strength of the encryption may depend on the data that is communicated, thus be specific for the application.

## 4 Application Policies

Meta-programs that implement non-functional requirements are often complex pieces of software that require specialized domain knowledge to construct and maintain. For optimal performance, these programs have to take application specific characteristics into account.

In our approach, application specific characteristics can be defined in policies. A policy specifies how the mechanisms that are provided by the meta-program should be applied for a specific application. Policies are strictly separated both from the application and from the meta-program. The idea is to use a separate policy object which is linked both to the application object and to the meta-level object. This strict separation enables the reuse of meta-programs for multiple applications.

The definition of policies is the task of the system integrator. They are declared at a high level of abstraction in specialized languages. These languages are defined by the meta-level programmer as general templates. From the point-of-view of the meta-level programmer, application policies are then specializations of these templates.

At run-time, policies are interpreted by the meta-program. This allows the meta-program to ensure that the required mechanisms are executed. Vice versa, it also means that the semantics of the templates is ultimately defined by the meta-program. Finally note that the interpretation is in terms of the general template which keeps meta-programs independent of specific applications.

This idea is explained in further detail in the following two sections. In section 5, the definition of policies is discussed in detail. Section 6 explains the interpretation.

## 5 Defining Policies

In this section we show how policies can be defined. First two examples are presented that illustrate the general idea. This is followed by a more abstract description of the proposed technique.

### 5.1 Load balancing

As a first example, consider a subsystem that performs load balancing. Such a subsystem requires application specific information for the allocation of new objects and for measuring the load. The following template expresses this.

```
distributor {

  constructorproperty creation = BALANCED | LOCAL | CUSTOMISED;

  constructorproperty Host allocate(Host[] h) { return h[0]; }

  objectproperty migration = NONE | BALANCED;

  objectproperty double getLoad() { return 1.0; }

}
```

The template defines four properties. The first, **creation**, is a constructor property. It indicates where a new object should be created. There are three possibilities.

- *BALANCED*: The static allocation of objects of this class should be under control of the load management system. This is the default option.
- *LOCAL*: The new object must be created on the local host, i.e. it should reside on the same host as its creator.
- *CUSTOMIZED*: The new object should be allocated on the host indicated by the application programmer.

In the case of customised allocation, the application programmer can implement the constructor property **allocate** to indicate the correct host. The implementation of this constructor property should be a side-effect free function that returns one of the hosts in the array that is passed as a parameter. The default behaviour of this operation is to return the first host in the array.

Dynamic load balancing is controlled through the **migration** property. Objects whose class has this property set to BALANCED, are dynamically migrated during their life-time to ensure a proper load balance over the different nodes of the distributed system. The object property **getLoad()** indicates the load the object generates. The implementation of this property can make use of the internal state of the (base-level) object. By default, the load generated by an object is 1.

System integrators can then instantiate these templates. As an example, consider the following allocation policy for class **WorkUnit**. This class performs an iterative calculation over a square area.

```
distributor WorkUnit {

  WorkUnit(int xPos, int yPos, Dimension size) {

    creation = CUSTOMISED;

    Host allocate(Host[] h) {

      return host[xPos % h.length];

    }

  }

  migration = BALANCED;

  double getLoad() { return mySize.x * mySize.y; }

}
```

This policy specifies that the allocation of new **WorkUnit**s is customised and that objects of this class can be migrated for reasons of load management. A **WorkUnit** that is created with the indicated constructor shall be allocated on a host depending on the **xPos** constructor parameter. The load index of a **WorkUnit** instance is equal to the geometric surface of the object. This value can be easily computed based on **mySize**, which is an instance variable of the **WorkUnit** class.

### 5.2 Secure execution

A similar template can be constructed to specify a security policy.

```
securitypolicy {

  objectproperty principal = INHERITED | USER | NONE;

  objectproperty auditing = NONE | INCOMING | OUTGOING | BOTH;

  methodproperty security = NONE | WEAK | STRONG;

  methodproperty boolean allow(Principal mp, Principal sp,

       Object sender) { return true; }

}
```

This template defines two object properties. The **principal** property defines how a principal is attached to the object when it is created. There are three possibilities.

- *USER*: the object will be acting on behalf of a user.
- *INHERITED*: the new object will inherit the principal of the object that created it. This is useful if a several objects together form one logical client. The 'main' object in the cluster can delegate responsibilities to other objects it creates. This is the default.
- *NONE*: the object does not act on behalf of someone. In this case, an anonymous principle is attached to the object.

The **auditing** property specifies what interactions will be logged. Choices are: no logging, only invocations of other objects on this object (INCOMING), only invocations made by this object (OUTGOING) or both.

Two additional properties are specified on a per method basis. The first one, **security**, defines what security (secrecy, integrity, etc.) is applied to invocations of the method. This ranges from no security applied, over only weak security, to strong. The implementation of weak and strong is left to the security subsystem. Its selection of the algorithms (RSA, DES, ...) will depend on the environment of the involved objects.

Access control can be specified with the **allow()** method property. The parameters allow for a fine-grained specification of access control. The two principals, one for the sender and one for the receiver, make it possible to check role membership, compare identity, or query other attributes. The third parameter is a reference to the sending object. This can be used to extract application specific information in cases where a complex access control policy is needed.

This template can be instantiated as follows for a class in an electronic commerce application that manages electronic payment instruments.

```
securitypolicy UserPayment {

  String[] getPaymentMethods() {

    security = WEAK;

    boolean allow

      (Principal mp, Principal sp, Object sender) {

        return (sender instanceof User);

    }

  }

  UserPaymentInstrument create(String n) {

    security = STRONG;

    boolean allow

     (Principal mp, Principal sp, Object sender) {

       return mp.equals(sp);

    }

  }

}
```

Two important methods for this class have custom security requirements. The first method gives a list of available payment methods. As the names of these methods are not too critical, weak security is sufficient. For access control, the policy states that only users (objects of type **User**) can do the query. The other method is more critical, so it requires strong security measures. Only users acting on behalf of the same principal as the principal of the **UserPayment** object are able to ask to create a new payment instrument.

### 5.3 Definition

Our approach makes a difference between *templates* and *policies*. A template defines a declarative language that indicates the various possible customisations an application might require. A policy is an instantiation of such a template. Instantiating a template consists of making a selection between a number of possible customisations and completing certain missing information. The scope of the resulting policy is a single application class.

A template is expressed as a set of *properties*. In the simplest case, a property is an enumeration of a set of atomic values. In more complex cases, a property is expressed as a function of the internal state of the object and of the current state of the environment. In this context, the environment is defined as a set additional parameters that contain essential information for that specific property. The array of **Host**s in the **Distributor** template is an example of such a parameter. It is the responsibility of the meta-program to provide these parameters. Orthogonal to this, a property can be specified for an entire application class or for a single method/constructor.

The template also defines the default value that is used in case no policy can be found for a certain class. For enumerations, the default value is always the first element. For properties that depend on the internal state of the object, a default implementation of the abstraction function has to be defined. Examples of this were given in the two previous subsections.

A policy is the instantiation of a template. It is always related to a single application class. A policy can instantiate any number of properties of its template. Properties that are not instantiated have the default value. Instantiating an enumeration consists of selecting a single value from the set. To instantiate a function, an alternative implementation must be given. This implementation can make use of the instance variables of the application class (as free variables).

A grammar for the syntax of templates and policies can be found in appendix A.

## 5.4 Conclusion

It is important to understand that the examples are not meant as the ultimate policy templates for distribution and security that fit every application. Developing a general template would also mean building a general subsystem for some non-functional requirement, foreseeing all possible demands applications can have. This is clearly a very difficult task. In practice, the template is usually constructed as a result of the dialog between system integrators, giving rough descriptions of the non-functional requirements, and meta-level programmers explaining the available options.

A second point is that the semantics of the various properties in a policy template is ultimately defined by the meta-level program that interprets the policy objects. This enables a declarative policy specification at a high level of abstraction. Only a high level of abstraction enables system integration without the need for highly detailed knowledge on algorithms and implementation strategies.

# 6 Interpreting Policies

In the previous section, we have shown how an application programmer can specify an application specific policy. We now show how such a specification is transformed into a class and then used by the meta-program at run-time.

Note that the transformation from policy and template to Java class is simple and can be handled automatically by a tool. This tool is independent of both applications and meta-programs.

## 6.1 Representing templates

Each template is transformed into an abstract class. This class serves as interface for the meta-program to query the policy an application wants. They don't contain any code related to the implementation of a certain policy.

The interface of a template class is fairly simple. Each property is represented as a public operation on this class. In case of an enumeration property, the operation returns a number that indicates which enumeration value the policy chooses. A state-dependent property has the return type that was defined in the template. All these operations have the appropriate parameters depending on the type of property. This is shown in table 6.1. As before, the environment is defined as the set of additional parameters that are delivered by the non-functional meta-program.

**Table 1.** Parameters of properties

| | Enumeration | State-dependent value |
|---|---|---|
| Object property | / | Internal state of object<br>Environment |
| Constructor property | Constructor | Constructor<br>Constructor parameters<br>Environment |
| Method property | Method | Internal state of object<br>Method<br>Method parameters<br>Environment |

Finally, a static operation is defined that can be used to get the policy object for a certain application class.

Figure 1 shows this class for the Distributor template. The **Distributor** class defines five integer constants, one for each enumeration value. Two get-operations are defined to retrieve respectively the creation and the migration property. The **getLoad** and the **allocate** property are implemented as operations on the **Distributor** class. The former property only depends on the internal state of the object. Therefore, the **getLoad()** operation has this object as

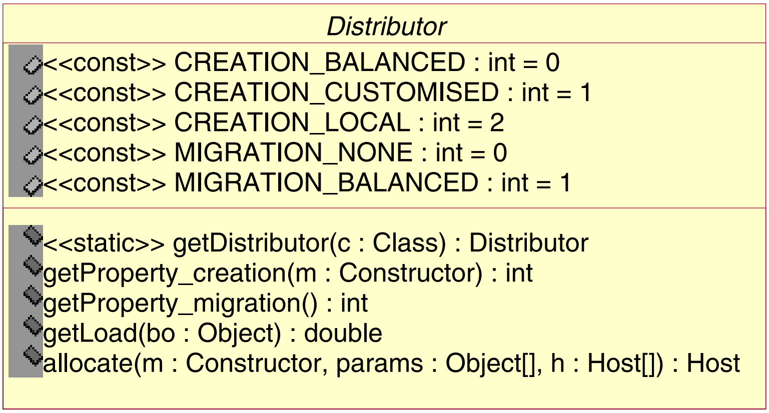

Distributor
<<const>> CREATION_BALANCED : int = 0
<<const>> CREATION_CUSTOMISED : int = 1
<<const>> CREATION_LOCAL : int = 2
<<const>> MIGRATION_NONE : int = 0
<<const>> MIGRATION_BALANCED : int = 1
<<static>> getDistributor(c : Class) : Distributor
getProperty_creation(m : Constructor) : int
getProperty_migration() : int
getLoad(bo : Object) : double
allocate(m : Constructor, params : Object[], h : Host[]) : Host

**Fig. 1.** Implementing the Distributor policy

only parameter. The latter is a constructor property and takes the constructor and the constructor parameters as parameters. In addition, as specified in the policy template, the operation has as final parameter an array of hosts.

### 6.2 Representing policies

Policies for application classes are transformed into specializations of the abstract template classes. Policy classes implement of course the property operations of the template class. Note that all **Policy** classes are singletons.

Figure 2 shows this for the **SecurityPolicy** template and two security policies of application classes (**User** and **UserPayment**).

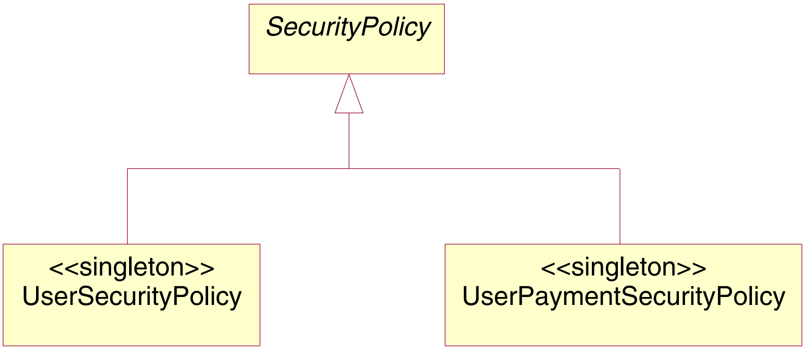


**Fig. 2.** Implementing templates and policies

Details on the implementation of properties can be found in appendix B.

## 7 Application in Correlate

We've applied this technique in the Correlate meta-level architecture. Correlate is a research project of the DistriNet research group of the department of Computer Science of the Katholieke Universiteit Leuven. The overall aim of the project is to support the development of distributed application software. A prototype in Java has been built and is available on the web[12].

Correlate is a concurrent object-oriented language extension based on Java. It supports a strictly implicit meta-object protocol for behavioral reflection at run-time. The MOP[13] focuses on the run-time events of object creation/deletion and object invocation. In this sense, it is related to OpenC++ version 1[1]. A Correlate meta-program reacts to these events and implements an execution environment for the base-level application objects.

Using this MOP, we have developed a customised execution environment that implements distributed execution. This meta-program implements a remote communication mechanism and a mechanism for object migration. The abstract interface defined by the MOP ensures a strict separation of concerns between application and execution environment. Therefore, this meta-program can be reused for several applications.

However, application specific requirements need to be taken into account. For example, a genetic search agent application[10] has been built that solves the traveling salesman problem (TSP). In this application, the optimal object allocation scheme is known to the application programmer before the execution of the program. A numerical solver for the heat equation, has a quite different policy. Dynamic refinement of objects at run-time causes major shifts in the work load generated by this application. Because these refinements cannot always be computed in advance, dynamic load balancing techniques have to be applied.

The distributor template has been successfully used to handle both these application-specific policies. As an example, we'll show how the creation property is implemented. The Correlate MOP lets a meta-program control the creation of new objects. In the distribution meta-program, this is implemented by the **handle()** operation which is invoked each time a new application object is created[13]. The following code fragment shows the simplified implementation.

```
void handle(ConstructionMessage msg) {
  Distributor d =
   Distributor.getDistributor(msg.getCorrelateClass());
  switch(d.getProperty_creation(msg.getConstructor())) {
    case CREATION_BALANCED:
      location = loadManager.getNewLocation(...);
      break;
    case CREATION_CUSTOMISED:
      location = d.allocate(msg.getConstructor(),
        msg.getParams(), Host.getHosts());
      break;
    default:
      location = Host.getLocalHost();
  }
  ... // allocate new object at location
}
```

First, we search for the distributor object that implements the policy for the class of which a new instance must be created. Depending on the **creation** property, the location of the new object is then computed. In case of a balanced allocation, the local load manager is consulted to find a good location. In case of customized allocation, the location is computed based on the allocate object property. The third case, local allocation, is straightforward and needs no further explanation.

Other properties, such as dynamic load balancing and security for example, are more complex and require more sophisticated mechanisms at the meta-level. In the case of load balancing for instance, an information gathering component and a dealing component are necessary. The former component monitors the load distribution. This information is then used by the latter component that decides when to migrate which object(s) to which host. Nevertheless, the principle stays the same. The meta-program implements the mechanism and consults the policy objects whenever application-specific information is required.

At the moment, we only applied our approach to the Correlate prototype. Nevertheless, we believe that the technique as described is general and can therefore be applied in any sufficiently powerful meta-level architecture that supports run-time reflection.

## 8 Discussion

The major advantage of the approach as described in this paper, is that application specific policies can be expressed as a separate entity. The base application remains totally independent from any meta-program; no hooks or comment lines are present that link application code to meta-level objects. The meta-program contains no application specific code. The only links to the application it supports are through the abstract interface of the MOP and through a policy template, both independent of a specific application. Because of this separation, reuse of both parts is achieved more easily. In addition, changes in the meta-program will have no effect on the application, and vice versa. All modifications are concentrated in the set of policies for the application. However, the effort is reduced as a result of the high-level declarative style of the specifications. To help the system integrator manage the properties in all policies, a tool could even be constructed. For example, in the case of security specifications, the properties for access control would probably be derived from a more general policy statement.

In comparison to aspect-oriented programming, experimentation with the functionality of the meta-program and the expressions of the templates is less difficult. Instead of constantly having to rewrite the aspect weaver for every change in the aspect language, one only has to modify some property definitions in the template. The general parser will generate new policy objects so that the meta-program can query them in an object-oriented fashion. This flexibility of policy templates lets the meta-programmer focus on implementing the necessary algorithms instead of the adaptation of base application code.

Finally, consider the problem of composition. Suppose an application needs both security and distribution. A promising approach is to organize the meta-programs into a meta-tower. In this case, composition simply means describing an order of stacking the different meta-programs on each other. For instance, such a system is described in [2] for a combination of fault-tolerance, security and distributed execution. However, when two meta-programs need to control the same base application, there is a problem. As the meta-program only sees the level that lies directly beneath it, the higher meta-levels can make no decisions about the state of the base application. Although it is too early yet to make a definite statement, we claim our approach can overcome this latter problem for many practical cases. The reason for this is that a meta-program can define its policies based on the policies of the base-level that it controls.

For example, suppose the security meta-program is to be used together with the distribution meta-program. The distribution policy of a security meta-level object then depends on the distribution policy of the base-level object it controls. Consider for instance the allocation property. A particular security policy could be to only allow creation of **WorkUnit** instances on trusted hosts. The **allocation** property of the security policy will thus first query the property of its base object. If the host on which the base object wants to be created is trusted, that host is selected by the security policy. If not, a different but trusted host is selected. This means controlled access to the state of the base application is achieved, through mapping policies from one level onto the next. The complexity of the mapping will depend on how orthogonal the non-functional requirements and their meta-programs are.

## 9 Conclusion

This paper described a new approach that enables application-specific policies for non-functional requirements to be expressed at a high level of abstraction. These policies are represented as objects that are separated both from the base and the meta-level. Our approach has been applied in the Correlate meta-level architecture.

As part of our future work, we want to apply our approach in the domains of distributed security and real-time systems to validate its expressiveness. Also of particular interest is the applicability of our approach for the composition of complex non-orthogonal requirements. Another interesting track would be to apply this research to architectures that support compile-time reflection.

## Acknowledgements

We gratefully thank all the anonymous reviewers for the many helpful comments that helped us to improve the presentation of this work.

## References

1. Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In Proceedings ECOOP '93, pages 483-502, Kaiserslautern, July 1993. Springer-Verlag.
2. Jean-Charles Fabre and Tanguy Prennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. In IEEE Transactions on Computers, 47(1), January 1998.
3. Brendan Gowing, Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. In Proceedings of Reflection'96, San Francisco, 1996.
4. Wouter Joosen. Load Balancing in Distributed and Parallel Systems. PhD Thesis, Department Computer Science K.U.Leuven, 1996.
5. Ciaran McHale. Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance. Phd Thesis, University of Dublin, Trinity College, 1994.
6. G. Kiczales, J. des Rivieres and D. Bobrow. The Art of the Meta-Object Protocol, MIT Press, 1991.
7. Christina Lopes. D: A Language Framework for Distributed Programming. PhD thesis, Graduate School of the College of Computer Science of Northeastern University, 1997.
8. Robert J. Stroud and Zhixue Wue. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmermann, editor, Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996.
9. R. J. Stroud and Z. Wu. Using Metaobject Protocols to Implement Atomic Data Types. In Proceedings of ECOOP'95, Lecture Notes in Computer Science 952, 168-189, 1995.

10. Romain Slootmaekers, Henk Van Wulpen, Wouter Joosen. Modeling Genetic Search Agents Using a Concurrent Object-Oriented Language. In Proceedings of HPCN Europe 1998, Lecture Notes in Computer Science 1401, Amsterdam 1998.
11. Ian Welch and Robert Stroud. Dalang - A Reflective Java Extension, OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, October 1998. To be published as part of the OOPSLA'98 Workshop Reader.
12. Correlate Home Page. http://www.cs.kuleuven.ac.be/x̃enoops/CORRELATE/.
13. Bert Robben, Wouter Joosen, Frank Matthijs, Bart Vanhaute, and Pierre Verbaeten. A Metaobject Protocol for Correlate. In ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems, 1998.

## Appendix A: Grammars

The grammar for policy templates is defined as follows. A template is written as a name followed by a sequence of properties surrounded by curly braces. A property is defined as a context modifier followed by either an enumeration or a state-function. In this context, a modifier is one of the keywords "constructorproperty", "objectproperty" or "methodproperty". An enumeration consists of a name and a sequence of identifiers separated by the | operator. A state-function is a simple Java method declaration. Simple in this context means that the method doesn't declare any thrown exceptions and doesn't have any method modifiers.

```
<template> = <name> "{" <property>+ "}"

<property> = <modifier>

   ( <enumeration> | <state-function> )

<modifier> = "constructorproperty" | "objectproperty" |

             "methodproperty"

<enumeration> = <identifier> "=" <value-list> ";"

<value-list> = <identifier> [ "|" <value-list> ]

<state-function> = <simple-method-declaration>
```

The grammar for a policy is similar to that for a template. The major difference is that constructor and method policies have to declare to which constructor or method they apply.

```
<policy> = <template-name> <application-class-name>
```

```
    "{" <policy-property>+ "}"

<policy-property> = <c-prop> | <m-prop> | <o-prop>

<c-prop> = <constructor-signature>

    "{" <simple-property>+ "}"

<m-prop> = <method-signature>

    "{" <simple-property>+ "}"

<o-prop> = <simple-property>

<simple-property> = <selection> | <completion>

<selection> = <identifier> "=" <identifier> ";"

<completion> = <simple-method-declaration>
```

The grammar for policies that is described here is independent of any template. For a policy to be valid, it must be an instantiation of a template for a certain application class. This means that the simple properties of the policy are defined in the policy template and that all methods and constructors that are mentioned refer to actual methods and constructors of the application class. Also, note that methods defined in policies access the instance variables of the application class. Completions for constructors and methods can use the constructor / method parameters.

## Appendix B: Translating policies to classes

In section 6, we discussed the representation of policies and templates as classes. However, we only showed the external interface. This appendix explains a straightforward implementation.

While implementing policy classes, we face two interesting problems. On the one hand, we need to give an implementation for the three different kinds of properties. On the other hand, we must ensure that properties can access the environment (for instance the state of the base-level object).

The first problem is not very hard to solve in Java. A simple solution is to make use of the Java reflection feature that reifies an operation in a Method object. On the policy class, a single operation is defined for each method property. This operation takes a Method object as parameter. In its implementation, a simple if-then-else structure is used to select the appropriate code. The following code fragment shows this for the allow property. The code is for a policy that redefines the property for two operations. For a constructor property, a similar implementation strategy can be followed.

```
public boolean allow(Method m, ...) {
  if (m.equals(METHOD_1)) {
    ... // code for METHOD_1
  } else if (m.equals(METHOD_2)) {
    ... // code for METHOD_2
  }
  return super.allow(m, ...); // default property
}
```

The second problem concerns properties that are specified as state functions. The code of these functions can almost directly be used as implementation of the operation that implements the property. We must only ensure that references to instance variables and invocation parameter are correctly transformed. This is necessary because the operation that implements the property is defined on the policy class and not on the application class. To access instance variables, we define the policy class to be in the same package as the application class. All non-private instance variables are then reachable. To access private variables, the Correlate compiler generates special hidden get-functions on the application class. These functions have package visibility and can thus be used by the policy classes[3]. Consider for instance the **getLoad()** property for the **WorkUnit** class as discussed earlier. This property depends on the **mySize** instance variable of the **WorkUnit** class. The following code fragment shows the implementation.

```
public double getLoad(Object o) {
  return ((WorkUnit)o).mySize().x * ((WorkUnit)o).mySize().y;
}
```

We've developed a tool that automates this translation.

[3] Note that the $Sun^{Tm}$ Java Compiler uses a similar solution when Java inner classes need to access instance variables of an outer class.

# Invited Talk 2
# On the Reflective Structure of Information Networks

*Jean-Bernard Stefani*
*France Télécom - Centre National d'Etudes des Télécommunications (CNET)*
*28 Chemin du Vieux Chêne BP 98*
*38243 Meylan, FRANCE*
*E-mail : jeanbernard.stefani@cnet.francetelecom.fr*

## Introduction

The notion of information networking has been the subject of much work, in the past ten years, from the "classical" telecommunications community under the auspices of the TINA (Telecommunications Information Networking Architecture) Consortium. Although much of this work has turned up in the end to be predicated upon a relatively traditional view of networking, and has been overshadowed by the recent phenomenal growth of the Internet and the World Wide Web, the notion of an information network, as described e.g. in [1, 2], embodies a small but powerful set of architectural principles that have passed largely un-noticed. In this talk we will review these principles and spend some time reviewing the resulting network view, which characterizes an information network as an essentially reflective structure. In the process, we will end up discussing the significance of binding as a key constituency in a full-fledged reflective distributed system and pointing at relevant works in the area.

## Information Network Architectural Principles

The terms "information network" and "information networking" have been coined to emphasize the evolution of telecommunications networks towards communication systems providing a broad range of services, from standard, low-level transmission services — bit pipes — to high-level, value-added, information processing services — e.g. WWW-based services.

Briefly, an information network architecture can be generated by adopting three architectural principles :

- networks as open distributed systems ;
- reifying network resources as computational objects ;
- telecommunications services as binding services.

An information network can thus be seen as consisting in a superposition (in the technical sense of the term) of two different sub-systems: a transport network and a control system. The transport network corresponds to the set of network elements and communications resources that provide the transfer of information from one point of the network to another. The control system encompasses the

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 93-94, 1999

set of information processing resources and capabilities that are used to manage and control the transport network, as well as to support services constructed on top of the transport network services.

The resulting structure is inherently reflective in that the effectiveness of the control system is predicated upon the ability to reify network elements as full-fledged programming objects. It is also inherently recursive in that several information networking structures may stack-up in a tower of transport/information networks.

Within such a structure, many different network views can be reconciled, from traditional signalling in POTS to active networks.

## References

1. J.B. Stefani: "A reflexive architecture for intelligent networks" – 2nd TINA International Workshop, Chantilly, France, May 1991.
2. J.B. Stefani : "Open Distributed Processing: an architectural basis for information network" – Computer Communications vol. 18, no 11, November 1995.
3. D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, G. Minden : "A survey of active network research" – IEEE Communications Magazine, vol. 35, no 1, January 1997.

# Reflective Media Space Management Using RASCAL

Wayne Robbins and Nicolas D. Georganas

Multimedia Communications Research Laboratory
Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa
161 Louis Pasteur
Ottawa, Ontario, Canada
K1N 6N5
{robbins, georganas}@mcrlab.uottawa.ca

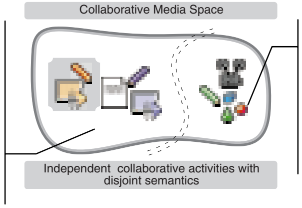
**Abstract.** The advent of interactive shared media spaces has augmented the traditional role of multimedia by providing a natural and intuitive means for interpersonal communication. These shared media-rich environments serve as a natural basis for distributed collaboration through a seamless blend of presentational, conversational and interactive multimedia. This integration, however, presents challenges ranging from the need to support various media types to managing real-time object interaction. This work addresses how to facilitate media space design by employing reflection as a primary design, implementation and management technique within a pattern-based meta-level architecture. Reflection is used to isolate system-level issues such as behavioural coordination from low-level, media-specific computation as well as to facilitate dynamic adaptation of differing behavioural requirements. The architectural framework and underlying topology are illustrated along with the model's application to a distance education system. Finally, the prototype and its use of the ObjecTime CASE tool are overviewed.

## 1 Introduction

The traditional role of multimedia systems has been to disseminate information. From textual to graphical, auditory to visual, such systems were designed to augment the presentation of ideas by representing them in the most appropriate format(s). *Collaborative multimedia*, however, goes beyond the role of exposition to provide support for distributed work groups. By melding multimedia within computer-supported collaborative work (CSCW) environments, media offer a natural and intuitive means to facilitate group interaction and the collaborative exchange of ideas.

The classical model of collaborative computing offered by teleconferencing can be seen as multimedia. However, its potential falls short because it does not adequately allow users to mimic traditional shared work spaces. In real-world collaborative exchanges, people see and talk to each other, manipulate and point at shared props, sketch ideas on shared surfaces and view other media (such as film clips) in a mutually common environment. In short, participants and the objects they use form a *shared media space* in which users interact with each other through the experience

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 95-114, 1999


Collaborative Media Space
Independent collaborative activities with disjoint semantics

*Collaborating group consisting of shared document editing and real-time audio/video conference*

*Collaborating group consisting of graphical editing and a synthetic audio/video movie playback*

**Fig. 1.** Logical View of an Example Media Space

and manipulation of multiple media (Fig. 1). The expressiveness of a diverse media palette combined with a distributed work environment goes not only beyond the utility of teleconferencing, but also the conventional classifications of presentational, conversational and interactive multimedia.

In order to effectively utilize this expressive power, activities performed within the space must be understood by all those involved. That is, user actions must be integrated into a *semantically coherent workspace* that suits the needs and expectations of participants as they interact. An important part of maintaining semantic coherence is the very often subtle and implicit *coordination* of activities that is part of the collaborative process [1]. This coordination [2] is necessary to achieve an environment in which the actions performed upon objects and the interactions between them make sense; that is, meaning must be maintained through synchronizing participants and their actions.

These actions can be seen to constitute participant *behaviours* within the shared environment. Consequently, specific actions can be viewed as *behavioural components* within the collaborative scenario while well-defined groups of actions form the notion of *behavioural patterns*. While multimedia synchronization has traditionally referred to temporal relationships amongst media [3], collaborative systems must address participant behaviours (personal interaction, media processing, and so forth) in an integrated fashion to facilitate a naturally fluid, predictable and reasonable work environment. Behaviours must be coordinated to ensure that they maintain their own semantics (such as timeliness), while interacting activity patterns must also occur in the correct temporal order (precedence relationship) to ensure causal dependencies of collaborative actions are reflected and maintained throughout the shared space [2][4]. In short, the media space must provide a containment framework for the integration of diverse entities and behaviours used within a particular collaborative context. It must do so in a way that mirrors the dynamic way people work together as well as how their collaborative behaviours change over time.

This inherently open and dynamic nature of the media space, however, creates significant difficulty in the construction of such systems. In particular, how to best provide for the possible variety of entities and behaviours while still being able to enforce the rules and requirements governing them. This work proposes an infrastructural framework that facilitates both structural and behavioural flexibility based on a system that can potentially analyze and modify itself. Rather than retrofit concerns such as synchronization in an ad hoc "after the fact" manner, behaviour and structure are regarded as orthogonal building blocks from which systems can be built, modified or replaced throughout their lifetime.

To this end, this paper presents a work in progress that proposes a reflective meta-level approach to collaborative media space management. Emphasis is given to the design of an infrastructural framework for building a broad range of media-based collaborative environments, with specific support for addressing diverse behavioural requirements. How to provide for these various facets in a flexible and adaptive fashion illustrates the importance of addressing how "the pieces of the puzzle" are put together. Consequently, the architecture's design, its underlying software engineering and the role of its communication infrastructure are considered.

The remainder of this paper is organized as follows: section two overviews the definition and management of a media space while section three discusses the flexibility achieved through the use of patterns within a reflective, meta-level management architecture. Section four presents the proposed model, which is then illustrated in terms of a telelearning scenario. On-going work towards development of a prototype utilizing ObjecTime [5], a CASE tool for the design of real-time and distributed systems, is then discussed. Finally, the paper summarizes and concludes.

## 2 A Media Space and Its Management

The term *media space* originated from work at Xerox Parc [6][7] in the mid to late 80's. A broad discussion in [8] illustrated its potential group for supporting group interaction between co-workers at remote sites in both social and work settings. These initial media spaces were based on the use of an analog audio/video network in parallel with a digital network of workstations. In contrast, our work is based on the premise of an "all digital" media space and that users interact using an integrated, more closely unified "point of presence" within the space. Collaboration is facilitated via whatever *objects* (media, applications and so forth) are shared between users and how they are manipulated. Indeed, various media and interfaces (including user interface peripheral devices such as video capture/display) are integrated into the space as adjunct objects rather than separate parallel systems. Our approach attempts to provide an integrated solution so that different aspects of collaboration are sufficiently modular but not disjoint streams of activity.

Traditional collaborative space research has dealt with collaboration from process, psychological and technological perspectives: (1) specific application environments [9][10]; (2) human-computer interaction [1][11]; and (3) communication and/or application protocols [4][12]. The first perspective deals with a program's functionality, while the second addresses elements of the user interface. In each, coordination is often either ignored or statically embedded within the application. Additionally, collaborative potential is limited since an application's utility is "wrapped up" in its design and complexity. While protocol-based solutions are more versatile, their premise is that data exchange rules are the main issue, resulting in a "message sequence chart"-based approach. In all cases, there is little focus on extensibility, abstraction or adaptivity.

Of specific interest is the work by Dourish [13], in which reflection/meta-techniques are used to provide extensibility in CSCW toolkits. While borrowing some intent from Dourish's body of work, ours differs (in part) via its focus on multimedia systems and its emphasis on behavioural components. While not explored in this paper, a significant benefit of our approach is its potential to address mixed mode synchronization issues that are not well served by other approaches.

This work therefore addresses the need for a design framework and methodology which can be used to build a versatile and enabling infrastructure for user-level collaborative applications. The proposed work can be viewed as a potential a way to organize, relate and integrate different approaches (including user interface technologies, applications and protocols) as required. To facilitate this, the work observes a key underlying principle: management and domain functionality must be strictly separated to facilitate a flexible, modifiable and adaptive system. In particular, the coordination and synchronization of both the environment and individual entities within it should be managed separately from their domain functionality. To facilitate both the abstraction of these components as well as their integration in a mutually effective manner, reflection is employed to provide the "causally connected glue" between functional and behavioural components. In particular, the primary areas of concern include: (1) how to address behavioural characteristics within the space; and (2) how to facilitate a versatile and flexible infrastructure to accommodate new uses.

## 2.1 Characterizing Media Space Behaviours

The usual concern of multimedia synchronization has been to ensure timely media rendering. Collaborative multimedia, however, goes beyond simple presentation, and so too does its synchronization requirements. While traditional concerns are important, it is the coordination of the collaborative activities that ultimately defines media space utility. As discussed later, by representing behaviours as first-class building blocks, it is easier to address the variety of media, interaction styles and the typical characteristics of inter-personal communication. Shown in Fig. 2, these characteristics form a hierarchy which can parameterize and constrain the behavioural patterns used within the media space [14].

## 2.2 Extending the Media Space

A significant benefit of the media space approach is its integration of many different kinds of media and activities into a single environment. As such, it naturally suits the

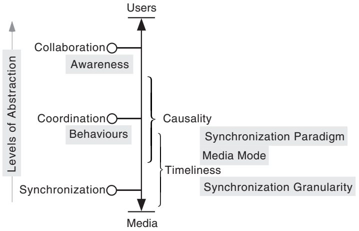


**Fig. 2.** Facets of Media Space Management

collaborative process by offering a “what you need is what you get” solution. However, even in a very feature-rich system, there will always be the need for another type of media, another application, the introduction of a new behaviour, a change in policy and so on. In other words, the utility of the space drives it to be a dynamic entity whose composition must be extensible and adaptive to different uses over time. Therefore, a media space framework must be able to facilitate changes both in terms structure and behaviour.

*Structural extensibility* means that the media space can be modified and extended through the introduction of new types of objects within the space. This is done by dynamically installing/replacing functional components, such as a new media type or application. For example, if users need to refer to a particular medium (e.g. use a particular codec) or perform a specific task (e.g. use a specific application) during a collaborative session, the media space integrates the required modules into its environment. This allows the media space framework to address real-world diversity and the “on-the-fly” nature of the collaborative process by “growing” the infrastructure as necessary. Such a *component-based* approach can also be applied to the user interface (UI) metaphors offered to media space users. By abstracting the manner in which users interact with the space, different types of physical interaction can be supported, ranging from common CSCW tools to virtual environments.

*Behavioural adaptation* primarily refers to modifying an object’s coordination and synchronization within the media space. Examples range from changing the frame rate for video playback to modifying the rules which govern the interaction between users who are jointly editing a shared document. Behavioural adaptation can be seen to cover a broad range of activities including performance monitoring of object mechanisms to the specification and evolution of collaborative policies. By treating object behaviours separately from their functionality, descriptions of behaviours can be manipulated as first-class abstractions. The result is a *behavioural component* approach which mirrors the structural component model. Behavioural adaptation then becomes a task of dynamically installing/replacing/modifying the behavioural components which describe entity behaviour and interaction within the space.

## 3 Facilitating Management Through Design

As a real-time system, the viability of a media space can be significantly impacted by its ability to meet the specific needs of a usage scenario. For example, reliability is paramount in a tele-surgical application while much less important for video-on-demand. In each case, however, the details typical of real-time systems that must be addressed to “get the job done right” can cloud issues of proper system and software design. The very issues that are vital to system deployment are often over-complicated or inadequately addressed because they are all jumbled together; areas of concern are inextricably woven together rather than properly separated into disjoint areas of responsibility. In multimedia, system complexity is often exacerbated due to the desire for an immediate and sleek solution. Hence, it is extremely important to address the organization and management of a collaborative multimedia environment.

### 3.1 Responsibility-Driven Organization

Object orientation is a technique which attempts to address these concerns by considering a system as a collection of interacting objects, each of which represents and encapsulates the details of some (real-world) entity. Unfortunately, an object's complexity is often increased as a result of this composite view. A video-on-demand system, using a movie object with audio and video tracks, would address issues such as data retrieval, decoding, intra/inter-stream synchronization, user interface management and so forth. While most designs would not attempt a strictly monolithic solution, it is quite common to blend at least some of these aspects together in implementation-specific ways. The result is difficulty in isolating which aspects of the system deal with different parts of the problem, ultimately affecting how easy it is to debug, modify and maintain it. Levels of functionality are often woven together and blur the distinction between solution, policy and mechanism [15].

Within collaborative environments, this lack of separation affects the ability to scale and adapt such systems because the range of usage scenarios can be overwhelming. Combined with the integration of presentational techniques, live media and interactive exchanges, there is clearly a need to simplify when, where and how to address a system's overall design. Object orientation is a necessary but insufficient means of isolating system detail because such an object addresses issues relating to its own behaviour as well as its interaction with others. The result is a mesh of interacting objects, each communicating with the other in a very tangled manner. In general, objects are both busy doing their own work (i.e. domain computation) and telling others what to do (i.e. coordinating them), either explicitly or implicitly through their direct communication with each other. This leads to a system that is a highly interdependent and interconnected collection of objects which must directly deal with the coordination of other entities in its environment. Consequently, the entities within the space are highly configuration-dependent and their ability to be re-used within a general framework is reduced.

Therefore, there is a need to organize the system along lines of the responsibility in which a system's execution (i.e. domain computation) is separated from its coordination (i.e. the management of domain computation). The coordination meta level manages the base-level computation in which these two aspects of an entity are explicitly separated. Doing so facilitates cleaner system design and re-use at a behavioural level in addition to a structural (i.e. functional) level (as in the standard object-oriented paradigm). This takes the standard hierarchical approach to structure (e.g. classes, types, objects) and mirrors it in a behavioural context; that is, lower-level domain activities are managed by higher-level coordination activities.

In collaborative systems, this approach reflects the difference between the elements of a media space. Media are synchronized at multiple granularities (objects, frames, actions, groups of...) and in terms of their individual media-specific requirements. Collaborators who use these media must be coordinated according to their behaviours, which are oriented towards achieving a specific goal. Consequently, collaboration is achieved by coordinating users' behaviours via synchronizing the media they use. Since these tasks deals with different entities and at different levels of abstraction within the media space, each forms a meta level to the one below it.

This meta-level approach is the basis for applying *reflection* [16][17][18] as a technique to abstract and inter-relate different levels of functionality and behaviour within the media space. With the use of multiple meta levels, a *reflective tower* can be used to provide a dynamically extensible and adaptive model for both structure and behaviour. And as such, it can thought of as a *pattern* for structuring the organization of the system into causally connected levels of responsibility.

## 3.2 Employing a Pattern-Based Architecture

Increasingly prevalent within the domain of software engineering, *pattern-oriented software architectures* [19] are extending the notion of encapsulation and re-use from simple code modules (such as procedures and objects) to entire subsystems of interacting components (i.e. component-based systems). Each component is an independently developed and tested entity which also has well-defined interfaces to connect to and function as part of a larger system. Patterns offer a way to re-use a general solution methodology across a variety of specific problems, ranging from architectural patterns for software system organization, design patterns for detailing subsystem refinements and idioms which hilight implementation constructs. Well-known patterns include the client/server, proxy and broker models.

The structural view of an architecture is one in which an entity's functional role and interconnections are governed according to a specific well-known configuration. This view addresses the design of collaborative multimedia systems as patterns of collaborating entities, each of which support/enable a particular kind of collaborative activity or multimedia functionality. For example, a video-on-demand system would employ the client/server pattern using an video data server and playback client.

The behavioural view, however, constitutes a logical perspective of entities and their interaction as part of the collaborative process. The interaction between participants and the media they use is a highly purposeful dialogue in which participants' activities and the exchanges between them are part of a mutually understood "goal-oriented" behaviour. Participants can be seen to engage in specific collaborative *behavioural patterns* which guide their mutual understanding and

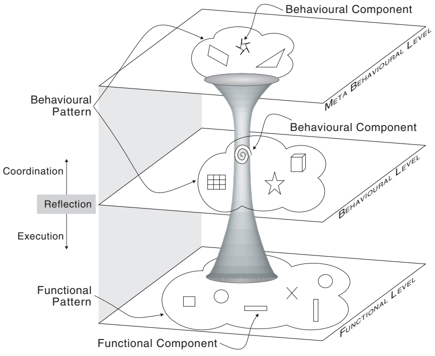


**Fig. 3.** Functional and Behavioural Patterns

activities, both as individuals as well as members of a collaborating group. Complex behaviours are a combination of simpler ones, with each behaviour being a pattern of entities which interact in a specific well-defined manner. The result is a composition of complex patterns from simpler ones, functioning much like *behavioural components* in the overall collaborative effort (Fig. 3).

The key benefit of this approach is the customization or *tailorability* offered to system designers, implementers and users. The different elements of the system can be integrated according to the requirements of the system at a range of granularities based on behavioural patterns (i.e. perform a required function, meet specific performance levels, work at a specific level of abstraction, etc.). The proposed architecture considers a media space as a collection of component entities which interact in a coordinated manner using an "intelligent" communications service. The space's coordination is separated from media-specific computations using patterns to specify policy and specific collaborative rules, in which behavioural components and patterns are first-class entities which are instantiated and manipulated in their own right. Rather than being an inherent part of the structural model, activity coordination involves behavioural (i.e. coordination/synchronization) meta-objects in which synchronization characteristics (Sect. 2.1) would be applied through the use of behavioural patterns.

An important aspect of this approach is its use of *active objects* to realize the environment's components. Each element, be it a user-level collaborative object or a part of the media space system itself, is considered an active object with its own run-time behaviour/thread of execution. Rather than use passive computational structures (i.e. traditional objects) to handle the system's various operational states, an entity's active nature allows it to perform computations and execute actions to fulfill its role. Changes in behaviour or function are achieved by using the appropriate active entity which "knows how" to perform a particular type of activity.

## 4 The RASCAL Collaborative Infrastructure

The proposed model [20][21][22][14] provides both a framework and a methodology from which to build collaborative media spaces. The *Reflectively Adaptive Synchronous Coordination Architectural framework* (more affectionately known as *RASCAL*) blends support for traditional multimedia synchronization with a more holistic approach to shared space coordination. The media space is taken as a collection of interacting objects which are managed using an "intelligent network coordination service" which separates the space's behavioural issues from its media-specific computation based on a reflective meta-architecture.

### 4.1 The RASCAL Media Space Topology

Based on the architectural premise of separating execution and coordination, data and control mechanisms form two distinct subsystems within the media space. As shown in Fig. 4, the architecture's topology can be represented as a collection of hosts (participants in the collaborative environment) joined by a *coordinated*

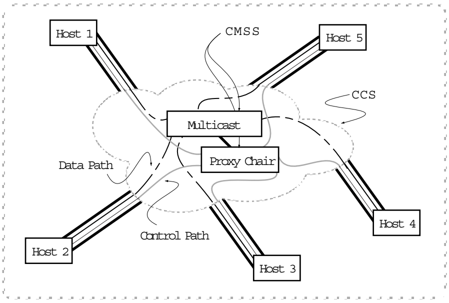


**Fig. 4.** RASCAL's Underlying Topology

*communications subsystem* (CCS). Most communications systems are simply viewed as a "passive" medium that link participants in a conference; however, the principle in RASCAL is to provide an "active" communications infrastructure which intelligently coordinates media space activities. The hosts are "points of presence" in the media space at which users connect to the collaborative environment. The basic element used to collaborate is the *object* which can represent a medium, application or other technological entity in the space, such as an autonomous intelligent agent. Users at different hosts manipulate and/or communicate through objects to interact with remote collaborators. Example objects include live audio/video objects (for conferencing), a shared editing application, a synthetic MPEG movie or a bitmap image. Objects are used in the context of a *collaborative session* which in which users participate and interact according to some behavioural pattern. Each of these objects has their own synchronization requirements and each collaborative session coordinates its constituent objects relative to the overall collaborative effort.

The CCS is divided into two complementary elements: a *coordination sphere* and an *execution sphere* (shown in Fig. 5). The execution sphere represents the flow of data, which is produced and consumed by objects within the collaborative environment; that is, execution constitutes the "media" within the media space. The data that forms this sphere are the media and interactions that compose the objects and the manipulations performed on them by participants. Coordination of these activities, on the other hand, is considered separately. The coordination of the activities addresses the notion of synchronization within the shared space. The coordination of execution (i.e. the synchronization of the media data) utilizes separate data and control paths within the CCS. The data from different participants is multicast to send and receive data between participants; the intention is to utilize an underlying network that would offer simultaneous data delivery as well as support for QoS guarantees [3]. Control information that describes the consumption of the generated data forms the execution sphere of the CCS. The control information is also multicast but under the auspices of the *collaborative multimedia support system*, or CMSS.

Coordination of the space, as performed by the CMSS, logically pairs a *dedicated coordinator* with the multicast switch. This coordinator, called the *Proxy Chair,* manages control messages that describe the data flowing through the multicast network. Data is not channeled through the Proxy Chair (abbreviated as *Proxy*), nor does the Proxy perform any other function other than coordinate the space through the synchronization of control messages. From the collaborator's perspective, the network is providing the necessary coordination and synchronization, while from the network's point of view, the coordinator is a specialized entity within it that offers a particular service, like those associated with *intelligent networks* [23]. Hence, the Proxy Chair is *not* a functionality to be assumed by a host participant; rather, it is to be a dedicated component of the CCS. Rather than waste resources, this separation offers the following benefits:

1. A complete and straight-forward separation of concerns in terms of communication. The separation of paths enables appropriate QoS levels for different objects' data as well as control messages. Multicasting is still used for transmitting coordination constructs but at the discrimination of the Proxy Chair.
2. A complete and straight-forward separation of concerns in terms of processing. End hosts are often taxed by the manipulation of media data (e.g. decoding and user interface); therefore, placing coordination in a subservient position to such variable and demanding duties is ill-considered. More than the use of a centralized architecture, it is this often-assumed multiplexing of functionality which leads to bottlenecks; in short, the temptation to minimize the number of components to perceive a more effective use of resources needs careful consideration.
3. The autonomy of coordination facilitates more straight-forward, scaleable and flexible designs for both end systems and the Proxy. End hosts typically function as input/output devices for participants within the shared space. For specific usage scenarios, the calibre of these machines could vary considerably. However, if considerable coordination logic is made part of the end host design, these machines necessarily become more complex and expensive. Furthermore, the potential orthogonality of host design is lessened because components are now context dependent as to their role within the media space; therefore, the adaptability and scalability of the architecture are now determined by application topology. As proposed, the architecture partitions system workload into two areas, each of which can be managed separately. The modification or addition of new coordination policies is more easily managed by using a central coordinator combined with host-independent, role-specific control software. The Proxy only functions as a "chairperson" for coordination purposes and is independent of any application-specific management or "chairperson" entity. While the Proxy forms a single, independent entity within the CCS, it can best be viewed as a *logical* entity. Should media space scale, performance or failure be an issue, a group of coordinators could be defined to function as a distributed Proxy Chair. In the case of failure, the Proxy could be dynamically replaced; or if the space scaled beyond the capacity of a single Proxy, multiple coordinators, each of which would manage a specific region of the media space, could be used to transparently migrate select sessions to another coordinator.

### 4.2 Applying the Meta-Level Approach

As part of the collaborative effort, the activities which ultimately achieve and affect the state of the collaboration can be seen to causally affect media space state (i.e. the users' environment). Therefore, in as much as participants and media interact in a coordinated manner, these entities can be modelled in a reflective relationship which maintains the semantics of the collaboration by using meta-information about their interactions. Hence the complexity of different entities (such as a particular medium or activity) is abstracted away and isolated from other parts of the system. The bi-directional nature of the reflective relationship allows the intention of more abstract layers to be reflected "down" to the lower implementation levels while their status is reflected "up" in order for the system to monitor and adapt itself.

The relationship between the coordination sphere (controlled by policy) and the execution sphere (enabled by mechanism) can be applied throughout the entire media space as shown in Fig. 5. The coordination meta level addresses issues of managing user behaviour (as well as media synchronization) according to collaborative policies (and associated goals). Meanwhile, the execution level deals with computations which realize the media and other "physical" components of the media space. The reflective relationship between the coordination and execution levels link the various entities within the space while still maintaining their encapsulation (both structurally and behaviourally) at different levels of abstraction.

Reflection can therefore be seen as an organizational pattern which bridges various levels of responsibility and function within the media space. While an important aspect of the architecture is to facilitate flexible and adaptive design through the explicit separation of concerns, there is a need to integrate them in a manner where changes in one area are "felt" in another; that is, there is a need to provide a *causal connection* between the various entities within the system. Fig. 6 illustrates (using pseudo-OMT notation [24]) different elements of a collaborative session within the space, ranging from users and mechanisms at the lowest base level to the abstract definition of collaborative rules (via patterns) at the highest meta level. The result is a reflective organization in which upper meta levels monitor and configure lower levels according to their status/performance. As with other reflective systems, each media space entity provides both a standard API by which specific domain functionality

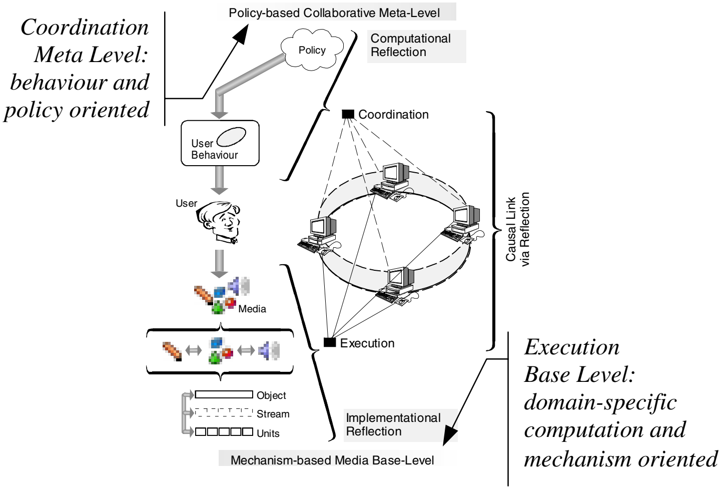

Coordination Meta Level: behaviour and policy oriented
Policy-based Collaborative Meta-Level
Policy
Computational Reflection
Coordination
User Behaviour
User
Causal Link via Reflection
Media
Execution
Object
Stream
Units
Implementational Reflection
Mechanism-based Media Base-Level
Execution Base Level: domain-specific computation and mechanism oriented

**Fig. 5.** Overview of the Media Space's Reflective Meta Architecture

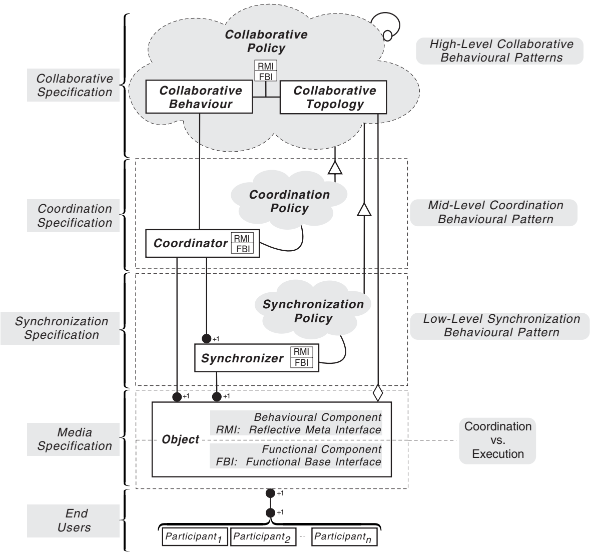

Collaborative Policy
RMI
FBI
Collaborative Behaviour
Collaborative Topology
High-Level Collaborative Behavioural Patterns
Collaborative Specification
Coordination Policy
Coordination Specification
Mid-Level Coordination Behavioural Pattern
Coordinator
RMI
FBI
Synchronization Policy
Synchronization Specification
Low-Level Synchronization Behavioural Pattern
Synchronizer
RMI
FBI
+1
+1
+1
Media Specification
Object
Behavioural Component
RMI: Reflective Meta Interface
Functional Component
FBI: Functional Base Interface
Coordination vs. Execution
End Users
+1
+1
Participant$_1$
Participant$_2$
Participant$_n$

**Fig. 6.** Elements of a Collaborative Session

(such as media rendering) is accessed. In addition to this *functional base interface* (or FBI), a secondary interface is provided to manipulate and examine how the entity provides its base functionality. This *reflective meta interface* (or RMI) offers a structured way to detail how specific aspects of an entity's functionality are facilitated and managed through the use of appropriate meta objects. These interfaces are applied across the entire system, such that "physical" system entities (like the Proxy Chair) as well as the logical ones (like collaborative behaviours) are all subject to reflective manipulation.

This encapsulation of responsibility promotes flexibility and increased adaptability in terms of implementation, design and usage; for example, the ability to support different "awareness widgets" using adaptive user interface meta objects. It also allows for *self-adaptive behaviours* in which collaborative behaviour patterns can themselves be designed in a reflective manner (i.e. causally connected meta/base levels); that is, it facilitates a hierarchical behavioural specification in which more abstract collaborative levels can monitor and modify lower-level coordination and synchronization rules. Such a specification would utilize multiple meta levels to specify behavioural patterns in terms of other behavioural patterns. By reifying these behaviours, the system can examine and modify itself using behavioural components that can be dynamically added to/removed from a collaborative session. Since all media space components are "active", they can perform whatever computation is necessary to provide different kinds of collaborative services based on the semantics deemed appropriate by the users/designers of the collaborative session. Additionally, "physical" components could (conceivably) reconfigure themselves based on system performance to better meet the needs of the media space users (such as dynamic load balancing for a distributed Proxy Chair as mentioned in Sect. 4.1). The result is a *self-investigative collaborative environment* that can adapt itself to both system performance as well user-specific behavioural idiosyncrasies.

## 5 Telelearning: An Example Media Space

Given the proposed model, its application as the coordination infrastructure for a distributed telelearning system will now be illustrated. The telelearning system will consist of multiple students and a single lecturer interconnected by a communications network. The environment will constitute a media space in that real-time audio/visual capture and display will be used in combination with support for a shared whiteboard, the distributed playback of synthetic media such as video clips, and so forth.

In the proposed model, the primary coordination logic is contained in the CCS (i.e. network), while the standard "multiplexed" method has the coordination and synchronization logic placed in highly role-specific entities. The standard method results in a more complex and less orthogonal system where functionality is determined more by hardware and topological assumptions rather than software control mechanisms. The proposed model promotes an orthogonal system in which coordination is provided by a coordinated communications subsystem under the control of host-independent software. Therefore, the priority and management hierarchy of the media space is determined by control software at individual hosts which provide a management interface to the CCS. This provides more flexibility in development and extension of the system by encouraging high-level configuration of the media space using adaptable, policy-based software rather than application-dependent hardware. While end hosts require some secondary synchronization logic, coordination is primarily achieved through the intelligent communications system which is a more topological-independent location for policy-enabling mechanisms.

Consider a telelearning environment where rather than placing all the synchronization and coordination logic in a single, complex and expensive lecturer machine, the lecturer is a simple entity with an interface to the synchronization and coordination logic within the CCS/Proxy Chair. By increasing host equity (reducing differences in calibre between student and teacher machine requirements), the system's instructional policy can be enforced by role-based control software, rather than hardware layout. The ability to reduce the impact of resource requirements on student machines is also important given the burgeoning trend to low-cost network appliances such as set-top boxes or digital entertainment terminals (DETs). Consequently, by taking a network services approach, topology and functionality are more orthogonal and changes to either are easier. Specifically, the architecture offers more flexibility in the way that students can be involved within the media space. Since a participant's role is not determined by the complexity of synchronization mechanisms in their local host, the CMSS allows both students and lecturer to participate in or lead the "classroom" based on policy, not topology.

As an example, consider the collaborative scenario outlined in Fig. 1, which maps the media space to a virtual classroom. A pattern-based view of classroom entities and their relationships are shown in Fig. 7. Within this scenario, the highest meta level is a global "classroom policy" which guides all interactions in the classroom, such as regulations as to how individuals interact, what kinds of group activity are allowed (such as brain-storming or working at the blackboard) and so forth. These policies are represented as patterns which specify the role and interaction between entities within the media space. Therefore, if entity interaction at a given level of abstraction conforms to the (behavioural) pattern specified at its meta level, then the

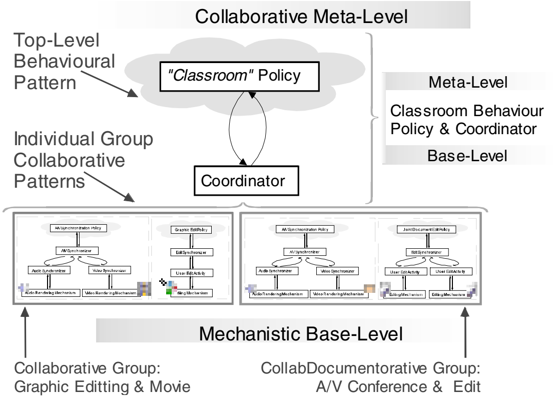

Collaborative Meta-Level
Top-Level Behavioural Pattern
"Classroom" Policy
Meta-Level
Classroom Behaviour Policy & Coordinator
Base-Level
Individual Group Collaborative Patterns
Coordinator
A/V Synchronization Policy
A/V Synchronizer
Audio Synchronizer
Video Synchronizer
Audio Rendering Mechanism
Video Rendering Mechanism
Graphic Edit Policy
Edit Synchronizer
User Edit Activity
diting Mechanism
A/V Synchronization Policy
A/V Synchronizer
Audio Synchronizer
Video Synchronizer
Audio Rendering Mechanism
Video Rendering Mechanism
Joint Document Edit Policy
Edit Synchronizer
User Edit Activity
User Edit Activity
Editing Mechanism
Editing Mechanism
Mechanistic Base-Level
Collaborative Group: Graphic Editting & Movie
CollabDocumentorative Group: A/V Conference & Edit

**Fig. 7.** High-Level Meta/Base Pattern View of Example Scenario

given entities are coordinated according to the intended collaborative behaviour. This argument can then be applied throughout the system at different levels of abstraction, including low-level media synchronization (Fig. 8).

The Proxy Chair functions as a *coordinator* who is the entity responsible for managing the coordination of the space (the classroom) according to its management policy (the classroom policy). It is important to note that the coordinator (i.e. Proxy Chair) does *not* represent the lecturer but an entity which enforces the guidelines of the space for all participants within it; therefore, the coordinator guides the lecturer as well as the students. Within the given scenario, there are two separate collaborating groups: (1) two participants with live audio/visual conferencing are jointly editing a (text) document; and (2) a graphic is being worked on while a movie is being played. In each case, other students can be observing these active collaborators and form passive participants in the collaborative effort.

While the classroom policy forms the meta level for the coordinator, the coordinator is itself a meta entity to the collaborating groups. The coordinator governs the coarse-grain behaviour of the individual groups using the classroom policy while individual groups are governed in a similar manner but at a finer

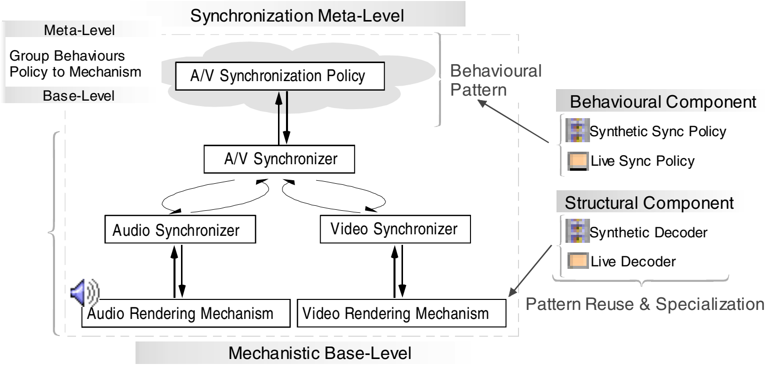


**Fig. 8.** Local Meta/Base View of an Audio/Video Collaborative Pattern

granularity and using the appropriate behavioural pattern. For example, while the maintenance of audio/video "lip-sync" is done on a per pair basis, it is coordinated with respect to sibling collaborators under the control of the coordinator using a "conversation" behavioural pattern. This non-monolithic approach enables different paradigms and granularities of synchronization to be specified as part of a specific group's local synchronization policy. Additionally, the appropriate considerations for specific mixed mode semantics (live and synthetic media combined) can also be dealt with in a similar manner. And finally, the mechanisms which actually perform the operations are at the lowest level and represent the media-specific execution sphere.

The tailorability offered by using such an approach means that different educational requirements can be met by integrating different components (i.e. specific demonstration applets or informational material, such as an on-line textbook). It also offers support for a wide range of classroom settings and learning styles, such as specifying interactions between students and the tools they can use. Such components could conceivably be intelligently modified or replaced under the control of the system itself (based on self-investigative meta-computations by higher-level entities). The ability to monitor and adapt based on the notion of reflection would effectively provide a media space which could dynamically "learn" how best to support its participants based on their usage characteristics. Possibilities include changes in synchronization granularity according to user satisfaction with the provided quality of service, switching to a less stringent synchronization paradigm based on frequency of access and the dynamic reconfiguration of how media objects are related (i.e. composition of the collaborative groupings in Fig. 7).

## 6 MSpace: The RASCAL Media Space Prototype

Currently, efforts are focusing on the creation of a prototype to illustrate the use of a reflective meta-architecture as applied to media space design and implementation. A major emphasis is being placed on providing for the flexible and adaptive coordination of the collaborative process involving both live and synthetic media.

Development is being done under Windows NT using MFC/C++ and ObjecTime Developer [5] in order to facilitate a *simulation-based prototype*. ObjecTime is a computer-assisted software engineering (CASE) tool designed for the design, implementation and simulation of object-oriented, distributed real-time systems. The given approach blends implementation of a user interface front-end and a distributed coordination/communications backbone; the former being implemented using OLE (Object Linking and Embedding) and ActiveX technologies under Windows while the latter is built with ObjecTime. Using a visual design paradigm, ObjecTime models are 'drawn' using the *Real-Time Object Oriented Modelling (ROOM)* technique. In addition to production-level code generation, the toolset supports internally built test harnesses as a means to validate its models. This offers the ability to independently simulate, test and analyze any of the prototype's ObjecTime-based components, including the reflection service and the behavioural pattern mechanism.

Figure 9 illustrates ObjecTime design metaphor as applied to the top-level structural definition of the media space. Each rectangle represents an entity in the

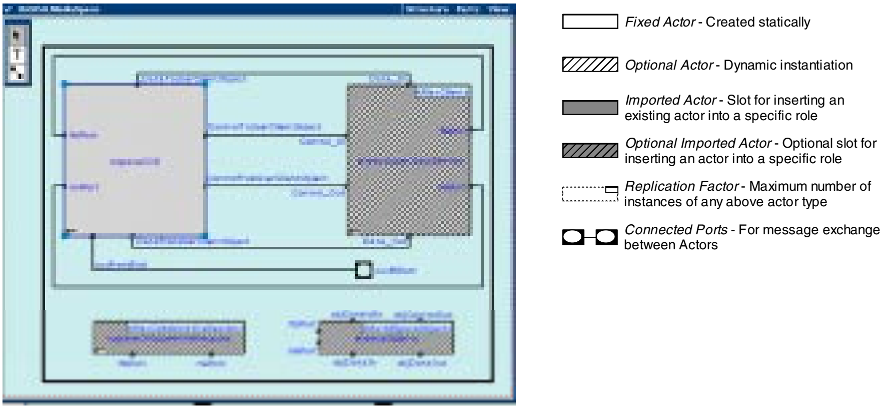

Fixed Actor - Created statically
Optional Actor - Dynamic instantiation
Imported Actor - Slot for inserting an existing actor into a specific role
Optional Imported Actor - Optional slot for inserting an actor into a specific role
Replication Factor - Maximum number of instances of any above actor type
Connected Ports - For message exchange between Actors

**Fig. 9.** Top-Level ObjecTime Structural Model of a RASCAL Media Space

system and is modelled by an *actor*. *Actor class*es define *active objects* (entities with their own run-time behaviour) that can be created statically or dynamically, in single or multiple instances and imported/exported to/from ‘slots’ as required. Each actor runs independently, can contain other actors using structural containment and communicates with others through the exchange of messages (using ports) defined via *protocol class*es. These messages (or events) then trigger changes in actor behaviour, which is specified using hierarchical finite state machines (FSMs). The definition of both actors and protocols is done in an object-oriented manner and uses inheritance in order to specialize their definitions. Therefore, a media space implementation that wishes to customize the types of entities within the space (for specific types of media or specific collaborative patterns) can take advantage of pre-existing designs and build upon them. This includes both structural and behavioural elements as well as protocols definitions, enabling the derivation of specialized meta-object protocols.

The ObjecTime model considers the media space as an instance of the actor class *RASCALMediaSpace* which acts as a container for all other actors in the system. This includes the CCS (*mspaceCCS* of class *RBE_CCS*) and a set of client “point of presence” workstations (*mspaceUserClientStations* of class *RBE_UserClient*). Each of these would also contain other active objects as part of their structural definition, such as the CCS which contains an instance of the multicast facility (*RBE_MulticastFacility*) and the Proxy Chair (*RBE_ProxyChair*). Other entities, such as *mspaceCollaborativeSession*s and *mspaceObjects*, would be built in a similar manner. These particular entities are kept as a “global pool” within the model such that they are potentially used by other entities within the space, including importation into collaborative sessions and behaviours.

While a complete examination of the RASCAL object framework (partially shown in Fig. 10) is beyond the scope of this paper, it can be seen that both structural (e.g. *RMSApplication*) and behavioural entities (e.g. *RMSCollaborativePattern*) are first-class objects than can be: (1) specialized through subclassing; (2) instantiated as active objects with their own thread of execution; and (3) have their own behavioural definition (via its FSM). For example, a video class could be defined based on the *RMSMediaObject* class and itself later subclassed to refer to a variety of synthetic or

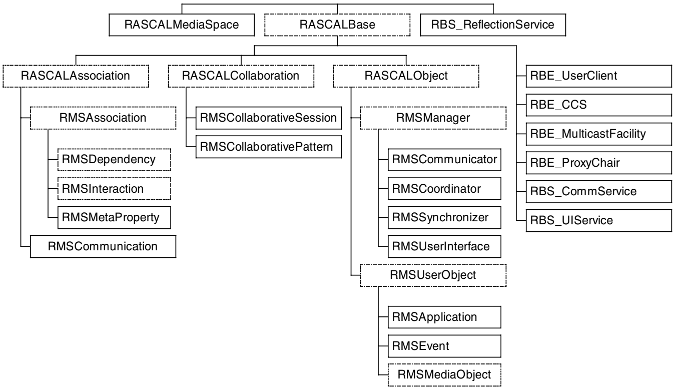


**Fig. 10.** Abbreviated RASCAL Active Object Framework

live video encoding standards. Similarly, different types of behaviours could be derived from a general audio/video synchronization pattern based on granularity, paradigm or media mode. As an active entity, a collaborative behavioural pattern is defined as a set of actor(s), each of which executes their own behaviour as specified by their individual finite state machines.

The flexibility of behaviours is brought about by the use of *behavioural slots* in which specific behaviour patterns and components can be inserted. Pools of collaborative behaviours can be predefined and brought into the ObjecTime environment and dynamically imported into slots which define policies and patterns for behaviours; Fig. 11 illustrates this technique as applied to the definition of the both collaborative session and collaborative pattern objects. This change would result from the system performing a reflective act in which the system would reify to the current object's meta level which would decide whether behavioural modifications were necessary and import an appropriate actor into the behavioural slot. Upon deification, (i.e. returning to the thread of execution belonging to the actor in the slot), the new behaviour would take effect. Consequently, by changing the actor whose execution is performing either a behavioural or functional role in the system, a natural causal link occurs relative to the system's execution when that slot resumes its base level computation (since it is now a different actor). Orthogonally applicable to policies, patterns and mechanisms at various levels in the architecture, Fig. 12 illustrates this notion with respect to changing the style of classroom discussion between the classic lecture style and group brainstorming.

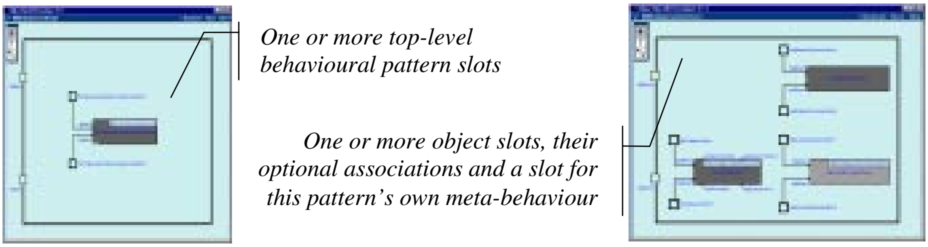

*One or more top-level behavioural pattern slots*
*One or more object slots, their optional associations and a slot for this pattern's own meta-behaviour*

**Fig. 11.** ObjecTime View of a Collaborative Session (*left*) and Behavioural Pattern (*right*)

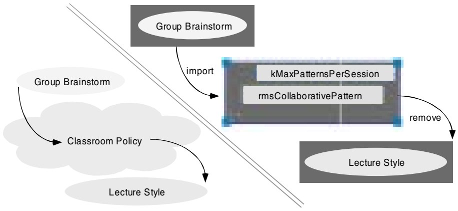


**Fig. 12.** Switching Classroom Behaviour Policies

As an alternative example, consider the physical entities within RASCAL's topology, such as the Proxy Chair. The software governing these entities (Fig. 10) is derived from *RASCALBase*, and from there they inherit support for reflection and the ability to perform reflective acts. They therefore have the means by which to monitor and potentially adapt themselves via reflection in order to meet the evolving needs of a media space. As a logical entity, the Proxy Chair could be designed in such a way that it could dynamically distribute its workload to sibling coordinators (moving from a strictly centralized to a distributed Proxy) based on issues such as type of media traffic, number of participants and so forth. Similarly, the multicast facility could adapt itself, with respect to QoS issues, to address particular types of traffic and/or congestion. The policies and mechanisms to address these kinds issues, however, could vary considerably and would best be provided using a meta-level approach.

This flexibility within the framework is mirrored in the prototype's user interface shown in Fig. 13. The prototype's main window represents the whole media space and functions as a container of different collaborative sessions (realized as child windows). Each session consists of objects that represent media and applications that the users utilize within the collaboration. To illustrate the notion of implementational reflection, the prototype utilizes OLE technology to facilitate inserting a wide variety of object types and functionality into the space. The sample screen shot illustrates the use of a graphic, textual document and live video camera input. Each of these objects is independently manipulated by its own application program. Events generated through the manipulation of these objects via the user interface, which is written in C++, are then passed to the ObjecTime backbone to be coordinated.

Therefore, the prototype clearly separates the system's coordination from its execution. Events generated through the manipulation of user interface elements

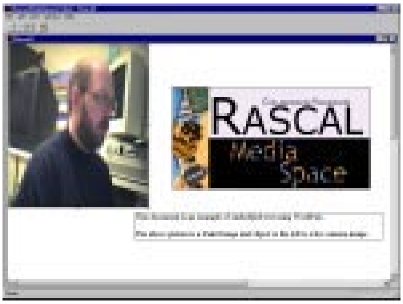


**Fig. 13.** Snapshot of Prototype's User Interface

represent media space activities which are coordinated according to the behavioural patterns governing a collaborative session. In both cases, the user interface and behavioural patterns form a meta-level which abstracts a specific aspect of a media space entity. Doing so simplifies their modification and promotes their re-use as structural and behavioural components. While shown through the use of different media and application objects via an OLE interface, the premise can be extended to support less conventional user interfaces, like those in virtual reality systems.

## 7 Summary and Conclusion

This paper has presented the notion of a collaborative media space and introduced RASCAL as a versatile and flexible foundation on which to build a broad range of media space systems. In particular, the framework was shown to provide for highly modular and adaptive systems, both in terms of structure and behaviour. As part of the framework's reflective meta-architecture, RASCAL explicitly separates the system's coordination from its execution. The architecture and its role as a management and support infrastructure for collaborative multimedia computing were presented along with the model's application to the field of distance education.

RASCAL provides a flexible and adaptive way to design, implement and extend a media space, both in terms of structure and behaviour. Both of these elements are regarded as orthogonal building blocks from which systems can be built, modified or replaced throughout their lifetime. Structural diversity is shown through the potential to address a variety of objects (media, applications and so forth) while behavioural aspects are shown in terms of how the model can supports various coordination strategies and synchronization mechanisms. The versatility of the proposed framework was shown amenable to a range of media space applications without the need to embed aspects of the system's coordination in its topology.

On-going work related to RASCAL includes development of the prototype along with continued evolution and exploration of its framework. The prototype is being developed under Windows NT using MFC/C++ and the ObjecTime Developer CASE tool. The dual approach blends an implementation of the user interface along with a model of the media space's communications system in order to concentrate on meta-architectural and behavioural issues rather than network-specific ones. Additionally, ObjecTime offers the potential to address performance analysis related to those components involved in using reflection as a behavioural management technique. A topic for future investigation may also be to assess the feasibility of integrating a collaborative knowledge base within a RASCAL media space so it can intelligently modify itself based on observing the behavioural patterns of its users.

As applied to the development of media space systems, RASCAL illustrates the importance of using design and implementation methods that inherently mirror the system being modelled. The highly dynamic and complex nature of multimedia systems needs to be addressed using techniques that can express this complexity in a simple and understandable manner. Consequently, RASCAL's use of reflection and its meta-architectural approach can be seen to offer an elegant way to deal with media space complexity in a straightforward, scalable, flexible and adaptive fashion.

## References

[1] Gutwin, C., Greenberg, S.: Workspace Awareness in Real-Time Distributed Groupware. Technical Report 95-575-27, Dept of Computer Science, University of Calgary (1995)

[2] Malone, T. W., Crowston, K.: The Interdisciplinary Study of Coordination. ACM Computing Surveys 16.1 (1994) 87 – 119

[3] Steinmetz, R., Nahrstedt, K.: Multimedia: Computing, Communications and Application. Prentice-Hall, Upper Saddle River (1995)

[4] Yavatkar, R.: MCP: A Protocol for Coordination and Temporal Synchronization in Multimedia Collaborative Applications. Proc. ICDCS, IEEE Com. Soc. (1992) 606 – 613

[5] Selic, B., Gullekson, G., McGee, J., Engelberg, I.: ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. Proc. CASE'92, IEEE Com. Soc. (1992)

[6] Stults, R.: Media Space. Technical Report, Xerox Corporation, Palo Alto (1986)

[7] Harrison, S., Minneman, S.: The Media Space: An Electronic Setting for Design. Technical Report SSL-89-63, Xerox Corporation, Palo Alto (1989)

[8] Bly, S. A., Harrison, S. R., Irwin, S.: Media Spaces: Bringing People Together in a Video, Audio and Computing Environment. Comm. of the ACM 36.1 (1993) 28 – 47

[9] Ahuja, S. R., Ensor, J. R., Horn, D. N.: The Rapport Multimedia Conferencing System. *Proc. ACM COIS*, ACM Press (1988) 1 – 8

[10] Eriksson, H.: MBone: The Multicast Backbone. Comm. of the ACM 37.8 (1994) 56 – 60

[11] Lauwers, J. C., Lantz, K. A.: Collaboration Awareness in Support of Collaboration Technology: Requirements for the Next Generation of Shared Window Systems. Proc. CHI'90, ACM Press (1990) 303 – 311

[12] Akyildiz, F., Yen, W.: Multimedia Group Synchronization Protocols for Integrated Services Networks. IEEE J. Selected Areas in Comm. 14.1 (1996) 162 – 173

[13] Dourish, P.: Open Implementation and Flexibility in CSCW Toolkits. Ph. D. Thesis, University College London (1996)

[14] Robbins, W., Georganas, N. D.: Reflective Pattern-Based Approach to Media Space Management. In: Kandlur, D., Jeffay, K., Roscoe, T. (*eds.*): Proc. MMCN'99, Vol. 3654. SPIE, Bellingham (1999) 29 – 40

[15] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R.: MMConf: An Infrastructure for Building Shared Applications. Proc. CSCW'90, ACM Press (1990) 637 – 650

[16] Maes, P.: Computational Reflection. Ph. D. Thesis, Vrije Universiteit Brussel (1987)

[17] Ibrahim, M. H.: Reflection in Object-Oriented Programming. Intl. J. Artificial Intelligence Tools 1.1 (1992) 117 – 136

[18] Kiczales, G., Paepcke, A.: Open Implementations and Metaobject Protocols. Tutorial Notes, Xerox Corporation, Palo Alto (1996)

[19] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons Ltd., Chichester (1996)

[20] Robbins, W., Georganas, N. D.: Shared Media Space Coordination: Mixed Mode Synchrony in Collaborative Multimedia Environments. Proc. IEEE ICMCS'97, IEEE Com. Soc. (1997) 466 – 473

[21] Robbins, R. W.: Collaborative Media Space Coordination Using RASCAL. Ph. D. Candidacy Paper, School of Information Technology and Engineering (Electrical and Computer Engineering), University of Ottawa (1997)

[22] Robbins, W., Georganas, N. D.: Collaborative Media Space Architectures. *Proc. CCBR'98*, OCRI Publications (1998) 284 – 295

[23] ITU-T, Principles of Intelligent Network Architecture. *Recomm. Q.1201*, Geneva (1992)

[24] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (1991)

# The Design of a Resource-Aware Reflective Middleware Architecture

Gordon S. Blair[1], Fábio Costa[1], Geoff Coulson[1], Fabien Delpiano[2], Hector Duran[1], Bruno Dumant[2], François Horn[2], Nikos Parlavantzas[1], Jean-Bernard Stefani[2]

[1]Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK.
{gordon, geoff, fmc, duranlim, parlavan}@comp.lancs.ac.uk

[2]CNET, France Télécom, 38-40 rue Général Leclerc, 92794 Issy les Moulineaux, CEDEX 9, Paris, France.
{fabien.delpiano, bruno.dumant, francois.horn, jeanbernard.stefani}@cnet.francetelecom.fr

**Abstract.** Middleware has emerged as an important architectural component in supporting distributed applications. With the expanding role of middleware, however, a number of problems are emerging. Most significantly, it is becoming difficult for a single solution to meet the requirements of a range of application domains. Hence, the paper argues that the next generation of middleware platforms should be both configurable and re-configurable. Furthermore, it is claimed that reflection offers a principled means of achieving these goals. The paper then presents an architecture for reflective middleware based on a multi-model approach. The main emphasis of the paper is on resource management within this architecture (accessible through one of the meta-models). Through a number of worked examples, we demonstrate that the approach can support introspection, and fine- and coarse- grained adaptation of the resource management framework. We also illustrate how we can achieve multi-faceted adaptation, spanning multiple meta-models.

## 1. Introduction

Middleware has emerged as a key architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. The importance of the topic is reflected in the increasing visibility of industrial standardisation activities such as OMG's CORBA and Microsoft's DCOM.

The expanding role of middleware is however leading to a number of problems. Most importantly, middleware is now being applied to a wider range of application domains, each imposing their own particular requirements on the underlying middleware platform. Examples of this phenomenon include real-time multimedia systems (requiring an element of quality of service management), and embedded systems (requiring small footprint technologies). In our opinion, this implies that future middleware platforms should be *configurable* in terms of the components used in the construction of a given instance of a platform. With the advent of mobile

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 115-134, 1999

computing, changes may also occur in the underlying network environment. Given this, it is also becoming increasingly important that middleware platforms are *re-configurable*, i.e. underlying components can be changed at run-time. Finally, and of central importance to this paper, we also believe that future middleware platforms should provide sophisticated facilities for *resource management*, in terms of allocating (potentially) scarce resources, being aware of patterns of resource usage, and adapting to changes in resource patterns.

The authors believe the general solution to these problems is to introduce more openness and flexibility into middleware technologies. In particular, we advocate the use of reflection [Smith82, Kiczales91] as a means of achieving these goals. Reflection was first introduced in the field of programming languages and has recently been applied to a range of other areas including operating systems and windowing systems (see section 5). We believe that similar techniques can usefully be applied to middleware technology. This solution contrasts favourably with existing approaches to the construction of middleware platforms, which typically adopt a black box philosophy whereby design decisions are made by the platform developers in advance of installation and then hidden from the programmer at run-time.

Note that some flexibility has been introduced into middleware products. For example, CORBA now supports a mechanism of interception whereby wrappers can be placed round operation requests. However, this can be viewed as a rather ad hoc technique for achieving openness. The Portable Object Adaptor can also be criticised in this way. Some work has been carried out on developing more open ORBs. For example, the TAO platform from the University of Washington offers a more open and extensible structure, and also features the documentation of software patterns as used in the construction of TAO [Schmidt97]. TAO also offers access to key resource management functionality. Similarly, BBN's QuO platform employs techniques of open implementation to support quality of service management, in general, and adaptation, in particular [Vanegas98]. Finally, a number of researchers have recently experimented with reflective middleware architectures (see section 5). These platforms address many of the shortcomings of middleware platforms. However, none of these initiatives offer a complete and principled reflective architecture in terms of configurability, re-configurability and access to resource management.

Based on these observations, researchers at Lancaster University and CNET are developing a reflective middleware architecture as part of the Open ORB Project. This architecture supports introspection and adaptation of the underlying components of the middleware platform in terms of both structural and behavioural reflection [Watanabe88]. Crucially, the architecture also supports reification of the underlying resource management framework. This paper discusses the main features of this architecture with particular emphasis on the resource management aspect of our work.

The paper is structured as follows. Section 2 presents an overview of the reflective middleware architecture highlighting the multi-model approach that is central to our design. Section 3 then focuses on resource management, discussing the resource management framework in detail and considering the integration of this framework into the reflective architecture. Following this, section 4 presents three contrasting examples illustrating how our architecture can support introspection, fine- and coarse-grained adaptation, and resource-aware adaptation. Section 5 then discusses some related work and section 6 offers some concluding remarks.

## 2. Open ORB Architecture

### 2.1. Overall Approach

In common with most of the research on reflective languages and systems, we adopt an *object-oriented model of computation*. As pointed out by Kiczales et al [Kiczales91], there is an important synergy between reflection and object-oriented computing:

> *"Reflective techniques make it possible to open up a language's implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language's implementation and behaviour to be locally and incrementally adjusted".*

The choice of object-orientation is important given the predominance of such models in open distributed processing [Blair97]. Crucially, we propose the use of the RM-ODP Computational Model, using CORBA IDL to describe computational interfaces. The main features of this object model are: i) objects interact with each other through *interfaces*, ii) objects can have *multiple interfaces*, iii) interfaces for *continuous media interaction* are supported, iv) interfaces are uniquely identified by *interface references*, and v) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding object*). The object model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details of this object model can be found in [Blair97]. In contrast, with RM-ODP, however, we adopt a consistent object model throughout the design [Blair98].

As our second principle, we adopt a *procedural* (as opposed to a declarative) approach to reflection, i.e. the meta-level (selectively) exposes the actual program that implements the system. In our context, this approach has a number of advantages over the declarative approach. For example, the procedural approach is more primitive (in particular, it is possible to support declarative interfaces on top of procedural reflection but not vice versa[1]). Procedural reflection also opens the possibility of an infinite tower of reflection (i.e. the base level has a meta-level, the meta-level is implemented using objects and hence has a meta-meta-level, and so on). This is realised in our approach by allowing such an infinite structure to exist in theory but only to instantiate a given level on demand, i.e. when it is reified. This provides a finite representation of an infinite structure (a similar approach is taken in ABCL/R [Watanabe88]). Access to different meta-levels is important in our design although in practice most access will be restricted to the meta- and meta-meta-levels.

The third principle behind our design is to support *per object* (or *per interface*) meta-spaces. This is necessary in a heterogeneous environment where objects will have varying capacities for reflection. Such a solution also provides a fine level of control over the support provided by the middleware platform; a corollary of this is that problems of maintaining integrity are minimised due to the limited scope of

[1] Indeed, we have already experimented with constructing more declarative interfaces on top of our architecture in the context of the resources framework.

change. We do recognise however that there are situations where it is useful to be able to access the meta-spaces of sets of objects in one single action; to support this, we also allow the use of (meta-object) *groups* (see discussion in 2.2 below).

The final principle is to structure meta-space as a number of closely related but distinct *meta-space models.* This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [Okamura92]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. Further details of each of the various models can be found below.

## 2.2. The Design of Meta-space

### 2.2.1. Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content of a given object [Watanabe88]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the *encapsulation* and *composition* meta-models. We introduce each model in turn below.

The *encapsulation meta-model* provides access to the representation of a particular interface in terms of its set of methods and associated attributes, together with key properties of the interface including its inheritance structure. This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. Clearly, however, the level of access provided by the encapsulation model will be language dependent. For example, with compiled languages such as C access may be limited to introspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python, more complete access is possible such as being able to add or delete methods and attributes. This level of heterogeneity is supported by having a type hierarchy of encapsulation meta-interfaces ranging from minimal access to full reflective access to interfaces. Note however that it is important that this type hierarchy is open and extensible to accommodate unanticipated levels of access.

In reality, many objects will in fact be composite, using a number of other objects in their construction. In recognition of this fact, we also provide a *compositional meta-model* offering access to such constituent objects. Note that this meta-model is associated with each object and not each interface. More specifically, the same compositional meta-model will be reached from each interface defined on the object (reflecting the fact that it is the object that is composite). In the meta-model, the composition of an object is represented as an *object graph* [Hokimoto96], in which the constituent objects are connected together by *local bindings*[2]. The interface offered by the meta-model then supports operations to inspect and adapt the graph structure, i.e. to view the structure of the graph, to access individual objects in the graph, and to adapt the graph structure and content.

This meta-model is particularly useful when dealing with binding objects [Blair98]. In this context, the composition meta-model reifies the internal structure of the

[2] This RM-ODP inspired concept of local binding is crucial in our design, providing a language-independent means of implementing the interaction point between interfaces.

binding in terms of the components used to realise the end-to-end communication path. For example the object graph could feature an MPEG compressor and decompressor and an RTP binding object. The structure can also be exposed recursively; for example, the composition meta-model of the RTP binding might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol. It is argued in [Fitzpatrick98] that open bindings alone provide strong support for mobile computing.

### 2.2.2. Supporting Behavioural Reflection

Behavioural reflection is concerned with activity in the underlying system [Watanabe88]. This is represented by a single meta-model associated with each interface, the *environmental meta-model.* In terms of middleware, the underlying activity equates to functions such as message arrival, enqueueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [Watanabe88, McAffer96].

Again, different levels of access are supported. For example, a simple meta-model may enable the insertion of pre- and post- methods. Such a mechanism can be used to introduce, for example, some additional levels of distribution transparency (such as concurrency control). In addition, additional functions can be inserted such as security functions. A more complex meta-model may allow the inspection or adaptation of each element of the processing of messages as described above. With such complexity, it is likely that such a meta-model would itself be a composite object with individual components accessed via the compositional meta-space of the environmental meta-space. As with the other meta-models, heterogeneity is accommodated within an open and extensible type hierarchy.

### 2.2.3. The Associated Component Library

To realise our open architecture, we provide an open and extensible library of components that can be assembled to build middleware platforms for a range of applications. By the term component, we mean a robust, packaged object that can be independently developed and delivered. Our components adhere to a component framework that offers a built in event notification service and also access to each of the meta-models.

The component library consists of both primitive and composite components, the distinction being that primitive components are not open to introspection or adaptation in terms of the compositional meta-model. Examples of primitive components include communication protocols such as UDP and TCP, and end system functions such as continuous media filters. Composite components then represent off-the-shelf configurations of components such as end-to-end stream bindings. Importantly, we also provide *group bindings* (or simply groups) as composite objects. The role of groups is to provide a uniform mechanism for invoking a set of interfaces whether they are at the base-level, meta-level, meta-meta-level, etc. Furthermore, through access to their underlying object graph of the group component, it is possible to tailor and adapt the semantics of group message distribution, e.g. by introducing a new ordering protocol.

### 2.3. Management of Consistency

In reflective architectures, care must be taken to ensure the consistency of the underlying virtual machine. This is particularly true in the reflective middleware architecture described above, as the internal structure of the underlying platform can be changed significantly by using the meta-object protocols associated with the meta-models. For example, using the encapsulation meta-model, it is possible to introduce new methods or attributes into a running object. Similarly, the compositional and environmental meta-models allow the programmer to change the construction of a composite object and the interpretation of method invocation respectively. Given this, steps must be taken to ensure that consistency constraints are maintained.

As can be seen from the architecture, the most common style of adaptation is to alter the structure of an object graph. Consequently, it is crucial that such run-time alterations can be made in a safe manner. Some support is provided by the strongly-typed object model in that local bindings will enforce type correctness across interacting objects. This is however not a complete solution to the problem. The following steps are also necessary[3]:

1. Any activity in the affected region of the graph must be suspended;
2. The state of such activity must be preserved;
3. Replacement components must maintain the desired level of quality of service.

In addition, it is also necessary to maintain global consistency across the graph, i.e. changes of one component may require changes to another component (as in the case when replacing a compression and decompression object).

Responsibility for enforcing consistency constraints may be left to the application in our architecture. However, it is much more likely that the application will devolve responsibility to QoS management components that monitor the underlying object graph and implement an appropriate adaptation policy. The design of such a QoS management infrastructure for our reflective architecture is discussed in [Blair99].

We return to this important issue in section 4 below.

## 3. Resource Meta-Model

### 3.1. The Resource Framework

### 3.1.1. The Overall Approach

The main aim of the resource framework is to provide a complete and consistent model of all resources in the system at varying levels of abstraction. For example, a consistent approach should be taken to the management of processing capacity, memory, and network resources. Similarly, the framework should encompass more abstract resources such as virtual memory, persistent virtual memory, connections, teams of threads, and more general collections of resources.

[3] This is similar to the problem of (nested) atomic transactions, but with the added requirement of maintaining the quality of service constraints [Mitchell98].

This is achieved by recursively applying the structure illustrated in figure 1.

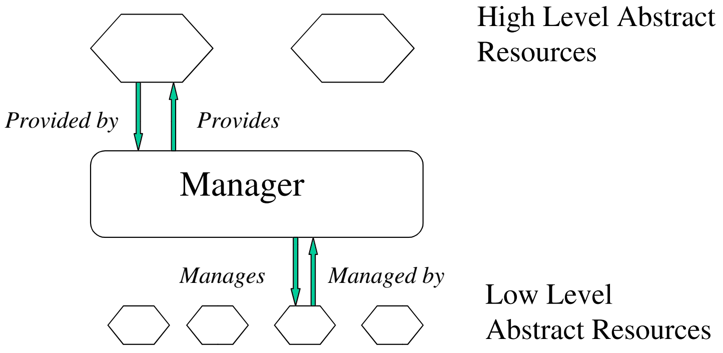


**Fig. 1.** The Overall Resource Framework

This diagram shows how higher level resources are constructed out of lower level resources by resource manager objects [ReTINA99]. Managers *provide* the higher level resources and *manage* the lower level resources. Similarly, higher level resources are *provided by* managers and lower level resources are *managed by* managers (we return to this in section 3.3). This structure maps naturally on to facilities offered by most modern operating systems. Note that there is an assumption that access to resource management is possible. At the very least, this will be possible in terms of user level resources such as user level threads. The level of access can also be enhanced through techniques such as split-level resource management [Blair97].

Both the high level and low level resources are of type `AbstractResource` as defined by the following interface:

```
interface AbstractResource {
 Manager provided_by();
 Manager managed_by();
 void release();
}
```

The `provided_by()` operation enables an application to locate the manager for a high level resource. Similarly, the `managed_by()` operation can be used to locate the manager for a given low level resource. Finally, `release()` should be called when the (abstract) resource is no longer needed.

Different styles of manager can be created. For example, some managers may multiplex high level resources on to the set of low level resources (e.g. schedulers), or they can map directly on to low level resources (e.g. memory allocation), with the set of available resources either being static or dynamic. Similarly, they can compose low level resources to create a more abstract high level resource (e.g. to create a Unix-like process), or they can provide direct access to a low level resource (cf binders).

In general, managers offer the following interface:

```
interface Manager {
 void register(Interface ref,...);
 void unregister(Interface ref);
```

```
  AbstractResource[] manages(AbstractResource high);
  AbstractResource[] provides();
 }
```

The `register()` and `unregister()` operations allow a programmer to associate additional low level resources with the manager. The precise interpretation of this will vary between different styles of manager as seen when we consider factories and binders below. The `manages()` and `provides()` operations return the high level and low level abstract resources associated with this manager respectively. The `high` parameter to `manages()` enables the manager to return only the low level resources supporting that given high level resource.

Note that this approach treats resource as a relative term. An object *r* is a resource *for* another object *u* - a *user* -, that itself may be a resource for a third object *v*, or possibly for *r*. In other words, resource and user are roles, similar to server and client.

### 3.1.2. Factories and Binders

*Overview*
Suppose now that an object requires access to an abstract resource. There are two ways in which this can happen. Firstly, it can create a new instance of the required abstract resource. Secondly, it can locate and bind to an existing instance of an abstract resource. These two scenarios are supported in the framework by *factories* and *binders* respectively.

As will be seen, both factories and binders are specialisations of `Manager` in the architecture in that they both have the role of providing an `AbstractResource` for an object (the `Manager()` class described above should be viewed as an abstract class, which is realised by factories, binders, or indeed other styles of manager). In this way, they can be viewed as different *patterns* of object management.

We look at each in turn below.

*Factories*
As stated above, factories support the creation of new objects of type `AbstractResource`. In our architecture, they also have a second role, i.e. performing the run-time management of the resultant resource(s). Combining the two roles has a number of advantages, e.g. in integrating policies for static and dynamic resource management.

A factory must conform to the following interface:

```
interface Factory extends Manager {
  AbstractResource new(...);
}
```

The `new()` method returns a reference to an interface of an abstract resource. Its arguments may contain information needed by the factory to build and manage the new resource, like specifications of the type and quality of service of the expected resource, indications of lower level resources or factories to be used, etc. The `register()` operation defined in the super class effectively provides the factory with additional raw material to create the new high level abstract resource, e.g. in terms of providing a new virtual processor to support threads. The `unregister()` operation has the opposite effect of removing the low level abstract resource from the factory.

In a typical environment, an extensible range of factories will be provided corresponding to the various levels of abstract resources discussed in section 3.1.1.

*Binders*

As stated earlier (section 2.1), an object is only aware of other objects through knowledge of interface references. However, holding an interface reference does not mean that it is possible for the holder of the reference to interact with the corresponding object. The object may be inaccessible or unusable in its current form.

The role of a binder object is to make identified interfaces accessible to their user. More specifically, in the resources framework, they support the location of appropriate resources and also undertake the necessary steps to make the interface accessible. For example, if an interface is located on a remote machine, the binder will establish an appropriate communications path to the object or indeed migrate the remote object to the local site.

The interface for a binder object is shown below:

```
interface Binder extends Manager {
 AbstractResource bind(Interface ref,...);
}
```

To be managed by a binder, an interface must register itself using the inherited `register()` method. In this context, this is similar to registering an interface with a name server or trading service. This means that the binder is now entitled to let users access and interact with that interface. The binder may also take steps to make this resource accessible. The precise policies used for allocation may be implicit in the binder, encapsulated in the provided interface reference, or made explicit in parameters to the `register()` method. Additional parameters may also specify management policies (cf the policies provided in CORBA in the Portable Object Adapter specification). The corresponding `unregister()` method then removes this abstract resource from control of that binder.

The role of the `bind()` method is to allow the specified interface to be used. This method returns an interface reference that should then be used to interact with that object. This is likely to be an interface reference for the object itself. The architecture however also allows this to refer to an intermediary interface that will provide indirect access to the appropriate object (cf proxy mechanisms).

## 3.2. Application to Processing Resources

### 3.2.1. A Resource Framework for Processing

To illustrate the use of the resource management framework, we now discuss a particular instantiation of the framework to support the management of processing resources. As an example, we present a two-level structure whereby lightweight threads are constructed using virtual processors, which are themselves built on top of the underlying processor(s). The corresponding structure is illustrated in figure 2.

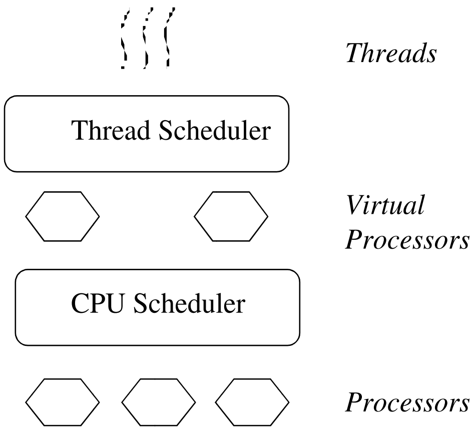


**Fig. 2.** An Example Scheduling Framework

From this diagram, it can be seen that *schedulers* fulfil the role of managers (more specifically factories). The CPU scheduler provides the abstraction of virtual processors out of the underlying physical processors. Similarly, the thread scheduler creates the abstraction of threads and maps them on to the underlying virtual processor(s). In both cases, the scheduling policy is encapsulated within each of the schedulers but may be changed dynamically (see below).

Processors, virtual processors and lightweight threads can all be viewed as a particular type of abstract resource that we refer to as a *machine*. In general terms, machines consume messages, act on these messages (possibly changing the state of the corresponding object), and then emit further messages. We also introduce a specialisation of a machine, called a *job*, which can be configured with scheduling parameters.

### 3.2.2. The Major Objects Revisited

*Machines and Jobs*

As mentioned above, machines are abstract resources that are responsible for carrying out some activity. They may be processors or sets of processors, or may represent more abstract execution environments such as virtual processors or indeed interpreters for high level languages. They may also be sequential or parallel.

Machines offer the following interface:

```
interface Machine {
 void run(Message);
}
```

The `run()` method provides a target machine with a new message (assuming the machine is in an appropriate state to accept the message). A `Job` is then an abstract machine that uses lower-level machines controlled by the scheduler to execute its messages. It offers the following interface:

```
interface Job extends Machine, AbstractResource {
 SchedParams getSchedParams();
 void setSchedParams(SchedParams params);
}
```

The two methods, `getSchedParams()` and `setSchedParams()` can be used to inspect the current scheduling parameters used by the underlying scheduler and to change these parameters dynamically.

*Schedulers*

The precise definition of a scheduler is that it is a machine factory that is able to create abstract machines (jobs) and then multiplex them on to lower-level machines. Its interface is as follows:

```
interface Scheduler extends Factory {
 Job new(SchedParams params);
 void notify(Event event);
 void suspend(...);
 void resume(...);
}
```

As can be seen, the interface supports a `new()` method that returns an object of type `job`. The `params` argument represents the initial *scheduling parameters* that will be used by the scheduler to control the multiplexing. These may be expressed in terms of priorities, deadlines, periods, etc. The scheduler implements a function to decide at any moment how jobs should be allocated to the various machines under its control[4] (according to these scheduling parameters). For instance, threads may be understood as jobs offered by a scheduler; the scheduler will then allocate the processor(s) (the lower-level machine(s)) to them according to their priority (the scheduling parameter). The scheduler may also use some dynamic information, like the timestamp on specific events, to control the multiplexing. Obviously, if there are fewer machines than jobs, the scheduler must maintain a job queue.

The decisions taken by a scheduler to multiplex jobs on machines may be triggered by different events such as interruptions or timer notifications. More generally, an event is an asynchronous occurrence in the system (message reception, message emission, object creation, etc).

The remaining methods defined on `Scheduler` provide a level of event management. The `notify()` method informs the scheduler of the occurrence of a specific event. The `suspend()` method then specifies that the execution of one or more jobs, as given by the arguments of the call, should be stopped and blocked. Finally, the `resume()` method is called to make blocked jobs eligible for re-

[4] Note that, in our framework, we enforce a restriction that a machine can be controlled by one and only one scheduler, this being the sole entity allowed to call the `run()` method. In contrast, a single scheduler may manage and use several machines.

execution. Parameters may include events that should trigger the suspend/resume mechanism. Locks may also be provided as specific instantiations of these operations.

### 3.3. Incorporation into Open ORB

The resource management framework is incorporated into the Open ORB architecture as an additional meta-model, referred to as the *resources meta-model*. The complete Open ORB architecture therefore consists of four meta-models as shown in figure 3.

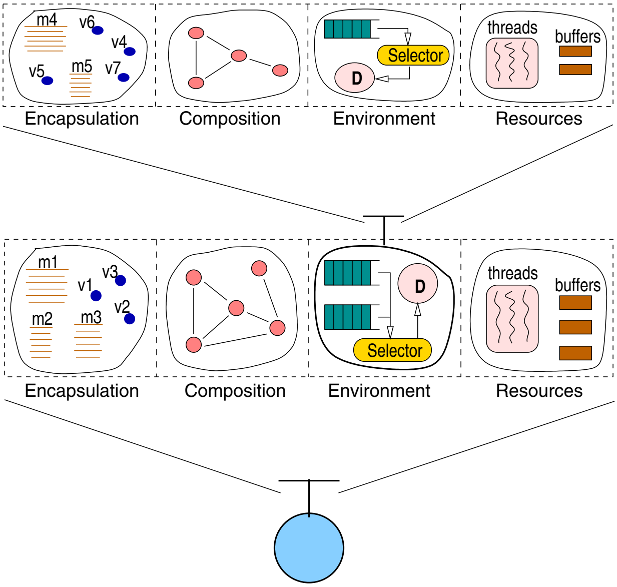


**Fig. 3.** The Open-ORB Reflective Architecture

It is important to stress that this meta-model is orthogonal to the others. For example, the environmental meta-model identifies the steps involved in processing an arriving message; the resources meta-model then identifies the resources required to perform this processing. More generally, the resources meta-model is concerned with the allocation and management of resources associated with any activity in the system. There is an arbitrary mapping from activity to resources. For example, all objects in a configuration could share a single pool of resources. Alternatively, each object could have its own private resources. All other combinations are also possible including objects partitioning their work across multiple resource pools and resources spanning multiple objects.

As an example, consider a configuration of objects providing an environmental model. One object could deal with the arrival and enqueuing of all messages from the network. This object could have a single high-priority thread and a dedicated pool of buffers associated with it. A further set of objects could deal with the selection, unmarshalling and dispatching of the message. In this case, the various objects could all share a pool of threads and a pool of buffers. Note that, once allocated to deal with a message, the thread (and associated buffer) will complete the entire activity, spanning multiple object boundaries. Synchronisation with other threads (and buffers) is only required when switching between abstract resources, e.g. when handing over from the arrival activity to the selection activity.

The resources meta-model supports the reification of the various objects in the resource framework. More specifically, the meta-model provides access to the (one or more) top-level abstract resources associated with a given interface. For example, a given interface might have a single abstract resource containing a global pool of threads and a dedicated buffer pool. It is then possible to traverse the management structure from the abstract resource(s). In particular, we can locate the manager for a given abstract resource by following a *provided by* link (see section 3.1.1). In turn, each manager maintains a list of resources that it *manages*, providing access to abstract resources one level down. This process can then be applied recursively to reach any particular object of the management structure. Similarly, from a resource, it is possible to follow a *managed by* link to locate the appropriate manager. It is then possible to trace the abstract resource that this manager *provides* (see figure 1).

Using such traversal mechanisms, it is possible to inspect and adapt the management structure associated with such abstract resources. We distinguish between fine-grained adaptation whereby the management structure remains intact and coarse-grained adaptation that makes changes to this structure. Examples of both styles of adaptation are given below.

# 4. Explanatory Examples

## 4.1. Introspection and Fine-grained Adaptation

This example demonstrates introspection and fine-grained adaptation for processing resources. We consider a single object, offering a single interface `iref`. This object is supported by a set of abstract resources that are bundled into a single top level abstract resource. In particular, this bundle contains a *team of threads* and some associated *virtual memory*. We assume a two-level management structure for threads as defined in section 3.2 above. We note that the threads are, in general, under-performing and decide to alter the priority of the underlying virtual processor (we assume that all threads in the team map on to one virtual processor).

This level of adaptation is achieved as follows:

```
top  = iref.Resources()[BUNDLE];
team = top.provided_by().manages(top)[THREADS];
vp   = team.provided_by().manages(top)[VP];
vp.setSchedParams (new_params);
```

The first line enters the resources meta-model, using a Resources() method as defined on all (reflective) objects. Note that this operation returns an array of all the top level abstract resources associated with the interface. We thus provide an index to select the appropriate resource (in this case the bundle, denoted by BUNDLE). From this, we can access the underlying team of threads and then the underlying virtual processor by using the provided_by() and manages() links. Note that we must also provide an index to select the appropriate low level resource from the array returned by the manages() operation. From there, it is straightforward to change the scheduling parameters on the appropriate virtual processor.

### 4.2. Coarse-grained Adaptation

In this example, we assume the same top-level abstract resource as before. This time, however, we want to change virtual memory to persistent virtual memory. This implies some fundamental changes to the management structure. In addition, this example raises important issues of consistency, i.e. the new persistent memory should mirror the original virtual memory and the change should be atomic.

We assume that virtual memory is provided by a single resource manager that maps to both the underlying physical memory and secondary storage resources. We also assume that we have access to a factory, denoted by `pvm_factory`, which can be used to create a new segment of persistent virtual memory.

The first few steps are similar to the example above:

```
top  = iref.Resources()[BUNDLE];
vm   = top.provided_by().manages(top)[VM];
team = top.provided_by().manages(top)[THREADS];
vp   = team.provided_by().manages();
```

In this case, however, we locate the virtual memory abstract resource *and* the underlying virtual processor. The latter is required as it is necessary to suspend all activity on the object while the changes are made, i.e. to make the change atomic. Note that this only works if we assume that no other threads will operate on the abstract resource. If this is not the case, then we must lock the resource instead.

We proceed as follows:

```
vp.suspend();
pvm   = pvm_factory.new(vm);
vm.release();
meta.provided_by().unregister(vm);
meta.provided_by().register(pvm);
vp.resume();
```

As can be seen, we use the state of the old virtual memory in the creation of the new persistent virtual memory (we assume that the factory supports this initialisation capability). The old virtual memory abstract resource can now be released. It then remains to alter the top level abstract resource (via its manager) to use the persistent virtual memory rather than the previous virtual memory.

### 4.3. Multi-faceted Adaptation

The final example illustrates how adaptation strategies can span multiple meta-models. In particular, we consider adaptation of a continuous media binding object, referring to the compositional and resources meta-models (the former provides access to the components used in the construction of the binding, and the latter provides information on underlying resource usage). The binding object is responsible for the delivery of audio in a mobile environment, where the available bandwidth may change significantly on roaming between different network types. The object has an initial structure as shown in figure 4.

As can be seen, the configuration contains a GSM encoder and decoder with a throughput requirement of approx. 13kbits/sec. The role of the monitoring object is to report if throughput drops below this threshold. Should this happen, the compression and decompression objects must be changed to a more aggressive scheme such as LPC-10 (which only requires 2.4kbits/sec). This is only feasible, however, if there is spare processing capacity as the computational overhead of LPC-10 is significantly greater than GSM. To support this, we assume the existence of a monitoring object in the resources meta-model that collects statistics on the available processing resources.

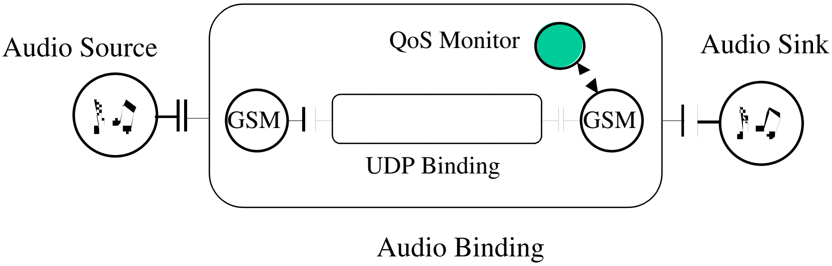


**Fig. 4.** The Composite Binding Object

The solution is outlined below. (As this is a more complex example, we do not go to the level of providing code.)

Firstly, it is necessary to locate the relevant monitoring objects in the compositional and resource meta-spaces. On detecting a drop in available bandwidth, it is then necessary to check if there is spare processing capacity. If there is, then adaptation can proceed (if not, it may be possible to obtain additional processing capacity, e.g. by increasing the priority of the underlying virtual processor as described in section 4.1. above[5]).

To achieve adaptation, it is first necessary to suspend activity in the binding (as in section 4.2 above). With this being a distributed object, however, this will involve the suspension of activity in more than one abstract resource. We can then use graph manipulation operations to replace the encoder and decoder with the new LPC-10 components[6]. Following this, activity can be resumed. Note that suspension of activity may not be sufficient to ensure consistency, i.e. some GSM packets may remain in transit until after the change-over. For the sake of this example, we assume that decoders will discard invalid packets.

In general, this is an example of a *resource aware QoS management* strategy. As an extension of this strategy, we can implement adaptation policies that monitor and

[5] Strictly speaking, such a step could violate QoS contracts in other parts of the system and so should involve QoS renegotiation [ReTINA99].

[6] Again, this step could involve QoS re-negotiation.

adapt one or more meta-models. Further discussion of such multi-faceted adaptation and QoS management can be found in [Blair99].

## 5. Related Work

### 5.1. Reflective Architectures

As mentioned earlier, reflection was first introduced in the programming language community, with the goal of introducing more openness and flexibility into language designs [Kiczales91, Agha91, Watanabe88]. More recently, the work in this area has diversified with reflection now being applied to areas such as operating system design [Yokote92], atomicity [Stroud95] and atomic transactions [Barga96], windowing systems [Rao91], CSCW [Dourish95]. There is also growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer [McAffer96]. With respect to middleware, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [Singhai97]. The level of reflection however is coarse-grained and restricted to invocation, marshalling and dispatching. In addition, the work does not consider the reification of resource management. Researchers at APM have developed an experimental middleware platform called FlexiNet [Hayton97]. This platform allows the programmer to tailor the underlying communications infrastructure by inserting/ removing layers. Their solution is, however, language-specific, i.e. applications must be written in Java. Manola has carried out work in the design of a "RISC" object model for distributed computing [Manola93], i.e. a minimal object model which can be specialised through reflection. Researchers at the Ecole des Mines de Nante are also investigating the use of reflection in proxy mechanisms for ORBs [Ledoux97].

Our design has been influenced by a number of specific reflective languages. As stated above, the concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [Okamura92]. Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [Watanabe88] and CodA [McAffer96]. Both these systems feature decompositions of meta-space in terms of the acceptance of messages, placing the message in a queue, their selection, and the subsequent dispatching. Finally, the design of ABCL/R2 includes the concept of groups [Matsuoka91]. However, groups in ABCL/R2 are more prescriptive in that they enforce a particular construction and interpretation on an object. The groups themselves are also primitive, and are not susceptible to reflective access.

### 5.2. Reflective Resource Management

In general, existing reflective architectures focus on structural reflection, behavioural reflection, or a combination of both. Most architectures do not address the reification of resource management. There are three notable exceptions, namely AL/1D, the ABCL/R* series of languages and Apertos. We consider each in turn below.

As mentioned above, Al/1D [Okamura92] offers a resource model as one of its six meta-models. This resource model focuses on resources on the local host and does not consider issues of distribution. AL1/D, like the other systems considered in this section, has an active object model. In terms of resources, this maps on to execution environments, called *contexts*, and memory abstractions, referred to as *segments.* These are managed by *schedulers* and *garbage collectors* respectively (with an object having precisely one of each). In terms of reflection, the resource model supports reification of contexts, segments, schedulers and garbage collectors. Consequently, this meta-model supports reification of only selected aspects of resource management; it does not attempt to provide as comprehensive a framework as we offer.

In ABCL/R2 [Matsuoka91], resource management is closely related to their concept of *groups* (see section 5.1). In their design, structural reflection is accessed through a meta-object, but behavioural reflection is controlled by a meta-group. This meta-group is implemented by a group kernel object offering a meta-object *generator* (for the creation of meta-objects), an evaluator (which acts as a sequential interpreter for the language), and a group manager (which bears the identity of the group and has reference to the meta-object generator and the evaluator). It is assumed that objects belonging to a group share a pool of common resources. Resource management policies are then contained within the various components of the group kernel object. For example, evaluators are responsible for scheduling of activity relating to the group. Such meta-behaviour can be changed dynamically. For example, as the evaluator is an object itself, the scheduling policy can be changed by inserting a call to a user-defined scheduler in the code of its meta-object. In contrast to ABCL/R2, our approach separates out the concepts of group and resource management, and also offers a more explicit resource management framework.

The successor, ABCL/R3 [Masuhara94] has a quite different resource model. In particular, the designers have added the concepts of node and node management to make the locality of computations more explicit at the meta-level (this is transparent to the base-level). A node is the meta-level representation of a processor element. Node management can then be used, for example, to control where objects are created, e.g. to introduce an element of load balancing. The language also makes node scheduling more explicit (again as a meta-level object on each node). This is an interesting development but again is less comprehensive than our proposed approach.

Finally, as a reflective operating system, one of the primary goals of Apertos is to provide open access to resource management [Yokote92]. In Apertos, every object has an associated meta-space offering an execution environment for that object (in terms of for example a particular scheduling policy, implementation of virtual memory, etc). This meta-space is accessed via a reflector object, which acts as a façade for the meta-space and offers a set of primitives for message passing, memory management, etc. Apertos adopts the computational model of active objects and pushes it to the limits. Every object managed by the system owns its computational resources, called a context), which includes, for example, memory chunks or disk blocks. This leads to prohibitive overheads in the system due to constant context switching. Our approach is more flexible with respect to active/ passive resource association, allowing high level active objects to co-exist with passive but efficient low level resources. In addition, in Apertos meta-space is structured as a set of interacting meta-objects, including resource objects and resource managers. These are not accessed directly; rather a form of coarse-grained adaptation is achieved by migrating an object from one meta-space to another (equivalent) meta-space (e.g. one that offers persistent virtual memory). The validity of migration is checked by calling a primitive defined on all reflectors called `canSpeak()`. The granularity of adaptation is therefore fixed by the class of the reflectors. In our framework, objects can gain direct access to the resources they use at the desired level of abstraction (e.g. an object may want to know at different times all the time-slices, threads or transactions executing its methods). Thus, we achieve dynamic reconfiguration at any level of expressiveness, the validity of adaptations being verified by type checking.

## 6. Concluding Remarks

This paper has discussed the design of next generation middleware platforms which we believe should be more configurable and re-configurable and should also provide access to the underlying resource management structure. We argue that this is best achieved through the use of reflection. More specifically, we advocate a reflective architecture where every object has its own meta-space. This meta-space is then structured as a series of orthogonal meta-models offering structural and behavioural reflection, and also reification of the resource management framework. Associated meta-object protocols support introspection and adaptation of the underlying middleware environment in a principled manner. This reflective architecture is supported by a component framework, offering a re-usable set of services for the configuration and re-configuration of meta-spaces. In general, re-configuration is achieved through reification and adaptation of object graph structures.

The main body of the paper concentrated on the resource model. The underlying resource management framework provides a uniform mechanism for accessing resources and their management at different levels of abstraction. We also presented a specialisation of this framework for activity management. We demonstrated, through a set of worked examples, that the approach can support introspection, fine- and coarse- grained adaptation, and also multi-faceted adaptation.

The work presented in this paper is supported by a number of prototype implementations. For example, the reflective middleware architecture has been implemented in the object-oriented, interpreted language Python [Watters96]. Currently, this particular implementation supports a full implementation of structural and behavioural reflection, together with an initial implementation of the resources meta-model. An earlier version of this platform (without the resources meta-model) is described in detail in [Costa98]. The resources framework is inspired by the underlying structure of Jonathan [Dumant97]. A more complete implementation of the resources framework is also being developed in Jonathan v2. A similar approach is also taken in GOPI [Coulson98].

Ongoing research is addressing two main distinct areas: i) further studies of consistency and adaptation, and ii) an examination of the support offered by the architecture for QoS-driven transaction mechanisms. This latter activity, in particular, should provide a demanding examination of the approach.

## Acknowledgements

The research described in this paper is funded by CNET, France Telecom (Grant 96-1B-239). We would also like to thank a number of other people at CNET and Lancaster University who have contributed to the work described in this paper, including Anders Andersen, Lynne Blair, Mike Clarke, Tom Fitzpatrick, Lee Johnston and Katia Saikoski.

## References

[Agha91] Agha, G., “The Structure and Semantics of Actor Languages”, Lecture Notes in Computer Science, Vol. 489, pp 1-59, Springer-Verlag, 1991.

[Barga96] Barga, R., Pu,, C., “Reflection on a Legacy Transaction Processing Monitor”, In Proceedings of Reflection 96, G. Kiczales (ed), pp 63-78, San Francisco; Also available from Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, P.O. Box 91000, Portland, OR 97291-1000, 1996.

[Blair97] Blair, G.S., Stefani, J.B., “Open Distributed Processing and Multimedia, Addison-Wesley, 1997.

[Blair98] Blair, G.S., Coulson, G., Robin, P., Papathomas, M., “An Architecture for Next Generation Middleware”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), pp 191-206, Springer, 1998.

[Blair99] Blair, G.S., Andersen, A., Blair, L., Coulson, G., “The Role of Reflection in Supporting Dynamic QoS Management Functions”, Internal Report MPG-99-03, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., 199 January 1999.

[Costa98] Costa, F., Blair, G.S., Coulson, G., “Experiments with Reflective Middleware”, Proceedings of the ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP’98 Workshop Reader, Springer-Verlag, 1998.

[Coulson98] Coulson, G., "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol. 21, No. 9, pp 802-818, July 1998.

[Dourish95] Dourish, P., “Developing a Reflective Model of Collaborative Systems”, ACM Transactions on Computer Human Interaction, Vol. 2, No. 1, pp 40-63, March 1995.

[Dumant97] Dumant, B., Horn, F., Dang Tran, F., Stefani, J.B., “Jonathan: An Open Distributed Processing Environment in Java”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), Springer, September 1998.

[Hayton97] Hayten, R., “FlexiNet Open ORB Framework”, APM Technical Report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, October 1997.

[Hokimoto96] Hokimoto, A., Nakajima, T., “An Approach for Constructing Mobile Applications using Service Proxies”, Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS’96), IEEE, May 1996.

[Fitzpatrick98] Fitzpatrick, T., Blair, G.S., Coulson, G., Davies N., Robin, P., “A Software Architecture for Adaptive Distributed Multimedia Systems", IEE Proceedings on Software, Vol. 145, No. 5, pp 163-171, October 1998.

[Kiczales91] Kiczales, G., des Rivières, J., and Bobrow, D.G., “The Art of the Metaobject Protocol”, MIT Press, 1991.

[Ledoux97] Ledoux, T., “Implementing Proxy Objects in a Reflective ORB”, Proc. ECOOP’97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, June 1997.

[Manola93] Manola, F., "MetaObject Protocol Concepts for a "RISC" Object Model", Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, December 1993.

[Masuhara94] Masuhara, H., Matsuoka, S., Yonezawa, A., “An Object-Oriented Concurrent Reflective Language for Dynamic Resource Management in Highly Parallel Computing”, IPSJ SIG Notes, Vol. 94-PRG-18 (SWoPP'94), pp. 57-64, July 1994.

[Matsuoka91] Matsuoka, S., Watanabe, T., and Yonezawa, A., “Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming”, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP’91), Geneva, Switzerland, LNCS 512, pp 231-250, Springer-Verlag, 1991.

[McAffer96] McAffer, J., “Meta-Level Architecture Support for Distributed Objects”, In Proceedings of Reflection 96, G. Kiczales (ed), pp 39-62, San Francisco; Available from Dept of Information Science, Tokyo University, 1996.

[Mitchell98] Mitchell, S., Naguib, H., Coulouris, G., Kindberg, T., “Dynamically Reconfiguring Multimedia Components: A Model-based Approach”, Proc. 8th ACM SIGOPS European Workshop, Lisbon, Sep. 1998.

[Okamura92] Okamura, H., Ishikawa, Y., Tokoro, M., “AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework”, Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[Rao91] Rao, R., “Implementational Reflection in Silica”, Proceedings of ECOOP’91, Lecture Notes in Computer Science, P. America (Ed), pp 251-267, Springer-Verlag, 1991.

[ReTINA99] ReTINA, “Extended DPE Resource Control Framework Specifications”, ReTINA Deliverable AC048/D1.01xtn, ACTS Project AC048, January 1999.

[Schmidt97] Schmidt, D.C., Bector, R., Levine, D.L., Mungee, S., Parulkar, G., “Tao: A Middleware Framework for Real-time ORB Endsystems”, IEEE Workshop on Middleware for Real-time Systems and Services, San Francisco, Ca, December 1997.

[Singhai97] Singhai, A., Sane, A., Campbell, R., “Reflective ORBs: Supporting Robust, Time-critical Distribution”, Proc. ECOOP’97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems, Jyväskylä, Finland, June 1997.

[Smith82] Smith, B.C., “Procedural Reflection in Programming Languages”, PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982.

[Stroud95] Stroud, R.J., Wu, Z., “Using Metaobject Protocols to Implement Atomic Data Objects”, Proceedings of the European Conference on Object-Oriented Programming (ECOOP’95), pp 168-189, Aarhus, Denmark, August 1995.

[Vanegas98] Vanegas, R., Zinky, J., Loyall, J., Karr, D., Schantz, R., Bakken, D., “QuO’s Runtime Support for Quality of Service in Distributed Objects”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), pp 207-222, Springer, 1998.

[Watanabe88] Watanabe, T., Yonezawa, A., “Reflection in an Object-Oriented Concurrent Language”, In Proceedings of OOPSLA’88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of “Object-Oriented Concurrent Programming”, A. Yonezawa, M. Tokoro (eds), pp 45-70, MIT Press, 1987.

[Watters96] Watters, A., van Rossum, G., Ahlstrom, J., "Internet Programming with Python", Henry Holt (MIS/M&T Books), September 1996.

[Yokote92] Yokote, Y., “The Apertos Reflective Operating System: The Concept and Its Implementation”, In Proceedings of OOPSLA’92, ACM SIGPLAN Notices, Vol. 28, pp 414-434, ACM Press, 1992.

# A Formal Analysis of Smithsonian Computational Reflection

Inge M.C. Lemmens and Peter J. Braspenning

University Maastricht, P.O. Box 616, 6200MD Maastricht, The Netherlands
lemmens@cs.unimaas.nl,
WWW home page: http://www.cs.unimaas.nl/~lemmens/

Etymologically, to be reflective means: "to apply to oneself"; i.e., reflectivity occurs when the subject and the object of some action are roughly the same. The notion of reflection has attracted the attention of researchers in the field of Computer Science. Consequently, several theories of computational reflection have been developed. In a sense, all these theories come down to applying the symbolic processing paradigm to the inner workings of computational engines. As fascinating as the topic may be, the interest in it within Computer Science is not so constant, albeit rather persistent. We think that this situation is likely caused by the inherent cognitive complexity of the topic. It is our belief that the use of standard modelling methods w.r.t. the issue of computational reflection can assist in decreasing this cognitive complexity. Therefore, as a first step, we applied the Booch method to the theory of B.C. Smith, as described in [2]. This theory relies on the application of the *reflection hypothesis* and the use of *meta-circular* processes.

**Computational Processes**

The notion of *computational process* is the fundamental subject matter. According to the interpretive reduction model, a process $P$ consists of a structural field, $S_P$, and an ingredient process $P'$; i.e., the tupel $\langle S_P, P' \rangle$. The *structural field*, $S_P$, consists of data structures which specify a representation of the domain of process $P$. The *ingredient process* $P'$ involves the dynamics of the process. This dynamics can be achieved by running a *program* on a *processor*. Thus, the ingredient process $P'$ can be further decomposed into a program $S_{P'}$ and a processor $P$" (the tupel $\langle S_{P'}, P" \rangle$). The program of $P'$, $S_{P'}$, is a particular kind of structural field, consisting of symbol structures that represent the intended virtual machine instructions whose processing by $P$" engenders the process $P'$ (see figure 1, in the middle).

**Reflection Hypothesis Applied**

Smith formulated a reflection hypothesis which in essence states that it is possible for a process to reason about itself, in the *same way* that a process can reason about some part of the external world ([2]).

The first part of the theoretical account relies on the development of a process $Q$ which reasons about a process $P'$. This implies that the structural field of $Q$, i.e., $S_Q$ should contain a representation of the objects and events in $P'$ (see left side of figure 1).

In the *reflective process* we not only want to target something that reasons about itself, but we want something that is also affected by its ability to reason

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 135-137, 1999

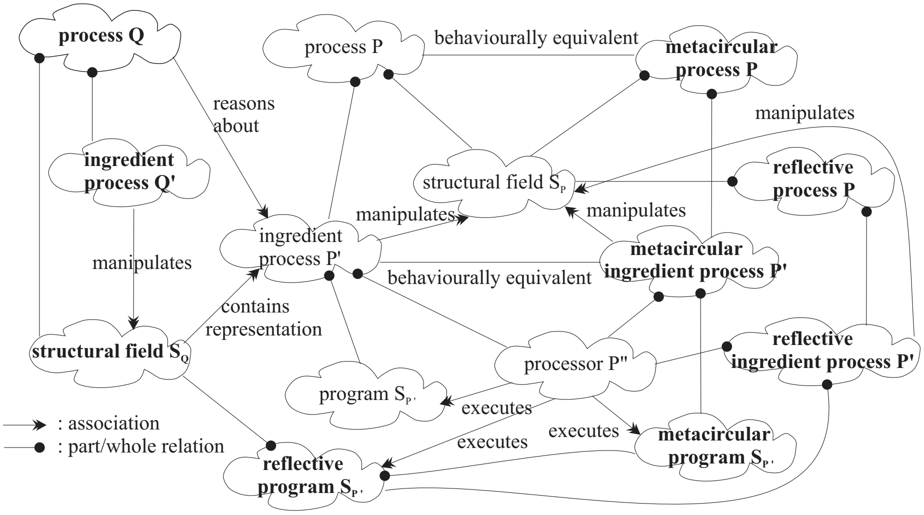


**Fig. 1.** The Booch method applied.

about itself. Therefore, in the end, we require the structural field $S_Q$ to be *causally connected* to $P'$. This means that $Q$ not only reasons about $P'$, but that the results of the reasoning process have a direct impact on the process $P'$.

Note, that this process of "reasoning about" can be iterated: we could construct a third process, say $R$, which reasons about $Q'$ in the same way that $Q$ reasons about $P'$. With every iteration we are, in fact, shifting what is considered to be the *theory* w.r.t. which the reasoning process takes place. For that reason, the relationship of "reasoning about" is *not* transitive.

**Meta-Circular Processes**

The second part of the theoretical account is based on the use of meta-circular processes. In any computational formalism able to model its own syntax and structures, it is possible to construct what are commonly known as *meta-circular processes*: if *programs* are accessible as first class symbolic structural fields, it is possible to construct meta-circular processes. The term *circular* indicates that the meta-circular process does not constitute a definition of the process: they have to be executed by the underlying process in order to yield any sort of behaviour (since they are programs and thus static, not dynamic) and the behaviour they would thereby engender can only be known if one knows beforehand what the process does (since it refers to the process) (see figure 2).

If one would consider processes to be functions from structure onto behaviour, then one should be able to prove that $P"(MCP_{P'}) \cong P'$, where $\cong$ means *behaviourally* equivalent. Thus, if an entire computation is mediated by the explicit processing of the meta-circular process, then the result will be the same as if the entire computation had been carried out directly. A meta-circular process is *not* causally connected to the process to which it refers. The interpretive reduction model allows the construction of a tower of meta-circular processes.

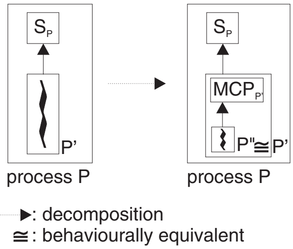


**Fig. 2.** A meta-circular process.

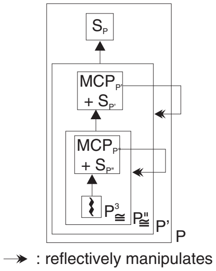


**Fig. 3.** A reflective process.

### Reflective Processes

The tower of meta-circular processes is not yet reflective since the causal connection link is lacking. One can, however, combine the theory of meta-circular processes with the application of the reflection hypothesis. We have shown that in order for a process $Q$ to reason about a process $P'$, the structural field $S_Q$ of $Q$ should contain a representation of the relevant facts of $P'$. Suppose now that we incorporate within the meta-circular process $MCP_{P'}$ the structural field which contains a representation of the relevant facts of $P'$ (a representation like in $S_Q$). Then this incorporation gives rise to a reflective process since the causal connection requirement is now fulfilled. This can be explained as follows: the underlying process, $P$", manipulates the program, $MCP_{P'} + S_{P'}$, which may cause a change in $S_{P'}$. This change in $S_{P'}$ will, due to the circularity, change the process $P'$. Furthermore, $P"(MCP_{P'}) \cong P'$ and $P"(MCP_{P'} + S_{P'}) \cong P'$. This implies that a change of $P'$ leads to a change of $P"(MCP_{P'} + S_{P'})$, and in particular in a change of $S_{P'}$.

### Future Research

Research involved the comparison of different theories of computational reflection (see [1]). Future research will concentrate on the application of standard modelling techniques on the theories. It is our belief that some interesting differences will become apparent. This may lead to two possible research directions: the development of a unified theory, or the development of parallel theories of computational reflection.

## References

1. Lemmens, I.M.C., Braspenning, P.J.: Computational Reflection: Different Theories Compared. Report CS99-02, University Maastricht (1999).
2. Smith, B.C.: Reflection and Semantics in Lisp. MIT Technical Report MIT/LCS/TRT-272, Massachusetts Institute of Technology (1982).

# Reflection for Dynamic Adaptability: A Linguistic Approach using LEAD++

Noriki Amano and Takuo Watanabe

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-1292, Japan
Email: {n-amano, takuo}@jaist.ac.jp

Nowadays open-ended distributed systems and mobile computing systems have come into wide use. In such systems, we cannot obtain accurate information of dynamically changing runtime environments beforehand. The changes of runtime environments have given a strong influence on the execution of programs, which we cannot ignore. Thus, the software systems that can adapt themselves to dynamically changing runtime environments are required. We call such software systems *dynamically adaptable software systems.*

In this work, we propose a software model called *DAS* [1] and its description languages *LEAD++* for dynamically adaptable software systems. The DAS model has the mechanism to adapt software systems to dynamically changing runtime environments. We had designed & implemented the language *LEAD* [1] [2] based on the DAS model. We are currently working on object-oriented reflective language LEAD++. Its prototype is a pre-processor of Java. By using them, we can systematically describe dynamically adaptable software systems.

To realize dynamically adaptable software systems, it is an effective way to change the behaviors of each software system depending on the states of runtime environments. However, it is not practical to develop several versions of the same software system depending on each runtime environment and/or its states. Moreover, such behaviors of each software system depending on the states are related with any other parts of it. Thus, it is difficult to control such behaviors of each software system from its outside. From such reasons, each software system should have the ability that can adapt itself to dynamically changing runtime environments. We call such ability of software systems *dynamic adaptability.* The dynamically adaptable software systems (namely, software systems with dynamic adaptability) not only adapt themselves to dynamically changing runtime environments, but also change their own functionalities flexibly to make full use of the properties in the runtime environments.

However, there is a limitation on runtime environments and their states that software engineers can anticipate beforehand. Thus, there is also a limitation on the dynamic adaptability that the software engineers can give to software systems beforehand. From the reason, the mechanism of dynamic adaptability must be extensible. Namely, the mechanism must be able to change depending on various runtime environments and their states afterward.

[1] LEAD is based on the restricted DAS model and its prototype is an extension of Scheme.

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 138-140, 1999

We propose a software model with dynamic adaptability in extensible way called DAS. The following are main concepts of the DAS model.

- The basic components of adaptable behaviors in a software system are procedures, and the control of adaptable behaviors is based on procedure's invocation.
- The DAS model has a variant of generic procedures (functions) based on the states of runtime environments called *adaptable procedures*.
- The control mechanism of adaptable procedures (called *adaptation mechanism*) is realized by using reflection.
- A software system based on the DAS model forms a meta-level architecture.

These concepts are based on our following objectives.

- High compatibility for existing software systems.
- High flexibility of dynamic adaptability.
- High readability, maintainability and extensibility.

We have a great many requirements that we want to use many existing software systems effectively in open-ended distributed systems and mobile computing systems. Therefore, we require the methodology that we can introduce dynamic adaptability into software systems in existing languages. In the DAS model, adaptable behaviors in a software system are realized as *adaptable procedures*. An adaptable procedure consists of several *adaptable methods* that have runtime codes depending on the states of runtime environments. When an adaptable procedure is invoked, a well-suited adaptable method is selected depending on the states, and its runtime codes are executed. This approach can be applied to many existing languages that have the procedure invocation's mechanism in principle. Consequently, we have a great potentiality for effective utilization of existing software systems. Actually, we had applied the DAS model to two different languages Scheme and Java, and had designed & implemented the languages LEAD (based on Scheme) & LEAD++ (based on Java). Moreover, we had implemented a dynamically adaptable workflow system in LEAD++ by using an existing mobile code system in Java.

In our approach, to realize dynamic adaptability of software systems in flexible way, the adaptation mechanism must be able to change depending on the states of runtime environments. In the DAS model, the mechanism itself is realized using an adaptable procedure in reflective way. Therefore, it can adapt itself to the states (namely, it can change its own behaviors depending on the states). By changing the mechanism depending on the states, we can realize adaptable behaviors of software systems in flexible way such as the temporary restrictions & releases of their functionalities without modifying their original programs. Moreover, since the mechanism is realized by using an adaptable procedure, its customization is also realized by adding adaptable methods incrementally without modifying its structure.

A dynamically adaptable software system based on the DAS model forms a meta-level architecture. The descriptions of the mechanism that controls adaptable behaviors in the software system (meta-level) are completely separated from the descriptions of the primary subject domain in it (base-level). The descriptions of the control mechanism are hidden into the meta-level. This separation of concerns improves the readability, maintainability and extensibility of dynamically adaptable software systems.

Toward several related works, the following are features of the DAS model.

- It is a generic model that is independent of specific paradigms & description languages (such as object-oriented paradigm & languages, etc.).
- The adaptation mechanism can adapt itself to dynamically changing runtime environments.
- The adaptation mechanism (in other words, adaptation strategy) can be modified/extended by adding adaptable methods incrementally without modifying the structure.
- It provides asynchronous events mechanism (event object) depending on the abstractions of the virtual & local runtime environments.

Moreover, although existing reflective languages have potential for realizing dynamically adaptable software systems, their design is mainly aimed for language (or runtime) extension. On the contrary, the DAS model aims to realize dynamic adaptability of software systems using reflection. In the DAS model, reflection is used to realize the adaptation mechanism. Thus, the mechanism itself can also realize adaptable behaviors depending on the states of runtime environments. As the result, we can realize dynamic adaptability of software systems in flexible way.

However, there are also disadvantages of the DAS model. For example, when we develop dynamically adaptable software systems based on the DAS model in practice by using LEAD++, the following problems occur.

- When we describe adaptable procedures, we must consider the adaptation strategies for them.
- We must pay attention to the alteration of adaptation strategy sufficiently.
- It is difficult to apply the DAS model to software systems that require dynamic adaptability for the real time processing.

As the future works on this research, there are the following issues.

- Evaluation of dynamically adaptable software systems.
- Verification of dynamically adaptable software systems.
- Introducing concurrency & synchronization into the DAS model.

## References

1. N. Amano and T. Watanabe. A Procedural Model of Dynamic Adaptability and its Description Language. In *Proceedings of the ICSE'98 International Workshop on Principles of Software Evolution*, pages 103–107, April 1998.
2. N. Amano and T. Watanabe. LEAD: Linguistic Approach to Dynamic Adaptability for Practical Applications. In *Proceedings of the System Implementation 2000*, pages 277–290. International Federation for Information Processing (IFIP), Chapman & Hall, February 1998.

# Networking and reflection: a strong combination

Frank Matthijs, Peter Kenens, Wouter Joosen, and Pierre Verbaeten

Distrinet - Computer Science Department
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven BELGIUM
Frank.Matthijs@cs.kuleuven.ac.be

**Abstract.** Current meta-levels for distribution tend to ignore the networking aspect of distribution. The solution we propose is to build those meta-levels using a flexible protocol stack, which can be customised by specialised programmers. We present a framework with which such a protocol stack can be built, and briefly discuss some trade-offs and issues that have to be solved to make this approach feasible as a main-stream solution.

## 1 Introduction

Modern distributed systems run on a variety of hosts, from huge servers to credit card sized smart cards, with object invocations traveling over various communication technologies. Therefore, the need to control how those invocations are sent over the network becomes more and more apparent. Current meta-levels for distribution tend to ignore this aspect of distribution. The solution we propose is to construct a meta-level controlling the "execution" of base-level object invocations using a flexible protocol stack. Reflection can be used to dynamically reconfigure the protocol stack.

## 2 Protocol stacks at the meta-level

In the rest of this text, we assume we have available an object-oriented system with a meta-level architecture, where object invocations are reified and available at the meta-level. In our case, the meta-level for distribution will consist of protocol stack objects.

Such a meta-level can for example be built using our protocol stack framework [1]. The framework uses *components* as its basic building blocks. Each component is responsible for exactly one function in the stack, like fragmentation, encryption, flow control, etc. Components may be grouped in layers. Unlike in many other flexible protocol stack systems (e.g., microprotocols in [3]), the framework decouples header handling from protocol stack composition: any group of components (including layers but this is no requirement) can have its own header and a component does not necessarily have a header. Multiplexing is also decoupled from the other concepts, which avoids multiplexing problems [12] while

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 141-143, 1999

still retaining layers as a unit of composition. Architecture wise, components are connected by *connectors*. Using different kinds of connectors, programmers can create their own protocol stack architecture, such as common layered stacks, or more horizontal configurations (like [10][11]). Network packets are explicitly passed between components as objects, according to the connectors between the components. Components can only communicate implicitly, through *meta-information* attached to packets. As a result, components can be added and removed dynamically, since there are no direct dependencies on other components.

One connector which deserves more attention is the *ReflectionPoint*. A ReflectionPoint observes the Packet objects flowing through and can decide to alter the destination component dynamically. As an illustration of what reflection points can do, consider an object oriented web server application. All object invocations flow through our meta-level for distribution. A reflection point in the metalevel could notice, by observing the object identifiers attached to the packets flowing through, that certain invocations tend to contain large amounts of data (probably invocations of graphics objects), and fine tune some parameters of the transport protocol or switch to a different transport protocol altogether (one which is optimized for bulk-data transfers) for these invocations. Somewhere else in the meta-level, a different reflection point could decide to apply encryption to certain invocations (for example only those which retrieve the contents of entry field objects in a secure page).

Note that this kind of dynamic behaviour on an invocation basis could also be achieved by setting up a separate, custom communication session for each object invocation. However, this approach prevents optimisations such as maintaining a communication channel pool which reuses open channels. Moreover, one would have to use a different endpoint identifier (e.g., port number) for each custom channel. This may lead to an exponential explosion of endpoint identifiers [10]. Our simple example would already require 4 different endpoint identifiers.

## 3 Discussion

ReflectionPoints introduce limited reflective capabilities in the protocol stack. They are very useful for normal stack reconfigurations. More sophisticated reconfigurations are probably handled better if the interactions between components are themselves reified and handled at the meta-meta-level. A potential disadvantage of this, however, is decreased performance. Using explicit reflection points as connectors between components allows specialised programmers to introduce them only where necessary (selective "inlining" of reflective parts).

Meta-levels for distribution like we propose here can also be constructed using other flexible protocol stack systems, such as Ensemble [6], Horus [5], x-kernel [3] / Morpheus [4], Scout [7], Plexus [2], DaCaPo [9], FCSS [8], Tau [10], HOPS [11], etc. Unlike our framework, these systems are in general not designed to form a meta-space. In our opinion, the most important requirement in this regard is the ability to convey control information between components, allowing application specific information to flow through the stack, and making specialised behaviour per invocation possible. This poses problems in many of these systems, certainly if we extend the requirement to the complete protocol stack, not just the upper layers (transport and above). Some systems, such as HOPS and the x-kernel can only be configured at bind time. Going meta-meta is also not trivial with many of these protocol stack systems.

Since applying flexible protocol stacks in a meta-level for distribution is not yet common practice (to say the least), many issues have to be studied in more detail. We believe that the protocol stack framework presented here, by decoupling concepts such as components, layers, multiplexing and encapsulation, and by its ability to attach any kind of meta-information to packets, is well suited as a basis for experimentation not only with protocol stack architecture and functionality, but also with dynamic reconfiguration, stability issues when reconfiguring the meta-level at run-time, usefulness of meta-meta levels, etc.

## References

1. Frank Matthijs, Pierre Verbaeten. The Distrinet Protocol Stack framework. CW Report 276. 1999.
2. Marc Fiuczynski, Brian Bershad. "An Extensible Protocol Architecture for Application-Specific Networking." In Proceedings of the 1996 Winter USENIX Conference, 55-64. January 1996.
3. N. C. Hutchinson and L. L. Peterson. "The x-Kernel: An architecture for implementing network protocols." In IEEE Transactions on Software Engineering, 17(1), 64-76, Jan. 1991.
4. M. B. Abbott and L. L. Peterson. "A language-based approach to protocol implementation." In IEEE/ACM Transactions on Networking, 1(1), 4-19. Feb. 1993.
5. Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. "A Framework for Protocol Composition in Horus." In Proceedings of Principles of Distributed Computing. August 1995.
6. M. Hayden."The Ensemble System." PhD Dissertation, Dept. of Computer Science, Cornell University, USA. Hayton, 1997.
7. A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting. "Scout: a communications-oriented operating system." In Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, 1995.
8. M. Zitterbart, B. Stiller and A. Tantawy. "A Model for Flexible High-Performance Communication Subsystems." In IEEE Journal on Selected Areas in Communications 11(4), 507-518, 1993.
9. T. Plagemann, B. Plattner, M. Vogt and T. Walter. "Modules as Building Blocks for Protocol Configuration." In Proceedings of the IEEE International Conference on Network Protocols, San Francisco, 106-113, 1993.
10. R. Clayton, K. Calvert. "A Reactive Implementation of the Tau Protocol Composition Mechanism." In Proceedings of the First IEEE Conference on Open Architectures and Network Programming, San Francisco, California.
11. Z. Haas. "A protocol structure for high-speed communication over broadband ISDN." In IEEE Network 5(1), 64-70, January 1991.
12. D. L. Tennenhouse, "Layered Multiplexing Considered Harmful." In Protocols for High-Speed Networks, Rudin and Williamson (Ed.), North Holland, Amsterdam, 1989.

# Towards Systematic Synthesis of Reflective Middleware

*Petr Tůma, Valerie Issarny, Apostolos Zarras*

*INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France*
{tuma, issarny, zarras}@irisa.fr
http://www.irisa.fr/solidor/work/aster.html

**Abstract.** In this paper, we present a method for systematic synthesis of middleware based on the meta-level requirements of the application that stands on top of it. Particular attention is paid to the ability to accommodate evolving requirements of a running application.

## 1 Introduction

Following the definition of programming languages supporting reflection [3], we say that a middleware supports *reflection* if the application that lies on top of it is provided with means for reasoning about the middleware properties and for customizing the properties as necessary. Many middleware infrastructures, such as FlexiNet and reflective Orbs, expose functionality in a way that can be used to customize the properties. This process is called *reification* [2] of the middleware functionality. Reification alone, however, does not equal reflection. What is missing are the means for reasoning about the properties of the middleware. The reasoning itself is left to the application designer, who is free to make unconstrained changes to the reified functionality and has to deduce the impact of the changes on the middleware properties himself. We remedy this drawback by designing a method for the systematic synthesis of middleware based on the application requirements, detailed in [4] and outlined in Section 2. Here, we concentrate more on how to use the systematic synthesis to accommodate changes in the requirements of a running application, as detailed in Section 3.

## 2 Systematic Middleware Synthesis

The systematic synthesis of middleware uses an architectural description of the application consisting of two views. The *structural view* describes interconnection of the individual components of the application; the *property view* describes meta-level properties of the middleware that implements the interconnections. The meta-level properties are described formally using temporal logic, a theorem prover is used to check the relationship between properties.

The available middleware services are stored in a middleware repository together with descriptions of the properties they provide. Based on the specification of requirements on the middleware properties, the appropriate middleware

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 144-146, 1999

services are retrieved from the repository, assembled into a middleware and incorporated into the application [4].

## 3 Dynamic Reflection

Here, we extend the systematic synthesis of middleware with the ability to build middleware that can be dynamically exchanged so as to reflect changes in the requirements of a running application. The problem of exchanging the middleware is twofold. Immediately visible are the technical issues related to dynamic loading and unloading of the middleware code, where many solutions have been proposed. Less visible is the issue of what impact the dynamic change of the middleware code has on the properties provided by the middleware.

At the time a new middleware $M'$ is to replace the old middleware $M$, there might be requests issued through $M$ for which the application requirements are not satisfied yet. Examples of these requests include an unfinished remote procedure call when reliable delivery is required, or an open distributed transaction when some of the ACID transaction properties are required. The new middleware $M'$ must guarantee that the requirements $R(req)$ valid at the time these *pending requests* were issued will be satisfied. During system execution, the information necessary to complete pending requests is a part of the middleware state. The new middleware should start from an initial state that contains this information and should be able to satisfy requirements related to the pending requests. Given a specific mapping $Map(\sigma)$ between the states of the old and the new middleware, we define a *safe state* $\sigma_M$ in which it is possible to perform the exchange while satisfying the requirements:

$$\begin{aligned}&SafeState(\sigma_M, M, M', Map) \equiv\\&\quad [\sigma_M] \wedge \forall req \mid Pending_M(req) : [Map(\sigma_M)] \Rightarrow R(req)\end{aligned}$$

A safe state is defined in relation to the ability to map the state from the old to the new middleware. When no state mapping is available, we refine the $SafeState$ predicate into a stronger criterion, $IdleState$, defined only with respect to the old middleware properties. In an *idle state* [1], no requests are pending and hence no state needs to be mapped:

$$IdleState(\sigma_M, M) \equiv [\sigma_M] \wedge (\forall req : \neg Pending_M(req))$$

Based on a straightforward utilization of the safe state definition, the general strategy of the exchange is to reach the safe state of the middleware, block incoming requests to remain in the safe state during update, exchange the middleware implementation, and unblock incoming requests. To detect whether the middleware reached a safe state, an idle state detection code is incorporated into it at the time it is being synthesized. The definition of the idle state makes this code independent of the middleware implementation; it is retrieved from the repository based on the middleware properties only. During exchange, this code is used to determine when it is safe to perform it; alternatively, a safe state detection code specific to the particular update can be installed.

It is generally not guaranteed that the middleware will reach a safe state within finite time during normal execution. We therefore selectively block requests from the application, so as to prevent activities that do not participate in driving the middleware into a safe state from issuing requests that would keep the middleware away from the safe state. It can be shown that the decision whether to block a request depends only on the safe state used during the update. This decision is therefore taken by the safe state detectors associated with the update.

For the purpose of the exchange, the middleware is separated into a static and a dynamic part. The static part of the middleware contains the proxy through which the application accesses the middleware, and the code for blocking requests. The dynamic part contains the safe state detection code and the middleware itself, both retrieved from the repository based on the required properties. The exchange is directed by a coordinator component, responsible for exchange within the scope of the changed property. When requested to perform an exchange and given the new code of the dynamic middleware parts, the coordinator instructs the static middleware parts to block the requests as described above. After a safe state is reached, the coordinator directs the middleware to unload the existing dynamic parts and install the new code.

## 4 Conclusion

The approach presented in this paper tackles the problem of changing middleware properties in a running application. Its advantage is in synthesizing the middleware systematically based on the required properties, as opposed to only exposing the functionality through reification. The basic concepts of the approach were prototyped using several CORBA platforms and the STeP theorem prover. The current work focuses on improvements related to granularity and timing of the middleware exchange, and on using the approach to synthesize adaptive middleware.

*Acknowledgement:* this work has been partially supported the C3DS LTR Esprit project.

## References

1. J. Kramer and J. Magee. The Evolving Philosophers Problem. *IEEE Transactions on Software Engineering*, 15(1):1293–1306, November 1990.
2. J. Malenfant, M. Jacques, and F.N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of REFLECTION '96*. ECOOP, 1996.
3. B. C. Smith. *Procedural Reflection in Programming Languages.* PhD thesis, MIT, 1982. Available as MIT Techical Report 272.
4. A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of* MIDDLEWARE *'98*, pages 257–272. IFIP, Sept 1998.

# An Automatic Aspect Weaver with a Reflective Programming Language

Renaud Pawlak, Laurence Duchien, and Gérard Florin
{pawlak, duchien, florin}@cnam.fr

Laboratoire CEDRIC-CNAM 292 rue St Martin, Fr 75141 Paris Cedex 03

**Abstract.** This short paper presents A-TOS (Aspect-TOS (TCL Object System)), an aspect-oriented framework that allows the programmer to define its own specialized aspects and to weave or remove them at runtime. Since A-TOS is based on a configurable and semantic-reliable object-wrapping technique, it is able to solve what we call the Aspect-Composition Issue (ACI), i.e., to detect and solve many semantic conflicts when weaving the different aspects together.

***Keywords:*** *Aspect-Oriented Programming, Weaver, Wrapping.*

## Introduction

Recent approaches like the Aspect-Oriented Programming approach [KLM$^+$97] consist in finding an accurate and consistent mean to achieve separation of concerns within complex application programs.

However, since it does not exist any general and semantic reliable solution when aspects are composed together, existing weavers are *ad hoc* and fixed solutions to a given problem.

In this short paper, we present A-TOS (for Aspect-Tcl Object System), a general purpose Aspect-Oriented Framework that we have implemented with TOS [Paw99], a class-based reflective language. A-TOS uses wrappers [BFJR98] to ensure a sequential and ordered composition of the aspect codes by the weaver and allows the meta-programmer to easily add or remove any aspect in a controlled and semantic reliable way.

The first section presents the Aspect-Composition Issue and proposes an aspect-composition model based on wrapping. Section 2 briefly describes how the ACI is automatically solved in A-TOS (by ordering the wrappers).

## 1 A Wrapper-Based Composition Model

When programming with aspects, an application programmer can deal with different parts of the problem within different programs. However, none of the proposed solutions allows the programmer to easily add or remove aspects in a secure and automatic way — we say that a weaver is automatic when the

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 147-149, 1999

programmer does not need to explicitly extend the weaver capabilities when a new aspect is added. For instance, D [LK97] fixes the number of aspects (two) that can be used to program an application. The main reason for this limitation comes from true difficulties to reliably control the resulting program semantics when a new aspect is added — we call this problem the *Aspect-Composition Issue* (ACI). For example, if the composition of a synchronization and an authentication aspects makes all the authentication processes to be synchronized, it can have tragic effects on the system performances.

Since it may be very difficult to solve the ACI in a general case, A-TOS proposes a wrapping-based system where the aspect composition is realized by some *WRAPS* links (see figure 1). In this system, wrappers are sequentially called in a given order so that the ACI is reduced to an ordering problem.

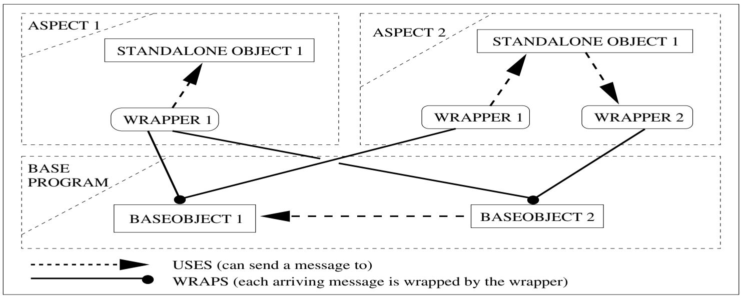


**Fig. 1.** An overview of an aspect system based on wrapping.

## 2 From Composition to Automatic Composition

Since our composition scheme consists in sequentially calling the wrappers, solving the object-level ACI (as defined in 1) consists in finding the good wrappers order to realize the required semantics. Many criteria may influence the way wrappers are composed with each other. However, those criteria can be reduced to four main wrapper properties:
- the wrapper does not always call the base object — it is a **conditional** wrapper,
- it always has to be called — it is a **mandatory** wrapper,
- it has to be called only if the base object is effectively called — it is an **exclusive** wrapper,
- it sends messages to other objects — it is a **modificator** wrapper.

Those properties can be composed[1] and finally give five different classes of wrappers: conditional wrappers ($CW$), mandatory w. ($MW$), exclusive w. ($EW$), mandatory conditional w. ($MCW$), and exclusive conditional w. ($ECW$).

[1] Mandatory and exclusive properties cannot be composed. Moreover, for simplification sake, we assume that the modificator property is not relevant for ordering the wrappers.

When knowing the wrapper classes, we can deduce ordering constraints when composing wrappers. For instance, a mandatory wrapper always has to be placed *before* a conditional wrapper to always respect the mandatory wrapper definition (i.e., it is always called).

Finally, to resolve the wrapper list order when adding a wrapper to a base object, A-TOS adds a new class-level property that represents the wrapper class and that must be filled by the wrapper programmer. When composing aspects, A-TOS respects the following ordering constraint that is a garantee for a correct resulted semantics ($C1 > C2$ means that class *C1* wrappers must always be called before class *C2* ones): `MW>MCW>CM>ECW>EW.`

## Conclusion

In this paper, we propose A-TOS, a generic and automatic aspect-composition framework based on a simple wrapper classification. This classification determines automatic and semantic-reliable composition rules (by ordering the wrappers) and partially solves the Aspect-Composition Issue (ACI). This solution, by forcing the programmer to think about the objects semantics when programming an aspect, can be considered as a tradeoff between a fully automatic solution (where all the conflicts would be solved by the weaver) and with a handmade solution like D.

In the future, we will also address some difficult points. Especialy, since we assumed that all the modificators modifications are commutative (see section 2), we must now provide some support when it is not the case. Besides, this framework points out that, in an open environment, some aspect administration has to be done to prevent redundancies and semantics conflicts.

## References

[BFJR98] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *Prodeedings of ECOOP'98*, 1998.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *SIGPLAN Notices*, pages 317–326. ACM Press, October 1989.

[KdRB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[LK97] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, 1997.

[Paw99] R. Pawlak. Tos: a class-based reflective language on tcl. Technical Report 9902, Laboratoire CEDRIC-CNAM, January 1999.

# Using Compile-Time Reflection for Objects'State Capture[1]

Marc-Olivier Killijian, Juan-Carlos Ruiz-Garcia, Jean-Charles Fabre

LAAS-CNRS, 7 Avenue du Colonel Roche
31077 Toulouse cedex, France
{killijian, fabre, ruiz}@laas.fr

## 1 Motivations

Checkpointing is a major issue in the design and the implementation of dependable systems, especially for building fault tolerance strategies. Checkpointing a distributed application involves complex algorithms to ensure the consistency of the distributed recovery state of the application. All these algorithms make the assumption that the internal state of the application objects can be obtained easily. However, this is a strong assumption and, in practice, it is not so easy when complex active objects are considered. This is the problem we focus on in our current work. The available solutions to this problem either rely on :

- a specific hardware/operating-system/middleware that can provide runtime information about the mapping used between memory and application objects;
- a reflective runtime which provides access to class information, such as Java Serialization [1] based on the Java Reflection API;
- intervention at the language level, rewriting the original code to add checkpointing mechanisms (e.g. Porch [2] for C programs);
- the provision of state handling mechanisms by the user; since the implementation of these mechanisms must be error-free, this is a weak solution.

When neither the underlying system nor the language runtime is sufficiently reflective to provide type information about the object classes, the only solution is to obtain this information at the language level. Two approaches can be investigated here: (i) developing a specific compiler (as Porch does) or (ii) using a reflective compiler such as OpenC++ [3] or OpenJava [4]. Since diving into a compiler is very complex and can lead to the introduction of new software faults, we think that the latter is a better solution. Clearly, compile-time reflection opens up the compilation process without impacting the compiler itself. Off-the-shelf compilers can thus be used to produce the runtime entities.

[1] This work has been partially supported by the European Esprit Project n° 20072, DEVA, by a contract with FRANCE TELECOM (ref. ST.CNET/DTL/ASR/97049/DT) and by a grant from CNRS (National Center for Scientific Research in France) in the framework of international agreements between CNRS and JSPS (Japan Society for the Promotion of Science).

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 150-152, 1999

## 2 Approach Overview

To provide state handling facilities to target objects, we have to implement two new methods for each class: `saveState` and `restoreState`. The former is responsible for saving the state of an object and the latter for restoring this state; these methods are similar to the `readObject` and `writeObject` methods provided by the Java runtime.

These methods must save/restore the values of each attribute: basic and structured types of the language, arrays, and inherited fields. Furthermore, they must be able to traverse object hierarchies, i.e. save/restore the objects included in the object currently processed. To generate these methods we use OpenC++, an open-compiler for C++ which provides a nice API for both introspection and intercession of application classes. Thanks to the static type information it provides, we are able to analyse the structure of the classes and can thus generate the necessary methods for checkpointing.

However, since C++ is an hybrid between object-oriented and procedural languages, some of its features are very difficult to handle. Features which break the encapsulation principle must be avoided from a dependability viewpoint: friend classes or functions and the pointer model of C++. We decided to handle only a single level of indirection for pointers and to forbid the use of pointer arithmetic; pointers can thus be seen as simple object references as in Java.

Filtering programs and rejecting those that don't respect these restrictions is easy to implement using compile-time reflection. Enforcing other programming restrictions is also useful for validation aspects. Dependability can be significantly enhanced by restricting to a subset of a language [5] in many respects.

It is worth noting that this approach does not deal with multithreaded objects. Indeed, compile-time reflection is not sufficient to save the state of threads, thread introspection is needed [6].

## 3 Current Implementation Status

We have implemented this approach using OpenC++ 2.5.1 and GCC 2.8.1 on both Solaris and Linux operating systems. Firstly the application program is parsed by the metaclasses we designed, these metaclasses generate the necessary facilities (save/restore state methods and related state buffer classes). Secondly, the C++ compiler compiles the instrumented code.

We are currently working on the validation of this toolkit. We are using some application benchmarks in order to evaluate the correctness of the states obtained (the coverage of the class' structure analysis), and to evaluate the efficiency of the state capture/restoration compared to Java serialization and Porch checkpointing. The first efficiency results obtained on simple examples are promising.

We have also implemented an optimisation of this approach that saves only the modified attributes of an object. This technique uses runtime reflection in order to know which of the object's members have been modified since the last checkpoint.

The idea is to save/restore a delta-state instead of the full object state. Since an object's methods often modify only a small subset of its attributes, this approach is more efficient in practice. It is worth noting that this new technique can only be applied for checkpointing. Cloning an object requires its full state to be available.

## 4 Conclusion

The work briefly describe here is part of a reflective architecture for dependable CORBA applications [7] which implements fault-tolerance mechanisms as metaobjects on any off-the-shelf ORB. The metaobject protocol controls both object interaction and object state. It has been developed for C++ objects and is currently being ported for Java objects using OpenJava. The implementation relies on a combined use of tools and techniques, such as open compiler, IDL compiler and reflective runtime when available.

**Acknowledgements.** We would like to thank Shigeru Chiba and Michiaki Tatsubori for their contribution to this work and invaluable assistance in using OpenC++.

## References

[1] Sun Microsystems, "Java Object Serialization Specification".

[2] V. Strumpen and B. Ramkumar , "Portable Checkpointing for Heterogeneous Architectures," in *Fault-Tolerant Parallel and Distributed Systems*, D. Avresky, R. and D. Keli, R., Eds.: Kluwer Academic Press, pp. 73-92, 1998.

[3] S. Chiba, "A Metaobject Protocol for C++," presented at OOPSLA, Austin, Texas, USA, pp. 285-299, 1995.

[4] M. Tatsubori, "An Extensible Mechanism for the Java Language", Master of Engineering Dissertation, Graduate School of Engineering, University of Tsukuba , University of Tsukuba, Ibaraki, Japan, Feb 2, 1999.

[5] P. J. Plauger, "Embedded C++", appeared in C/C++ Users Journal, vol.15, issue 2, February 1997.

[6] M. Kasbekar, C. Narayanan, and C. R. Dar, "Using Reflection for Checkpointing Concurrent Object Oriented Programs" Center for Computational Physics, University of Tsukuba, UTCCP 98-4, ISSN 1344-3135, October 1998.

[7] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-Garcia, and S. Chiba, "A Metaobject Protocol for Fault-Tolerant CORBA Applications," presented at IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA, pp. 127-134, 1998.

# Invited Talk 3
# Past, Present, and Future of Aperios

*Yasuhiko Yokote*
*Sony Corporation*
*PSD Center, ESDL (3G5F)*
*6-7-35 Kitashinagawa, Shinagawa-ku,*
*Tokyo 141 JAPAN*
*E-mail : ykt@arch.sony.co.jp*

Aperios is a software platform for Networked Consumer Devices such as STB, Digital TV, and Home Server. It has originally developed at Sony Computer Science Laboratory, and transferred to Sony Corporation as an operating system for an Interactive Videocommunication System in 1995. Since then, Aperios has been evolved to be used for software platform of consumer devices development and the first Aperios product has introduced in the Japan market on May 1999.

Firstly, some technical aspects of Aperios are presented in terms of reflection in an operating system. Then, our experience of Aperios introduction to the product development is discussed: How have we introduced a new idea, reflection, to engineers?; How has reflection contributed to the product?; What is the most difficulty in that process?; etc. Also, our policy of the Aperios release and some business related topics are presented. In conclusion, the next step (or direction) of Aperios is briefly described.

*Yasuhiko Yokote got his PhD in Electrical Engineering from Keio University in 1988. From 1988 to 1996, he was a resarcher at the Sony Computer Science Laboratory in Tokyo. Then he joins Sony Corporation. His interests include object-oriented operating systems, programming languages, distributed computing, and embedded systems. He worked on the design and implementation of well known object-oriented systems including ConcurrentSmalltalk, the Muse Operating System, the Apertos Operating System, and Aperios. He is a member of ACM, IEEE Computer Society, and JSSST.*

P. Cointe (Ed.): Reflection'99, LNCS 1616, p. 153, 1999

# Reflecting Java into Scheme

Kenneth R. Anderson[1], Timothy J. Hickey[2]

[1]BBN Technologies, Cambridge, MA, USA KAnderson@bbn.com,
[2]Brandeis University, Waltham, MA, USA tim@cs.brandeis.edu

**Abstract.** We describe our experience with SILK, a Scheme dialect written in Java. SILK grazes symbiotically on Java's reflective layer, enriching itself with Java's classes and their behavior. This is done with three procedures. `(constructor)` and `(method)` provide access to a specific Java constructor or method respectively. `(import)` allows the entire behavior of a class to be imported easily. `(import)` converts Java methods into generic functions that take methods written in Java or Scheme. In return, SILK provides Java applications with an interactive development and debugging environment that can be used to develop graphical applications from the convenience of your web browser. Also, because SILK has introspective access into Java, it can also be used for compile time metaobject scripting. For example, it can generate a new class using an old one as a template.

## 1 Introduction

Java's reflective layer provides access to metaobjects that reflect primitive, class, array, field, constructor, and method objects. There are obvious limitations to its reflective capabilities. For example, one can only affect the behavior of the running program by invoking methods and accessing fields of objects. One cannot define or redefine a new class or method. Also, one cannot subclass any metaobjects.

Despite these restrictions, Java's reflective capabilities are used successfully by many Java facilities, such as serialization, remote method invocation (RMI), and Java Beans. However, programming in base Java, is different from programming in the reflective layer. While Java is a statically typed language, the reflective layer is more dynamic. This suggests that a dynamic language, such as Scheme, could be ideal for programming the reflective layer. Scheme could be used as a dynamic language at runtime, and as a dynamic metalanguage at compile time.

Here we describe SILK (Scheme in about 50 K), a compact Scheme dialect written in Java. Because Java is reflective, considerable power can be gained relatively easily. For example, only two functions, `(constructor)` and `(method)`, need to be added to Scheme to provide complete access to the underlying Java.

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 154-174, 1999

Following Common Lisp, SILK provides generic functions. These generic functions accept several method types:

- Java static methods, instance methods, and constructors
- Methods written in Scheme that dispatch on Java classes

By arranging for Java to do most of the work, method dispatch is efficient. For most generic functions, the method dispatch overhead is only one Java method invocation beyond the normal overhead of invoking a reflected Java method through Scheme.

The foreign function interface between Scheme and Java is virtually transparent. The function `(import <class name>)` makes the methods and final static fields of the Java class named `<class name>` accessible to Scheme.

SILK started out simply as a Scheme implementation in Java. However, as its use of reflection has grown, it has become a powerful tool for controlling and developing Java applications. The following sections describe what programming in SILK is like, focusing on issues related to using reflection and the implementation of generic functions.

As this paper is concerned with Scheme, and Java reflection, we begin with a very brief introduction to the essential features of these topics. Anyone familiar with these topics can simply skip to the next section. Anyone with some knowledge of object-oriented programming and reflection should be able to follow this paper easily.

### 1.1 Java Reflection

Java is a statically typed object-oriented language, with a syntax similar to C. Java is a hybrid language with both primitive types, like `int` and `double`, and class based object types descending from the class `Object`. A Java program is built out of a set of classes. Java classes support single inheritance of implementation (extends) and multiple inheritance of interfaces (implements). An interface is a class that describes an object in terms of the protocol (set of methods) it most implement, but provides no implementation itself.

A class definition can contain other class definitions, static and instance fields, and static and instance methods.

A Java method has a signature, which is its name, and the types of each of its parameters. The signature is used at compile time to focus the choice of which method is to be invoked.

Java has two types of methods. An "instance method" has a distinguished first argument as well as possibly some parameters. When such a method is invoked, the dynamic type of this argument is used to select the appropriate method to invoke at runtime, while the declared types of the parameters are used to choose a signature at compile time.

A "static method" does not have a distinguished argument, but may have other parameters. It fills the role of a function in other languages. The choice of which static method to invoke is determined completely at compile time.

The java.lang and java.lang.reflect packages provide metaclasses that reflect the class and its components. To invoke a method, one must first ask the class that defined the method for the method's metaobject, using getMethod(), which takes the methods name, and an array of argument types as parameters. The method can be invoked by using its invoke() method which takes a parameter which is an object array of the arguments. Java provides a set of wrapper classes that let one convert a primitive type to an object. Thus for example, an int can be represented as an instance of class Integer. Here's an example of using the reflection layer to construct a hash table of size 10 and then invoke a put () method on it:

```
package elf;
import java.lang.reflect.*;
import java.util.Hashtable;

public class Reflect {
  public static void main(String[] args) {
    try {
      // Hashtable ht = new Hashtable(10);
      Class[] types1 = new Class[] { Integer.TYPE };
      Constructor c =
        Hashtable.class.getConstructor(types1);
      Object[] args1 = new Object[] {new Integer(10)};
      Hashtable ht = (Hashtable) c.newInstance(args1);

      // ht.put("Three", new Integer(3))
      Class[] types2 = new Class[] {Object.class,
                                    Object.class };
      Method m = Hashtable.class.getMethod("put",
                                           types2);
      Object[] args2 =
        new Object[] { "Three", new Integer(3) };
      m.invoke(ht, args2);

      System.out.println(ht);  // Prints: {Three=3}
    } catch (Exception e) { e.printStackTrace(); }}}
```

## 1.2 Scheme

Scheme is a dynamically typed language that uses simple Lisp syntax. A Scheme program is built out of a sequence of expressions. Each expression is evaluated in turn.

Scheme provides a set of primitive types. The following exhibit shows the Scheme type name and its Java implementation used in SILK:

| **Scheme** | **Java** |
|---|---|
| boolean | Boolean |
| symbol | silk.Symbol |
| char | Character |
| vector | Object[] |
| pair | silk.Pair |

| | |
|---|---|
| procedure | silk.Procedure |
| number | Number |
| exact | Integer |
| inexact | Double |
| string | char[] |
| port | |
| inputport | silk.InputPort |
| outputport | java.io.PrintWriter |

A Scheme procedure takes arguments of any type. Type predicates can be used to identify the type of an object at runtime.

While Scheme would not be considered "object oriented", the Scheme community has developed SLIB [1], a standard library that provides more complex data structures built from these types, including several object oriented extensions.

The power of Scheme comes from such features as:

1. A compact and clear definition, requiring only 50 pages, including its denotational semantic, example, references, and index.
2. A procedure can construct and return a new procedure, as you would expect from a functional language.
3. The syntax is so simple that new minilanguages can extend the language easily using functions or Scheme's macro facility, `(define-syntax)`.

Both Java and Scheme provide garbage collection.

## 2 SILK's Access to Java

SILK offers two distinct interface to Java which are discussed in the following sections.

### 2.1 Primitive Access to Java is Easy

Originally, SILK provided two primitive procedures for accessing Java behavior:

```
(constructor CLASSNAME ARGTYPE1 ...)
(method METHODNAME CLASSNAME ARGTYPE1 ...)
```

The `(constructor)` procedure is given a specification of a Java constructor, in terms of a class name and a list of argument type names, and returns a procedure implementing that constructor. The `(method)` is given a specification of a Java method, in terms of a method name, class name, and a list of argument type names, and returns a procedure implementing that method.

So, for example, we can use the `java.math.BigInteger` package to see if the number, 12345678987654321, is probably prime (with a probability of error of less than 1 part in $2^{-10}$ when the result is `#t`):

```
> (define isProbablePrime
    (method "isProbablePrime" "java.math.BigInteger"
            "int"))
isProbablePrime
```

```
> (define BigInteger
    (constructor "java.math.BigInteger"
                 "java.lang.String"))
BigInteger
> (isProbablePrime (BigInteger "12345678987654321") 10)
#f
```

It is useful to have additional access to Java's reflection layer. Here we define the procedure `(class)` that returns a class object given its full name string:

```
> (define class (method "forName" "java.lang.Class"
                        "java.lang.String"))
class
> (define HT (class "java.util.Hashtable"))
HT
> HT
class java.util.Hashtable
```

Here we define a procedure, `(get-static-value)` that given a class and a string naming a static field, returns the value of the corresponding static field. We then ask for the value of the `TYPE` field of the class `Void`. In Java, that would be simply `Void.TYPE`.

```
>(define get-field
    (method "getField" "java.lang.Class"
           "java.lang.String"))
get-field
>(define get-field-value
    (method "get" "java.lang.reflect.Field"
                 "java.lang.Object"))
get-field-value
>(define (get-static-value class field-name)
   (get-field-value (get-field class field-name) '()))
get-static-value
>(get-static-value (class "java.lang.Void") "TYPE")
void
>
```

### 2.2 But, Procedures Aren't Generic Enough

This approach was used extensively for a year. However, two problems became apparent:

1. Procedures are not generic. Procedures must be carefully named apart to avoid name conflicts. There are many potential conflicts, such as:
    - classes `java.util.Hashtable` and `java.lang.reflect.Field` both provide a `put` method.
    - class `java.lang.Object` provides a `toString` method, while `java.lang.reflect.Modifier` provides a static `toString` method.

   - methods can be overloaded, so for example, class java.lang.StringBuffer provides 10 methods named append.

2. For each Java method or constructor one needs, it must be named and defined separately. Even if only a few methods of a class are used, this can be a fair amount of work.

### 2.3 (import) Lifts Java Classes into SILK Wholesale

To overcome this problem, an (import) function was added used to import the static and instance methods of a class. The goal was to make this similar to the import statement in Java. Here we import Hashtable:

```
> import "java.util.Hashtable")
importing java.util.Hashtable in 161 ms.
#t
>
```

The result of the (import) is that:

- The global variable Hashtable.class is given the value of the class Hashtable.
- Each public instance method applicable to Hashtable is made a generic function. This includes inherited methods, such as clone(), and toString() that are inherited from java.lang.Object. Generic functions, whose name conflicts with an existing Scheme procedure, have "#" add to the name as a suffix. Such conflicts include load#, substring#, length#, apply#, list#, append#, and print#.
- Generic functions are also made for each static method, but they must be named apart from instance methods. They are given a name that looks like Class.method. See the example uses of such methods below.
- A generic constructor function named after the class, Hashtable, is defined. Thus one does not need to say "new", just use the name of the class, Hashtable. This approach is similar to Haskell or ML. It makes constructing nested objects a bit more compact. It also fits well with Scheme's syntax.
- Each global constant is represented in Java as a public static final field is assigned to a global variable of the form Class.field.

We can immediately start using Hashtables:

```
>(define ht (Hashtable 20))
ht
>ht
{}
>(put ht 'clone 1)
()
>ht
{clone=1}
```

```
>(put ht 'zone 2)
()
>ht
{clone=1, zone=2}
>(get ht 'clone)
1
>
```

The procedure (import) creates generic functions. For example, (get) is a generic function with three methods:

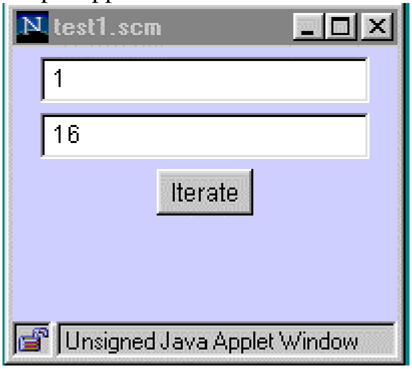
```
> get
{silk.Generic get[3]}
> (for-each print (methods get))
{InstanceMethod Object Map.get(Object)}
{InstanceMethod Object silk.GlobalEnv.get(Symbol)}
{InstanceMethod Object Field.get(Object)}
#t
>
```

Here's an example of using a static method:

```
> (import "java.lang.Float")
importing java.lang.Float in 81 ms.
#t
> (Float.parseFloat "17.42")
17.42
```

## 2.4 Here's an Example of Using (import)

To show what programming with (import) and generic functions is like, here's a simple applet:



1

16

To use it, type a number into the second textfield. The top textfield contains the current estimate of its square root. Each time the "iterate" button is pressed, an iteration of Newton's method is performed to better approximate the square root.

Here's the code. The EasyWin class is borrowed from JLIB.

```
(import "java.lang.Double")
```

```
(import "java.awt.Color")
(import "java.awt.Button")
(import "java.awt.TextField")
(import "jlib.EasyWin")

(define (test1)
  ;; Construct a test window.
  (let ((win (EasyWin "test1.scm" this-interpreter))
        (g (TextField "1" 20))
        (x (TextField "16" 20))
        (go (Button "Iterate")))
    (define (action e)           ; Define call back.
      (setText g
               (toString
                (f
                 (Double.valueOf (getText g))
                 (Double.valueOf (getText x))
                 ))))
    (define (f g x)              ; Newton's method.
      (/ (+ g (/ x g)) 2.0))
    (resize win 200 200)         ; Size the window.
    (add win g)                  ; Add the components.
    (add win x)
    (add win go)
    (addActionCallback win action)
    (setBackground win (Color 200 200 255))
    (show win)))

(test1)                          ; Try it out.
```

## 3 The Generic Function Protocol

Compared to Common Lisp, or even Tiny CLOS, the SILK generic function protocol is simple. Because the protocol is defined in terms of non generic Java methods, metacircularity is not an issue [3]. Here are the essential abstract classes and methods:

```
public abstract class Procedure extends SchemeUtils {
  // Apply the Procedure to a list of arguments.
  public abstract Object apply(Pair args, Engine eng);
}

public abstract class Generic extends Procedure {
  // Add a method to the generic.
  public abstract void addMethod(GenericMethod m);
}

public abstract class GenericMethod extends Procedure {
  // Two GenericMethod's match if they have equal
  // lists of parameterTypes.
  public abstract boolean match(GenericMethod x);

  // Is the method applicable to the list of arguments?
  public abstract boolean isApplicable(Pair args);

  // Returns the method that is more applicable.
  public abostract GenericMethod moreApplicableMethod
                                  (GenericMethod m2);
}
```

A Procedure is an object that can be applied to a list of arguments. It is the basis for function calling in SILK. Class SchemeUtil simply provides many convenient utilities (a common Java idiom). Engine is used to execute code fragments that allows tail call optimization and need not be considered further here.

A Generic is a Procedure defined in terms of a set of GenericMethod's The addMethod() method is used to add a GenericMethod to the set. No two GenericMethod's in the set are allowed to match().

There are currently four classes of GenericMethod. ConstructorMethod, StaticMethod, and InstanceMethod are wrapper classes for each of the corresponding Java metaobjects. A SchemeMethod is used to define methods whose behavior is written in Scheme.

Since (import) assigns constructors, static methods and instance methods to different generic functions, the methods in a generic function tend to be of only one type. However, a generic function can have any subclass of GenericMethod. This allows Scheme methods to be added to any Generic, for example.

### 3.1 Choosing the Applicable Method

To choose the applicable method, we follow Java semantics closely. This may seem surprising since Java chooses a method based on the dynamic type of its first argument, and the declared type of its other arguments. SILK simply makes this choice at runtime based on the types of arguments passed to the Generic.

For example, here are the methods for the (list#) Generic looks like after importing java.io.file and javax.swing.JFrame:

```
> list#
{silk.Generic list#[7]}
> (for-each print (methods list#))
{InstanceMethod String[] File.list(FilenameFilter)}
{InstanceMethod String[] File.list()}
{InstanceMethod void Component.list(PrintStream, int)}
{InstanceMethod void Component.list(PrintWriter, int)}
{InstanceMethod void Component.list()}
{InstanceMethod void Component.list(PrintWriter)}
{InstanceMethod void Component.list(PrintStream)}
```

To list the contents of a directory, the second method would be chosen:

```
> (define f (File "d:/java/jlib3/src/silk/"))
f
```

```
> (for-each* print (list# f))
Closure.java
Code.java
ConstructorMethod.java
...
>
```

The only complication with this approach is Scheme datatypes must be mapped to an appropriate Java type during method invocation. Currently, there are two issues.

1. SILK's numeric types include only `Integer` and `Double`. So, methods such as `java.util.Hashtable(int, float)` can't be invoked directly. One must first convert the required `float`, using `(Float 0.75)` for example. While SILK does not use `Float` or `Long` objects, once such objects are constructed it treats them as normal numbers.
2. Scheme symbols and strings (represented as Java `char[]`) are mapped to class `String` during method invocation. This can lead to an ambiguity, such as in the `append()` method of `java.lang.StringBuffer`:

```
>(import "java.lang.StringBuffer")
importing java.lang.StringBuffer in 70 ms.
#t
 >append#
{silk.Generic append#[10]}
 >(for-each print (methods append#))
{InstanceMethod StringBuffer.append(char[], int, int)}
{InstanceMethod StringBuffer.append(char[])}    ; ***
{InstanceMethod StringBuffer.append(boolean)}
{InstanceMethod StringBuffer.append(String)}    ; ***
{InstanceMethod StringBuffer.append(Object)}
{InstanceMethod StringBuffer.append(char)}
{InstanceMethod StringBuffer.append(long)}
{InstanceMethod StringBuffer.append(int)}
{InstanceMethod StringBuffer.append(float)}
{InstanceMethod StringBuffer.append(double)}
```

Since this `Generic` has methods on both `String` and `char[]`, SILK can't decide which to use. In such a case, the user must invoke a particular method using `(method)`.

### 3.2 Use Only The Most General Method

To minimize the method lookup overhead, for Java instance methods we let Java's single argument dispatch do most of the work. To do that, we only store the most general Java methods in a generic function. For example, for the generic function `(toString)` we only need the `Object.toString()` method:

```
>(methods toString)
({InstanceMethod String Object.toString()})
>
```

We call such a method, a "most general method".

The feasibility of this approach was studied using the 494 classes reachable from the class `javax.swing.Jframe`. Here are some statistics:

| **Count** | **What** |
|---|---|
| 494 | classes |
| 458 | public classes |
| 52 | Exception classes |
| 176 | static most general methods |
| 2759 | instance most general methods. |
| 2935 | total most general methods. |

There were 134 generic functions that contain only static methods. 93% of them have two or fewer methods:

| **Count** | **# Methods,** | **Cumulative % and examples** |
|---|---|---|
| 110 | 1 | 82.1% |
| 15 | 2 | 93.3% |
| 6 | 3 | |
| 1 | 4 | createPackedRaster |
| 1 | 5 | getKeyStroke |
| 1 | 9 | valueOf |

The 2,759 most general instance methods fall into 1525 generic functions. 91% of these have three or fewer methods:

| **Count** | **# Methods,** | **Cumulative % and examples** |
|---|---|---|
| 1058 | 1 | 69.4% |
| 255 | 2 | 86.1% |
| 75 | 3 | 91.0% |
| 39 | 4 | |
| 24 | 5 | |
| 23 | 6 | |
| 17 | 7 | |
| 10 | 8 | |
| 6 | 9 | |
| 1 | 10 | |
| 5 | 11 | |
| 1 | 12 | |
| 1 | 13 | |
| 2 | 16 | |
| 1 | 17 | get |
| 1 | 18 | contains |
| 3 | 20 | clone insert print |
| 1 | 22 | println |
| 1 | 24 | remove |
| 1 | 36 | add |

So, most generic functions have one or two methods so our approach favors such situations.

Methods are only added to a generic function when a class is imported. So, the above statistics reflect what you get if you imported all 494 classes. The number of actual methods is likely to be substantially lower than this. For example, when using only javax.swing.JFrame, (add) only has six methods, not 36.

### 3.3 A Few Discriminator States Are Adequate

Since the number of methods per generic function tends to be small, we focus on optimizing the method lookup for such cases. We use a discriminator function with a small number of states. The state of the discriminator is recomputed whenever a method is added to the generic function. The states are chosen based on the static statistical analysis above. In contrast, Common Lisp chooses discriminator states dynamically so performance is adapted to each run of an application [4]. Here is a description of the states:

1. NOMETHODS - No methods, an error occurs if invoked. The discriminator is in this state when the generic function is first constructed, before any methods have been added to it.
2. ONEMETHOD - One method simply invoke the method.
3. TWOMETHODINSTANCEFIXED - Two instance methods with the same number of arguments. Check the first argument of the first method. If it is appliable, apply it, otherwise apply the second method. This works because the types of the first arguments are disjoint because of the most general method requirement.
4. TWOMETHODNOTFIXED - Two methods of any type with different numbers of arguments. Choose the method based on the number of arguments.
5. GENERAL - Most general lookup. Compute the most applicable method based on all of the arguments of all methods.

For the most likely case of a generic function only having one or two methods, discrimination is little more than a switch jump and a subclass test or two. For cases where an error would occur, we simply invoke the wrong method and let Java signal the error.

Most of the cost of invoking a generic function is in crossing the Scheme/Java frontier to actually apply the method. Currently this involves converting a list of arguments from the Scheme side to an array of arguments on the Java side. String and symbol arguments are also converted to appropriate Java types. The return value must also be converted. For example, a boolean result must be inverted to either `#t` or `#f`.

### 3.4 Scheme Methods

Besides using Java methods in generic functions, it is useful to define methods directly in Scheme, using the (define-method) macro:

```
(define-method name ((arg class) ...)
  (form) ..)
```

Where class names a Java class.

For example, here we define the generic function (iterate collection action) that maps the function action over the elements of collection:

```
(import "java.util.Iterator")
(import "java.util.Collection")
(import "java.util.Map")
(import "java.util.Vector")
(import "java.util.Hashtable")

(define-method iterate ((items java.util.Iterator)
                        action)
  (if (hasNext items)
      (begin (action (next items))
             (iterate items action))))

(define-method iterate ((items java.util.Collection)
                        action)
  (iterate (iterator items) action))

(define-method iterate ((items java.util.Map) action)
  (iterate (entrySet items) action))

(define-method iterate ((items silk.Pair) action)
  (action (car items))
  (let ((items (cdr items)))
    (if (pair? items) (iterate items action))))

(define-method iterate ((items java.lang.Object[])
                action)
  (let loop ((i 0)
           (L (vector-length items)))
    (if (< i L) (begin (action (vector-ref items i))
               (loop (+ i 1) L)))))
```

The collection argument can be an array, a Scheme list (of type silk.Pair), or any of the Java 1.2 collection types. This type of integration is not easy in Java because new instance methods cannot be added to existing classes. Here's an example use:

```
>(define h (Hashtable 50))
h
>(put h 'fred 3)
()
>(put h 'mary 4)
()
>(iterate h print)
fred=3
mary=4
()
>
```

Scheme methods are treated like Java static methods. Currently, there is no provision for (call-next-method). One must invoke such a method directly using (method).

Java classes can be defined directly in Scheme using a (define-class) macro, using the compiling technique described below. Such classes only have constructor and field accessor methods. Scheme methods can be added to the class using (define-method).

## 4 Creating New Code From Old

It should be clear that SILK fulfills one of Scheme's important roles as an embedded scripting and prototyping language [5]. Perhaps a greater strength is that Scheme can be used as a compile time scripting language to generate new code, perhaps in another language, such as C or Java. Two examples of this are described in references [6] and [7] where Scheme is used to set up a complex numerical problem such as a complex computer visualization. Partial evaluation is then used to generate an efficient algorithm to compute the solution of the problem, in C.

Java development environments, such as Sun's Java Bean box, compile small glue classes automatically. This generated code usually follows a standard template and requires some introspection of an existing class. We can do the same thing in SILK. Essentially, Scheme becomes a macro language for Java.

For example, the normal Java runtime environment does not provide a way to trace individual methods. Here we sketch how to add such tracing capability. The basic idea is to generate a subclass of an existing class which allows its methods to be traced. If SILK has enough access to the Java application, an instance of this traceable class can be substituted into the application without changing any Java code. As part of this process, we would have a method, m2, say the method Hashtable.put(), and generate a traced method from it using (gen-trace-method m2):

```
>m2
public synchronized Object Hashtable.put(Object,
                                         Object)
>(emit (gen-trace-method m2))
public synchronized java.lang.Object
    put(java.lang.Object a1, java.lang.Object a0) {
  if(trace) Trace.enter(this + ".put(" + a1 + ", " +
                        a0 + "}");
  java.lang.Object result = super.put(a1, a0);
  if(trace) Trace.exit(result);
  return result;
  }
#t
```

Here's some Scheme code that does this:

```
(define-method method-return-type
  (m java.lang.reflect.Method)
  (getName (getReturnType m)))

(define-method gen-trace-method
  (m java.lang.reflect.Method)
  `(,(gen-method-signature m)
    { ,(gen-trace-method-body m) }))

(define-method gen-method-signature
  (m java.lang.reflect.Method)
  `(,(Modifier.toString (getModifiers m))
    ,(method-return-type m)
    ,(getName m) "("
    ,(map-args
      (lambda (arg) `(,(arg-type arg) ,(arg-name arg)))
      ","
      (arg-types m))
    ")"))

(define-method gen-trace-method-body
  (m java.lang.reflect.Method)
  `(if "(" trace ")"
       Trace.enter "(" this +
       ,(quotify "." (getName m) "(")
       ,(map-args (lambda (arg) `(+ ,(arg-name arg)))
                  "+ \", \""
                  (arg-types m))
       + ,(quotify "}") ")" ";"
       ,(method-return-type m) result = super.
       ,(getName m) "("
       ,(map-args arg-name "," (arg-types m))
       ")" ";"
       if "(" trace ")" "Trace.exit(result)" ";"
       return result ";"))
```

This code generation system is extremely basic. (emit) takes a list structure, produced by (gen-trace-method) here, and formats it into readable Java code. Scheme's backquote macro characters, `(,  ,@) are used to construct the necessary list structures. (map-args) is like (map) but adds a separator between each argument, to generate a comma separated argument list, for example.

SILK can automatically compile Java files using:

```
(import "java.lang.String")
(import "java.lang.reflect.Array")
(import "sun.tools.javac.Main")
(import "java.lang.System")
(define (compile-file file)
  ;; Compile the *.java file, file,
  ;; using the current CLASSPATH.
  (let ((as (Array.newInstance String.class 3))
        (main (Main (get-field System.class 'out)
                    'silkc)))
    (vector-set! as 0 "-classpath")
    (vector-set! as 1 (System.getProperty
                       "java.class.path"))
    (vector-set! as 2 file)
    (compile main as)))
```

In this simple example, Java's Method metaclass was used directly to generate the traced code. A more realistic example would provide a compile time metaobject protocol that would be used to do code generation more formally. While this example is simple, it should be clear that SILK can be used as a useful compile time software development environment without a substantial amount of work.

# 5 Related Work

## 5.1 Other Scheme Implementations

There are three other Scheme implementations in Java we are aware of, which we briefly describe: The following exhibit shows statistics from these implementations.

**Scheme implementation statistics**

| **Implementation** | **java files** | **lines** | **Scheme files** | **lines** | **Generics** |
|---|---|---|---|---|---|
| Silk 1.0 | 12 | 1905 | 0 | 0 | No |
| Silk 2.0 | 20 | 2778 | 0 | 0 | No |
| Generic Silk 2.0 | 28 | 3508 | 5 | 510 | Yes |
| Skij [8] | 27 | 2523 | 44 | 2844 | Yes |
| Jaja [9] | 66 | 5760 | ? | ? | No |
| Kawa [10] | 273 | 16629 | 14 | 708 | No |

**Skij:** Skij is a Scheme advertised as a scripting extension for Java. It is similar in capabilities to SILK and has extensive Java support including `(peek)` and `(poke)` for reading and writing slots, `(invoke)` and `(invoke-static)` for invoking methods, and *`(new)`* for constructing new instances of a Java class.

`(new)` `(invoke)` and `(invoke-static)` invoke the appropriate Java method using runtime looked up based on all of its arguments. This approach is similar to SILK's. However, SILK's generic functions also allow Scheme methods to be added.

**Jaja:** Jaja is a Scheme based on the Christian Queinnec's wonderful book "Lisp in Small Pieces"[11]. It includes a Scheme to Java compiler written in Scheme, because it requires only 1/3 the code of a Java version. Compared to Silk, Jaja is written in a more object oriented style. Like Silk, Jaja uses a super class `(Jaja in Jaja, and SchemeUtils in Silk)` to provide globals and utilitiy functions. Unlike Silk, in Jaja, each Scheme type has one or more Java classes defined for it. Also, in Jaja, the empty list `'()` is represented as an instance of the class `EmptyList`, while in Silk it is represented by null. All Jaja objects are serializable.

**Kawa:** Kawa is an ambitious Scheme implementation. It includes a Scheme to Java byte code compiler. Each function becomes a Java class compiled and loaded at runtime.

### 5.2 Generic Function Dispatch

In any language that uses generic functions, efficient method lookup is extremely important. Method lookup is basically a two-dimensional table lookup in a very sparse table. There has been extensive research recently on table compression methods [12] [13] [14][15].

While these techniques are quite interesting, they seemed too complex to implement for our tiny Scheme environment. Instead, we follow the approach taken in Common Lisp, and associate each generic function with a list of its methods. The trick then becomes, given a generic function and a set of arguments, choose the appropriate method.

In Common Lisp, [4] a generic function does method lookup using one of several strategies. The strategy used is based on the number of different methods the generic function has actually invoked. Backing up these strategies is a per generic function caching scheme. The advantage of such a dynamic approach is that it tailors itself to the running application.

In SILK, a generic function uses a static strategy based on the number and types of methods that have been added to the generic function, as described above. This approach seems to work well, but we have not had a chance to analyze this in any detail. However, other evidence suggests that a static approach such as ours is not unreasonable.

For example, STK[14] is a Scheme dialect that uses an object system based on Tiny CLOS [3]. Method lookup is done by searching for the appropriate method each time, without any caching. Our belief is that this approach works reasonably because most methods only have a few methods.

We have verified this with one typical Common Lisp application (Allegro CL 5.0 + CLIM). There were 1,568 generic functions and 952 classes. The following is a cumulative histogram of the number of methods per generic function.

| #methods | Cumulative % |
|---|---|
| 1 | 59.3 |
| 2 | 79.6 |
| 3 | 89.1 |
| 4 | 91.7 |
| 5 | 94.4 |
| 10 | 97.6 |
| 20 | 99.1 |
| 100 | 100.0 |

From this we see that 59% of the generic functions have one method. These are likely to be assessor methods. 89% of the methods have three or fewer methods. However, one generic function, `(print-object)` has 100 methods.

Thus we expect our static approach to be reasonable. However, adding a Scheme method to a generic function can often force a full lookup to be done. Thus as more Scheme methods are used over Java methods, exploring other approaches will become important. Queinnec [9] describes a method lookup approach that uses discrimination net. This approach looks promising for SILK.

## 6 Conclusion

There are many Scheme implementations, This is partly because it relatively easy to implement a reasonably efficient implementation, and techniques to build a high performance implementation are well understood. It is also partly because, while Scheme is a useful language in its own right, it has found and important role as a scripting language embedded in an application. Guile and STK are two examples of embeddable Schemes implemented in C [16].

Beckman argues that scripting languages are inevitable [5]. In the 80's Jon Bently popularized the idea of Little Languages [17]. Such a language can be used to describe in a compact way, one aspect of your project, graphical layout for example. Beckman argues that this decoupling of aspects is essential, because the only other option is to keep changing all the source code. He also argues that little languages often grow to become more complete languages, adding control structure, classes, etc. TCL and Visual Basic are unfortunate examples of this trend. Beckman further argues that Scheme is an excellent choice for a little language, because it is also a complete language in which other extension languages can be easily embedded.

We have tried to show that SILK makes effective use of Java's reflective capabilities:

1. The implementation of its data types were carefully chosen to match those of Java closely. This minimizes the impedance mismatch between Scheme and Java. Currently the main remaining mismatch is that strings in Scheme are represented as `char[]` because strings are mutable in Scheme. Making Scheme strings immutable, as they are in Java would allow us to use the String class. This change would only require dropping the procedure `(string-set!)` from SILK.

2. Using `(import)` SILK has almost transparent access to Java. A Scheme implemented in another languages requires substantial foreign function interface to be written. While it is possible to automatically generate this interface from C headers files, for example, it is a fair amount of work.
3. Once SILK is available, it can be used for scripting in the base language, providing the benefits that Beckman suggests. For example, a SILK extension is used to layout Java Swing components. A graphical designer lays out the components and a Java programmer wires in the underlying behavior.
4. Because SILK has direct access to the reflection of any Java class, SILK can be used as a metalevel scripting language to generate new Java classes at compile or runtime. While other Java systems have similar capabilities, we've tried to show that the amount of work required in SILK is small.
5. Reflection could be used more aggressively in the implementation of the underlying Scheme interpreter itself. For example, the Scheme primitives could be implemented as Java classes and imported into a kernel language. The Scheme interpreter and Scheme-to-Java compiler, which are written in Java, could also be accessed via reflection, and even modified at runtime using a custom classloader.

## Acknowledgments

The authors wish to thank Peter Norvig for developing the first version of SILK and for keeping it simple enough that we could easily use and extend it. We thank Geoffrey S. Knauth, Richard Shapiro, Tom Mitchell, Bruce Roberts, and Jeff Tustin for reviewing drafts of this paper. We thank Rusty Bobrow who thought SILK was simple enough that he suggested it to his daughter for programming a High School science project. And we'd like to thank Rusty's brother, Danny Bobrow for long ago teaching us that "Meta is Beta!" That two generations of one family use reflection, reflects well on reflection.

## References

1. http://www-swiss.ai.mit.edu/~jaffer/SLIB.html
2. Aubrey Jaffer, r4rstest.scm, ftp://ftp-swiss.ai.mit.edu/pub/scm/r4rstest.scm.
3. Gregor Kiczales, Tiny CLOS, file://parcftp.xerox.com/pub/mops/tiny/
4. Gregor Kiczales and Luis Rodriguez, Efficient method dispatch in PCL, proceedings 1990 ACM Conference on LISP and Functional Programming, Nice, France, June 1990, 99-105.
5. Brian Beckman, A scheme for little languages in interactive graphics, Software Practice and Experience, 21, 2, p. 187-208, Feb, 1991.
6. A. Berlin and D. Weise, Compiling scientific code using partial evaluation. Technical Report CSL-TR 90-422, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1990.
7. Clifford Beshers Steven Feiner, Generating efficient virtual worlds for visualization using partial evaluation and dynamic compilation, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), p. 107-115.

8. Mike Travers, Skij, IBM alphaWorks archive, http://www.alphaworks.ibm.com/formula/Skij
9. Christian Queinnec, JaJa: Scheme in Java, http://www-spi.lip6.fr/~queinnec/WWW/Jaja.html
10. Per Bothner, Kawa the Java-based Scheme System, http://www.cygnus.com/~bothner/kawa.html
11. http://www-spi.lip6.fr/~queinnec/Papers/dispatch.ps.gz
12. Shih-Kun Huang and Deng-Jyi Chen, Efficient algorithms for method dispatch in object-oriented programming systems, Journal of Object-Oriented Programming, 5(5):43-54, September 1992.
13. E.Amiel, O. Gruber, and E. Simon, Optimmizing multi-method dispatch using compressed dispatch tables. In OOPSLA '94, October 1994.
14. Jan Vitek and R. Nigel Horspool, Taming message passing: Efficien method lookup for dynamically typed languages. ECOOP '94 - 8th European Conference on Object-Oriented Programming, Bologna (Italy), 1994.
15. Weimin Chen and Volker Turau, Multiple-dispatching base on automata, Theory and Practice of Object Systems, 1(1):41-60, 1995.
16. http://kaolin.unice.fr/STk/
17. J.L. Bentley, More Programming Pearls, Addison-Wesley, Reading, MA, 1988.

# *jContractor*: A Reflective Java Library to Support Design By Contract

Murat Karaorman[1,2], Urs Hölzle[2], John Bruno[2]

[1]Texas Instruments Inc.,
315 Bollay Drive, Santa Barbara, California 93117
muratk@ti.com

[2]Department of Computer Science,
University of California, Santa Barbara, CA 93106
{murat,urs,bruno}@cs.ucsb.edu

**Abstract.** *jContractor* is a purely library based approach to support Design By Contract specifications such as preconditions, postconditions, class invariants, and recovery and exception handling in Java. *jContractor* uses an intuitive naming convention, and standard Java syntax to instrument Java classes and enforce Design By Contract constructs. The designer of a class specifies a contract by providing contract methods following *jContractor* naming conventions. *jContractor* uses Java Reflection to synthesize an instrumented version of a Java class by incorporating code that enforces the present *jContractor* contract specifications. Programmers enable the run-time enforcement of contracts by either engaging the *jContractor* class loader or by explicitly instantiating objects using the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not. Since *jContractor* is purely library-based, it requires no special tools such as modified compilers, modified JVMs, or pre-processors.

## 1 Introduction

One of the shortcomings of mainstream object-oriented languages such as C++ and Java is that class or interface definitions provide only a signature-based application interface, much like the APIs specified for libraries in procedural languages. Method signatures provide limited information about the method: the types of formal parameters, the type of returned value, and the types of exceptions that may be thrown. While type information is useful, signatures by themselves do not capture the essential semantic information about what the method does and promises to deliver, or what conditions must be met in order to use the method successfully. To acquire this information, the programmer must either analyze the source code (if available) or

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 175-196, 1999

rely on some externally communicated specification or documentation, none of which is automatically checked at compile or runtime.

A programmer needs semantic information to correctly design or use a class. Meyer introduced *Design By Contract* as a way to specify the essential semantic information and constraints that govern the design and correct use of a class [6]. This information includes assertions about the state of the object that hold before and after each method call; these assertions are called *class invariants*, and apply to the public interface of the class. The information also includes the set of constraints that must be satisfied by a client in order to invoke a particular method. These constraints are specific to each method, and are called *preconditions* of the method. Each precondition specifies conditions on the state of the object and the argument values that must hold prior to invoking the method. Finally, the programmer needs assertions regarding the state of the object after the execution of a method and the relationship of this state to the state of the object just prior to the method invocation. These assertions are called the *postconditions* of a method. The assertions governing the implementation and the use of a class are collectively called a *contract*. Contracts are specification constructs which are not necessarily part of the implementation code of a class, however, a runtime monitor could check whether contracts are being honored.

In this paper we introduce *jContractor*, a purely library-based system and a set of naming conventions to support *Design By Contract* in Java. The *jContractor* system does not require any special tools such as modified compilers, runtime systems, modified JVMs, or pre-processors, and works with any pure Java implementation. Therefore, a programmer can practice *Design By Contract* by using the *jContractor* library and by following a simple and intuitive set of conventions.

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine (JVM) `class` file format [10]. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. *jContractor* utilizes the meta-level information encoded in the standard Java class files to instrument the bytecodes on-the-fly during class loading. During the instrumentation process *jContractor* parses each Java class file and discovers the *jContractor* contract information by analyzing the class meta-data.

The *jContractor* design addresses three key issues which arise when adding contracts to Java: how to express preconditions, postconditions and class invariants and incorporate them into a standard Java class definition; how to reference entry values of attributes, to check method results inside postconditions using standard Java syntax; and how to check and enforce contracts at runtime.

An overview of *jContractor*'s approach to solving these problems is given below:

1. Programmers add contract code to a class in the form of methods following *jContractor*'s naming conventions: *contract patterns*. The *jContractor* class loader recognizes these patterns and rewrites the code to reflect the presence of contracts.

2. Contract patterns can be inserted either directly into the class or they can be written separately as a *contract class* where the contract class' name is derived from the target class using *jContractor* naming conventions. The separate contract class approach can also be used to specify contracts for interfaces.

3. The *jContractor* library instruments the classes that contain *contract patterns* on the fly during class loading or object instantiation. Programmers enable the run-time enforcement of contracts either by engaging the *jContractor* class loader or by explicitly instantiating objects from the *jContractor* object factory. Programmers can use exactly the same syntax for invoking methods and passing object references regardless of whether contracts are present or not.

4. *jContractor* uses an intuitive naming convention for adding *preconditions, postconditions, class invariants, recovery* and *exception handling* in the form of `protected` methods. Contract code is hence distinguished from the functional code. The name and signature of each contract method determines the actual method with which the contract is associated.

5. Postconditions and exception handlers can access the *old* value of any attribute by using a special object reference, *`OLD`*. For example `OLD.count` returns the value of the attribute `count` just prior to the execution of the method. *jContractor* emulates this behavior by transparently rewriting class methods during class loading so that the entry values of *`OLD`* references are saved and then made available to the postcondition and exception handling code.

6. *jContractor* provides a class called *`RESULT`* with a `static boolean` method, `Compare.` Inside a method's postcondition it is possible to check the *result* associated with the method's execution by calling *`RESULT.Compare(<expression>)`*. The call returns true or false by comparing the value of the *`<expression>`* with the current result.

This paper presents an extension to Java to support Design By Contract by introducing *jContractor* as a pure library-based approach which utilizes the meta-level information found in Java class files and takes advantage of dynamic class loading in order to perform "reflective", on-the-fly bytecode modification.

## 2 *jContractor* Library and Contract Patterns

*jContractor* is a purely library-based approach to support Design By Contract constructs using standard Java. Table 1 contains a summary of key Design By Contract constructs and the corresponding *jContractor* patterns. One of the key contributions of *jContractor* is that it supports all Design By Contract principles using a *pure-Java*, *library-based* approach. Therefore, any Java developer can immediately start using Design By Contract without making any changes to the test, development, and deployment environment after obtaining a copy of *jContractor* classes.

**Table 1. Summary of *jContractor* Design By Contract Constructs**

| ***Construct*** | ***jContractor Pattern*** | ***Description*** |
|---|---|---|
| ***Precondition***<br>*(Client's obligation)* | `protected boolean`<br>***`methodName_PreCondition(`***<br>*`<arg-list>)`* | Evaluated just before `methodName` with matching signature is executed. If the precondition fails the method throws a *`PreConditionException`* without executing the method. |
| ***Postcondition***<br>*(Implementor's promise)* | `protected boolean`<br>***`methodName_PostCondition(`***<br>*`<arg-list>)`* | Evaluated just before `methodName` returns (i.e. *normal termination*).<br>If the postcondition fails, a *`PostConditionException`* gets thrown. |
| ***Exception Handler***<br>*(Implementor's attempt)* | *`protected Object`*<br>***`methodName_OnException(`***<br>*`Exception e)`*<br>*`throws Exception`* | Called when `methodName's` execution ends in *abnormal termination*, throwing an Exception. The exception handler provides an opportunity to do recovery by restoring invariants, resetting state, etc., |
| ***Class invariant***<br>*(Implementor's promise)* | *`protected boolean`*<br>***`className_ClassInvariant(`***<br>***`)`*** | Evaluated once before each invocation of a public method, *`m`*, *<u>and</u>* once before *`m`* is about to return -- *normal termination.* If class invariant fails, a *`ClassInvariantException`* is thrown instead of returning the result. |
| ***old*** | ***`OLD`***`.`*`attr`*<br>***`OLD`***`.`*`foo( )`* | Expression evaluates to *value* of *`attr`* on method entry. *OLD* methods can only be used inside postcondition and exception handler methods; *`attr`* can be any non-private class attribute. |
| ***Result*** | ***`RESULT.compare`***`(`*`expr`*`)` | Evaluates true/false depending on if current result matches the expr. `RESULT` class is part of *jContractor* distribution. |

```
class Dictionary  … {
    protected Dictionary OLD;

    Object put(Object x, String key)
    {
          putBody();
    }
    protected boolean put_PreCondition(Object x,
                                       String key)
    {
          return ( (count <= capacity)
                &&  !(key.length()==0)  );
    }
    protected boolean put_PostCondition(Object x,
                                        String key)
    {
          return (   (has (x))
                  &&  (item (key) == x)
                  &&  (count == OLD.count + 1) )
    }
    protected Object put_OnException(Exception e)
                       throws Exception
    {
          count =  OLD.count;
          throw e;         //rethrow exception.
    }
    protected boolean Dictionary_ClassInvariant()
    {
          return (count >= 0);
    }
 …}
```

**Figure 1-a. Dictionary Class Implementing Contract for *put* Method**

A programmer writes a contract by taking a class or method name, say *put*, then appending a suffix depending on the type of constraint, say *_PreCondition*, to write the *put_PreCondition.* Then the programmer writes the method body describing the precondition. The method can access both the arguments of the *put* method with the identical signature, and the attributes of the class. When jContractor instrumentation is engaged at runtime, the precondition gets checked each time the *put* method is called, and the call throws an exception if the precondition fails.

The code fragment in Figure 1-a shows a *jContractor* based implementation of the put method for the *Dictionary* class. An alternative approach is to provide a separate *contract class*, *Dictionary_CONTRACT*, as shown in Figure 1-b, which

```
class Dictionary_CONTRACT extends Dictionary …
{
    protected boolean put_PostCondition(Object   x,
                                                String key)
    {
          return (    (has (x))
                   && (item (key) == x)
                   && (count == OLD. count + 1) )
    }

    protected boolean Dictionary_ClassInvariant() {
                return (count >= 0);
    }
}
```

**Figure 1-b. Separate Contract Class for `Dictionary`**

contains the contract code using the same naming conventions. The contract class can (optionally) extend the target class for which the contracts are being written, which is the case in our example. For every class or interface *X* that the *jContractor ClassLoader* loads, it also looks for a separate contract class, *X_CONTRACT*, and uses contract specifications from both *X* and *X_CONTRACT* (if present) when performing its instrumentation. The details of the class loading and instrumentation will be presented in subsequent sections.

Table 2 shows the informal contract specifications for inserting an element into the *dictionary*, a table of bounded capacity where each element is identified by a certain character string used as key.

**Table 2. Contract Specification for Inserting Element to Dictionary**

| | ***Obligations*** | ***Benefits*** |
|---|---|---|
| ***Client*** | *(Must ensure precondition)*<br>Make sure table is *not full* & key is a *non-empty* string | *(May benefit from postcondition)*<br>Get updated table where the given element now appears, associated with the given key. |
| ***Supplier*** | *(Must ensure postcondition)*<br>Record given element in table, associated with given key. | (*May assume precondition)*<br>No need to do anything if table given is *full*, or key is *empty* string. |

## 2.1 Enabling Contracts During Method Invocation

In order to enforce contract specifications at run-time, the contractor object must be instantiated from an instrumented class. This can be accomplished in two possible ways: (1) by using the *jContractor class loader* which instruments all classes containing contracts during class loading; (2) by using a factory style instantiation using the *jContractor* library.

The simplest and the preferred method is to use the *jContractor class loader*, since this requires no changes to a client's code. The following code segment shows how a client declares, instantiates, and then uses a `Dictionary` object, *`dict`*. The client's code remains unchanged whether *jContractor* runtime instrumentation is used or not:

```
Dictionary  dict;          // Dictionary (Figure-1) defines contracts.

dict = new Dictionary();   // instantiates dict from instrumented or
                           // non- instrumented  class depending on
                           //  jContractor classloader being engaged.
dict.put(obj1,“name1”);    //  If jContractor is enabled, put-contracts
                           // are enforced,  i.e. contract  violations
                           // result in an exception being thrown.
```

The second approach uses the *jContractor* object factory, by invoking its *`New`* method. The factory instantiation can be used when the client's application must use a custom (or third party) class loader and cannot use *jContractor class loader*. This approach also gives more explicit control to the client over *when* and *which* objects to instrument. Following code segment shows the client's code using the *jContractor* factory to instantiate an instrumented `Dictionary` object, *`dict`*:

```
dict = (Dictionary) jContractor.New(“Dictionary”);
                                       // instruments  Dictionary

 dict.put(obj1,“name1”);  //  put-contracts are enforced
```

Syntactically, any class containing *jContractor* design-pattern constructs is still a pure Java class. From a client's perspective, both instrumented and non-instrumented instantiations are still Dictionary objects and they can be used interchangeably, since they both provide the same interface and functionality. The only semantic difference in their behavior is that the execution of instrumented methods results in evaluating the contract assertions, (e.g., *`put_PreCondition`*) and throwing a Java runtime exception if the assertion fails.

Java allows method overloading. *jContractor* supports this feature by associating each method variant with the pre- and postcondition functions with the matching argument signatures.

For any method, say `foo`, of class `X`, if there is no `boolean` method by the name, `foo_PreCondition` with the same argument signature, in either `X`, `X_CONTRACT` or one of their descendants then the default precondition for the `foo` method is "true". The same "default" rule applies to the postconditions and class invariants.

## 2.2 Naming Conventions for Preconditions, Postconditions and Class Invariants

The following naming conventions constitute the *jContractor* patterns for pre- and postconditions and class invariants:

| | | | |
|---|---|---|---|
| *Precondition*: | *protected boolean* | *methodName* | *+ _PreCondition + ( <arg-list>)* |
| *Postcondition*: | *protected boolean* | *methodName* | *+ _PostCondition + ( < arg-list >)* |
| *ClassInvariant:* | *protected boolean* | *className* | *+ _ClassInvariant ( )* |

Each construct's method body evaluates a `boolean` result and may contain references to the object's internal state with the same scope and access rules as the original method. Pre- and postcondition methods can also use the original method's formal arguments in expressions. Additionally, postcondition expressions can refer to the old values of object's attributes by declaring a pseudo object, `OLD`, with the same class type and using the `OLD` object to access the values.

## 2.3 Exception Handling

The postcondition for a method describes the contractual obligations of the contractor object only when the method terminates successfully. When a method terminates abnormally due to some exception, it is not required for the contractor to ensure that the postcondition holds. It is very desirable, however, for the contracting (supplier) objects to be able to specify what conditions must still hold true in these situations, and to get a chance to restore the state to reflect this.

*jContractor* supports the specification of general or specialized exception handling code for methods. The instrumented method contains wrapper code to catch exceptions thrown inside the original method body. If the contracts include an exception-handler method for the type of exception caught by the wrapper, the exception handler code gets executed.

If exception handlers are defined for a particular method, each exception handler must either re-throw the handled exception or compute and return a valid result. If the exception is re-thrown no further evaluation of the postconditions or class-invariants is carried out. If the handler is able to recover by generating a new result, the postcondition and class-invariant checks are performed before the result is returned, as if the method had terminated successfully.

The exception handler method's name is obtained by appending the suffix, *"_OnException"*, to the method's name. The method takes a single argument whose type belongs to either one of the exceptions that may be thrown by the original method, or to a more general exception class. The body of the exception handler can include arbitrary Java statements and refer to the object's internal state using the same scope and access rules as the original method itself. The *jContractor* approach is more flexible than the Eiffel's "*rescue*" mechanism because separate handlers can be written for different types of exceptions and more information can be made available to the handler code using the exception object which is passed to the handler method.

## 2.4 Supporting Old Values and Recovery

*jContractor* uses a clean and safe instrumentation "trick" to mimic the Eiffel keyword, *old*, and support Design By Contract style postcondition expressions in which one can refer to the "*old*" state of the object just prior to the method's invocation. The trick involves using the "syntax notation/convention", *OLD.x* to mean the value that x had when method body was entered. Same notation is also used for method references as well, e.g., *OLD.foo()* is used to refer to the result of calling the member *foo()* when entering the method. We will later explain how the *jContractor* instrumentation process rewrites expressions involving *OLD* to achieve the desired effect. First we illustrate its usage from the example in Figure 1. The class Dictionary first declares *OLD*:

```
        private Dictionary OLD;
```

Then, in the postcondition of the put method taking *<Object x, String key>* arguments, the following subexpression is used

```
        (count == OLD.count + 1)
```

to specify that the execution of the corresponding put method increases the value of the object's count by 1. Here *OLD.count* refers to the value of count at the point just before the *put*-method began to execute.

*jContractor* implements this behavior using the following instrumentation logic. When loading the Dictionary class, *jContractor* scans the postconditions and exception handlers for *OLD* usage. So, when it sees the *OLD.count* in *put_PostCondition* it inserts code to the beginning of the put method to allocate a unique temporary and to save count to this temporary. Then it rewrites the expression in the postcondition replacing the *OLD.value* subexpression with an access to the temporary. In summary, the value of the expression *OLD.expr* (where *expr* is an arbitrary sequence of field dereferences or method calls) is simply the value of *expr* on entry to the method.

It is also possible for an exception handler or postcondition method to revert the state of `attr` to its old value by using the *OLD* construct. This may be used as a basic recovery mechanism to restore the state of the object when an invariant or postcondition is found to be violated within an exception-handler. For example,

```
attr = OLD.attr;
```

or

```
attr = OLD.attr.Clone();
```

The first example restores the object reference for `attr` to be restored, and the second example restores the object state for `attr` (by cloning the object when entering the method, and then attaching the object reference to the cloned copy.)

### 2.5 Separate Contract Classes

*jContractor* allows contract specifications for a particular class to be externally provided as a separate class, adhering to certain naming conventions. For example, consider a class, *X*, which may or may not contain *jContractor* contract specifications. *jContractor* will associate the class name, *X_CONTRACT*, with the class *X*, as a potential place to find contract specifications for *X*. *X_CONTRACT* must extend class *X* and use the same naming conventions and notations developed earlier in this paper to specify pre- and postconditions, exception handlers or class invariants for the methods in class *X*.

If the implementation class *X* also specifies a precondition for the same method, that precondition is *logical-AND*'ed with the one in *X_CONTRACT* during instrumentation. Similarly, postconditions, and class invariants are also combined using *logical-AND*. Exception handlers in the contract class override the ones inherited from *X*.

The ability to write separate contract classes is useful when specifying contracts for legacy or third party classes, and when modifying existing source code is not possible or viable. It can be used as a technique for debugging and testing system or third-party libraries.

### 2.6 Contract Specifications for Interfaces

Separate contract classes also allow contracts to be added to interfaces. For example, consider the `interface` *IX* and the class *C* which implements this interface. The class *IX_CONTRACT* contains the pre- and postconditions for the methods in *IX*. Methods defined in the contract class are used to instrument the class "implementing" the interface.

```
interface IX {

      int foo(<args>);
}

class IX_CONTRACT  {

  protected boolean foo_PreCondition(<args>) { … }
  protected boolean foo_PostCondition(<args>){ … }
}
```

Contracts for interface classes can only include pre- and postconditions, and can only express constraints using expressions involving the method's arguments or interface method calls, without any references to a particular object state. If the implementation class also specifies a precondition for the same method, the conditions are *logical-AND*'ed during instrumentation. Similarly, postconditions are also combined using *logical-AND*.

## 3 Design and Implementation of *jContractor*

The *jContractor* package uses *Java Reflection* to detect Design By Contract patterns during object instantiation or class loading. Classes containing contract patterns are instrumented on the fly using the *jContractor* library. We begin by explaining how instrumentation of a class is done using the two different mechanisms explained in section 2.1. The rest of this section explains the details of the instrumentation algorithm.

The primary instrumentation technique uses the *jContractor class loader* to transparently instrument classes during class loading. The scenario depicted in Figure 2 illustrates how the *jContractor Class Loader* obtains instrumented class bytecodes from the *jContractor instrumentor* while loading class Foo. The *jContractor class loader* is engaged when launching the Java application. The instrumentor is passed the name of the class by the class loader and in return it searches the compiled class, Foo, for *jContractor* contract patterns. If the class contains contract methods, the instrumentor makes a copy of the class bytecodes, modifying the public methods with wrapper code to check contract violations, and returns the modified bytecodes to the class loader. Otherwise, it returns the original class without any modification. The object instantiated from the instrumented class is shown as the *Foo<Instrumented>* object in the diagram, to highlight the fact that it is instrumented, but syntactically it is a Foo object.

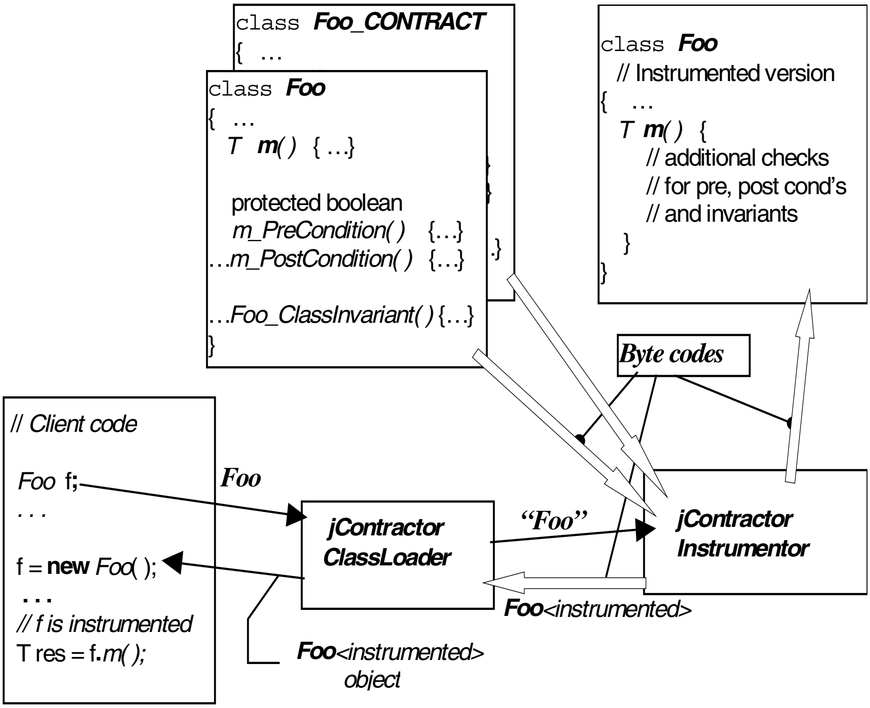
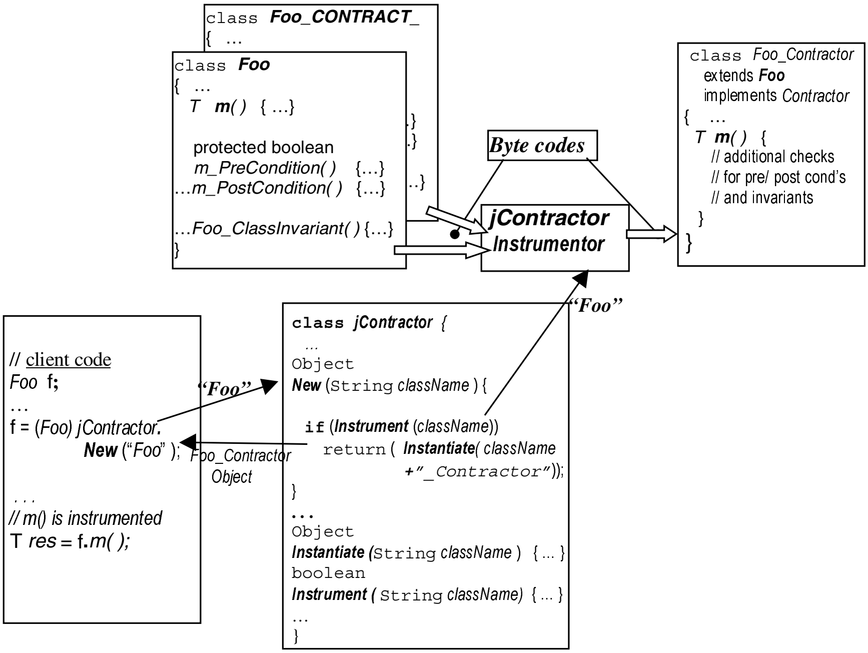


**Figure 2.** *jContractor* **Class Loader based Instrumentation**

If the command line argument for *jContractor* is not present when starting up the application, the user's own (or the default) class loader is used, which effectively turns off the *jContractor* instrumentation. Since contract methods are separate from the public methods, the program's behavior remains exactly the same except for the runtime checking of contract violations. This is the preferred technique since the client's code is essentially unchanged and all that the supplier has to do is to add the *jContractor* contract methods to the class.

The alternative technique is a factory style object instantiation using the *jContractor* library's New method. New takes a class name as argument and returns an instrumented object conforming to the type of requested class. Using this approach the client explicitly instructs *jContractor* to instrument a class and return an instrumented instance. The factory approach does not require engaging the *jContractor* class loader and is safe to use with any pure-Java class loader. The example in Figure 3 illustrates the factory style instrumentation and instantiation using the class `Foo`. The client invokes *jContractor*.`New`() with the name of the class, "*`Foo`*". The New method uses the *jContractor* Instrumentor to create a subclass of `Foo`, with the name, `Foo_Contractor` which now contains the instrumented version of `Foo`. New instantiates and returns a `Foo_Contractor` object to the client. When the client



```
class Foo_CONTRACT_
{ ...
```

```
class Foo
{ ...
  T m() {...}

  protected boolean
  m_PreCondition() {...}
...m_PostCondition() {...}

...Foo_ClassInvariant() {...}
}
```

```
class Foo_Contractor
   extends Foo
   implements Contractor
{ ...
  T m() {
    // additional checks
    // for pre/ post cond's
    // and invariants
  }
}
```

```
// client code
Foo f;
...
f = (Foo) jContractor.
         New ("Foo" );
. . .
// m() is instrumented
T res = f.m( );
```

```
class jContractor {
  ...
Object
New (String className ) {

  if (Instrument (className))
    return ( Instantiate( className
                   +"_Contractor"));
}
...
Object
Instantiate (String className ) { ... }
boolean
Instrument ( String className) { ... }
...
}
```

**Figure 3** *jContractor* **Factory Style Instrumentation and Instantiation**

invokes methods of the returned object as a `Foo` object, it calls the instrumented methods in `Foo_Contractor` due to the polymorphic assignment and dynamic binding.

The remainder of this section contains details of the instrumentation algorithm for individual *jContractor* constructs.

## 3.1 Method Instrumentation

*jContractor* instruments contractor objects using a simple code rewriting technique. Figure 4 illustrates the high level view of how code segments get copied from original class methods into the target instrumented version. *jContractor*'s key instrumentation policy is to inline the contract code for each method within the target method's body, to avoid any extra function call. Two basic transformations are applied to the original method's body. First, `return` statements are replaced by an assignment statement – storing the result in a method-scoped temporary – followed by a labeled `break`, to exit out of the method body. Second, references to "old" values, using the *`OLD`* object reference are replaced by a single variable – this is explained in more detail in a subsection. After these transformations, the entire method block is placed inside a wrapper code as shown in the instrumented code in Figure 4.

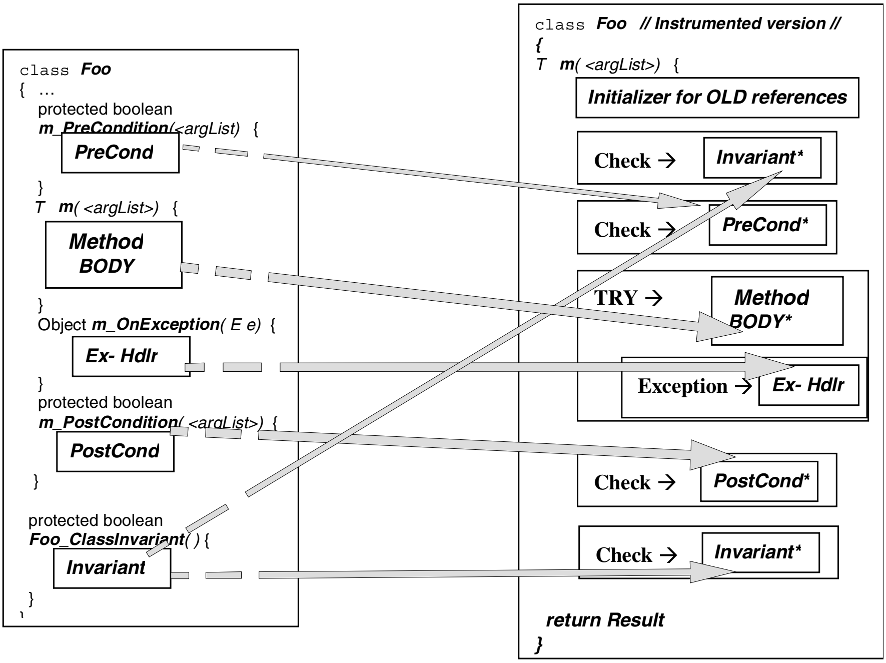


**Figure 4.** *jContractor* **Instrumentation Overview**

A *check wrapper* checks the boolean result computed by the wrapped block and throws an exception if the result is `false`. A *TRY wrapper* executes the wrapped code inside a try-catch block, and associates each exception handler that the contract specifies with a catch phrase inside an exception wrapper. *Exception wrappers* are simple code blocks that are inserted inside the catch clause of a try-catch block with the matching `Exception` type. Typically, exception handlers re-throw the exception, which causes the instrumented method to terminate with the thrown exception. It is possible, however, for the exception handler to recover from the exception condition and generate a result. Figure 5 shows a concrete example of the instrumented code that is generated.

## 3.2 Instrumentation of OLD References

*jContractor* takes the following actions for each unique *OLD-expression* inside a method's postcondition or exception handler code. Say the method's name is `m()` and the expression is `OLD.attr`, and `attr` has type T, then *jContractor* incorporates the following code while rewriting `m():`

```
T  $OLD_$attr = this.attr;
```

```
class Dictionary_Contractor  extends Dictionary …{
  …
  Object put(Object x, String key)
  {
      Object   $put_$Result;
      boolean  $put_PreCondition,
               $put_PostCondition,
               $ClassInvariant;

      int      $OLD_$count = this.count;

      $put_PreCondition = (  (count <= capacity)
                          && (! key.length()==0) );

      if (!$put_PreCondition) {
            throw new PreConditionException();
      }
      $ClassInvariant = (count >= 0);
      if (!$ClassInvariant) {
            throw new ClassInvariantException();
      }
      try {
            $put_$Result = putBody();
      }
      catch (Exception e) {      // put_OnException
            count = $OLD_$count; //restore(count)
            throw e;
      }
      $put_PostCondition =((has(x)) &&
                          (item (key) == x) &&
                          (count == $OLD_$count + 1));
      if (!$put_PostCondition) {
            throw new PostConditionException();
      }
      $ClassInvariant = (count >= 0);
      if (!$ClassInvariant) {
            throw new ClassInvariantException();
      }
      return $put_$Result;
    }
  }
```

**Figure 5. Factory Instrumented Dictionary Subclass.**

The effect of this code is to allocate a temporary, *$OLD_$attr*, and record the value of the expression, *attr*, when the method code is entered. The code rewriting logic then replaces all occurrences of *OLD.attr* inside the contract code with the temporary variable *$OLD_$attr* whose value has been initialized once at the beginning of the method's execution.

### 3.3 Instrumentation of RESULT References

*jContractor* allows the following syntax expression inside a method's postcondition method to refer to the result of the current computation that led to its evaluation:

```
RESULT.Compare(expression)
```

*RESULT* is provided as part of the *jContractor* library package, to facilitate this syntax expression. It exports a single *static boolean* method, *Compare(),* taking a single argument with one variant for each built-in Java primitive type and one variant for the *Object* type. These methods never get invoked in reality, and the sole purpose of having them (like the *OLD* declarations discussed earlier) is to allow the Java compiler to legally accept the syntax, and then rely on the instrumentation logic to supply the right execution semantics.

During instrumentation, for each method declaration, *T m()*, a temporary variable *$m_$Result* is internally declared with the same type, *T*, and used to store the result of the current computation. Then the postcondition expression shown above is rewritten as:

```
($m_$Result == (T)(expression))
```

### 3.4 Use of Reflection

Each class and interface in a Java program corresponds to a translation unit with a machine and platform independent representation as specified by the Java Virtual Machine `class` file format. Each class file contains JVM instructions (bytecodes) and a rich set of meta-level information. During the instrumentation process *jContractor* parses and analyzes the meta-information encoded in the class byte-codes in order to discover the *jContractor* contract patterns. When the class contains or inherits contracts, *jContractor* instrumentor modifies the class bytecodes on the fly and then passes it to the class loader. The class name and its inheritance hierarchy; the method names, signatures and code for each class method; the attribute names found and referenced in class methods constitute the necessary and available meta information found in the standard Java class files. The presence of this meta information in standard Java class byte codes and the capability to do dynamic class loading are essential to our way building a pure-library based *jContractor* implementation.

Core Java classes include the `java.lang.reflect` package which provides reflection capabilities that could be used for parsing the class information, but using this package would require prior loading of the class files into the JVM. Since jContractor needs to do its instrumentation *before* loading the class files, it cannot use core reflection classes directly and instead uses its own class file parser.

# 4 Discussion

## 4.1 Interaction Of Contracts With Inheritance And Polymorphism

Contracts are essentially specifications checked at run-time. They are not part of the functional implementation code, and a "correct" program's execution should not depend on the presence or enabling of the contract methods. Additionally, the exceptions that may be thrown due to runtime contract violations are not checked exceptions, therefore, they are not required to be part of a method's signature and do not require clients' code to handle these specification as exceptions. In the rest of this section we discuss the contravariance and covariance issues arising from the way contracts are inherited.

The inheritance of preconditions from a parent class follows *contravariance*: as a subclass provides a more specialized implementation, it should weaken, not strengthen, the preconditions of its methods. Any method that is redefined in the subclass should be able to at least handle the cases that were being handled by the parent, and in addition handle some other cases due to its specialization. Otherwise, polymorphic substitution would no longer be possible. A client of *X* is bound by the contractual obligations of meeting the precondition specifications of *X*. If during runtime an object of a more specialized instance, say of class *Y* (a subclass of *X*) is passed, the client's code should not be expected to satisfy any stricter preconditions than it already satisfies for *X*, irrespective of the runtime type of the object.

*jContractor* supports contravariance by evaluating the a *logical-OR* of the precondition expression specified in the subclass with the preconditions inherited from its parents. For example, consider the following client code snippet:

```
// assume that class Y extends class X
X x;
Y y = new Y();       // Y object instantiated
x = y;               // x is polymorphically attached
                     // to a Y object
int i = 5; …
x.foo(i);            // only PreCondition(X,[foo,int i]) should be met
```

When executing `x.foo()`, due to dynamic binding in Java, the *foo()* method that is found in class Y gets called, since the dynamic type of the instance is Y. If *jContractor* is enabled this results in the evaluation of the following precondition expression:

$$PreCondition(X,[foo,int\ i]) \vee PreCondition(Y,[foo,int\ i])$$

This ensures that no matter how strict *PreCondition(Y,foo)* might be, as long as the *PreCondition(X,foo)* holds true, `x.foo()` will not raise a precondition exception.

While we are satisfied with this behavior from a theoretical standpoint, in practice a programmer could violate contravariance. For example, consider the following precondition specifications for the `foo()` method defined both in X and Y, still using the example code snippet above:

| | | |
|---|---|---|
| *PreCondition(X, [foo,int a]) :* | $a > 0$ | *(I)* |
| *PreCondition(Y, [foo,int a]) :* | $a > 10$ | *(II)* |

From a specification point of view *(II)* is stricter than *(I),* since for values of a*:* $0<a<=10$, *(II)* will fail, while *(I)* will succeed, and for all other values of a, *(I)* and *(II)* will return identical results. Following these specifications, the call of previous example:

```
x.foo(i);      // where i is 5
```

does not raise an exception since it meets *PreCondition(X,foo,int a).* However, there is a problem from an implementation view, that Y's method `foo(int a)` effectively gets called even though its own precondition specification, *(II),* is violated. The problem here is one of a design error in the contract specification. Theoretically, this error can be diagnosed from the specification code using formal verification and by validating whether following *logical-implication* holds for each redefined method `m()`:

$$PreCondition(ParentClass,m) \Rightarrow PreCondition(SubClass,m)$$

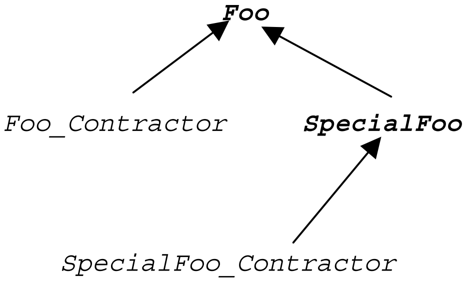
For the previous example, it is easy to prove that *(I)* does not logically-imply *(II).* It is beyond the scope of *jContractor* to do formal verification for logical inference of specification anomalies. *jContractor* does, however, diagnose and report these types of design anomalies, where any one of the *logical-OR'*ed precondition expressions evaluates to *false*. In the above example, *jContractor* would throw an exception to report that the precondition has been illegally strengthened in the subclass, thus forcing the programmer to correct the precondition.

A similar specification anomaly could also occur when a subclass strengthens the parent class's invariants, since *jContractor* checks the class invariants when preconditions are evaluated. The subclass' invariant's runtime violation is caught by *jContractor* instrumented code as an exception, with the correct diagnostic explanation.

The inheritance of postconditions is similar: as a subclass provides a more specialized implementation, it should strengthen, not weaken the postconditions of its interface methods. Any method that is redefined in the subclass should be able to guarantee at least as much as its parent's implementation, and then perhaps some more, due to its specialization. *jContractor* evaluates the *logical-AND* of the postcondition expression found in the subclass with the ones inherited from its parents. Similar anomalies as discussed above for preconditions can also appear in postcondition specifications due to programming errors. *jContractor* will detect these anomalies should they manifest during runtime execution of their respective methods.

### 4.2 Factory Style Instrumentation Issues

When factory style instrumentation is used, *jContractor* constructs a contractor subclass as a direct descendant of the original base class. Therefore, it is possible to pass objects instantiated using the instrumented subclass to any client expecting an instance of the base class. Other than enforcing the contract specifics, an instrumented subclass, say *Foo_Contractor,* has the same interface as the base class, *Foo*, and type-wise conforms to *Foo*. This design allows the contractor subclasses to be used with any polymorphic substitution involving the base class. Consider the following class hierarchy:



*jContractor* allows for the polymorphic substitution of either *SpecialFoo* objects or the instrumented *SpecialFoo_Contractor* objects with *Foo* objects.

## 5 Related Work

The idea of associating boolean expressions (assertions) with code as a means to argue the code's correctness can be traced back to Hoare [2] and others who worked in the field of program correctness. The idea of extending an object-oriented language using only libraries and naming conventions appeared in [3]. The notion of compiling assertions into runtime checks first appeared in the *Eiffel* language [7].

*Eiffel* is an elegant language with built-in language and runtime support for Design By Contract. *Eiffel* integrates preconditions (*require-clause*), postconditions (*ensure-clause*), class invariants, *old* and *rescue/retry* constructs into the definition of methods and classes. *jContractor* is able to provide all of the contract support found in *Eiffel*, with the following differences: *jContractor* supports exception-handling with finer exception resolution – as opposed to a single *rescue* clause; *jContractor* does not support the *retry* construct of *Eiffel.* We believe that if such recovery from an exception condition is possible, it is better to incorporate this handler into the implementation of the method itself, which forestalls throwing the exception at all. *jContractor'* support for *old* supports cloning semantics where references are involved, while Eiffel does not.

Duncan & Hölzle introduced Handshake[1], which allows a programmer to write external contract specifications for Java classes and interfaces without changing the classes themselves. Handshake is implemented as a dynamically linked library and works by intercepting the JVM's file accesses and instrumenting the classes on the fly using a mechanism called binary component adaptation (BCA). BCA is developed for on the fly modification of pre-compiled Java components (class bytecodes) using externally provided specification code containing directives to alter the pre-compiled semantics [5]. The flexibility of the approach allows Handshake to add contracts to classes declared `final`; to system classes; and to interfaces as well as classes. Some of the shortcomings of the approach are that contract specifications are written externally using a special syntax; and that Handshake Library is a non-Java system that has to be ported to and supported on different platforms.

Kramer's *iContract* is a tool designed for specifying and enforcing contracts in Java [4]. Using *iContract*, pre-, postconditions and class invariants can be annotated in the Java source code as "comments" with tags such as: @pre, @post. The *iContract* tool acts as a pre-processor, which translates these assertions and generates modified versions of the Java source code. *iContract* uses its own specification language for expressing the boolean conditions.

Mannion and Philips have proposed an extension to the Java language to support Design By Contract [8], employing *Eiffel*-like keyword and expressions, which become part of a method's signature. Mannion's request that Design By Contract be directly supported in the language standard is reportedly the most popular "non-bug" request in the Java Developer Connection Home Page (bug number 4071460).

Porat and Fertig propose an extension to C++ class declarations to permit specification of pre- and postconditions and invariants using an assertion-like semantics to support *Design By Contract* [9].

## 6 Conclusion

We have introduced *jContractor,* a purely library-based solution to write Design By Contract specifications and to enforce them at runtime using Java. The *jContractor* library and naming conventions can be used to specify the following Design By Contract constructs: pre- and postconditions, class invariants, exception handlers, and *old* references. Programmers can write contracts using standard Java syntax and an intuitive naming convention. Contracts are specified in the form of protected methods in a class definition where the method names and signatures constitute the *jContractor* naming conventions. *jContractor* checks for these patterns in class definitions and rewrites those classes on the fly by instrumenting their methods to check contract violations at runtime.

The greatest advantage of *jContractor* over existing approaches is the ease of deployment. Since *jContractor* is purely library-based, it does not require any special tools such as modified compilers, runtime systems, pre-processors or JVMs, and works with any pure Java implementation.

The *jContractor* library instruments the classes that contain *contract patterns* during class loading or object instantiation. Programmers enable the run-time enforcement of contracts by using a command line switch at start-up, which installs the *jContractor* instrumenting class loader. *jContractor* object factory provides an alternative mechanism that does not require engaging the *jContractor* ClassLoader to instantiate instrumented objects. Clients can instantiate objects directly from the *jContractor* factory, which can use any standard class loader and does not require a command line switch. Either way, clients can use exactly the same syntax for invoking methods or passing object references regardless of whether contracts are present or not. Contract violations result in the method throwing proper runtime exceptions when instrumented object instances are used.

We also describe a novel instrumentation technique that allows accessing the *old* values of variables when writing postconditions and exception handling methods. For example, `OLD.count` returns the value of the attribute `count` at method entry. The instrumentation arranges for the attribute values or expressions accessed through the `OLD` reference to be recorded at method entry and replaces the `OLD` expressions with automatically allocated unique identifiers to access the recorded value.

## References

1. Andrew Duncan and Urs Hölzle. *Adding Contracts to Java with Handshake*. Technical Report TRC98-32, University of California, Santa Barbara, 1998.
2. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

3. Murat Karaorman and John Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*. Vol.36, No.9, September 1993, pp.103-116.

4. Reto Kramer. *iContract – The Java Design by Contract Tool*. Proc. of TOOLS '98, Santa Barbara, CA August 1998. Copyright IEEE 1998.

5. Ralph Keller and Urs Hölzle. *Binary Component Adaptation*. Proc. of ECOOP '98, Lecture Notes in Computer Science, Springer Verlag, July 1998.

6. Bertrand Meyer. *Applying Design by Contract*. In Computer IEEE), vol. 25, no. 10, October 1992, pages 40-51.

7. Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

8. Mike Mannion and Roy Phillips. *Prevention is Better than a Cure*. Java Report, Sept.1998.

9. S.Porat and P.Fertig. *Class Assertions in C++*. Journal of Object Oriented Programming, 8(2):30-37, May 1995.

10. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

# OpenCorba: a Reflective Open Broker

Thomas Ledoux

École des Mines de Nantes
4 rue Alfred Kastler
F-44307 Nantes cedex 3, France
Thomas.Ledoux@emn.fr

**Abstract.** Today, CORBA architecture brings the major industrial solution for achieving the interoperability between distributed software components in heterogeneous environments. While the CORBA project attempts to federate distributed mechanisms within a unique architecture, its internal model is not very flexible and seems not to be suitable for future evolutions. In this paper, we present OpenCorba, a *reflective open broker*, enabling users to adapt dynamically the representation and the execution policies of the software bus.

We first expose the reflective foundations underlying the implementation of OpenCorba: i) metaclasses which provide a better separation of concerns in order to improve the class reuse; ii) a protocol which enables the dynamic changing of metaclass in order to allow run-time adaptation of systems.

Based on this reflective environment, OpenCorba enables the adaptability of the internal characteristics of the broker in order to change its run-time behavior (e.g. remote invocation, IDL type checking, IR error handling). OpenCorba gives a clear example of the benefits of reflective middleware.

## 1 Introduction

In the last couple of years, the success of the Internet emphasizes the need to quickly find solutions for interoperability between distributed heterogeneous environments. Reusability and composability of software components are one of the main concerns for the computer industry dealing with different languages, systems and locations.

By supporting specifications for portable and interoperable object components, the OMG (Object Management Group) consortium proposes a solution to deal with the construction of object-oriented distributed applications [OMG 95]. Indeed, OMG's standardization efforts led to the definition of a whole modular architecture approved by the computer industry. The specifications describe independent modules of a large system, from technical aspects to business objects, reviewing essential distributed services such as security, transactions, event notification, etc. known as the CORBA Services [OMG 97]. The CORBA (Common Object Request Broker Architecture) software bus is the main module of this architecture and has the responsibility of

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 197-214, 1999

achieving a transparent communication between remote objects [OMG 98]. The *modularity* of this architecture is a major advantage of the OMG solution.

However, the complexity of the broker specifications negatively impacts in the intended flexibility of the CORBA model. For example, the invocation mechanism is a "black box" described by fixed specifications. The introduction of a small evolution leads to a new version of the specifications, making obsolete the last one. Dealing with the invocation, the introduction of the interceptors mechanism in CORBA 2.2 and the request for proposal Messaging Service for CORBA 3.0, attempt to improve the existing specifications (and resulting applications). Thus, we can be sceptical about the stability of such improvement. In the current style of distributed systems, we must support the *dynamic* modification of the broker mechanisms to deal with changing contexts of execution (e.g. load balancing, fault tolerance). This dynamic capability of the bus makes possible the evolution of execution policies (e.g. object migration).

We propose an overall solution that allows the object bus to be *adaptive*, making the broker "plug and play". The classification of concurrent and distributed programming proposed by Briot et al. [BRI 98] constitutes an interesting framework for tackling the adaptive issue. The authors distinguish three approaches:

- *library* approach applies object-oriented concepts in order to structure the concurrent and distributed systems through class libraries;
- *integrative* approach consists in unifying concurrent and distributed systems concepts with object-oriented ones;
- *reflective* approach integrates protocol libraries dealing with concurrency and distribution within an object-based programming system.

Using this classification, we can notice that the OMG model both corresponds to an integrative and a library approach, with the minimal object model and the CORBA services perspective respectively. In this taxonomy, the reflective approach is presented as a solution for combining the advantages of the two previous approaches. So, reflection is the best choice for handling the adaptability of the object bus. Reflection [SMI 82] [MAE 87] [KIC 91] allows the extension of the initial OMG model with libraries of meta-protocols customizing mechanisms of distributed programming. Then, it is possible to introduce – in a transparent way – new semantics on the initial model such as concurrency, replication, security, etc. including ones currently unthought of.

In this paper, we present an overview of OpenCorba [LED 98]: a CORBA broker based on a reflective approach. Its architecture enables the reification of the internal characteristics of the software bus in order to modify and adapt them at *run-time*. Then, OpenCorba allows introspection and dynamic modification of the representation and the execution policies of the CORBA bus. Its implementation is based on the reflective language NeoClasstalk [RIV 97]. This language results from an implementation of a MOP (Meta Object Protocol) [KIC 91] in Smalltalk [GOL 89]. Its main contribution consists of an extension of dynamic aspects in Smalltalk: an efficient solution for handling message sending and a way of achieving dynamic behavior of a class.

This paper is presented as follows. In section 2, we explain the reflective foundations underlying the making of OpenCorba and show the advantages of reflection for building open systems. In section 3, after an introduction of OpenCorba itself, we present three possible reflective aspects of the bus. In section 4, we present related work, and in section 5, we discuss about our works in progress. Finally, we draw our conclusions on the contribution of *dynamic adaptability* to middleware.

## 2 Reflective Foundations for Building Open Systems

In this section, we show the benefits of the paradigm of metaclasses for building reusable and adaptable architectures.

### 2.1 Metaclasses and Separation of Concerns

Reflection allows us to separate what an object does (the base level) from how it does it (its meta level) [McA 95]. A reflective language encourages a clean separation between the basic functionalities of the application from its representations and controls. In class-based languages integrating reflective features [COI 87] [DAN 94], the class of a class – a *metaclass* – defines some properties concerning object creation, encapsulation, inheritance rules, message handling, etc. We call *class properties* the properties that denote behavior for classes themselves, independently from the behavior for their instances. In [LED 96], we present a taxonomy of reusable metaclasses which represent class properties such as ensuring that a class has one sole instance (like the pattern Singleton [GAM 95]), a class cannot be subclassed (like Java final [GOS 96]), a class provides pre/post conditions for its methods (like the Eiffel assertions [MEY 92]), etc.

The previous taxonomy was implemented in the MOP NeoClasstalk, which is a based on a new kernel of metaclasses inside the Smalltalk world [RIV 96]. An additional metaclass named `StandardClass`, is defined and strapped into the initial Smalltalk meta-level architecture. `StandardClass` provides a starting point to the NeoClasstalk system. Then, new metaclasses can be derived from `StandardClass`, which describes common behavior for classes, by subclassing it.

**Fig. 1** shows a class `Account`, instance of the metaclass `BreakPoint` that owns the responsibility to set breakpoints in the methods of the class `Account`[1]. Message sending is handled by the method `#execute:receiver:arguments:` of the MOP NeoClasstalk: the metaclass `BreakPoint` intercepts messages received by the instances of `Account`.

[1] It could be interesting to control class interactions with the rest of the system during the debugging phase.

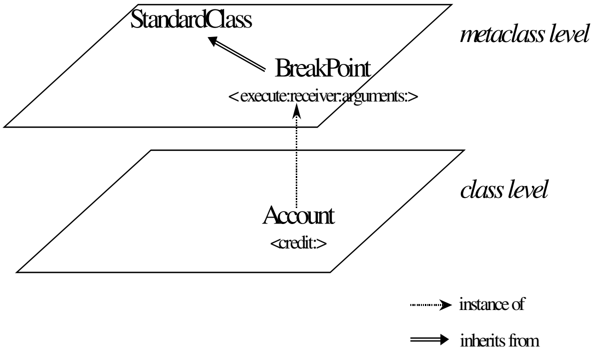


**Fig. 1.** Class properties and (meta)classes

The method `#execute:receiver:arguments:` described above, opens a debugger with the execution context of the trapped method (e.g. `#credit:`), then performs the originally intended method via traditional inheritance mechanisms.

*metaclass code*

```
BreakPoint>>execute: cm receiver: rec arguments: args
   "Set a breakpoint on my instance methods"
   self halt: 'BreakPoint for ', cm selector.
   ^super execute: cm receiver: rec arguments: args
```

*class code*

```
Account>>credit: aFloat
   "Make a credit on the current balance"
   self balance: self balance + aFloat
```

In this way, we avoid the mixing between the business code (bank account) and the code describing a specific property on it (breakpoints). By increasing the separation of concerns, class properties encourage readability, reusability and quality of code. Then, reusable metaclasses propose a better organisation of class libraries for designing open architectures.

### 2.2 Dynamic Change of Metaclass

The dynamic change of class introduced by NeoClasstalk is a protocol that makes it possible for the objects to change their class at run-time [RIV 97][2]. The purpose of

[2] This protocol compensates for the restrictions imposed by the method `#changeClassToThatOf:` found in Smalltalk.

this protocol is to take into account the evolution of the behavior of the objects during their life in order to improve their class implementations. The association of first class objects with this protocol allows the dynamic change of metaclass at run-time. Therefore, it is possible to dynamically add and remove class properties without having to regenerate code.

By taking again the previous example, we can temporarily associate the `BreakPoint` property with a class during its development. The class `Account` changes its original metaclass towards metaclass `BreakPoint`[3] for debugging the messages, then returns towards its former state reversing its change of class (cf. **Fig. 2**).

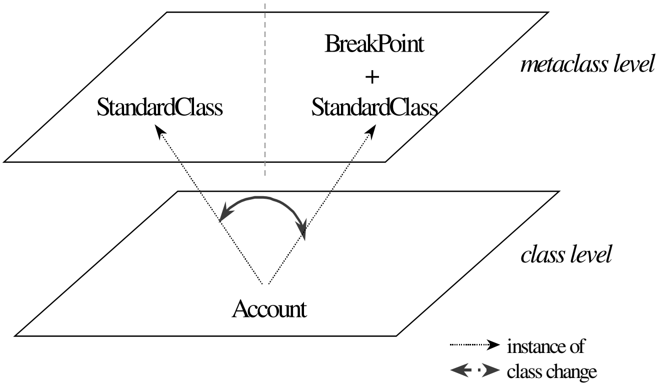


**Fig. 2.** Dynamic adaptability of class properties

The protocol for dynamically changing the class of a class (the metaclass) allows us to replace a class property by another, during execution. In our work dealing with adaptive brokers, this protocol is extremely helpful because it provides a "plug and play" environment for enabling the *run-time* modification of the distributed mechanisms.

# 3 OpenCorba

Our first implementation of an open architecture dealt with the CORBA platform [OMG 98] and gave place to the implementation of a software bus named OpenCorba. The goal of this section is to expose the major reflective aspects of OpenCorba. Others features of OpenCorba like the IDL compiler or the different layers of the broker are described in details in [LED 98].

[3] Or a composition of metaclasses dealing with `BreakPoint` and other class properties (cf. 5.1).

## 3.1 Introduction

OpenCorba is an application implementing the API CORBA in NeoClasstalk. It reifies various properties of the broker – by the means of explicit metaclasses – in order to support the separation of the internal characteristics of the ORB. The use of the dynamic change of metaclass allows to modify the ORB mechanisms represented by metaclasses. Then, OpenCorba is a reflective ORB, which allows to *adapt* the behavior of the broker at *run-time*.

In the following paragraphs, we present three aspects of the bus that have been reified:

1. the mechanism of remote invocation via a proxy;
2. the IDL type checking on the server class;
3. the management of exceptions during the creation of the interface repository[4].

First, we introduce the two basic concepts for creating classes in OpenCorba. This creation deals directly with the reflective aspects.

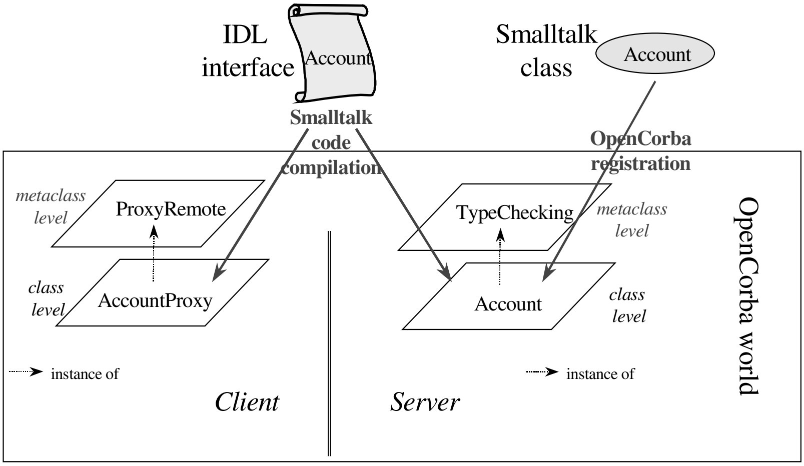


**Fig. 3.** Creation of OpenCorba classes

### IDL Mapping in OpenCorba.

Following the Smalltalk mapping described in the CORBA specifications [OMG 98], the OpenCorba IDL compiler generates a proxy class on the client side and a template class on the server side. The proxy class is associated with the metaclass

[4] Let us recall that the interface repository (IR) is similar to a run-time database containing all the IDL specifications reified as objects.

`ProxyRemote` implementing the remote invocation mechanisms; the template class, with the metaclass `TypeChecking` which implements the IDL type checking on the server. The left hand of **Fig. 3** shows the results from IDL mapping of the `Account` interface in OpenCorba: the proxy class `AccountProxy` is instance of `ProxyRemote` and the template class `Account` is instance of `TypeChecking`.

### Feature Smalltalk2OpenCorba.

OpenCorba allows the Smalltalk developers to transform any Smalltalk standard class into a Smalltalk server class in the ORB. This feature was introduced to reuse existing Smalltalk code, and to free the programmer from the writing of the bulk of IDL specification. Technically, a semi-automatic process is applied to the Smalltalk class in order to generate the interface repository and the IDL file. Then, to become a server class in OpenCorba, the class must be an instance of `TypeChecking` (cf. **Fig. 3**, right hand). By analogy with similar techniques, we called this special feature Smalltalk2OpenCorba.

## 3.2 The OpenCorba Proxy

In distributed architectures, the proxy object is a local representation in the client side of the server object. Its purpose is to ensure the creation of the requests and their routing towards the server, then to turn over the result to the client. In order to remain transparent, a proxy class adapts the style of local call to the mechanism of remote invocation [SHA 86].

### Separation of Concerns.

The remote invocation mechanism is completely independent of the semantics of the IDL interface (e.g. `Account`). That is, remote invocation refers to the control of the application and not to the application functionality; it deals with meta-level programming. The OpenCorba IDL compiler automatically generates the proxy class on the basic level of the application and associates the proxy class with the metaclass `ProxyRemote` in charge of calling the real object. The remote invocation remains thus transparent for the client. We can manipulate the proxy class with the traditional Smalltalk tools (browser, inspector).

*Base level features.* Methods of the proxy class are purely descriptive and represent interfaces like IDL operations [LED 97]. In the Smalltalk browser, the methods do not contain any code, only one comment denoting that OpenCorba generated them. **Table 1** presents some examples of mapping for OpenCorba. The type of the returned values is put in comments to announce IDL information to the programmer.

**Table 1.** Examples of IDL mapping

| ***IDL attribute/operation*** | ***Methods of a proxy class OpenCorba*** |
|---|---|
| `readonly attribute float balance;` | `balance`<br>*"Generated by OpenCORBA*<br>****   DO NOT EDIT   ***"*<br>"^aFloat " |
| `void credit(in float amount);` | `credit: aFloat`<br>*"Generated by OpenCORBA*<br>****   DO NOT EDIT   ***"*<br>"^nil" |

*Meta level features.* By definition, the sending of a message to a proxy object involves a remote message send to the server object it represents. The idea is to intercept the message at its reception time by the proxy object and to launch a remote invocation. Thus, the control of the message sending is suitable for our purpose. The metaclass `ProxyRemote` redefines the method `#execute:receiver:arguments:` of the MOP NeoClasstalk to intercept the messages received by a proxy. This redefinition carries out the remote invocation by using the DII CORBA API according to [OMG 98].

### Dynamic Adaptability.

To allow the dynamic adaptability of the invocation mechanisms in OpenCorba, it is possible to develop others metaclasses. We distinguished them in two categories:

- The first one deals with the possible variations on the proxy mechanisms. We think of policies modelling Java RMI [SUN 98], a future version of the CORBA DII or a local invocation. This last mechanism was implemented in OpenCorba in order to model proxies with cache.
- The second category considers extensions of the proxy concept for introducing new mechanisms like object migration [JUL 88] [OKA 94] or active replication [BIR 91] [GUE 97].
    - Object migration consists in transferring a server object towards the client side in order to optimize the performances of the distributed system. This mechanism reduces the bottlenecks of the network and minimizes the remote communications.
    - Replication is another mechanism of administration of the objects distribution. It consists of a duplication of the server in several replicas, which are the exact representation of the original server object. The mechanism of active replication supposes that the message is sent by the client to the replicas – via the proxy – thanks to an atomic protocol of diffusion (broadcast).

Thus, the dynamic adaptability of metaclasses allows to implement some variations on the remote invocation mechanism, without upsetting existing architecture.

## 3.3 IDL Type Checking on the Server Classes

The CORBA standard [OMG 98] specifies that the server side uses the interface repository to check the conformity of the signature of a request by checking the argument and return objects. On the other hand, it does not specify how it must be carried out.

### Separation of Concerns.

We are convinced that it falls within the competence of the server class to check if one of its methods can be applied or not. Indeed, if this kind of checking could be carries out more upstream by the ORB (during the unmarshalling of request by the server for example), that would introduce a more inflexible management: a little change in the mechanism of control involves a rewriting of the lower layers of the broker. On the contrary, we propose an externalisation of the control of these layers towards the server class, which will test if the application of a method is possible or not.

Moreover, the type checking is independent of the functionalities defined by the server class: it can be separated from the base code and constituted as a class property that will be implemented by a metaclass. Thus, the code of the server class does not carry out any test on the type of the data in arguments. The developer can then write, modify or recover its code without worrying about the control of the type handled by the metaclass `TypeChecking`.

Technically, this metaclass controls the message sending on the server class – via the method `#execute:receiver:arguments:` – to interrogate the interface repository before and after application of its methods. This query enables us to check the type of the arguments and the type of the result of the method.

In conclusion, OpenCorba externalises the type checking, at the same time from the lower layers of the ORB, and from the server class.

### Dynamic Adaptability.

Thanks to our approach, we can bring new mechanisms for type checking, without modifying the existing implementation. For example, we can develop a new metaclass managing a system of cache for the types of the parameters. During the first query of the interface repository, OpenCorba locally backs up the type of each argument of the method[5]. Then, with the next invocations of the method, the metaclass questions memorized information in order to carry out the type checking.

Another example: we can also remove the control of the type for reasons of performance or when the type of the parameters is known before hand. A dynamic

[5] For example, in a Smalltalk shared dictionary or in the byte code of the compiled method.

change of metaclass allows then to associate the server class with the default metaclass of the OpenCorba system (i.e. `StandardClass`).

### 3.4 Interface Repository and Error Handling

There are two ways used to populate the interface repository: the IDL mapping and the feature Smalltalk2OpenCorba. In the first case, the generation of each proxy class is handled in the creation of the objects in the repository. In the second case, this creation is allowed by a table of Smalltalk equivalence towards IDL (*retro-mapping*). Technically, these two mechanisms do not have the same degree of intercession:

- In the case of Smalltalk2OpenCorba, the Smalltalk compiler already carried out the syntactic analysis and the semantic checking of the class. Also, the installation of a Smalltalk class in the ORB does not cause errors during the creation of the objects in the repository;
- On the other hand, in the IDL case, it is an authentic compilation where the semantic actions check the integrity of IDL specifications before the creation of objects in the repository (e.g. the same attribute duplicated).

Thus, in order to distinguish the Smalltalk case from the IDL case where creation can lead to error handling, we must encapsulate the CORBA creational APIs of the interface repository. Let us recall that these creational APIs are implemented by the *container* classes of the interface repository specified by CORBA standard [OMG 98].

**Separation of Concerns.**

By analyzing the tests of integrity necessary for the IDL compilation, it appears that they are generic and independent from the creational APIs of the *containers*. Thus, they can be externalized from the *container* classes to constitute a class property which will be implemented by a metaclass. The code of the *container* classes does not carry out any tests. It is then helpful to differentiate the IDL case from the Smalltalk case, in order to associate a given metaclass or not to the *container* classes.

The metaclass `IRChecking` plays this role. It specializes the method `#execute:receiver:arguments:` of the MOP NeoClasstalk in order to intercept the messages received by the instances of the *container* class. For the creational APIs, the integrity tests are carried out and an exception is raised if an error occurs.

**Dynamic Adaptability.**

Our design leads to a greater flexibility to carry out or transform the integrity tests without having to modify the creational APIs. Thus, it allows the distinction between the Smalltalk case and the IDL case at run-time. In the Smalltalk case, creation is carried out normally (default metaclass `StandardClass`); in the IDL case, there is a dynamic adaptability of the behavior to carry out creation only if the integrity tests do not raise an error (metaclass `IRChecking`). OpenCorba connects the appropriate metaclasses at run-time according to needs of the system.

### 3.5 Implementation and Performance Issues

Reflective aspects of OpenCorba are essentially based in the ability of handling message sends. Therefore, the extra cost of the reflective broker is highly dependent on the performance of the message sending.

The implementation of this control in NeoClasstalk is based on a technique called *method wrappers* [BRA 98]. The main idea of this feature is the following: rather than changing the method lookup process directly at run-time, we modify the compiled method objects that the lookup process returns. In NeoClasstalk, the method wrappers deal both with compile-time and run-time reflection. We briefly describe the two steps involved:

- *At compile-time*

  The original method defined in a class is wrapped so that it sends to the class itself the message `#execute:receiver:arguments:` defined in its class (the metaclass). The arguments are i) the original compiled method which is stored in the method wrapper, ii) the object originally receiving the message, iii) the arguments of the message.

- *At run-time*

  Method lookup is unmodified: the activation of the method wrapper does the call to the metaclass and starts the meta-level processing.

By analysing these two steps, we conclude that the cost is actually significant. First, the meta-level indirection occurring at run-time involves additional method invocations. Since the meta-level indirection is applied to each message send, the performance of the whole system decreases. However, we can notice that since OpenCorba deals with distributed environment, the network traffic moderates the impact of this overhead.

Secondly, at compile-time, there is a real difficulty in applying the method wrappers concept: which are the methods that we need to wrap? It depends on the problem at hand. For example, in OpenCorba, all the methods of a server class and all the inherited ones must be wrapped. Thus, the technique of method wrappers imposes some decisions at design time, which imply a cost that should not be overlooked.

In summary, the flexibility provided by reflection impacts on the system efficiency. Many works attempt to find a way for efficient reflective systems such as the optimization of virtual machine for meta-programming, the reification categories which provide the opportunity to specifically reify a given class [GOW 96], the partial evaluation for MOP [MAS 98], etc.

# 4 Related Work

## 4.1 Reflective Adaptive Middleware

Like the OpenCorba broker, other research projects are under development in the field of adaptive middleware. The ADAPT project of the University of Lancaster has investigated the middleware implementation for mobile multimedia applications which are capable of dynamically adapting to QoS fluctuations [BLA 97]. Its successor, the OpenORB project, studies the role of reflection in the design of middleware platforms [BLA 98]. The implementation of the current reflective architecture is based on a per-object meta-space (structured as three distinct meta-space models) and on the concept of open bindings, differing from the per-class reflective environment and the MOP of OpenCorba. These two design approaches result in two different ways of investigating reflective middleware.

The FlexiNet platform [HAY 98], a Java middleware, proposes to reify the layers of the communication stack into different meta-objects (in order to customize them). Each meta-object represents a specific aspect such as call policy, serialization, network session, etc. OpenCorba currently reifies characteristics dealing the upper layers of the invocation mechanism, and could reify the lower layers (e.g. marshalling, transport).

Researchers at the University of Illinois developed dynamicTAO, a CORBA-compliant reflective ORB that supports run-time reconfiguration [ROM 99]. Specific strategies are implemented and packed as dynamically loadable libraries, so they can be linked to the ORB process at run-time. Rather than implementing a new ORB from scratch as done in OpenCorba, they chose to use TAO [SCH 99], causing a dependency to this ORB.

Finally, since its experiment in the development of MOP [GOW 96], the distributed system team of the Trinity College of Dublin has recently started the Coyote project whose goal is to provide the adaptation of distributed system (e.g. administration of telecommunication network, CORBA bus).

## 4.2 Reflection in Distributed Systems

The use of reflection in the concurrent and distributed systems is certainly not recent [WAT 88], but seems to take a new rise in the last years. Indeed, many research projects in the field of reflection or in the distributed domain use the reification of the distributed mechanisms to modify them and specialize them. Let us quote the mechanisms of migration [OKA 94], marshalling [McA 95], replication [GOL 97], security [FAB 97], etc. The CORBA architecture is a federate platform of the various mechanisms of distribution. Also, it seems interesting to study these projects to implement their mechanisms within OpenCorba.

### 4.3 Programming with Aspects

The mixing in the system's basic functionality of several technical aspects (e.g. distribution, synchronization, memory management) dealing with the application domain, constitutes one of the major obstacles to the reusability of the software components. A possible solution is then to consider the isolation of these specific aspects for their individual reuse. The "tangled" code is separated and aspects can evolve independently thus formulating the paradigm of separation of concerns [HUR 95].

There are some models and techniques allowing the separation of concerns: composition filters [BER 94], adaptive programming [LIE 96], aspect-oriented programming (AOP) [KIC 97]. However, the latter implement particular constructs to achieve the programming with aspects. We preferred a reflective approach that does not impose a new model, but extends the existing languages to open them. Moreover, contrary to these models, our solution allows the dynamic adaptability of aspects.

## 5 Work in Progress

### 5.1 Metaclass Composition

Distributed architectures are complex and require many mechanisms for their implementation. The question of how to compose these mechanisms is essential. For example, the functionality of "logging within a class the remote invocations" uses the mechanism of "logging" combined with the mechanism of "remote invocation". As we have seen previously, each of these mechanisms corresponds with a class property and is implemented by a metaclass. Thus, the combination of several mechanisms raises the problem of the composition of the metaclasses: this composition causes conflicts a priori when there is a behavior overlap.

To tackle this problem, our recent work consists in the definition of a "metaclass compatibility model" in order to offer a reliable framework for the composition of the metaclasses [BS 98]. We attempt to define a classification of the various distribution mechanisms to prevent possible overlappings of behavior.

### 5.2 Towards Specifications of a Reflective Middleware

By studying concurrent and distributed architectures, we can notice that they deal with well-known mechanisms and policies, which are independent from the system's basic functionality (i.e. business objects) and could be implemented at the meta-level. Then, we can suggest a first classification of middleware features related to reflective components:

- Distribution
  proxy mechanism, replication, migration, persistence, etc.

- Communication
  synchronous, asynchrounous, no reply, future, multicast, etc.
- Object concurrency
  inter-objects, intra-object (readers/writer model, monitor), etc.
- Reliability
  exceptions, transactions, etc.
- Security
  authentification, authorisation, licence, etc.
- Thread management
  single-threaded, thread-per-message, thread-pool, etc.
- Data transport
  sockets, pipes, shared memory, etc.

The richness of future middlewares will depend on their capabilities to dynamically adapt and (fine) tune such features. Then, we are strongly convinced that the first step to build such a middleware is to find a good reflective model and environment.

In OpenCorba, we experimented with a per-class reflective environment on the top of Smalltalk. We are currently investigating other models like meta-object or message reification models in order to compare them with our metaclass approach. We noticed that the design of reflective distributed mechanisms could be common to any reflective language dealing with similar features. For example, the handling of the message sending is an important feature because several mechanisms have to deal with it (in their implementation). Then, most of the reflective languages support the *same design* of the meta-level in order to implement a given distributed mechanism (e.g. primary replication [CHI 93] [McA 95] [GOL 97]).

To reason about a language independent environment, we sketched out an abstract MOP allowing the control and customization of the message sending and state accessing. Thus, we plan to describe the specifications of distributed mechanisms in the context of meta-programming, i.e. reflective distributed components.

## 6 Conclusion

In this paper, we developed the idea in which reflection is a tremendous vector for the building of open architectures. Languages considering classes as full entities allow separation of concerns through metaclasses, improving reusability and quality of code. Then, associated to the dynamic change of class, metaclasses offer an exclusive reflective scheme for the building of reusable and adaptable, open architectures.

In the context of distributed architectures, these reflective foundations offer a dynamic environment allowing the reuse and the adaptability of mechanisms related to the distribution. Our first experimentation enabled us to "open" some internal characteristics of the CORBA software bus, such as the invocation mechanism. The result – OpenCorba – is an open broker capable of dynamically adapting the policies of representation and execution within the CORBA middleware.

**Acknowledgements**

The research described in this paper was funded by IBM Global Services (France) during the author's PhD (1994-1997). The author wishes to thanks Noury Bouraqadi-Saâdani, Pierre Cointe, Fred Rivard for many discussions about OpenCorba, Xavier Alvarez for his reviewing of the final version of the paper.

## References

[BER 94] BERGMANS L. — *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, Netherlands, June 1994.

[BIR 91] BIRMAN K., SCHIPER A., STEPHENSON P. — Lightweight Causal and Atomic Group Multicast. In *ACM Transactions on Computer Systems*, vol.9, n°3, p.272-314, 1991.

[BLA 97] BLAIR G.S., COULSON G., DAVIES N., ROBIN P., FITZPATRICK T. — Adaptive Middleware for Mobile Multimedia Applications. In *Proceedings of the 8th International Workshop on NOSSDAV'97*, St-Louis, Missouri, May 1997.

[BLA 98] BLAIR G.S., COULSON G., ROBIN P., PAPATHOMAS M. — An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98,* Springer-Verlag, p.191-206, N. Davies, K. Raymond, J. Seitz Eds, September 1998.

[BRA 98] BRANT J., FOOTE B., JOHNSON R., ROBERTS D. — Wrappers to the Rescue. In *Proceedings of ECOOP'98*, Springer-Verlag, Brussels, Belgium, July 1998.

[BRI 98] BRIOT J.P., GUERRAOUI R., LÖHR K.P. — Concurrency and Distribution in Object-oriented Programming. In *ACM Computer Surveys*, vol.30, n°3, p.291-329, September 1998.

[BS 98] BOURAQADI-SAÂDANI N., LEDOUX T., RIVARD F. — Safe Metaclass Programming. In *Proceedings of OOPSLA'98*, ACM Sigplan Notices, Vancouver, Canada, October 1998.

[CHI 93] CHIBA S., MASUDA T. — Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP'93*, p.482-501, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993.

[COI 87] COINTE P. — Metaclasses are First Class : the ObjVlisp Model. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.156-167, Orlando, Florida, October 1987.

[DAN 94] DANFORTH S., FORMAN I.R. — Reflections on Metaclass Programming in SOM. In *Proceedings of OOPSLA'94*, ACM Sigplan Notices, Portland, Oregon, October 1994.

[FAB 97] FABRE J.C, PÉRENNOU T. — A metaobject architecture for fault tolerant distributed systems: the FRIENDS approach. In *IEEE Transactions on Computers*. Special Issue on Dependability of Computing Systems, vol.47, n°1, p.78-95, January 1998

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES John — *Design Patterns*, Addison-Wesley Reading, Massachusetts, 1995.

[GOL 89] GOLDBERG A., ROBSON D. — *Smalltalk-80 : The Language*, Addison-Wesley, Reading, Massachusetts, 1989.

[GOL 97] GOLM M. — *Design and Implementation of a Meta Architecture for Java*. Master's Thesis, University of Erlangen, Germany, January 1997.

[GOS 96] GOSLING J., JOY B., STEELE G. — *The Java Language Specification*. The Java Series, Addison-Wesley, Reading, Massachusetts, 1996.

[GOW 96] GOWING B., CAHILL V. — Meta-Object Protocols for C++ : The Iguana Approach. In *Proceedings of Reflection'96*, Ed. Kiczales, San Francisco, California, April 1996.

[GUE 97] GUERRAOUI R., SCHIPER A. — Software Based Replication for Fault Tolerance. In *IEEE Computer*, vol.30 (4), April 1997.

[HAY 98] HAYTON R., HERBERT A., DONALDSON D. — FlexiNet - A flexible component oriented middleware system. In *Proceedings of ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HUR 95] HÜRSH W.L., LOPES C.V. — *Separation of Concerns*. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[JUL 88] JUL E., LEVY H., HUTCHINSON N., BLACK A. — Fine-grained mobility in the Emerald system. In *ACM Transactions on Computer Systems*, vol.6(1), p.109-133, February 1988.

[KIC 91] KICZALES G., DES RIVIERES J., BOBROW D.G.— *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.M, IRWIN J. — Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241, Springer-Verlag, p.220-242, Jyväskyla, Finland, June 1997.

[LED 96] LEDOUX T., COINTE P. — Explicit Metaclasses as a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96*, LNCS 1049, p.38-55, Springer-Verlag, Kanazawa, Japan, March 1996.

[LED 97] LEDOUX T. — Implementing Proxy Objects in a Reflective ORB. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finland, June 1997.

[LED 98] LEDOUX T. — *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk*. PhD thesis, Université de Nantes, École des Mines de Nantes, March 1998. *(in french)*

[LIE 96] LIEBERHERR K.J. — *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[MAE 87] MAES P. — Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.147-155, Orlando, Florida, October 1987.

[MAS 98] MASUHARA H., YONEZAWA A. — Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP'98*, LNCS 1445, p.418-439, Springer-Verlag, Brussels, Belgium, July 1998.

[McA 95] MCAFFER J. — Meta-level architecture support for distributed objects. In *Proceedings of IWOOOS'95*, p.232-241, Lund, Sweden, 1995.

[MEY 92] MEYER B. — *Eiffel: The Language*. Prentice Hall, second printing, 1992.

[OKA 94] OKAMURA H., ISHIKAWA Y. — Object Location Control Using Meta-level Programming. In *Proceedings of ECOOP'94*, p.299-319, LNCS 821, Springer-Verlag, July 1994.

[OMG 95] OBJECT MANAGEMENT GROUP — *Object Management Architecture Guide*, Revision 3.0. OMG TC Document ab/97-05-05, June 1995.

[OMG 97] OBJECT MANAGEMENT GROUP — *CORBAservices: Common Object Services Specification*. OMG TC Document formal/98-07-05, November 1997.

[OMG 98] OBJECT MANAGEMENT GROUP — *The Common Object Request Broker : Architecture and Specification*, Revision 2.2. OMG TC Document formal/98-07-01, February 1998.

[RIV 96] RIVARD F. — A New Smalltalk Kernel Allowing Both Explicit and Implicit Metaclass Programming. In *Workshop "Extending the Smalltalk Language"*, OOPSLA'96, San Jose, California, October 1996.

[RIV 97] RIVARD F. — *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Université de Nantes, École des Mines de Nantes, June 1997. *(in french)*

[ROM 99] ROMAN M., KON F., CAMPBELL R.H. — Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case. In *Workshop on Middleware*, ICDCS'99, Austin, Texas, May 1999.

[SCH 99] SCHMIDT D., CLEELAND C. — Applying Patterns to Develop Extensible ORB Middleware. In *IEEE Communications Magazine*, 1999. (to appear)

[SHA 86] SHAPIRO M. — Structure and Encapsulation in Distributed Systems : The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, p.198-204, Cambridge, MA, May 1986.

[SMI 82] SMITH B.C. — *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, January 1982.

[SUN 98] SUN MICROSYSTEMS — *Java Remote Method Invocation (RMI)*. At http://java.sun.com/products/jdk/rmi/index.html

[WAT 88] WATANABE T., YONEZAWA A. — Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*, ACM Sigplan Notices, p.306-315, San Diego, California, September 1988.

# OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection

Yukihiko Sohda, Hirotaka Ogawa, and Satoshi Matsuoka

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Oo-okayama, Meguro-ku, Tokyo 152-8552
TEL: (03)5734-3876 FAX: (03)5734-3876
email: {sohda,ogawa,matsu}@is.titech.ac.jp

**Abstract.** Platform portability is one of the utmost demanded properties of a system today, due to the diversity of runtime execution environment of wide-area networks, and parallel programs are no exceptions. However, parallel execution environments are VERY diverse, could change dynamically, while performance must be portable as well. As a result, techniques for achieving platform portability are sometimes not appropriate, or could restrict the programming model, e.g., to simple message passing. Instead, we propose the use of *reflection* for achieving platform portability of parallel programs. As a prototype experiment, a software DSM system called OMPC++ was created which utilizes the compile-time metaprogramming features of OpenC++ 2.5 to generate a message-passing MPC++ code from a SPMD-style, shared-memory C++ program. The translation creates memory management objects on each node to manage the consistency protocols for objects arrays residing on different nodes. Read- and write- barriers are automatically inserted on references to shared objects. The resulting system turned out to be quite easy to construct compared to traditional DSM construction methodologies. We evaluated this system on a PC cluster linked by the Myrinet gigabit network, and resulted in reasonable performance compared to a high-performance SMP.

## 1 Introduction

Due to rapid commoditization of advanced hardware, parallel machines, which had been specialized and of limited use, are being commoditized in the form of workstation and PC clusters. On the other hand, commodity software technologies such as standard libraries, object-orientation, and components are not sufficient for guaranteeing that the same code will work across all parallel platforms not only with the same set of features but also similar performance characteristics, similar fault guarantees, etc. Such *performance portability* is fundamentally difficult because platforms differ in processors, number of nodes, communication hardware, operating systems, libraries, etc., despite commoditization.

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 215-234, 1999

Traditionally, portability amongst diverse parallel computers have been either achieved by standard libraries such as MPI, or parallel programming languages and compilers such as HPF and OpenMP[Ope97]. However, such efforts will could require programming under a fixed programming model. Moreover, portable implementation of such systems themselves are quite difficult and require substantial effort and cost. Instead, *Reflection* and *open compilers* could be alternative methodologies and techniques for *performance portable* high-performance programs.

Based on such a belief, we are currently embarked on the OpenJIT[MOS$^+$98] project. OpenJIT is a (reflective) Just-In-Time open compiler written almost entirely in Java, and plugs into the standard JDK 1.1.x and 1.2 JVM. At the same time, OpenJIT is designed to be a compiler framework similar to Stanford SUIF[Uni], in that it facilitates user-customizable high-level and low-level program analysis and transformation frameworks. With OpenJIT, parallel programs of various parallel programming models in Java compiled into Java bytecode, will be downloaded and executed on diverse platforms over the network, from single-node computers to large-scale parallel clusters and MPPs, along with customization classes for respective platforms and programming models using compiler metaclasses[1].

The question is, will such a scheme be feasible, especially with strong requirements for performance of high-performance parallel programs? Moreover, how much metaprogramming effort would such an approach take? So, as a precursor work using OpenC++, we have employed reflection to implement DSM (distributed shared memory) in a portable way, to support Java's chief model of parallel programming i.e., the multithreaded execution over shared memory. More specifically, we have designed a set of compiler metaclasses and the supportive template classes and runtimes for OpenC++2.5[Chi95, Chi97] that implements necessary program transformations with its compile-time MOP for efficient and portable implementation of software-based DSM for programs written in (shared-memory) SPMD style, called *OMPC++*. A multithreaded C++ program is transformed into message-passing program in MPC++[Ish96] level0 MTTL (multithread template library), and executed on our RWC(Real-World Computing Partnership)-spec PC-cluster, whose nodes are standard PCs but interconnected with the Myrinet[Myr] gigabit network, and running the RWC's SCore parallel operating system based on Linux.

The resulting OMPC++ is quite small, requiring approximately 700 lines of metaclass, template class, and runtime programming. Also, OMPC++ proved to be competitive with traditional software-based DSM implementations as well as hardware-based SMP machines. Early benchmark results using numerical core programs written in shared-memory SPMD-style programs (a fast parallel CG-kernel, and parallel FFT from SPLASH2) shown that, our reflective DSM implementation scales well, and achieves performance competitive with that of high-performance SMPs (SparcServer 4000, which has dedicated and expensive

[1] OpenJIT is in active development, and is currently readying the first release as of Feb. 1, 1999.

hardware for maintaining hardware memory consistency). Not only this result serves as a solid groundwork for OpenJIT, but OMPC++ itself serves as a high-performance and portable DSM in its own right.

## 2 Implementation of a Portable, High-Performance Software DSM

DSM (Distributed Shared Memory) has had multitudes of studies since its original proposal by Kai Li[LH89], with proposals for various software as well as hardware assisted coherency protocols, techniques for software-based implementation, etc. However, there have not been so much work with platform (performance) portability in mind. In order for a program to be portable, one must not extensively alter the underlying assumptions on the hardware, OS, the programming language, which are largely commoditized. Moreover, as a realistic parallel system, one must achieve high performance and its portability across different platforms.

Below, we give an outline of our OMPC++ system with the above requirements in mind, as well as the building blocks we have employed.

### 2.1 Overview of the OMPC++ system

OMPC++ takes a parallel multithreaded shared-memory C++ program, transforms the program using compile-time metaprogramming, and emits an executable depending on various environments, those with fast underlying message passing in particular(Fig. 2). OMPC++ itself does not extend the C++ syntax in any way.

More concretely, OMPC++ defines several template classes (distributed shared array classes), and OpenC+ metaclasses for performing the necessary program transformations for implementing software DSM (such as read/write barriers to shared regions, and initialization/finalization). The OpenC++ compiler generates a customized program transformer, which then transforms the program into message-passing MPC++ program, which is further compiled into C++ and then onto a binary, and finally linked with DSM template libraries as well as MPC++ runtime libraries, resulting in an executable on the RWC cluster.

Our system exhibits competitive and sometimes superior performance to software DSM systems implemented as class libraries. This is because OMPC++ can analyze and pinpoint at compile time, exactly where we should insert runtime meta-operations (such as read/write barriers) that would result in performance overhead. Thus, we only incur the overhead when it is necessary. On the other hand, for traditional class-based DSM systems, either the programmer must insert such meta-operations manually, or it would incur unnecessary runtime overhead resulting in loss of performance.

### 2.2 Underlying 'Building-Blocks'

We next describe the underlying 'building-blocks', namely OpenC++, MPC++, SCore, and the PC Cluster hardware(Fig. 1). We note that, aside from OpenC++, the components could be interchanged with those with similar but different functionalities and/or interfaces, and only the metaclasses have to be re-written. For example, one could easily port OMPC++ to other cluster environments, such as Beowulf[Beo].

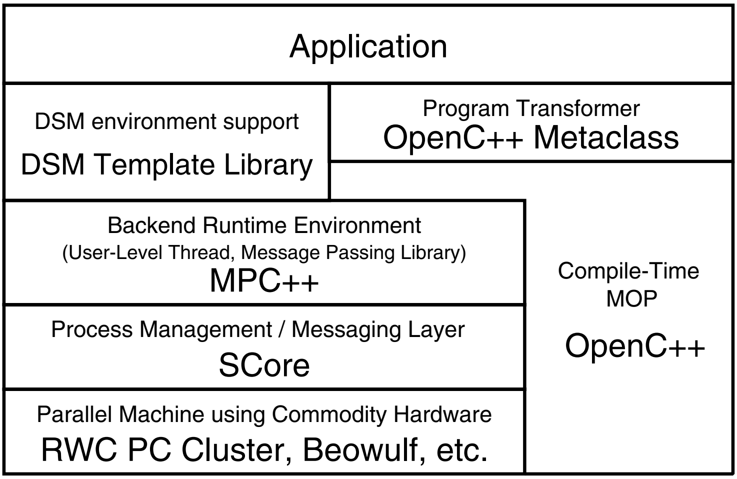

Application
DSM environment support
DSM Template Library
Program Transformer
OpenC++ Metaclass
Backend Runtime Environment
(User-Level Thread, Message Passing Library)
MPC++
Process Management / Messaging Layer
SCore
Parallel Machine using Commodity Hardware
RWC PC Cluster, Beowulf, etc.
Compile-Time
MOP
OpenC++

**Fig. 1.** Building-blocks hierarchy

**OpenC++** OpenC++ provides a compile-time MOP capability for C++. By defining appropriate compiler metaclasses, the OpenC++ compiler effectively generates a preprocessor that performs the intended program transformations. The preprocessor then can be applied to the base program for appropriate meta-operations. Compile-time MOP could eliminate much of the overhead of reflection, and coupled with appropriate runtime, could simulate the effects of traditional run-time MOPs. (OMPC++ currently employs OpenC++ 2.5.4.)

**MPC++** MPC++ 2.0 Level 0 MTTL is a parallel object-based programming language which supports high-level message-passing using templates and inheritance. MTTL is solely implemented using language features of C++, and no custom language extensions are done. For OMPC++, we employ most of the features of MTTL, such as local/remote thread creation, global pointers, and reduction operations[2].

[2] MPC++ Level 1[Ie96] supports compile-time MOP features similar to OpenC++. The reason for not utilizing them was not a deep technical issue, but rather for

**RWC PC Cluster and SCore** The default OMPC++ library emits code for RWC PC cluster, although with a small amount of metaclass reprogramming it could be ported to other cluster platforms. The cluster we employed is a subset of the RWC cluster II, with 8–10 Pentium Pro (200MHz) PC nodes interconnected with Myrinet gigabit LAN. The nodes in the cluster communicate with each other using PM, a fast message-passing library, and the whole cluster acts as one parallel machine using the SCore operating system based on Linux. SCore is portable in that it only extends Linux using daemons and device drivers.

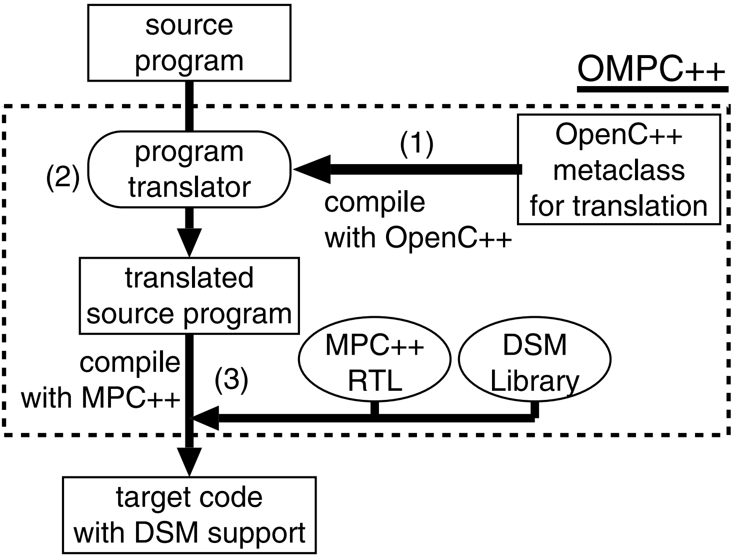


**Fig. 2.** Program Transformation and Compilation

## 3 Implementation of Software DSM in OMPC++

We outline the process of compilation and execution in OMPC++(Fig. 2).

- OpenC++ generates a preprocessor from the OMPC++ metaclasses (1).
- The source program is transformed using the preprocessor (2). The transformed program is a message-passing MPC++ program.
- MPC++ adds compiler variables such as the SCore library, and hands it off to the backend C++, which generates the executable (3).

practicality at the time of OMPC++ development such as stability and compilation speed. In theory we could replicate our work using MPC++ Level 1.

A real compilation session is depicted in Fig. 3. The numbers correspond to those in Fig. 2.

```
% ompc++ -m -- SharableClass.mc                   |(1)

% ompc++ -v -- -o cg cg.cc problem.o              |
[Preprocess...  /usr/score/bin/mpc++              |
  -D__opencxx -E -o cg.occ -x c++ cg.cc]          |
compiling  cg.cc                                  |(2)
[Translate... cg.occ into: cg.ii]                 |
Load SharableClass-init.so.. Load                 |
  SharableClass.so.. Initialize.. Done.           |

[Compile...  /usr/score/bin/mpc++                 |
                  -o cg cg.ii problem.o]          |
compiling  cg.ii                                  |(3)
[done.]                                           |
[GC happened 10 times.]                           |
```

**Fig. 3.** A Compilation Session for OMPC++

### 3.1 Program Transformation

There are 3 steps in the program transformation for implementing the DSM functionality:

1. Initialization and finalization of memory management objects created on each node.
2. Initialization of shared memory regions (creation of management tables, allocation of local memory).
3. Insertion of locks and read/write barriers to shared variables in the program.

The examples of program transformations are depicted in Fig. 4 and Fig. 5[3]. The program transformation possible with OpenC++ is restricted to transformation of base programs, and in particular those of class methods; so, for OMPC++, we assume the user program to conform to the following structure:

```
sdsm_main() { .. };   // main computing code
sdsm_init() { .. };   // application-specific initialization
sdsm_done() { .. };   // application-specific finalization
```

[3] Note that, there are restrictions of the template features in the metaclasses for the current version of OpenC++, forcing us to do some template expansions manually.

sdsm_init() is called following the system-level DSM initialization. sdsm_main() function is the body of parallel code, and is called independently in parallel on each PE after initialization of the shared memory region. sdsm_done() is called after sdsm_main() terminates, followed by system-level DSM finalization, after which the whole program terminates. sdsm_init() and sdsm_done() is sequentially executed on a single PE (normally PE 0), while sdsm_main() is executed on all the PEs.

The following functions are added as a result of program transformation:

```
mpc_main() { .. };          // the main MPC++ function
Initialize() { .. };        // initialization of shared region
Finalize() { .. };          // finalization: freeing of shared region
```

Because the resulting code of program transformation is MPC++ code, the program starts from mpc_main(). This in turn calls Initialize(), which then calls the application-specific sdsm_init(). After the computation completes, sdsm_done() is called, and finally Finalize() is executed on all the processors, and the program terminates.

```
Shared<double> a(100);                          // ..(1)
Shared<double> b;
sdsm_main() {
  b = (double*)malloc(sizeof(double) * 10); // ..(2)
  b[0] = a[0];  ...
};
sdsm_init() { a[0] = 10.0; ... };
sdsm_done() { ... };
```

**Fig. 4.** OMPC++ Program before Transformation

**Initialization and Finalization** The initialization process initializes the memory management object and the DSM objects. These are quite straightforward, and only involve initialization of locks (the MPC++ Sync object). Finalization involves freeing of all the memory regions and the associated management objects, and are also straightforward.

**Initialization of Shared Regions** There are two types of initialization of shared regions, depending on when the memory is allocated by the user. Type 1 is when the size of the shared region (on variable definition) is fixed, and initialization is done in Initialize(). Transformation from Fig. 4-(1) to Fig. 5-(1) is an example. Type 2 is when the user does not specify the size on variable definition, but instead dynamically allocates a memory region within sdsm_init, sdsm_main(); in this case, allocation is done on the spot as we see in Fig. 4-(2) to Fig. 5-(2).

```
Shared<double> a(100);
Shared<double> b;
sdsm_main() { b.allocate(10);              // ..(2)
              { double _sym52501_8 = a[0]; // ..(5)
                b.WriteStart(0);           // ..(3)
                b(0) = _sym52501_8;
                b.WriteEnd(0); } ... };    // ..(3)
sdsm_init() { { a.WriteStart(0);           // ..(4)
                a(0) = 10.0;
                a.WriteEnd(0); } ... };    // ..(4)
sdsm_done() { ... };
Initialize() { a.allocate(100); };         // ..(1)
Finalize() { a.free();
             b.free(); };
mpc_main() {
  invoke Initialize() on all PEs.
  sdsm_init();
  invoke sdsm_main() on all PEs.
  sdsm_done();
  invoke Finalize() on all PEs.
};
```

**Fig. 5.** After Transformation

**Access to Shared (Memory) Regions** For accessing shared regions, lock/unlock are inserted on writes. Fig. 5-(3),(4)). When a shared variable occurs on the RHS expression, writes are first performed to a temporary variable to avoid duplicate lockings (Fig. 5-(5)) on further RHS evaluation. Read access does not require any locks.

### 3.2 Program Transformation Metaclasses

In OpenC++, metaclasses are defined as subclasses of class Class, and by overriding the methods, one could change the behavior of programs. OMPC++ overrides the following three OpenC++ methods:

**TranslateInitializer** Called when the shared class object is created. We can then obtain the name, type, size, and other information of the distributed shared array object, and are used for initialization of the shared regions.

**TranslateAssign** Called when there is an assignment to the shared class object. We can then transform the initialization and the writes of the distributed shared array objects. We analyze the expression which is passed as a parameter by the OpenC++, and if there is an malloc (or other registered allocation functions), then we perform similar task as the TranslateInitializer to obtain the necessary information, and if it is within sdsm_main(), the initialization code for shared regions directly replaces the allocation. When a shared variable occurs on the RHS expression, TranslateAssign generates a new statement which assigns it to a temporary variable. For assignments to shared variables, their operators [] are transformed to operators (), and the WriteStart() and WriteEnd() methods are inserted to sandwitch the assignment.

**FinalizeInstance** Called upon the end of transformation. Here, we insert the initialization and finalization functions discussed earlier. More concretely, we generate Initialize(), Finalize() based on the information obtained with TranslateInitializer and TranslateAssign, and also generate mpc_main(), obtaining the whole program.

We illustrate a sample metaprogram of TranslateAssign in Fig. 6. Here, we sandwitch the assignment with lock method calls. Metaprograms are quite compact, with only 250 lines of code (does not include runtime as well as template code, etc.).

```
Ptree* SharedClass::
  TranslateAssign(Environment* env,
    Ptree* obj, Ptree* op, Ptree* exp){
      Ptree* exp0 = ReadExpressionAnalysis(env, exp);
      Ptree* obj0 = WriteExpressionAnalysis(env,obj);
      return Ptree::List(obj0, op, exp0);  };
Ptree* SharedClass::
  WriteExpressionAnalysis(Environment* env, Ptree* exp){
    Ptree *obj = exp->First();
    Ptree *index = exp->Nth(2);
    if (!index->IsLeaf())
      index = ReadExpressionAnalysis(env, index);
    InsertBeforeStatement(env,
      Ptree::qMake("`obj`.WriteStart(`index`);\n"));
    AppendAfterStatement(env,
      Ptree::qMake("\n`obj`.WriteEnd(`index`);\n"));
    return Ptree::qMake("`obj`(`index`)");  };
```

**Fig. 6.** Metaprogram Example

### 3.3 Distributed Shared Memory

OMPC++ does not implement DSM transformation with OpenC++ compile-time MOP alone; rather, it also utilizes C++ templates and operator overloading. Also, in OMPC++, read/write barriers are performed in software, instead of (traditional) hardware page-based strategies such as TreadMarks[ACD$^+$96]. Although such checks are potential sources of overhead, they provide the benefit of maintaining the coherency blocks small, avoiding false sharing. Recent work in Shasta[SGT96] has demonstrated that, with low-latency networks, software-based checks do not incur major overhead, even compared to some hardware

**Table 1.** Entry of the management table

| | |
|---|---|
| `addr` | pointer into the real (local) memory (4) |
| `copyowner` | pointer to a list of PEs holding a copy (4) |
| `owner` | the block owner (2) |
| `copyowers` | # of PEs with copies (2) |
| `havedata` | A flag indicating whether data is present (1) |
| `lock` | Lock (1) |
| `dummy` | Padding (2) |

() indicates the number of bytes for the field

based DSMs. Moreover, for portability, software-based checks are substantially better than paged-based checks, as the latter would incur adapting to differing APIs and semantics of trapping page faults and upcalling into the user code for a variety of operating systems.

We note that, with languages such as Java which does not have templates, more program transformation responsibility will be delegated to metaprogramming. It would be interesting to compare the real tradeoffs of the use of templates versus metaprograms from the perspective of performance, code size, ease-of-programmability, etc.

**Distributed Shared Array Class** Distributed Shared Array Class is a template class written in MPC++. The class implements a shared array as the name implies, and objects are allocated on all the PEs. For the current version, the memory consistency protocol is write-invalidate with sequential consistency, and weaker coherence protocols and algorithms such as LRC[Kel94] are not employed. The array elements are organized in terms of consistency block size, which is the unit of memory transfer among the nodes. The size of the block can be specified with `BlockSize`, and can be of arbitrary $2^n$ size. The management tables of the block resides entirely within the class. Each entry in the table is as shown in Table 1, and is aligned to 16 bytes to minimize the cost of address calculations. When the memory allocation method allocate(size) is called, (size/`BlockSize`) entry management table is created, and each PE allocates (size/#PE) memory for storage (excluding the copies).

**Read Access Processing** Because we employ the array access operator `[]`, we overload the `[]` operator in the distributed shared array class. The overloaded behavior is as follows:

1. If the `havedata` flag is true, then return that address.
2. If the `havedata` flag is not true, then allocate the necessary copy block, and request for copying of the block contents to the block owner by passing the MPC++ global pointer to the owner.
3. The requested owner of the block remotely writes the contents of the block onto the copy block pointed to by the global pointer.
4. After the write is finished, the local address of the copy block is returned.

**Write Access Processing** The write access overloads the () operator, but recall that the OMPC++ metaclasses have inserted the lock methods WriteStart() and WriteEnd(). The overloaded behavior is as follows, for write invalidation. Other protocols could be easily accommodated.

1. WriteStart() sets the lock flag of the corresponding management table entry. If it is already locked, then wait.
2. If the PE is the block owner, and there are copies elsewhere, then issue invalidation messages to the PEs with copies.
3. If the PE does not have the block data, then request for a copy in the same manner as the read.
4. If the PE is not the block owner, then transfer the ownership. The previous owner notifies to the other PEs that the ownership change has occurred.
5. The operator () checks the havedata flag, and returns its address.
6. WriteEnd() resets the lock flag.

**Optimizations** Minimizing software overhead is essential for the success of software DSM algorithms. Upon analysis, one finds that the overhead turns out to not be caused by invalidation, but primarily due to the reads/writes which require address translation and barrier checks. In OMPC++, we optimize this in the following way. For accessing the management table entries, we have made the entry size to be $2^n$ so that shifts could be used instead of multiplications and divisions. The overloaded [] and () operators as well as WriteStart() and WriteEnd() are inlined. Moreover, the block address computed in the last access is cached; as a result, if one accesses the same block repeatedly (which is the assumed action of "good" DSM programs in any case), read/write access speed are substantially improved.

One could further define sophisticated metaclasses to perform more through analysis and program transformation to physically eliminate coherency messages, as is seen in [LJ98]. As to whether such analysis are easily encodable in terms of compiler metaclasses of OpenC++ is an interesting issue, and we are looking into the matter.

## 4 Performance Evaluation

The current version of OMPC++ is approximately 700 lines of code, which consists of 250 lines of metaclass code, and 450 lines of template runtime code. This shows that, by the use of reflection, we were able to implement a portable DSM system with substantially less effort compared to traditional systems.

In order to investigate whether OMPC++ is sufficiently fast for real parallel programs, we are currently undertaking extensive performance, analysis, pitting OMPC++ with other software-based DSM algorithms, as well as large-scale SMP systems such as the 64-node Enterprise 10000. Here we report on some of our preliminary results on the PC cluster, including the basic, low-level benchmarks, as well as some parallel numerical application kernels, namely the CG kernel, and the SPLASH2[WOT$^+$95] FFT kernel and LU.

### 4.1 Evaluation Environment

Evaluation environment is a small subset of the PC Cluster II developed at RWCP (Real-World Computing Partnership) in Tsukuba, Japan. The subset embodies 8–10 200MHz Pentium Pro PC nodes with 64MB memory (of which 8 is used for benchmarking), and interconnected by the Myrinet Gigabit network (LinkSpeed 160MB/s). The OS for this system is the Linux 2.0.36-based version of SCore, and uses MPC++ Version 2.0 Level 0 MTTL as mentioned. The underlying compiler is pgcc 1.1.1, and the optimization flags are "-O6 -mpentiumpro -fomit-frame-pointer -funroll-loops -fstrength-reduce -ffast-math -fexpensive-optimizations". For comparative environment, we use the Sparc Enterprise Server 4000, with 10 250MHz UltraSparc II/1Mb, and 1Gb of memory. The Sun CC 4.2 optimization options are "-fast -xcg92 -xO5".

### 4.2 Basic Performance

As a underlying performance basis, we measured the read/write access times. For access patterns to shared arrays, we measured continues access, strided access, and write access with invalidation.

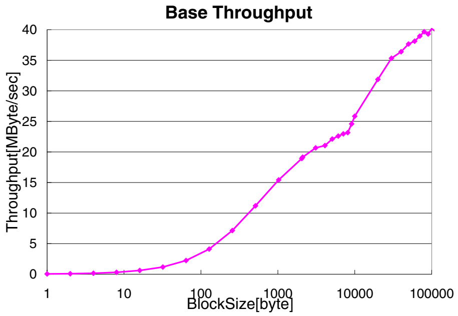


**Fig. 7.** Throughput of basic remote memory operations

**Continuous Read/Write Accesses** We measured the total read/write access times of size 1024×1024 double shared array. All the blocks are setup so that they reside on other PEs; in other words, none of the blocks are initially owned or cached. For reads, this obviously means that accesses must first start by fetching the copies. For writes, since none of the blocks are cached on any of the PEs, write access does not involve invalidation, and thus most of the incurred overhead is remote writes.

We show the averaged times of a single access for different block sizes in Fig. 8. For both reads and writes, average access time decreases with increased block size. This is naturally due to amortization of message handling and lock processing overhead. When block size increases further, we see a falloff due such amortization has mostly making the overhead negligible, and the speed is primarily determined by the network bandwidth. For `BlockSize` 64kbytes, it is 0.34$\mu$sec/1dword, which corresponds to 23.5Mbytes/sec. By comparison, raw MPC++ RemoteMemoryRead is approximately 40Mbytes/sec (Fig. 7). The difference here is the cost of address translation and barrier on the read operation.

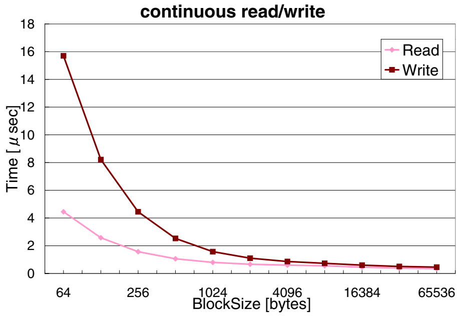


**Fig. 8.** Cost of Continuous Reads and Writes Accesses

**Strided Read/Write Accesses** Strided Access is a worse-case scenario where the arrays are serially accessed at a stride, and the stride is equal to `BlockSize`. As a result, no cached access can occur, but rather the entire block must be transferred per each memory access. The size and the initialization of memory is identical to the continuous access. However, the number of elements accessed would differ according to the stride. The results are shown in Fig. 9. As we can see, the access times increase along with the increase in `BlockSize`. This is due to increased memory transfer per each access. Also, the difference between read and write diminishes; this is because the transfer times become dominant, despite using a very fast network, as opposed to software overhead such as address translation and locking.

**Accesses with Invalidation** Because we currently employ the standard write-invalidate algorithm, the read cost will remain constant, while the write cost could increase as the number of shared copies of the block increases, as invalidate message has to be broadcast to all the PEs holding the copy. In order to measure the overhead, we varied the number of PEs with copies to 0,1,2,6, and the `BlockSize` to 1kbytes and 16kbytes, whose result is shown in Table 2. Here, we observe that for such small number of sharings, the differences are negligible. This is because message passing of MPC++ on SCore is physically

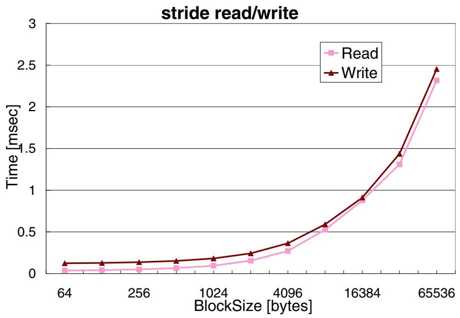


**Fig. 9.** Cost of Strided Read and Write Accesses

asynchronous, and as a result, multiple invalidations are being effectively issued in parallel. Also, access times is independent of block size, as invalidation does not transfer any memory, so its cost is independent of the `BlockSize`.

One might argue that for a larger number of sharing, parallelism in the network interface would saturate, resulting in increased cost. While this is true, research has shown that average number of sharing for invalidation-based protocol is typically below 3. This is fundamental to the nature of the write-invalidate algorithm, as shared blocks are thrown away per each write.

**Table 2.** Cost of Write Accesses with Invalidation (`BlockSize`=1kbytes,16kbytes)

| | Time($\mu$sec) | |
|---|---|---|
| Sharings (#PEs) | `BlockSize` 1kbytes | `BlockSize` 16kbytes |
| 0 | 4.80 | 5.26 |
| 1 | 31.27 | 33.05 |
| 2 | 46.06 | 48.27 |
| 6 | 76.68 | 78.24 |

### 4.3 CG (Conjugate Gradient Solver Kernel)

As a typical parallel numerical application, we employ the CG (Conjugate Gradient) Solver Kernel, without preconditioning. As a comparative basis, we execute the equivalent sequential program on one of the nodes of the PC cluster, and also on the Sun Ultra60(UltraSPARC II 300MHz, 256Mbytes, Sun CC 4.2). Here are the parameters for the measurements:

– Problem sizes: 16129 (197 iterations), 261121 (806 iterations)

- `BlockSizes`: 256, 1k, 4k, 16kbytes
- Number of PEs: 1, 2, 4, 8

The results are shown in Fig. 10 and Fig. 11.

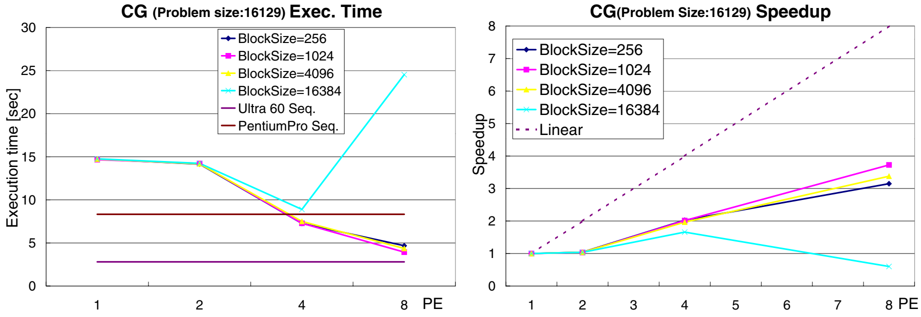


**Fig. 10.** CG Kernel Result, Problem Size=16129

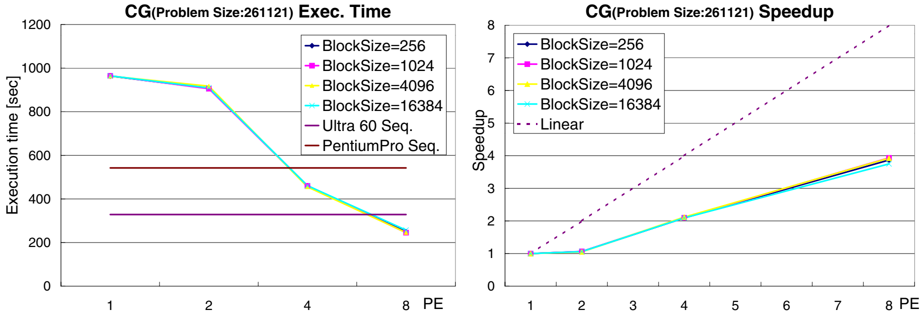


**Fig. 11.** CG Kernel Result, Problem Size=261121

According to the results of Fig. 10, we obtain only about 5% speedup from 1PE to 2PEs, whereas we observe approximately linear speedups from 2PEs to 8PEs. This is primarily due to the software overhead of DSM, in particular write-invalidation does not occur for 1PE, whereas they do occur for 2PEs and above. Even with 8PEs, we do not match the speed of sequential UltraSparc II 300MHz, which has much faster floating performance than a Pentium Pro 200MHz. We do attain about factor of 2 speedup over a sequential execution on a cluster PC node. For `BlockSize` of 16KB, we see that performance degrades significantly. This is due to the overhead of excessive data transfer.

With larger problem size as in Fig. 11, the overall characteristics remain the same. We still see speedups of range 3.8 to 4.0 with 8 processors. Here, `BlockSize` is almost irrelevant for overall performance; this is because, as the amount of data required for computation is substantially larger than `BlockSize`, and thus we do not have wasteful internote data transfers. On the other hand, compared to overall computation, the software overhead of message handling is negligibly small.

We also executed the same program on the Sun Enterprise 4000 (UltraSPARC II 250MHz, 10PEs). There the speedup was approximately 6 with 8 processors. The overhead is primarily due to barrier operations.

### 4.4 SPLASH2 FFT and LU

Finally, we tested the SPLASH2 [WOT$^+$95] FFT and LU kernel (Fig. 12) with 1kbytes BlockSize. In FFT, the problem size is 64Kpoints. In LU, the matrix size is 512.

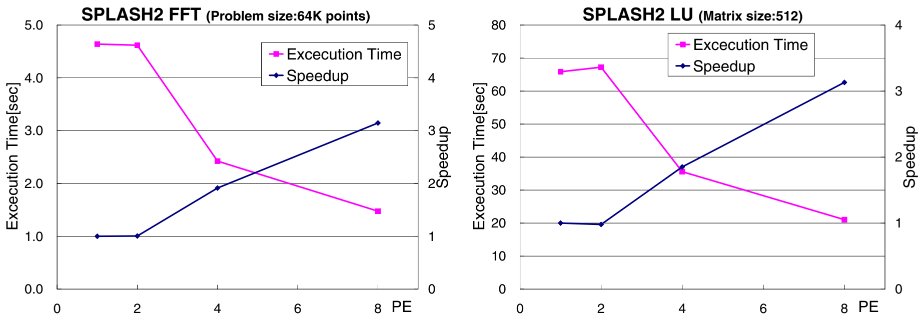


**Fig. 12.** SPLASH2 FFT and LU

As was with the CG kernel, speedup curves are similar. The required change to the original SPLASH2 was approximately 70 lines out of the 1008 in FFT, 80 lines out of the 1030 in LU, mostly the required top-level program structure (with sdsm_main, etc.) and memory allocations.

Shasta employs various low-level optimization strategies, that have not been implemented on OMPC++, such as bunching multiple read-/write-accesses. As an experiment, to qualify the effect of bunching technique, we manually applied it to the `daxpy` function in the LU kernel on OMPC++, attaining about factor of 2 speedup in terms of the total execution time(Fig. 13).

## 5 Related Work

There are several work on implementing DSM features using program transformations [SGT96]. All of them have had to develop their own preprocessor,

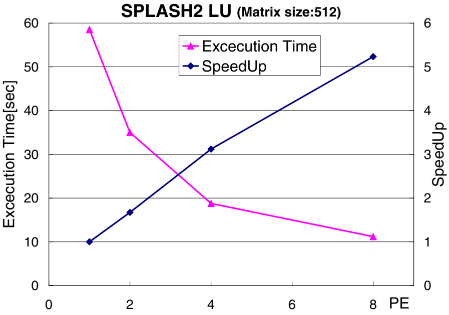


**Fig. 13.** SPLASH2 LU with bunched read-/write-accesses

and is thus expensive, and furthermore not portable nor adaptable to complex programming languages such as C++. By utilizing the reflective features of OpenC++, we have demonstrated that, developing a program transformer for an efficient DSM system was an easy effort with a small number of metaclass customizations, much easier to maintain and tune the code. In fact, substantial tuning and experimentation was possible by small changes to the metacode.

Shasta [SGT96] is a DSM system that performs program transformation at program load time, and is thus more language independent than other program transformation systems. Shasta is specifically tailored to work on DEC Alpha PEs interconnected by the MemoryChannel low-latency gigabit network. Shasta is extremely efficient, in that it optimizes the read/write barrier overhead for common cases down to two instructions. This is currently possible because Shasta performs program transformation at the binary level. On the other hand, Shasta is much harder to port to other platforms, as it is hardware and processor-binary dependent. There are numerous low-level optimization techniques that Shasta uses, that have not been implemented on OMPC++. We are planning to exploit such techniques with more metaprogramming, with the aid of better optimizing compilers such as Kai C++.

Midway[BZS93] and CRL[JKW95] are object-based DSM systems. Midway employs entry consistency as the coherency protocol, and read/write barriers are inserted by a special compiler. Thus, one must craft such a compiler, and as such portability and extensibility suffers. For example, it is difficult to add a new coherency protocol or porting to different hardware platform without detailed knowledge of the compiler. On the other hand, in CRL, the barriers must be inserted by the user. This is quite error-prone, and such bugs would be very hard to detect, as they will be dynamic, transient memory bugs. By using reflection, OMPC++ was created by only small set of customization in the underlying workings of the C++, and is easily understandable, thus being easily maintainable, customizable, and portable.

As a common shared memory programming API, OpenMP[Ope97] has been proposed as a standard. In OpenMP, the user adds appropriate annotations to the source code to indicate potential SPMD parallelism and distribution to sequential code, and the compiler emits the appropriate parallel code. OpenMP suffers from the same problem as Midway, as customized preprocessor/compiler must be built. It would be interesting to investigate whether OpenMP (or at least, its subset) could be implemented using reflective features to indicate DSM optimizations.

In perspective, with respect to the maintainability, customizability, and portability of DSM systems,

- Traditional page-based DSM systems must have OS-dependent code to trap memory reference via a page fault. Also, overhead is known to be significant.
- Link-time DSM systems such as Shasta is efficient, but is platform and processor-binary dependent.
- Class-based systems requires the user to insert appropriate read/write barriers, and is thus error prone and/or only a certain set of classes can be shared.
- Macros or ad-hoc preprocessor are hard to build and maintain, let alone customize or be portable, and complete support of complex programming languages such as C++ is difficult.

## 6 Conclusion and Future Work

We have described OMPC++, a portable and efficient DSM system implemented using compile-time reflection. OMPC++ consists of a small set of OpenC++ 2.5 metaclasses and template classes and is thus easy to customize and port to different platforms. OMPC++ is a first step to realization of reflective framework for portable and efficient support of various parallel programming models across a wide variety of machines. Such characteristics are necessary as increasingly in the days of network-wide computing, where a piece of code could be executed anywhere in a diverse network.

OMPC++ was shown to be efficient for a set of basic as well as parallel numerical kernel benchmarks on the RWC PC cluster platform.

As a future work, we are pursuing the following issues:

**More comprehensive benchmarking** The number of benchmarked programs, processors, and comparative platforms, are still small. We are currently conducting a set of comprehensive benchmarks by adding more SPLASH2 benchmarks (Water, Barnes), on larger platforms (64-processor RWCP cluster and 64-processor Sparc Enterprise Server 10000), pitting against other DSM platforms (developed at RWCP). We are also attempting alternative algorithms, both for coherency and barriers. We will report the findings in another paper after more analysis is done.

**More efficient implementation** Although OMPC++ went under some tuning process, the read/write barriers and locks are still expensive. Although in our preliminary benchmarks we are finding that OMPC++ is competitive with other DSM implementations, we need to further enhance efficiency. For example, Shasta only requires two instructions for barriers, much smaller than ours, and employs various low-level optimization strategies such as bunching multiple read-/write-accesses. Also, raw MPC++ RemoteMemoryRead throughput is 40MByte/sec, whereas OMPC++ throughput is 23.5MByte/sec. This is due to address translation and table accesses. We are considering altering the runtime data structures to further eliminate the overhead.

**Porting to other platforms** OMPC++ currently is implemented on top of MPC++ and RWC cluster. In order to port to different platforms, one must alter the dependency on MPC++, if it is not available. In principle this is simple to do, as one must only provide (1) threads, (2) remote memory read/write, and (3) global barriers. This can be easily implemented using MPI or other message passing platforms such as Illinois Fast Messages. Alternatively one could make MPC++ to be more portable, which is our current project in collaboration with RWC. In all cases, we must analyze and exhibit the portability and efficiency of OMPC++ to validate that implementing DSM systems with reflection is the right idea.

**Other coherency protocols** On related terms, we should support other coherency protocols, such as write-update, as well as weak coherency models such as LRC and ALRC. The latter will fundamentally require changes to the source code, and it would be interesting to investigate how much of this simplified using open compilers.

**Portable DSM on Java using OpenJIT** Finally, we are planning to implement a Java version of DSM using OpenJIT, our reflective Java Just-In-Time compiler. As of Feb. 1, 1999, We have almost completed the implementation of OpenJIT, and will start our implementation as soon as OpenJIT is complete.

## Acknowledgment

We would like to thank Shigeru Chiba for his support in the use of OpenC++, and the members of RWCP for their support in the usage of MPC++ plus the RWC cluster. Also, we would like to thank the Dept. of Information Science, the Univ. of Tokyo for allowing experiments on their Sparc Enterprise Server 10000. We finally thank the members of the Tsukuba TEA group for fruitful discussions on performance analysis of SPMD and DSM systems.

## References

[ACD$^+$96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Warkstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[Beo] Beowulf Project. Beowulf Homepage. http://beowulf.gsfc.nasa.gov/.

[BZS93] Brain N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdown. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE CompCon Conference*, 1993.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, pages 285–299, 1995.

[Chi97] Shigeru Chiba. *OpenC++ 2.5 Reference Mannual*, 1997.

[Ie96] Yutaka Ishikawa and et.al. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Proceedings of Reflection'96*, Apr. 1996.

[Ish96] Yutaka Ishikawa. Multiple Threads Template Library – MPC++ Version2.0 Level 0 Document –. TR 96012, RWCP, 1996.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symosium on Operating Systems Principles*, Dec. 1995.

[Kel94] Pete Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Dept. of Computer Science, Rice University, Dec. 1994.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov 1989.

[LJ98] Jae Bum Lee and Chu Shik Jhon. Reducing Coherence Overhead of Barrier Synchronization in Software DSMs. In *SC'98*, Nov. 1998.

[MOS+98] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, and Koichiro Hotta. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, Oct. 1998.

[Myr] Myricom, Inc. Myricom Homepage. http://www.myri.com/.

[Ope97] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming.*, Oct. 1997.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of ASPLOS VII*, Oct. 1996.

[Uni] Stanford Univ. SUIF Homepage. http://www-suif.stanford.edu/.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.

# Metaprogramming Domain Specific Metaprograms

Tristan Cazenave

Laboratoire d'Intelligence Artificielle,
Département Informatique, Université Paris 8,
2 rue de la Liberté,
93526 Saint Denis, France.
cazenave@ai.univ-paris8.fr

**Abstract**. When a metaprogram automatically creates rules, some created rules are useless because they can never apply. Some metarules, that we call impossibility metarules, are used to remove useless rules. Some of these metarules are general and apply to any generated program. Some are domain specific metarules. In this paper, we show how dynamic metaprogramming can be used to create domain specific impossibility metarules. Applying metaprogramming to impossibility metaprogramming avoids writing specific metaprogram for each domain metaprogramming is applied to. Our meta-metaprograms have been used to write metaprograms that write search rules for different games and planning domains. They write programs that write selective and efficient search programs.

## 1 Introduction

Knowledge about the moves to try enables to select a small number of moves from a possibly large set of possible moves. It is very important in complex games and planning domains where search trees have a large branching factor. Knowing the moves to try drastically cuts the search trees. Metaprogramming can be used to automatically create the knowledge about interesting and forced moves, only given the rules about the direct effects of the moves [4],[5]. Impossibility metaprograms enable to remove useless rules from the set of unfolded rules. These metaprograms can themselves be written by metametaprograms. From a more general point of view, metaknowledge itself can be very useful for a wide range of applications [23], and one of its fascinating characteristic is that it can be applied to itself to improve itself. We try to experimentally evaluate the benefits one can get from this special property.

The second section describes metaprogramming and especially metaprogramming in games. The third section uncovers how metametaprograms can be used to write impossibility metaprograms. The fourth section gives experimental results.

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 235-249, 1999

## 2 Metaprogramming in games and planning domains

Our metaprograms write programs that enable to safely cut search trees, therefore enabling large speedups of search programs. In our applications to games, metarules are used to create theorems that tell the interesting moves to try to achieve a tactical goal (at OR nodes). They are also used to create rules that find the complete set of forced moves that prevent the opponent to achieve a tactical goal (at AND nodes). Metaprogramming in logic has already attracted some interest [11],[2],[9]. More specifically, specialization of logic program by fold/unfold transformations can be traced back to [26], it has been well defined and related to Partial Evaluation in [15], and successfully applied to different domains [9]. The parallel between Partial Evaluation and Explanation-Based Learning [21],[18],[6],[13],[24] is now well-known [28],[7]. As Pitrat [23] points it, the ability for programs to reason on the rules of a game so as to extract useful knowledge to play is a problem essential for the future of AI. It is a step toward the realization of efficient general problem solvers.

In our system, two kinds of metarules are particularly important : impossibility metarules and monovaluation metarules. Other metarules such as metarules removing useless conditions or ordering metarules are used to speed-up the generated programs.

Impossibility metarules find which rules can never be applied because of some properties of the game, or because of more general impossibilities. An example of a metarule about a general impossibility is the following one :

```
impossible(ListAtoms):-
          member(N=\=N1,ListAtoms),var(N),var(N1),N==N1.
```

This metarule tells that if a rule created by the system contains the condition `'N=\=N1'` and the metavariables N and N1 contain the same variable, then the condition can never be fulfilled. So the rule can never apply because it contains a statement impossible to verify. These metarule is particularly simple, but this is the kind of general knowledge a metasystem must have to be able to reason about rules and to create rules given the definition of a game.

Some of the impossibility metarules are more domain specific. For example the following rule is specific to the game of Go :

```
impossible(ListAtoms):-
          member(color_string(B,C),ListAtoms),C==empty.
```

It tells that the color of a string can never be the color `'empty'`.

The other important metarules in our metaprogramming system are the monovaluation metarules. They apply when two variables in the same rules always share the same value. Monovaluation metarules unify such variables. They enable to simplify the rules and to detect more impossible rules. An example of a monovaluation metarule is :

```
monovaluation(ListAtoms):-
          member(color_string(B,C),ListAtoms),
          member(color_string(B1,C1),ListAtoms),
          B==B1,C\==C1,C=C1.
```

It tells that a string can only have one color. So if the system finds two conditions `'color_string'` in a rule such that the variables contained in the metavariables `'B'` and `'B1'` are equals, and if the corresponding color variables contained in the metavariables `'C'` and `'C1'` are not the same, it unifies `C` and `C1` because they always contain the same value.

Impossible and monovaluation metarules are vital rules of our metaprogramming system. They enable to reduce significantly the number of rules created by the system, eliminating many useless rules. For example, we tested the unfolding of six rules with and without these metarules. Without the metarules, the system created 166 391 568 rules by regressing the 6 rules on only one move. Using basic monovaluation and impossible metarules shows that only 106 rules where valid and different from each other.

The experiments described here use the goal of taking enemy stones to create rules for the game of Go. The game of Go is known to be the most difficult game to program [27],[1],[25]. Experiments in solving problems for the hardest game benefit to other games, and to other planning domains, especially if these experiments use general and widely applicable methods as it is the case for our metaprogramming methods.

## 3 Programs that write programs that write programs

Works on writing programs that write programs that write programs often refer to the third Futamura projection. Their goal is to speed up programs that write programs using self-application of Partial Evaluation. Self-applicable partial evaluators such as Goëdel [11] usually use a ground representation to enable self-application (a ground representation consists in representing variables in the programs by numbers, a parallel can be made with the numbering technique used by the mathematician Kurt Gödel to prove his famous theorem [10]). Our choice is rather to use general non-ground metaprograms that find domain specific metaknowledge to write the programs that write programs. On the contrary of fold/unfold/generalization and other program transformation techniques [20], our system only uses unfolding, and simple metaprograms can be written to decide when to stop unfolding: typically when generated rules have more condition than a pre-defined threshold.

Domain specific metarules in games and planning domains can be divided in different categories. We will focus on the 'board topology metarules' category in this section. Other categories that are often used are 'move metarules' or 'object metarules' for example.

The essential property of a game board is that it never changes whatever the moves are. The set of facts describing the board is always the same. Moreover it is a complete set: no facts can be added or removed.

In this paper, we will use a fixed grid to give examples of metaprogram generation. Grids are used in planning domains, for example for a robot to plan a path through a building, and in games such as Go or Go-Moku.

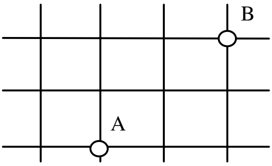


**Fig. 1.** A robot at point A has to choose a path on the grid to go to point B

The grid task consists in finding a path of length four between point A and point B in the figure 1. This task is easier to understand than the game of Go which is our principal application. A Go board is also a grid, and all the mechanisms described in this paper also apply to the rules generated by our metaprogram for the game of Go.

### 3.1 Metaprogramming impossibility metarules

The figure 2 gives all the points that are at distance three of point A. After each new instanciation of a variable containing a point, the rule verifies that the instanciated point is different from any previously instanciated one.

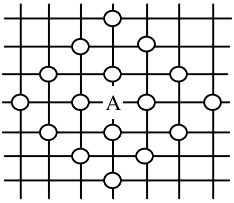


**Fig. 2.** All the points at a distance three of point A are marked.

The rule that find all these points is generated as follows:

```
distance(X,W,3):-
          connected(X,Y),connected(Y,Z),connected(Z,W).
```

This rule is generated unfolding the goal `'distance(X,W,3)'` defined below:

```
distance(X,X,0).
distance(X,W,N1):-
          N is N1-1,distance(X,Z,N),connected(Z,W).
```

On a grid, all the points that are at a distance two of X are not at a distance three. Moreover, when unfolding definitions in more complex domains, it often happens that two variables are unified, there is only one variable left after unfolding. Unfolding can lead to generate rules similar to this one:

```
distance(X,X,3):-
          connected(X,Y),connected(Y,Z),connected(Z,X).
```

This rule has been correctly generated by a correct metaprogram on a correct domain theory, but it is a rule that will never be applied, because no point on a grid is at a distance three of itself.

We can detect that this rule cannot be fired because we have access to the complete set of facts representing the topology of the board in this domain. In the generated rules, we only select the conditions that are related to the topology of the board (the connected predicates and the test that are done on the variables representing intersections of the grid). Then we fire this set of conditions on the complete set of facts. If the set of conditions never matches, we are confident that the system has generated an impossible set of conditions. In order to find the minimal set of impossible conditions, the system tries to remove the conditions one by one until each removed condition leads to a possible set of conditions. We now have a minimal set of conditions representing a subset of the initial rule that is impossible to match. In our simple example of the distance three goal, it is composed of the three conditions of the rule.

Once this subset is created, it is used to generate a new impossibility metarule. Impossibility metarules match generated rules to find subsets of impossible conditions. If an impossibility metarule succeeds, the generated rule is removed. The impossibility metarule generated in our example is:

```
impossible(ListAtoms):-
          member(connected(X,Y),ListAtoms),
          var(X),var(Y),X\==Y,
          member(connected(A,Z),ListAtoms),
          A==Y,var(A),var(Z),A\==Z,
          member(connected(B,C),ListAtoms),
          B==Z,var(B),var(C),B\==C,C==X.
```

The generation of impossibility metarules is useful in almost all the planning domains we have studied. Here is another example of its usefulness for the game of Abalone.

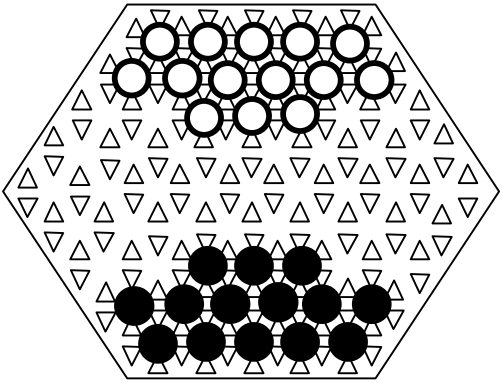


**Fig. 3.** An abalone board at the beginning of the game

The figure 3 represents an Abalone board, on this board, each position has six neighbors instead of four for the grid board. Each neighbor is associated to one of the six directions represented by numbers ranging from 1 to 6.

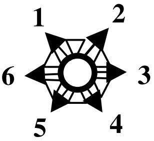


**Fig. 4.** The directions a stone can be moved to are marked, ranging from one to six.

In the rules of the game and therefore in the rules generated by our metaprogram, the connected predicates contain a slot for the direction. Here is an example of an impossible set of conditions in the game of Abalone :

```
connected(X,Y,Direction),connected(Y,X,Direction),
```

The corresponding impossibility metarule for the game of Abalone is:

```
impossible(ListAtoms):-
          member(connected(X,Y,D),ListAtoms),
          member(connected(A,B,C),ListAtoms),
          A==Y,B==X,C==D.
```

It is also generated using our metametaprogram that generate impossibility metarules.

### 3.2 Metaprogramming simplifying metarules

The system sometimes generates some rules that contain useless conditions. We give below a rule that finds a path of length four to go from one point of a grid to another one without going twice through the same point. After each new instanciation of a variable in the conditions, the rule verifies that the instanciated point is different from any previously instanciated one.

After each condition in the rule, we give the number of times the condition has been verified when matching the rule once on a set of facts.

```
 distance(X,D,4):-
          connected(X,Y),                     4
          X=\=Y,                              4
          connected(Y,Z),                    16
          Z=\=X,                             12
          Z=\=Y,                             12
          connected(Z,W),                    48
          W=\=X,                             48
          W=\=Y,                             36
          W=\=Z,                             36
          connected(W,D).                   144
```

However, in some cases, it is useless to verify that some points are different due to the topology of the grid. For example, two connected points are always different. We can use a metarule that tells to remove the condition `'X=\=Y'` if the condition `'connected(X,Y)'` is also present in the rule:

```
useless(X=\=Y,ListAtoms):-
          member(connected(X,Y),ListAtoms),
          member(A=\=B,ListAtoms),A==X,B==Y.
```

Another metarule, given below, removes the condition `'X=\=Y'` when there is a path of length three between the two points contained in `X` and `Y`, this is a consequence of the figure 2 that shows all the points that are at a three step path from point A: A is not at a three step path of itself.

```
useless(X=\=Z,ListAtoms):-
          member(connected(X,Y),ListAtoms),
          var(X),var(Y),X\==Y,
          member(connected(Y,Z),ListAtoms),
          var(Y),var(Z),Y\==Z,
          member(connected(Z,A),ListAtoms),
          var(Z),var(A),Z\==Y,A==X.
```

The initial rule makes 361 instanciations and tests. After firing the metarule of deletion on the initial rule, we obtain the rule below that only makes 261 instanciations or tests with the same results.

```
distance(X,D,4):-
          connected(X,Y),                     4
          connected(Y,Z),                    16
          Z=\=X,                             12
          connected(Z,W),                    48
```

```
        W=\=Y,                                  36
        connected(W,D).                        144
```

The simplifying metarules described here can also be generated using a complete set of facts representing the board topology.

For example, if our system analyzes a rule containing the conditions:

```
connected(X,Y),X=\=Y,
```

in this order. It observes that the condition `'X=\=Y'` is always fulfilled. So it tries to remove all the conditions of the rule one by one, provided the condition `'X=\=Y'` is always fulfilled when matching the set of remaining conditions. At the end of this process, the final set of conditions only contains the two above conditions, and the condition `'X=\=Y'` is always fulfilled for all the complete set of facts in the working memory. As the set of fact representing the topology of the board is complete, it can generate a new simplifying metarule that will apply to all the generated rules of this domain. This simplifying metarule is the one given above.

This method works in all planning domains where a complete set of facts can be isolated, such as the topology of the board, for games where the topology cannot be changed by the moves.

### 3.3 Metaprogramming ordering metarules

**Related Work**

P. Laird [14] uses statistics on some runs of a program to reorder and to unfold clauses of this program. T. Ishida [12] also dynamically uses some simple heuristics to find a good ordering of conditions for a production system. Our approach is somewhat different, it takes examples of working memories to create metarules that will be used to reorder the clauses. What we do, is automatically creating a metaprogram that is used to reorder the clauses, and not dynamically reordering conditions of the rules. One advantage is that we can create this metaprogram independently. Moreover, once the metaprogram is created, running it to reorder learned rules is faster than dynamically optimizing the learned rules. This feature is important for systems that use a large number of generated rules. The creation of the metaprogram is also fast.

We rely on the assumption that domain-dependent information can enhance problem solving [16]. This assumption is given experimental evidence on constraint satisfaction problems by S. Minton [17]. On the contrary of Minton, we do not specialize heuristics on specific problems instances, we rather create metaprograms according to specific distributions of working memories.

**Reordering conditions**

Reordering conditions is very important for the performance of generated rules. The two following rules are simple examples that show the importance of a good order of conditions. The two rules give the same results but do not have the same efficacy when X is known and Y unknown:

```
sisterinlaw(X,Y):-brother(X,X1),married(X1,Y),woman(Y).
sisterinlaw(X,Y):-woman(Y),brother(X,X1),married(X1,Y).
```

Reordering based only on the number of free variables in a condition does not work for the example above. In the constraint literature, constraints are reordered according to two heuristics concerning the variables to bind [17] : the range of values of the variables and the number of other variables it is linked to. These heuristics dynamically choose the order of constraints. But to do so, they have to keep the number of possible bindings for each variable, and to lose time when dynamically choosing the variable. It is justified in the domain of constraints solving because the range of value of a variable, affects a lot efficiency, and can change a lot from one problem to another. It is not justified in some other domains where the range of value a variable can take is more stable. We have chosen to order conditions, and thus variables, statically by reordering once for all and not dynamically at each match because it saves more time in the domains in which we have tested our approach.

Reordering optimally the conditions in a given rule is an NP-complete problem. To reorder conditions in our generated rules, we use a simple and efficient algorithm. It is based on the estimated number of following nodes the firing of a condition will create in the semi-unification tree. Here are two metarules used to reorder conditions of generated rules in the game of Go:

```
branching(ListAtoms,ListBindVariables,
          connected(X,Y),3.76):-
          member(connected(X,Y),ListAtoms),
          member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).

branching(ListAtoms,ListBindVariables,
          elementstring(X,Y),94.8):-
          member(elementstring(X,Y),ListAtoms),
          non_member_term(X,ListBindVariables),
          non_member_term(Y,ListBindVariables).
```

A metarule evaluates the branching factor of a condition based on the estimated mean number of facts matching the condition in the working memory. Metarules are fired each time the system has to give a branching estimation for all the conditions left to be ordered. When reordering a rule containing N conditions, the metarule will be fired N times: the first time to choose the condition to put at first in the rule, and at time number T to choose the condition to put in the T$^{th}$ place. In the first reordering metarule above, the variable X is already present in some of the conditions preceding the condition to be chosen. The variable Y is not present in the preceding conditions. The condition 'connected(X,Y)' is therefore estimated to have a branching factor of 3.76 (this is the mean number of neighbor intersections of an intersection on a 19*19 grid, this number can vary from 2 to 4), this is the mean number of bindings of Y.

The branching factors of all the conditions to reorder are compared and the condition with the lowest branching factor is chosen. The algorithm is very efficient, it orders rules better than humans do and it runs fast even for rules containing more than 200 conditions.

**Generating ordering metarules**

For each predicate in the domain theory that has an arity less or equal than three. Each variable of the predicate free or not, leading to $2^3$=8 possibilities for the three variables. So, for each predicate, we create between 1 and 8 metarules.

For predicates of arity greater than three, we only create the metarules that corresponds to the bindings of all but one of the variables of the predicate.

All the metarules are tested on some working memories. This enables to conclude on the priority to give to the metarule. The priority is the mean number of bindings the condition will create. The lower the priority, the sooner the condition is to be matched. When all the variables of a condition are instanciated, it is a test and it has a priority between zero and one, whereas predicates containing free variables have, most of the time, a priority greater than one.

## 4 Results

This section gives the results and the analysis of some experiments in generating metaprograms. We used a Pentium 133 with SWI-Prolog for testing.

### 4.1 Metaprogramming impossibility metarules

In the figure 5, the horizontal axis represents the number of rules unfolded by our metaprogram on one move. This experiment was realized using the game of Go domain theory associated to the subgoal of taking stones of the opponent. There are six unfolded rules, it means that six rules concluding on a won subgoal of taking stones where randomly chosen out of the total number of such rules created by our system. Each of these six rules has been unfolded using the rules of the game of Go, without the cuts of impossibility and monovaluation metarules. All of the six rules to unfold, match Go boards where the friend color can take the opponent string in less than two moves, whatever the opponent plays. The goal of the unfolding was to find all the rules that find a move that lead to match one of the six rules concluding on a won state. The vertical axis of the figure 5 represents the cumulated number of rules that have been created for each of the six rules. We did not match the impossible and monovaluation metarules on the resulting rules because it would have been to time consuming.

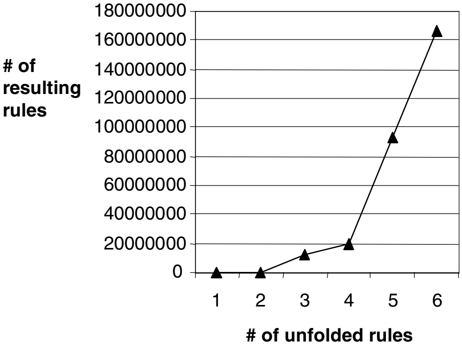


**Fig. 5.** Number of rules generated when unfolding six simple rules without impossibility and monovaluation metarules.

Instead of unfolding all the rules one move ahead and then destroying useless rules, we matched the monovaluation and impossibility metarules after each unfolding step (an unfolding step is the replacement of a predicate by one of its definitions). Each unfolding step is considered as a node in the unfolding tree.

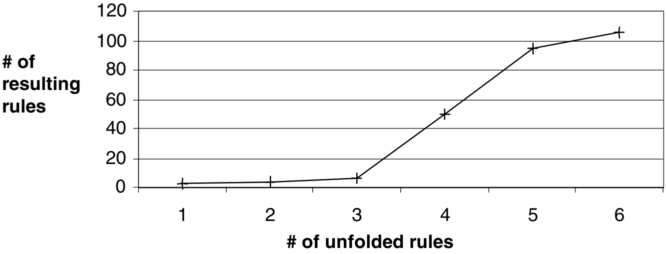


**Fig. 6.** Number of rules generated when unfolding six simple rules with impossibility and monovaluation metarules

This resulted in a very significant improvement of the specialization program, it was much faster and completely unfolding the six rules only gave 106 resulting rules concluding on winning moves to take stones 3 moves in advance. The results are shown in the figure 6, and can be compared with the results of the figure 5. It is important to see that among all the resulting rules of the figure 5, only the 106 resulting rules of the figure 6 are valid and different from each other.

This experiment also stresses the importance of the impossibility metarules. Without them, unfolding a goal on a domain theory is not practically feasible. Therefore impossibility metarules are necessary for such programs, and automatically generating them is a step further in the automatisation of planning programs development.

### 4.2 Metaprogramming ordering metarules

When no metarule concludes on the priority of the conditions left to be ordered, simple reordering heuristics are used. For example, the condition containing the less variables is chosen.

Following are two equivalent rules. The first one is ordered without metarules, and the second one is ordered using the learned metarules :

```
      threattoconnect(C,B,B1,I):-
            colorintersection(I1,empty),       55
            connected(I,I1),                  208
            connected(I1,I2),                 796
            I=\=I2,                           588
            liberty(I2,B1),                   306
            colorblock(B1,C),                 140
            liberty(I,B),                      84
            colorblock(B,C),                   36
            color(C).                          36
                                             ____
                                             2249

  threattoconnect(C,B,B1,I):-                   1
            color(C),                           2
            colorblock(B,C),                   14
            liberty(I,B),                      68
            connected(I,I1),                  240
            colorintersection(I1,empty),       96
            connected(I1,I2),                 350
            I=\=I2,                           254
            liberty(I2,B1),                    84
            colorblock(B1,C).                  36
                                             ____
                                             1145
```

Each condition is followed by the number of time it has been accessed during the matching of the rule. In this example, when choosing the first condition, a classic order gives `'colorintersection(I,empty)'` in the first rule. In the second rule, the two following metarules where matched among others to assign priorities to conditions :

```
  branching(ListAtoms,ListBindVariables,
            colorintersection(I,C),240.8):-
            member(colorintersection(I,C),ListAtoms),
            C==empty,
            non_member_term(I,ListBindVariables).

  branching(ListAtoms,ListBindVariables,color(C),2):-
            member(color(C),ListAtoms),
            non_member_term(C,ListBindVariables).
```

Therefore, the condition 'color(C)' has been chosen because it has the lowest branching factor.

The two rules given in the example are simple rules. Speedups are more important with rules containing more conditions.

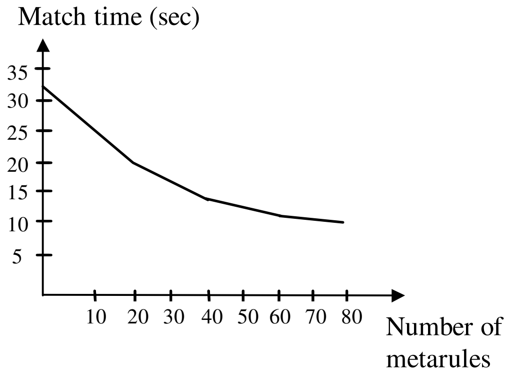


**Fig. 7.** The match time of generated rules decreases when more ordering metarules are generated and used.

The figure 7 gives the evolution of the matching time of a set of generated rules with the number of metarules generated. The evolution is computed on a test set of 50 problems. Problems in the test set are different from the problems used to generate the metarules.

## 5 Conclusion

Metaprogramming games and planning domains is considered as an interesting challenge for AI [19],[23]. Moreover it has advantages over traditional approaches: metaprograms automatically create the rules that otherwise take a lot of time to create, and the results of the search trees developed using the generated programs are more reliable than the results of the search trees developed using traditional heuristic and hand-coded rules.

The Go program that uses the rules resulting of the metaprogramming has good results in international competitions (6 out of 40 in 1997 FOST cup [8], 6 out of 17 in 1998 world computer Go championship). The metaprogramming methods presented here can be applied in many games and in other domains than games. They have been applied to other games like Abalone and Go-Moku, and to planning problems [3]. Using metaprogramming this way is particularly suited to automatically create complex, efficient and reliable programs in domains that are complex enough to require a lot of knowledge to cut search trees.

However metaprogramming large programs can be itself time consuming. We have proposed and evaluated methods to apply metaprogramming to itself so as to make it more efficient. These methods gave successful results. Moreover they tend to give even better results when generated programs become more complex.

## 6 References

1. Allis, L. V.: Searching for Solutions in Games an Artificial Intelligence. Ph.D. diss., Vrije Universitat Amsterdam, Maastricht 1994.
2. Barklund J. : Metaprogramming in Logic. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.
3. Cazenave, T.: Système d'Apprentissage par Auto-Observation. Application au Jeu de Go. Ph.D. diss., Université Paris 6, 1996.
4. Cazenave T.: Metaprogramming Forced Moves. Proceedings ECAI98, Brigthon, 1998.
5. Cazenave T.: Controlled Partial Deduction of Declarative Logic Programs. ACM Computing Surveys, Special issue on Partial Evaluation, 1998.
6. Dejong, G. and Mooney, R.: Explanation Based Learning : an alternative view. Machine Learning 1 (2), 1986.
7. Etzioni, O.: A structural theory of explanation-based learning. Artificial Intelligence 60 (1), pp. 93-139, 1993.
8. Fotland D. and Yoshikawa A.: The 3rd fost-cup world-open computer-go championship. ICCA Journal 20 (4):276-278, 1997.
9. Gallagher J.: Specialization of Logic Programs. Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.
10. Gödel K.: 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I', Monatsh. Math. Phys. 38, 173-98, 1931.
11. Hill P. M. and Lloyd J. W.: The Gödel Programming Language. MIT Press, Cambridge, Mass., 1994.
12. Ishida T.: Optimizing Rules in Production System Programs, AAAI 1988, pp 699-704, 1988.
13. Laird, J.; Rosenbloom, P. and Newell A. Chunking in SOAR : An Anatomy of a General Learning Mechanism. Machine Learning 1 (1), 1986.
14. Laird P.: Dynamic Optimization. ICML-92, pp. 263-272, 1992.
15. Lloyd J. W. and Shepherdson J. C.: Partial Evaluation in Logic Programming. J. Logic Programming, 11 :217-242., 1991.
16. Minton S.: Is There Any Need for Domain-Dependent Control Information : A Reply. AAAI-96, 1990.
17. S. Minton. Automatically Configuring Constraints Satisfaction Programs : A Case Study. Constraints, Volume 1, Number 1, 1996.
18. Mitchell, T. M.; Keller, R. M. and Kedar-Kabelli S. T.: Explanation-based Generalization : A unifying view. Machine Learning 1 (1), 1986.
19. Pell B.: A Strategic Metagame Player for General Chess-Like Games. Proceedings of AAAI'94, pp. 1378-1385, 1994. ISBN 0-262-61102-3.
20. Pettorossi, A. and Proietti, M.: A Comparative Revisitation of Some Program Transformation Techniques. Partial Evaluation, International Seminar, Dagstuhl Castle, Germany LNCS 1110, pp. 355-385, Springer 1996.

21. Pitrat J.: Realization of a Program Learning to Find Combinations at Chess. Computer Oriented Learning Processes, J. C. Simon editor. NATO Advanced Study Institutes Series. Series E: Applied Science - N° 14. Noordhoff, Leyden, 1976.
22. Pitrat, J.: Métaconnaissance - Futur de l'Intelligence Artificielle. Hermès, Paris, 1990.
23. Pitrat, J.: Games: The Next Challenge. ICCA journal, vol. 21, No. 3, September 1998, pp.147-156, 1998.
24. Ram, A. and Leake, D.: Goal-Driven Learning. Cambridge, MA, MIT Press/Bradford Books, 1995.
25. Selman, B.; Brooks, R. A.; Dean, T.; Horvitz, E.; Mitchell, T. M.; Nilsson, N. J.: Challenge Problems for Artificial Intelligence. In Proceedings AAAI-96, 1340-1345, 1996.
26. Tamaki H. and Sato T.: Unfold/Fold Transformations of Logic Programs. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.
27. Van den Herik, H. J.; Allis, L. V.; Herschberg, I. S.: Which Games Will Survive ? Heuristic Programming in Artificial Intelligence 2, the Second Computer Olympiad (eds. D. N. L. Levy and D. F. Beal), pp. 232-243. Ellis Horwood. ISBN 0-13-382615-5. 1991.
28. Van Harmelen F. and Bundy A.: Explanation based generalisation = partial evaluation. Artificial Intelligence 36:401-412, 1988.

# Aspect-Oriented Logic Meta Programming

Kris De Volder and Theo D'Hondt
{kdvolder|tjdhondt}@vub.ac.be

Programming Technology Lab, Vrije Universiteit Brussel

**Abstract.** We propose to use a logic meta-system as a general framework for aspect-oriented programming. We illustrate our approach with the implementation of a simplified version of the COOL aspect language for expressing synchronization of Java programs. Using this case as an example we illustrate the principle of *aspect-oriented logic meta programming* and how it is useful for implementing weavers on the one hand and on the other hand also allows users of AOP to fine-tune, extend and adapt an aspect language to their specific needs.

## 1 Introduction

The notion of aspect-oriented programming (AOP) [MLTK97,KLM$^+$97] is motivated by the observation that there are concerns of programs which defy the abstraction capabilities of traditional programming languages. At present the main idea behind software engineering is hierarchical (de)composition. Not surprisingly, current programming languages are designed from this perspective and provide mechanism of abstraction such as procedures, classes and objects, that constitute units of encapsulation specifically aimed at top-down refinement or bottom-up composition.

These abstraction mechanism however do not align well with what AOP terminology calls *cross-cutting design concerns*, such as synchronization, distribution, persistence, debugging, error handling etc. which have a wider, more systemic impact. Precisely because of their wider impact on the system, the code dealing with cross-cutting concerns can not be neatly packaged into a single unit of encapsulation. This results in their implementation being scattered throughout the source code and this severely harms the readability and maintainability of the program. Aspect-oriented programming addresses this problem by designing *aspect languages* which offer new language abstractions that allow cross-cutting aspects to be expressed separately from the base functionality. A so called *aspect weaver* generates the actual code by intertwining basic functionality code with aspect code.

Early instances of aspect languages where limited in scope and dealt only with very specific aspects in very specific contexts. The D [LK97] system for example proposes specific aspect languages to handle the aspects of synchronization and replication of method arguments in distributed Java programs. Similarly [ILGK97] describes an aspect-oriented approach in the specific context

P. Cointe (Ed.): Reflection'99, LNCS 1616, pp. 250-272, 1999

of sparse matrix algorithms. More recently, with AspectJ [LK98], AOP has taken a turn towards a more general aspect language applicable in a broader context.

What all aspect languages have in common is that they are declarative in nature, offering a set of declarations to direct code generation. Earlier incarnations of AOP support only specific declarations suited to one particular context, e.g. synchronization in Java. Typically these stand at a high level of abstraction, avoiding specific implementation details of the weaver. Consequently they are formulated in terms of concepts linked to their context, e.g. mutual exclusion in the context of synchronization. AspectJ on the other hand provides more generally applicable but at the same time also more low-level declarations which are more intimately linked with the weaving algorithm, directly affecting code generation. Typical AspectJ declarations provide code to be executed upon entry and exit of methods, variables to be inserted into classes, etc. The universal declarative nature of aspect languages begs for a single uniform declarative formalism to be used as a general uniform aspect language. This paper proposes to use a logic programming language for this purpose. Thus, aspect declarations are not expressed by means of a specially designed aspect language but are simply logic assertions expressed in a general logic language. These logic facts function as hooks into a library of logic rules implementing the weaver.

The objective of this paper is to clearly demonstrate the advantage gained by unifying the aspect declaration language and the weaver implementation language and using a logic language for both purposes. There are two points of view from which the advantages will be discussed: the side of the AOP implementor and the side of the AOP user. The *implementor* is responsible for identifying what kind of aspects he wants to tackle. He subsequently devises a set of aspect declarations that allows expressing the aspects conveniently. Then he implements a *weaver* that takes base functionality and aspects into account and produces code that integrates them. The *user* on the other hand, is a programmer who declares base functionality and aspects by means of the special purpose languages provided by the AOP implementor.

We have called our approach *aspect-oriented logic meta programming* (AOLMP), because it depends highly on using the logic meta language to reason *about aspects*. On the one hand the logic language can be used merely as a simple general purpose declaration language, using only facts, but more importantly, it can also be used to express *queries about aspect declarations* or to declare rules which *transform aspect declarations*.

To illustrate the advantages of AOLMP from both the user's and the implementor's point of view, we will look at one aspect: the aspect of synchronization. The concern for synchronization, to ensure integrity of data accessible simultaneously by several threads, is more or less orthogonal to the program's functionality. Nevertheless, synchronization code is spread all over the program thus making it completely unintelligible. Lopes [LK97] proposed a solution for the synchronization problem using the aspect-oriented approach. She defined the aspect language COOL for expressing the synchronization aspect separately from the base functionality. We will present an implementation for COOL-like aspect declarations, using the logic meta programming approach. Note that it is not the goal of this paper to propose a better solution to the particular problem of synchronization. Instead, the emphasis of the paper is on the technique of logic meta programming and how it relates to AOP. For reasons of clarity we therefore only provide support for a simplified subset of Lopes' declarations in our example implementation. Nevertheless, the extra flexibility of our logic meta programming approach will allow us to recover some of the omitted features and even go beyond. This results in clearer synchronization declarations because the reasoning underlying them can be made explicit through a logic program.

We used the TyRuBa system to conduct the experiments upon which this paper is based. TyRuBa was designed as an experimental system to explore the use of logic meta programming for code generation in our recent Ph.D. dissertation [DV98].

The structure of the paper is as follows. We start by briefly introducing logic meta programming and the TyRuBa system in section 2. Section 3 explains the synchronization problem and how it can be tackled using an AOP approach. This section is mostly based on Lopes' work on the COOL aspect language [LK97]. We show how it is possible to express the synchronization aspect by means of logic facts that represent COOL-like aspect declarations. Section 4 illustrates the usefulness of AOLMP from the viewpoint of an AOP user. Section 5 discusses the relevance of AOLMP for weaver implementation. Section 6 discusses related work. Section 7 summarizes the conclusions.

## 2 TyRuBa and Logic Meta Programming

### 2.1 The TyRuBa core language

The TyRuBa system is built around a core language which is basically a simplified Prolog variant with a few special features to facilitate Java-code manipulation. We assume some familiarity with Prolog and will only briefly discuss the most important differences with it. For more information about standard Prolog we refer to [DEDC96,SS94]. For more information about the TyRuBa language we refer to [DV98]. The examples throughout the paper are simple enough and sufficiently explained to be understandable to a reader not familiar with Prolog.

**Quoted code blocks** The most important special feature of TyRuBa is its quoting mechanism which allows pieces of Java code to be used as terms in logic programs. Quoted pieces of code may in turn contain references to logic variables or other compound terms. Pieces of quoted code containing logic variables can be used as a kind of code templates and are very useful in implementing code generators.

In the version of TyRuBa presented and used in this text the quoting mechanism is rudimentary. Basically a quoted Java term is nothing more than a kind of string starting after a "{" and ending just before the next balanced "}". Unlike a string it is not composed of characters but of Java tokens, logic variables, constant terms and compound terms. The Java tokens are treated as special name constants whose name is equal to the printed representation of the token. The following example of a quoted Java term illustrates most kinds of quoted elements:

```
{ void foo() { Array<?El> contents  = new Array<?El>[5];
               ?El        anElement = contents.elementAt(1); }
}
```

In the above example we see a compound term “Array<?El>”. We find several name constants “contents”, “new”, “anElement”, . . . There are two integer literals 5 and 1. The remainder of the tokens such as “=”, “.” and “(” are Java tokens treated as name constants with “strange names”. Note that a quoted code block may contain “{” and “}” tokens, as long as these are properly balanced. Let it be clear that a nested “{” or “}” is treated just as any other token and does *not* introduce a nested quoted code block.

The meaning of a quoted Java term in the context of a TyRuBa program is derived directly from its internal representation. A quoted Java term corresponds to a compound term of arity 1 with a special qualifier. Its single subterm is a TyRuBa list of quoted elements. The example given above is internally represented by the following compound term:

```
'{}'([void, foo, '(', ')', '{', Array<?El>, contents, '=', ... ])
```

Note that in this example the term Array<?El> is used in place of an identifier. Compound terms which occur inside blocks of quoted code are printed as mangled identifier names. This is very convenient for code generation purposes. It can be used to parameterize names for types, variables or methods. The term Array<?El> for example can be thought of as the name of a template class for representing arrays of type ?El. Also variable's and method's names can be parameterized this way. In the weaver implementation presented in section 5 we make explicit use of this feature to declare a synchronization variable per method by parameterizing the variable name with the method name.

**Lexical Conventions** Because terms and variables may occur inside of quoted code, TyRuBa's lexical conventions differ somewhat from Prolog's to avoid confusion. Variables are identified by a leading “?” instead of starting with a capital. This avoids confusion between Java identifiers and Prolog variables. Some examples of TyRuBa variables are: ?x, ?Abc12, etc. Consequently, any identifier, including identifiers starting with a capital, is considered to be a constant. Some examples constants are: x, 1, Abc123, etc. To avoid confusion with function or procedure calls in Java, TyRuBa compound terms are written with “<” and “>” instead of “(” and “)” as in Prolog.

## 2.2 Logic Meta Programming

The idea of logic meta-programming is very simple. A base program is represented indirectly by means of a set of logic propositions. The relationship between the base program and its logic representation is concretized under the form of a code generator: a program that queries the logic data repository and outputs source code in the base language.

Logic meta-programming is thus achieved because a logic program can be thought of as representing the set of logic propositions that can be proven from its facts and rules. These facts in turn can be thought of as indirectly representing a base-language program. The full power of the logic paradigm may thus be used to describe base-language programs indirectly. This offers great potential as we will try to illustrate in the rest of this paper.

The mapping scheme between logic representation and base program may vary and determines the kind of information that is reified and accessible to meta programs. In this paper we assume a mapping that represents classes by means of facts which state that the class has certain methods, instance variables or constructors.

The presence of a variable declaration in a class is represented by a fact of the form:

```
var(?Class,?VarType,?VarName,{...declaration code...}).
```

A method declaration is asserted by a fact of the following form:

```
method(?Class,?ReturnType,?MethodName,?ArgTypeList,
   {...declaration head...},
   {...method body...}).
```

Below is an example Java class declaration and its corresponding representation as a set of TyRuBa propositions.

```
class Stack {
  int pos = 0 ;
  Stack() {
    contents = new Object[SIZE];}
  public Object peek ( )  {
    return contents[pos];  }
  public Object pop ( )  {
    return contents[--pos];  }
  ... }
```

```
class(Stack).
var(Stack,int,pos,{int pos = 0;}).
constructor(Stack,[],{public Stack()},
  {contents = new Object[SIZE]; }).
method(Stack,Object,peek,[],
  {public Object peek()},{return...}).
method(Stack,Object,pop,[],
  {public Object pop()},{return...}).
...
```

## 3 The Synchronization Problem and AOP

In this section we introduce the example used throughout the rest of this paper: a simple aspect language and weaver for solving the problems involved in writing synchronization code for multi-threaded Java applications. Subsection 3.1 sketches the problem and subsection 3.2 describes the aspect declarations supported by our simple weaver.

### 3.1 The Synchronization Problem

The problem in writing multi-threaded Java applications is that synchronization code ensuring data integrity tends to dominate the source code completely. As a result it becomes entangled and unmanageable. As an illustration of the problem, consider the implementation of a Stack abstract data type which is given in figure 1. The figure just lists the “bare bones” version without synchronization code. This code is simple, straightforward and easy to read.

```
class Stack  {
  static final int MAX = 10 ;
  int pos = 0 ;
  Object[] contents = new Object [ MAX ] ;

  public void print ()  {
     System.out.print("[");
     for (int i=0 ; i<pos ; i++ ) {
        System.out.print(contents[i]+" ") ; }
     System.out.print("]");  }
  public Object peek ()  {
     return contents[pos];  }
  public Object pop ()  {
     return contents[--pos];  }
  public void push (Object e)  {
     contents [pos++]=e ;  }
  public boolean empty ()  {
     return pos == 0 ;  }
  public boolean full ()  {
     return pos == MAX ;  }
}
```

**Fig. 1.** The “bare bones” version of the class Stack

The readability of the class Stack with synchronization code added is much worse. It is even too complicated to fit comfortably onto a single page. Therefore we will only take a look at one of the methods in it. The other methods are messed up in a similar way. Figure 2 lists the declaration of the peek method, complete with synchronization code.

To implement synchronization at the granularity of methods, a number of counter instance variables will be added to the Stack class. One such counter will be declared for each method. A counter instance variable will therefore have a name such as BUSY_pop, BUSY_peek etc. Code must be added to the start and end of each method to increment and decrement these counters. Also added to the start of the method is a “guard condition” which verifies whether the method may start executing. If the guard is not satisfied the method must wait for the guard to become true. The peek method for example waits until there are no

```
public Object peek ( )  {
    while ( true ) {
      synchronized ( this ) {
        if (  (  BUSY_pop == 0  ) &&  (  BUSY_push == 0  )   ) {
           ++ BUSY_peek ; break ; } }
      try { wait ( ) ; }
      catch ( InterruptedException COOLe ) { } }
    try {
     return contents [ pos ] ;  }
    finally {
      synchronized ( this ) {
        -- BUSY_peek ;
        notifyAll ( ) ; } } }
```

**Fig. 2.** The peek method with synchronization code

more threads currently executing a push or a pop method. It is obvious from figure 2 that the synchronization code completely dominates the source code: almost all of the code in the figure is synchronization code. Aspect oriented programming solves this problem by providing a special purpose language, called an *aspect language*, with which the synchronization aspect can be described separately from the base functionality. A code generator, called an *aspect weaver* takes a base program without aspects and an aspect program and generates output code integrating both.

### 3.2 Synchronization-aspect Declarations

Our TyRuBa weaver for generating synchronization code supports a simplified version of the COOL aspect language proposed by Lopes [LK97]. The COOL aspect language is used to specify the synchronization aspect of a Java base program. Our approach differs from the traditional AOP approach. Instead of defining a special purpose aspect language we assume that the AOP programmer provides aspect declarations under the form of TyRuBa logic facts. The weaver is consequently implemented as a library of logic rules for generating code from the facts describing the aspects and the base functionality. We defer treatment of the weaver's implementation to section 5. In this section we introduce the various aspect declarations that are recognized and supported by it.

Mutual exclusion between methods is expressed by a logic fact as illustrated by the following example.

```
mutuallyExclusive(Stack,push,pop).
```

The same kind of declaration is also used to declare that a method should never be run concurrently with itself. For example, to declare that the method push is not allowed to be run concurrently with itself, one asserts a fact:

```
mutuallyExclusive(Stack,push,push).
```

Declaration of mutually exclusive methods triggers the weaver to insert the appropriate guard expressions at the beginning of methods. Additional guards, other than those derived from the above synchronization declarations, may be added to a method by declaring a fact:

```
requires(?c,?m,?condition).
```

This means that the method `?m` in class `?c` may not be started unless the `?condition` expression evaluates to `true`. The following example declarations ensure that no elements are ever popped from an empty stack nor pushed onto a stack which is full.

```
requires(Stack,push,{!full()}).
requires(Stack,pop,{!empty()}).
```

Finally, declarations of facts `onEntry` and `onExit` can be used to specify synchronization related actions that have to be performed upon entry and exit of a method.

```
onEntry(?class,?method,?statements).
onExit(?class,?method,?statements).
```

## 4 Aspect-Oriented Meta Programming

The fundamental advantage of using logic facts to declare aspects instead of a special-purpose aspect language is that aspect declarations can be accessed and declared by logic rules. This enables what we call *aspect-oriented logic meta programming*, i.e. writing logic programs which reason about aspect declarations. This technique is useful because it allows the user to extend or adapt the aspect language. This section presents two examples of the usefulness of AOLMP from the user's point of view.

### 4.1 Example: Self-exclusive and Mutually-exclusive Lists

The first example is a simple extension of the aspect language which adds some syntactic sugar on top of the pairwise declaration of mutually exclusive methods. This syntactic sugar allows expressing mutual exclusion by means of lists of mutually-exclusive and self-exclusive methods, in the same style as the declarations in Lopes' system. For example, mutual exclusion of three methods can be declared using the syntactic sugar as follows:

```
mutuallyExclusiveList(Stack,[push,pop,peek]).
```

This single declaration implies that any method in the given list is mutually exclusive with any other method in the list. A similar syntactic sugar is supported to assert that methods should not run concurrently with themselves:

```
selfExclusiveList(Stack,[push,pop,print]).
```

When the list of mutuallyExclusive methods is long the pairwise notation becomes cumbersome and less readable because of a combinatorial explosion of pairwise combinations. We would therefore like to be able to use the list notation as syntactic sugar. It is fairly easy to add support for this. All we need to do is include two simple rules into our aspect (meta) program. The first rule expresses that two methods ?m1 and ?m2 should be declared (pairwise) mutually exclusive if they both occur together as elements in the same mutually-exclusive list ?l.

```
mutuallyExclusive(?c,?m1,?m2) :- mutuallyExclusiveList(?c,?l),
    element(?m1,?l),
    element(?m2,?l),
    NOT(equal(?m1,?m2)).
```

Note that the above rule checks that the method ?m1 and ?m2 are not one and the same method. To actually infer that a method is mutually exclusive with itself, it must occur in a self exclusive list. This is taken care of by the following rule:

```
mutuallyExclusive(?c,?m,?m) :- selfExclusiveList(?c,?l),
    element(?m,?l).
```

The aspect program that describes the synchronization aspect of the Stack class, using the list-like notation is given in figure 3.

```
selfExclusiveList(Stack,[push,pop,print]).

mutuallyExclusiveList(Stack,[push,pop,peek]).
mutuallyExclusiveList(Stack,[push,pop,empty]).
mutuallyExclusiveList(Stack,[push,pop,full]).
mutuallyExclusiveList(Stack,[push,pop,print]).

requires(Stack,push,{!full()}).
requires(Stack,pop,{!empty()}).
```

**Fig. 3.** The Stack synchronization-aspect program

### 4.2 Example: Modifies and Inspects Declarations

This second example is a somewhat more sophisticated variation of the aspect language. The goal is to allow declaring synchronization of methods in an entirely different way. Rather than declaring which methods are mutually or self exclusive one may declare how methods modify or inspect state. This is often more convenient. For example in the Stack, when reasoning about what methods should be declared mutually exclusive with one another, one depends entirely on the knowledge of which methods inspect or modify what state. It would therefore be preferable to declare this information directly and explicitly. Therefore, instead of using exclusiveness declarations, we would like to write the following:

```
modifies(Stack,push,this).
modifies(Stack,pop,this).
inspects(Stack,peek,this).
inspects(Stack,empty,this).
inspects(Stack,full,this).
modifies(Stack,print,SystemOut).
inspects(Stack,print,this).
```

This style of declarations is much more convenient than exclusiveness declarations because they relate more closely to the semantics of the methods rather than to the way that synchronization is implemented. These declaration are also clearer because they explicitly reveal which is otherwise left implicit: the reason why synchronization code must be added.

This example really highlights the advantages of declaring aspects by means of a general purpose logic language. Using logic rules which reason about and declare aspects, we can easily provide support for these alternative synchronization aspect declarations, thereby explicitizing the reasoning underlying the aspect declarations. It is also important to note that we can do this aspect-language extension without having to reimplement the weaver! We can regard the `modifies` and `inspects` aspect declarations again as syntactic sugar on top of pairwise mutual exclusive declarations. This is possible because the `modifies` and `inspects` declarations provide sufficient information to derive `mutuallyExclusive` properties. Using AOLMP we can express elegantly and concisely how both kinds of declarations relate to one another by means of two simple logic rules. The first rule expresses that any two methods are mutually exclusive if one method inspects a state modified by the other one.

```
mutuallyExclusive(?class,?inspector,?modifier) :-
   inspects(?class,?inspector,?thing),
   modifies(?class,?modifier,?thing).
```

The second rule takes care of mutual exclusion between methods which modify the same state. This rule states that modification is implicitly a kind of inspection.

```
inspects(?class,?method,?thing) :- modifies(?class,?method,?thing).
```

The examples given so far illustrate how AOLMP is useful from the AOP user's point of view. They show how expressing aspects by means of logic assertions in combination with the availability of a full-fledged logic language fosters an enormous potential for the aspect-oriented programmer. It allows him to build on top of the existing aspect declarations in order to extend or modify the aspect language. It is important to note in these examples that in order to extend the aspect language, the aspect programmer did not have to descend to the level of the weaver's implementation. The core of the weaver's implementation has never been touched and consequently knowledge about its internal workings is not required. Extensions are simply defined as sophisticated syntactic sugar on top of already existing aspect declarations.

It might be argued that inspects/modifies declarations are less general and not able to express the synchronization aspect in some situations where exclusiveness style declarations could. This however only further highlights the power of our AOLMP approach: on the fly extensions of the aspect language can be implemented fairly easily and can be as general or as specific as a particular situation requires.

## 5 AOLMP and Weaver implementation

In this section we will have a look at the usefulness of AOLMP from the viewpoint of the AOP implementor. We will present the implementation of a weaver for the synchronization declarations proposed in section 3.2. For easy reference we have summarized these declarations in figure 4.

| Declaration | Meaning |
|---|---|
| `mutuallyExclusive(?c,?m1,?m2)` | The method `?m1` in class `?c` should not be run concurrently with the method `?m2`. |
| `requires(?c,?m,?e)` | The method `?m` in class `?c` should not be started unless expression `?e` evaluates to true. |
| `onEntry(?c,?m,?s)` | Execute the statement `?s` upon entry of method `?m`. |
| `onExit(?c,?m,?s)` | Execute the statement `?s` upon exit of method `?m`. |

**Fig. 4.** Basic synchronization-aspect declarations

### 5.1 Layers of Code-to-code Transformations

The architecture of the COOL weaver implementation in TyRuBa is depicted in figure 5. This is a layered architecture of code-to-code transformations. Every layer consists of logic facts describing Java source code. The Java input program with only the base functionality is parsed and turned into a set of logic facts which are inserted into the TyRuBa fact and rule base in the JCore[1] layer. A set of logic rules describes how to copy the code on the basic layer onto the COOL layer. Another set of rules describes how code is added (woven) into the COOL layer to support the aspect declarations. Finally there is one more set of rules which describes how the thus produced facts on the COOL layer should be "unparsed" into a form printable as Java source code.

[1] Named after JCore, the simplified Java language which is used in Lopes' system to express basic functionality.

Many of these rules, such as for example the copying rules and the unparsing rules just implement the general architecture and are not directly dependent on the aspect language. We will only discuss the set of rules which handles weaving of COOL aspect declarations into the COOL layer. For a more complete description we refer to [DV98].

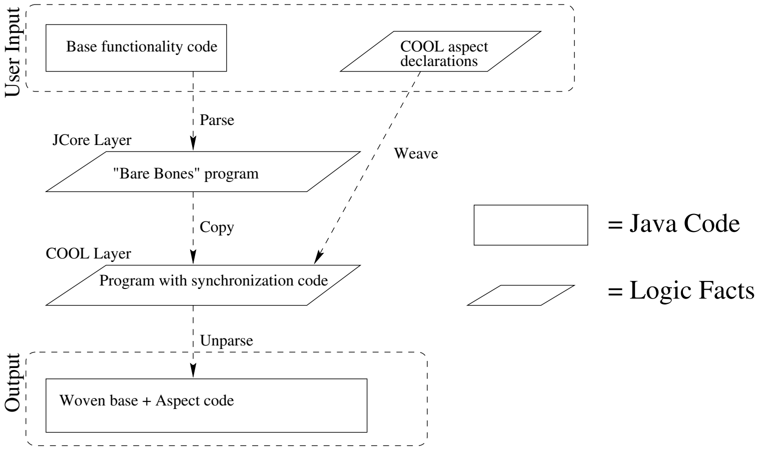


**Fig. 5.** The COOL code generator

Because the TyRuBa language is an experimental and simple logic language, without a module system, we could not rely on modules to divide the facts and rules into layers. We therefore partition the logic facts by adopting the convention that the first argument of every fact indicates the layer it belongs to. The JCore parser for example will insert the following fact into the rule base to indicate that the `Stack` has a `peek` method. The first argument is a symbol `JCore` indicating that this fact belongs to the `JCore` layer.

```
method(JCore,Stack,Object,peek,[],
  {public Object peek()},   //signature
  {return contents[pos]; }  //body
).
```

### 5.2 Synchronization-aspect Code

We will now have a look at the most important rules in the COOL weaver implementation: the rules that describe how the code on the COOL layer is generated, based on the code on the JCore layer and the synchronization aspect declarations.

Before continuing we note that part of the aspect language can be defined in terms of the more low-level features of the aspect language itself. The synchronization code for maintaining the counters could be added by means of `onEntry` and onExit declarations. Likewise, the guard conditions that prevent methods from being started based on the value of these counters can be added by means of requires declarations. This is a very important observation because it implies that the task of implementing the weaver is greatly simplified using AOLMP. We only need to provide support for generating code for the low-level features and can then implement the higher level declarations in terms of the more low-level declarations, in a similar way as in the previous examples. We therefore start by implementing support for the more low-level aspect declarations onExit, onEntry, and requires. Afterwards we will implement the mutuallyExclusive declaration easily in terms of the more low-level declarations.

**Low-level Aspect Declarations** The core of the COOL code generator is very simple. Basically it merely adds some wrapper code around the body of a JCore method declaration. Below is the rule which adds wrapper code around a method in the COOL layer. This wrapper code should look familiar since it has roughly the same layout as the example synchronization code we presented for the peek method in figure 2.

```
method(COOL,?class,?Return,?name,?Args,?head,{
    while (true) {
      synchronized ( this ) {
        if (?condition) {
           ?atStart
           break; } }
      try { wait ( ) ; }
      catch ( InterruptedException COOLe ) { } }
    try {?body}
    finally {
      synchronized(this) {
        ?atEnd
        notifyAll();} }}
) :- method(JCore,?class,?Return,?name,?Args,?head,?body),
      COOL_allRequired(?class,?name,?condition),
      COOL_atStartStatements(?class,?name,?atStart),
      COOL_atEndStatements(?class,?name,?atEnd).
```

A number of auxiliary predicates computes the ?condition expression and the ?atStart and ?atEnd statement lists to be inserted into the template wrapper code.

The auxiliary predicate COOL_allRequired collects all of the conditions declared by requires aspect declarations for a certain method. All of these are combined into a conjunction, i.e. a list of Java expressions combined together by means of the Java "&&" logical "and" operator.

```
COOL_allRequired(?class,?name,?exp) :-
  FINDALL(NODUP(?cond,requires(?class,?name,?cond)),
          ?cond,?conditions),
  JavaConjunction(?conditions,?exp).
```

The meta predicate FINDALL is a standard Prolog feature which can be used to collect results from a given query into a list. Here it is used to collect all of the guard conditions declared in requires declaration into the list ?conditions.

The meta predicate NODUP is a feature of TyRuBa to filter out duplicate solutions based on a comparison key. It was added to facilitate code generation. The use of NODUP here avoids duplicate conditions from being included more than once.

Two other auxiliary rules collect the statements to be inserted at the start and end of a method.

```
COOL_atStartStatements(?class,?name,?statements) :-
   FINDALL(onEntry(?class,?name,?stat),
          ?stat,?statements).

COOL_atEndStatements(?class,?name,?statements) :-
   FINDALL(onExit(?class,?name,?stat),
          ?stat,?statements).
```

**Higher-level Aspect Declarations** The rules presented in the previous section implement the core of the COOL code generator which supports the more low-level aspect declarations that insert synchronization statements and conditions at the right places into the synchronization wrapper code of a method. We can now relatively easily provide support for pairwise mutuallyExclusive declarations in terms of these.

First we observe that the mutuallyExclusive relationship is a symmetric relationship: whenever mutuallyExclusive(?c,?m1,?m2) holds this also implies mutuallyExclusive(?c,?m2,?m1). Rather than requiring the user to declare the symmetric pairs we let the weaver implementation take care of it and declare the symmetric closure of the mutuallyExclusive relationship declared by the user as follows[2].

```
mutuallyExclusiveSym(?c,?m1,?m2) :-
    mutuallyExclusive(?c,?m1,?m2);mutuallyExclusive(?c,?m2,?m1).
```

The following declaration adds the guard condition that makes sure that a method ?name is not started when another method with which it is mutuallyExclusive is already running[3].

```
requires(?class,?name,{BUSY<?other> == 0}) :-
   mutuallyExclusiveSym(?class,?name,?other).
```

[2] The ";" denotes a logical "or"

[3] The implementation of our simplified code generator also prohibits recursive calls from the same thread. This is usually not the intention. In Lopes' work this is patched by using a more complicated Lock object instead of a simple int counter. The Lock object also records which thread is locking the object and allows calls from the same thread explicitly. We could also support this more complicated locking strategy. All we need to change are the guard conditions and the declarations of the counter instance variables.

The guard expression consults a counter variable BUSY<?x> which registers how many times a method ?x has been entered. We still have to declare these variables and the onEntry and onExit code to increment and decrement the counters appropriately. The following rule adds a counter variable for every method which needs to be counted, i.e. every method that must conform to a mutuallyExclusive constraint. Note that the use of NODUP serves to avoid declaring the variable multiple times.

```
var(COOL,?class,int,BUSY<?name>,{
  private int BUSY<?name> = 0;
}) :- NODUP([?class,?name],
            mutuallyExclusiveSym(?class,?name,?other)).
```

Finally we present the rules that add administrative code for incrementing and decrementing the counter variables. Administrative code is added to every method for which a counter variable has been defined.

```
onEntry(?class,?name,{
        ++BUSY<?name>;
}) :- var(COOL,?class,int,BUSY<?name>,?declaration).
onExit(?class,?name,{
        --BUSY<?name>;
}) :- var(COOL,?class,int,BUSY<?name>,?declaration).
```

This concludes the implementation of our simplified version of the COOL aspect language and code generator. Due to lack of space we will not present the actually generated code. To get an idea of the the generated code, reexamine the peek method in figure 2. This is actually an excerpt taken from the generated code with only minor cosmetic changes to the indentation and renaming of messy "mangled TyRuBa-term identifiers" such as BUSY_Lpeek_R.

# 6 Related Work

## 6.1 Logic Meta Programming

To our knowledge, the connection between aspect-oriented programming and logic meta programming has never before been discussed or examined in the literature. The idea of logic meta programming itself, i.e. using a logic language as an expressive and powerful means to reason about programs is not new. A recent survey of the field can be found in [HG98,Bar95]. Logic programming languages are known to be good for implementing various kinds of meta programs, such as compilers, interpreters, type checkers, type inferencers etc. It's powerful unification and backtracking mechanism make it especially suitable for implementing these kinds of programs. Also, many features have been added to logic languages to facilitate meta programming. Prolog [DEDC96,CM81,SS94] for example has features to support meta programming. It offers definite clause grammars for example, a feature that facilitates the implementation of parsers. The programming language Gödel [HL94] is a declarative higher-order logic language designed for meta programming. Lambda Prolog [FGH+90] is an extension of Prolog with unification of lambda terms. Lambda terms are an extension specifically intended to facilitate the manipulation of formulas and programs [MN87]. It is especially useful in manipulating functional programs.

A logic programming approach has also been proposed for expressing sophisticated pattern matching and verification of programs [Wuy98,Cre97,BGV90,CMR92,Min96]. The power of the logic paradigm is exploited in two major ways in these approaches: as a verification/enforcing tool (e.g. Law Governed Architectures [Min96]), or as an information gathering tool [Cre97,BGV90,CMR92], or both at the same time (e.g. SOUL [Wuy98]). Both kinds of uses of logic are complementary to AOP. All deal with code tangling. AOP tries to avoid code tangling, by allowing aspects to be expressed separately. Using logic language as an information gathering tool on the other hand, its powerful pattern matching capabilities are exploited in recovering lost information from already tangled code. As a verification tool, a logic language is used in yet another way to deal with error prone tangled code by enforcing global correctness or consistency constraints.

### 6.2 AspectJ

The recent developments around AspectJ [KL98] concur with our ideas in many ways. Just like our approach, AspectJ moves away from the approach of only offering a fixed set of special purpose aspect languages. Instead, it tries to capture them as particular instantiations of a more general notion of an aspect. The major difference with our approach is that AspectJ is not offering a metaprogramming language in order to achieve generality. Instead, a much more restricted extension mechanism, resembling a form of subclassing on aspects, is offered. Incidentally AspectJ also incorporates a simple form of pattern matching using wild cards as an alternative for the pattern matching power we get almost for free through unification and backtracking. Our approach is more general and more expressive than AspectJ because our extension language is a general full-fledged (logic) programming language. Of course, telling which approach is best is not as clear cut as that and other criteria besides generality and expressiveness are also important. The AspectJ team explicitly does not want to offer the full power of a real meta-programming language to the AOP user and strives to obtain a simpler and more manageable extension language.

The question remains open however whether the simplicity and manageability is worth sacrificing the expressiveness. We feel given the experimental stage of development of AOP that a more liberal more expressive formalism, such as ours, might be more suitable, at least as means of exploration and experimentation. Also, extensions and adaptation of the logic meta language towards better AOP support, such as special syntactic sugar and a scoping and module mechanism to make it more closely resemble an AspectJ like syntax might eliminate the drawbacks of a more complicated notation for simple uses altogether.

Last but not least, the `modifies/inspects` example, is at least one compellingly simple application of AOLMP for extending an aspect language. This alone earns some merit for AOLMP as a suitable alternative for AspectJ's approach.

### 6.3 Reflection

Reflection is also a mechanism that can be used to deal with cross cutting. In our opinion there is however a large difference between true reflection and simpler forms of meta programming. Meta programs are programs which reason about *other* programs or aspects thereof. Reflection however, means programs which reason about *themselves*. Following the treatment of [Smi82] a reflective system has a "causally connected self representation". This means that a program has access to some kind of data structure which represents (reifies) its computational system or aspects thereof. This can be inspected or it can be acted upon. "Causally connected" means that acting upon the self representation directly affects the computational system (this is sometimes called absorption). For a more detailed explanation of this terminology and theory we refer to [Smi82,Ste94].

The self-referential nature of reflective systems makes them very complex both theoretically and with respect to implementation. Issues such as reflective overlap, meta-stability, infinite towers etc. need to be considered [Smi82,Smi84,Mae87,WF88,KdRB91,Ste94,DVS95]. The complications with reflection mainly have one common cause: its self-referential nature creates confusion between what is "meta" and what is "base". Sometimes what is "meta" can be "base" at the same time and vice versa. This confusion inevitably has its impact on the usability of reflective systems and programs because they tend to be very hard to understand. A "simple" meta system is much easier to understand and use because it has a clean separation of meta level and base level.

Our approach clearly falls into the category of "simple" meta programming. There is a very clear separation between the base program and the meta program. This can be seen easily because the base language and the meta language are actually different programming languages. The base language is Java and the meta language is a logic programming language.

This places our approach of AOLMP somewhere in between AOP and full reflection. Aspect-oriented programming is not really programming, in the sense that an aspect language is typically a restricted declarative formalism that allows asserting things about base programs, without offering the power of a programming language. Therefore aspect programs are "meta" since they are *about* programs. However, they are not real programs themselves. Our approach replaces the multitude of restricted declarative formalisms, that special purpose aspect languages typically are, by one general purpose (also declarative!) logic programming language. We however purposefully do not provide a fully reflective system because we want to keep a clear separation between meta level and base level. Mixing the two in moving towards full-fledged reflection would add confusion and complications without significantly improving the generality or expressive power of the system.

### 6.4 Synchronization

We have based the simple example weaver used throughout this paper on Lopes' work on D [LK97,Lop97]. The weaver presented in this text only serves as an example to illustrate the principle of AOLMP and its advantages. We did not intend to give a better solution to the particular problem of synchronization. Our example therefore is greatly simplified omitting many important features such as synchronization between multiple classes, distinction between synchronization per-class or per-instance, a more complicated and realistic locking strategy etc. However, all of these could be implemented with not too much difficulty in the logic framework we presented.

A great deal of the work involved in defining an aspect language for solving a particular problem, for example synchronization, is in deciding how exactly to describe a particular aspect. For the example of synchronization we were relieved of this task because we borrowed Lopes' design. Our approach does not offer a magical solution here: the AOP implementor still has a large responsibility in analyzing the problem and designing an aspect language for it. In this respect, the only difference is that one is designing an interface to a library of rules rather than inventing specialized syntax. Our example does illustrate how AOLMP simplifies weaver implementation once the interface to it has been designed. It also facilitates user defined variations of the aspect language. This alone may indirectly help in designing aspect languages by making it easier to experiment with alternatives.

### 6.5 The Transformational Approach to AOP

Work in progress by Fradet and Südholt [FS98] proposes a transformational approach to AOP. They focus on an interesting class of aspects which can be described by source to source program transformations. It is difficult to make very punctual comparisons with our approach because their work is still very much in progress. Nevertheless, we want to compare the approaches on some conceptual points because of the obvious similarities.

To a large extent our approach has a lot in common with their approach especially when considered from the AOP implementors point of view, which they seem to have focussed on. Our approach to weaver implementation in the given example is indeed a form of source to source transformation. From this point of view we simply use the logic language as a general purpose programming language which happens to be convenient as a transformation language because of its powerful pattern matching capabilities.

The transformational approach of Fradet and Südholt is based on a special purpose transformation language, specially designed for expressing transformations on abstract syntax trees. New kinds of aspect declarations consist of extensions to the abstract syntax and consequently also the concrete syntax of the language. They point out that only pattern matching on syntactic properties is not always sufficient to describe or guide the transformation and they propose to use static program analysis to remedy this.

Summarizing the above we discern three essential components in the system they envision:

1. A special purpose transformation language: To specify the aspect language. It can also serve as a generic weaver implementation language.
2. A static program analyzer construction toolkit: To deduce different kinds of static information needed in different kinds of aspects.
3. A parser generator toolkit: For creating new syntax for declaring aspects.

From the implementor's viewpoint, the added potential of our approach is found in its uniformity: the logic formalism provides a convenient substitute for all three components. A logic language can serve not only as a transformation language, but also as a general meta-programming language, well suited for implementing other kinds of meta programs, such as for example static program analyzers.

It might be useful to occasionally define special purpose syntax for some aspects. Therefore including a parser generator toolkit is not useless and could also be a useful addition to our approach. There is however no real need for it since the logic language itself can serve as a general-purpose aspect declaration language.

In many ways our work and their work is complementary in nature. Their approach is more ambitious with respect to formal underpinnings for transformational aspects, whereas ours is more directly inspired from a concrete implementation viewpoint. They also focus mostly on the aspect implementors viewpoint and consider an aspect language itself immutable once it has been defined. In contrast, this paper stresses the importance of user level programmability of aspects, for the purpose of extending the aspect language.

While it is not an central issue in this paper that the example weaver is transformational, this is still an interesting observation. It is to be expected that theoretical results from the transformational approach will be directly applicable in constructing a generic-weaver library of logic rules.

## 7 Conclusion

We have illustrated how an aspect language can be embedded in the logic paradigm by representing aspect declarations as logic facts. We illustrated this for a simplified version of the COOL aspect language proposed by Lopes.

That aspects are expressed by means of a full-fledged logic language represents an important advantage over using more limited special-purpose aspect languages: the logic language can serve uniformly as a formalism to declare aspects as logic facts and as a meta language for aspect-oriented logic meta programming using rules.

Aspect-oriented meta programming is useful for both users and implementors of AOP. AOP users can extend the aspect language on the fly, defining new kinds of aspect declarations. Some interesting examples where shown which do not require the user to descend to the level of the weaver implementation. This was possible because the new declarations could be defined in terms of already existing ones. Thus, the new declarations could be defined as a kind of sophisticated syntactic sugar and implemented by means of logic rules transforming new declarations into already existing ones. This kind of programming is very interesting because it allows making the reasoning underlying the aspect declarations explicit. This was illustrated clearly by the example, where AOLMP made it possible to explicitly capture the reasoning about how modification and inspection of state is the underlying motivation for the synchronization code in a `Stack` class. The same technique is also useful at the level of the weaver implementation because often some of the aspect declarations can be defined in terms of other more low-level aspect declarations.

### 7.1 About the generality of the conclusions

Despite the restricted and simplified nature of the example given in this paper, we can draw some conclusions about AOLMP which are valid in a broader context because:

1. The logic language is a full-fledged (Turing complete) programming language. Consequently, it is theoretically possible to implement any conceivable weaver in it. Some weavers will be harder to implement than others, but the same is true for weaver implementation in any other language.
2. There are other known aspects which have join points very similar to the synchronization aspect such as debugging and tracing. These at least would be equally easy to implement.
3. From the user's points of view, any aspect language implemented as a library of logic rules benefits from AOLMP for building on top of the aspect language, since this is independent of the internal complexity of the library implementation itself. In an extreme case, the library may also be a logic "front end" to a weaver implemented in another programming language altogether. This would yield the advantages of user-level AOLMP without requiring a logic implementation for the weaver itself.

It is important to realize that there is no magic. Designing a good aspect language is tricky business regardless of what medium is being used to implement it. This is also true when implementing them as libraries of logic rules. For example, one of the features omitted from our simplified example weaver is interclass synchronization. Theoretically implementing it in the logic language is not a problem. However, it is not possible to implement it by only building on top of the simplistic library exemplified in this paper. It would require the rewriting of at least a small portion of the library of rules since an inter-class synchronization policy would require a more general form of synchronization declaration which names the class together with the method:

```
mutuallyExclusive(Stack<push>,Stack<pop>).
```

The more general use of this type of declaration with methods in two distinct classes cannot be defined in terms of the building blocks provided by the simplified library. Note that it would be easy to go the other way around and provide the intra-class syntax on top of the more general inter-class syntax:

```
mutuallyExclusive(?cls<?m1>,?cls<?m2>) :- mutuallyExclusive(?cls,?m1,?m2).
```

This merely shows that aspect languages should be designed with care, just like every other programming language, library or software system.

## 8 Acknowledgments

I offer my gratitude to Kim Mens, Tom Tourwé, Tom Mens, Wolfgang De Meuter and Rob Nebbe for their proofreading and valuable comments on this paper and for the many inspiring discussions.

I also thank the organizers of the third AOP workshop: Cristina Lopes, Gregor Kiczales, Bedir Tekinerdogan and Wolfgang de Meuter, for their comments on an earlier version of this paper.

I thank Pascal Fradet and Mario Südholt for their comments on section 6.5 and for an interesting discussion about the subject over email.

## References

[Bar95] J. Barklund. Metaprogramming in logic. *Encyclopedia of Computer Science and Technology*, 33:205–227, 1995. Also available as UPMAIL Technical Report No. 80.

[BGV90] R. Ballance, S. Graham, and M. VanDeVanter. The Pan Language-Based Editing System for Integrated Development Systems. In *Proc. 4th ACM SIGSOFT Symp. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 77–93, 1990.

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

[CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, May 1992.

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

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.

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

[DVS95] Kris De Volder and Patrick Steyaert. Construction of the Reflective Tower Based on Open Implementations. Technical Report vub-prog-tr-95-01, Programming Technology Lab, Vrije Universiteit Brussel, 1995.

[FGH+90] Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. $\lambda$ Prolog: An extended logic programming language. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (Kaiserslautern, West Germany)*, volume 449 of *lncs*, pages 754–755, Berlin, 1990. sv.

[FS98] Pascal Fradet and Mario Südholt. Aop: towards a generic framework using program transformation and analysis. In Serge Demeyer and Jan Bosch, editors, *ECOOP 98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 394–397. Springer Verlag, 1998.

[HG98] P. M. Hill and J. Gallagher. Meta-programming in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5:421–498, January 1998.

[HL94] P. Hill and J. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, MA, 1994.

[ILGK97] J. Irwin, J.-M. Loingtier, J. R. Gilbert, and G. Kiczales. Aspect-oriented programming of sparse matrix code. *Lecture Notes in Computer Science*, 1343:249–??, 1997.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KL98] Gregor Kiczales and Cristina Videira Lopes. Tutorial 64: Aspect-oriented programming using aspectj. *OOPSLA'98 Tutorial Notes*, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-007 P9710047, Xerox Palo Alto Research Center, http://www.parc.xerox/aop, 1997.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in aspectj. In Serge Demeyer and Jan Bosch, editors, *ECOOP 98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, 1998.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, November 1997.

[Mae87] Patti Maes. *Computational Reflection.* Phd thesis, Vrije Universiteit Brussel, Artificial Intelligence Lab., Brussels, Belgium, January 1987.

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

[MLTK97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming. In Jan Bosch and Stuart Mitchell, editors, *ECOOP 97 Workshop Reader*, Lecture Notes in Computer Science, pages 483–496. Springer Verlag, 1997.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[Smi82] Brian C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, MIT, January 1982. Also available as MIT/LCS/TR-272.

[Smi84] Brian C. Smith. Reflection and semantics in LISP. Report ISL-3, ACM/ Xerox PARC, Intell. Systems Lab., Palo Alto, CA, June 1984.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., second edition, 1994.

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

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA'98*, 1998.

# Author Index