



## **Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin**

### **Copyright statement**

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

### **Liability statement**

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

### **Access Agreement**

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

# SUPPORTING META-TYPES IN A COMPILED, REFLECTIVE PROGRAMMING LANGUAGE

A thesis submitted to the  
University of Dublin, Trinity College,  
in fulfillment of the requirements for the degree of  
Doctor of Philosophy (Computer Science)

Tilman Schäfer

Distributed Systems Group  
Department of Computer Science  
Trinity College, University of Dublin

September 2001

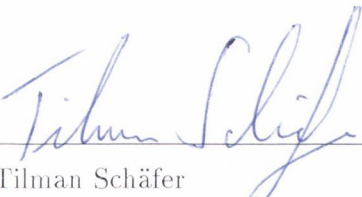




THESIS 6608

# Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

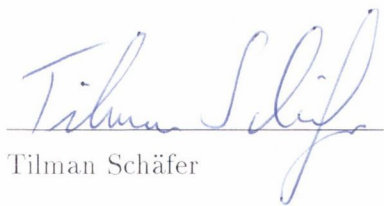


Tilman Schäfer

30 September 2001

# Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

A handwritten signature in blue ink, reading "Tilman Schäfer", written over a horizontal line.

Tilman Schäfer

30 September 2001

# Acknowledgments

The work described in this thesis would never have been possible without the help and support of a number of people. First of all, I would like to thank my supervisor, Dr. Vinny Cahill, for providing dedicated and profound guidance, moral support and constructive criticism over the entire time.

I would also like to thank the people in the Iguana project who have been a great company over the last couple of years and who contributed to many of the ideas presented in this thesis, namely Peter Haraszti, Jim Dowling and Barry Redmond.

Last but not least I want to thank all the other members of DSG for many fruitful and insightful discussions, in particular Johan Anderson for providing me with food and Marco Killijian for proof reading.



# Abstract

Software engineering in a distributed, heterogeneous environment is faced with a number of challenges. With distribution comes the need for synchronisation, transactions, and fault-tolerance while support for different operating systems also has to be taken into account. The distributed and dynamic nature of applications in such an environment also imposes special requirements in terms of adaptability: components being added to or removed from the system may require applications to adapt themselves to the changed environment. Development should moreover not be restricted to one particular programming language.

It soon becomes apparent that a large amount of effort in the development of distributed applications is spent on the specification and implementation of these non-functional requirements. Moreover, the actual functional parts of the application become more and more intertwined with and dependent on the parts that deal with the fulfillment of non-functional requirements. As a consequence, little changes in the design of an application may result in the modification of large amounts of code throughout the system.

In order to tackle these problems modern software engineering is clearly advancing towards middleware and component based systems such as the Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB) and the System Object Model (SOM). These systems define a platform-independent object model, usually with support for distribution, persistence and transactions, which allow the rapid development of applications in a heterogeneous, distributed environment.

However, these systems are primarily targeted towards the development of business applications and can hardly be employed for the development of low-level, performance critical applications such as operating systems and embedded systems. Moreover, commercial systems are 'closed' in the sense that users do not have direct control over the implementation of the various facilities that they provide and can, in general, not provide customised or additional services. Adaptation and customisation are however desirable properties in order to keep pace with evolving and changing environments that are characteristic of distributed systems.

Computational reflection and meta-level architectures offer an alternative approach towards the development of complex systems in that they provide a clean separation of concerns and a structured approach of making implementations more open. In order to provide generic language support for distributed computing as well as the advantages of component based programming, we have developed a composable and extensible meta-level architecture for a compiled, object-oriented programming language. The model proposed offers a high degree of flexibility and can be used to extend object-oriented programming languages to support a variety of application-independent behaviours such as distribution, persistence, transactions and fault-tolerance. The main contributions of the work described in this thesis consist of the provision of a reflective object-model that is

- **Composable:** We provide a mechanism that allows the *automatic* composition of individually defined object behaviours. If the default semantics does not yield the desired behaviour, user defined composition rules can be applied manually.
- **Dynamic:** Running applications can dynamically switch between different object models, allowing objects to evolve in changing environments. This kind of functionality could previously only be achieved in interpreted platforms.
- **Efficient:** By choosing a compiled language we achieve performance advantages over interpreted languages and present an architecture that is suitable for application domains such as legacy systems, operating systems and embedded systems.

As an outcome of our work we introduce the concept of a *meta-type*. Meta-types provide a further level of abstraction that encapsulate most of the functionality offered by the reflective language extension. In a number of case studies we have applied the reflective programming model in order to validate our claims.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                      | <b>9</b>  |
| 1.1      | Open Implementation . . . . .                            | 10        |
| 1.2      | Introduction to Reflection . . . . .                     | 11        |
| 1.2.1    | Computational Reflection . . . . .                       | 11        |
| 1.2.2    | Reification . . . . .                                    | 12        |
| 1.2.3    | Structural versus Behavioural Reflection . . . . .       | 13        |
| 1.2.4    | Base–Meta Level Separation . . . . .                     | 13        |
| 1.2.5    | Metaobjects and Metaobject Classes . . . . .             | 13        |
| 1.2.6    | Metaobject Protocol . . . . .                            | 14        |
| 1.2.7    | Metaclasses . . . . .                                    | 14        |
| 1.3      | Reflective Programming Languages . . . . .               | 14        |
| 1.4      | Aims and Objectives . . . . .                            | 15        |
| 1.5      | Iguana . . . . .   | 15        |
| 1.6      | Contribution of this Thesis . . . . .                    | 16        |
| 1.7      | Roadmap . . . . .  | 18        |
| <b>2</b> | <b>Reflective, Object-Oriented Programming Languages</b> | <b>19</b> |
| 2.1      | General overview . . . . .                               | 21        |
| 2.2      | Smalltalk . . . . .                                      | 23        |
| 2.3      | C++ . . . . .  | 24        |
| 2.4      | Objective-C . . . . .                                    | 26        |
| 2.5      | Java . . . . .   | 28        |

---

|       |                                  |    |
|-------|----------------------------------|----|
| 2.6   | OpenC++ v1 . . . . .             | 29 |
| 2.6.1 | Type of Reflection . . . . .     | 29 |
| 2.6.2 | Reflective Facilities . . . . .  | 29 |
| 2.6.3 | Programming Model . . . . .      | 30 |
| 2.6.4 | Known Applications . . . . .     | 32 |
| 2.6.5 | Performance . . . . .            | 33 |
| 2.6.6 | Support for Meta-Types . . . . . | 33 |
| 2.7   | OpenC++ v2 . . . . .             | 33 |
| 2.7.1 | Type of Reflection . . . . .     | 33 |
| 2.7.2 | Reflective Facilities . . . . .  | 34 |
| 2.7.3 | Programming Model . . . . .      | 35 |
| 2.7.4 | Performance . . . . .            | 37 |
| 2.7.5 | Known Applications . . . . .     | 38 |
| 2.7.6 | Support for Meta-Types . . . . . | 39 |
| 2.8   | CodA . . . . .                   | 40 |
| 2.8.1 | Type of Reflection . . . . .     | 40 |
| 2.8.2 | Reflective Facilities . . . . .  | 40 |
| 2.8.3 | Programming Model . . . . .      | 42 |
| 2.8.4 | Known Applications . . . . .     | 42 |
| 2.8.5 | Performance . . . . .            | 43 |
| 2.8.6 | Support for Meta-Types . . . . . | 43 |
| 2.9   | ABCL/R . . . . .                 | 44 |
| 2.9.1 | Type of Reflection . . . . .     | 44 |
| 2.9.2 | Reflective Facilities . . . . .  | 44 |
| 2.9.3 | Programming Model . . . . .      | 45 |
| 2.9.4 | Performance . . . . .            | 47 |
| 2.9.5 | Known Applications . . . . .     | 48 |
| 2.9.6 | Support for Meta-Types . . . . . | 48 |
| 2.10  | Discussion . . . . .             | 48 |



|          |   |           |
|----------|---|-----------|
| <b>3</b> | <b>The Iguana Reflective Programming Model</b>        | <b>50</b> |
| 3.1      | General Overview . . . . .                            | 51        |
| 3.2      | Reification Categories . . . . .                      | 51        |
| 3.2.1    | Structural Reification Categories . . . . .           | 52        |
| 3.2.2    | Behavioural Reification Categories . . . . .          | 52        |
| 3.2.3    | Extent of Reflection . . . . .                        | 53        |
| 3.2.4    | Dependencies between Reification Categories . . . . . | 53        |
| 3.2.5    | Protocols and Protocol Selection . . . . .            | 54        |
| 3.2.6    | Shared Behaviour . . . . .                            | 57        |
| 3.3      | Metaobject Composition . . . . .                      | 58        |
| 3.3.1    | Default Composition Semantics . . . . .               | 59        |
| 3.3.2    | Modifying Composition . . . . .                       | 61        |
| 3.3.3    | Discussion . . . . .                                  | 64        |
| 3.3.4    | The Iguana Syntax . . . . .                           | 64        |
| 3.4      | Summary . . . . .                                     | 65        |
| <b>4</b> | <b>The Iguana/C++ Implementation</b>                  | <b>67</b> |
| 4.1      | Applying the Model . . . . .                          | 67        |
| 4.2      | General Overview . . . . .                            | 68        |
| 4.3      | The Iguana Meta-Level Class Hierarchy . . . . .       | 69        |
| 4.4      | The Pre-Processor . . . . .                           | 71        |
| 4.5      | Source-to-Source Translation . . . . .                | 72        |
| 4.5.1    | Protocol Definitions . . . . .                        | 72        |
| 4.5.2    | Adding Introspection . . . . .                        | 73        |
| 4.5.3    | Bootstrapping . . . . .                               | 74        |
| 4.5.4    | Adding Intercession . . . . .                         | 75        |
| 4.5.5    | Instance protocol selection . . . . .                 | 79        |
| 4.5.6    | Run-time Checks . . . . .                             | 79        |
| 4.5.7    | Nested Expressions . . . . .                          | 80        |

---

|          |  |            |
|----------|--|------------|
| 4.6      | Dynamic Meta-Type Selection . . . . .                      | 81         |
| 4.6.1    | Meta-level Reconfiguration . . . . .                       | 81         |
| 4.7      | The C++ Default Protocol . . . . .                         | 85         |
| 4.7.1    | Method Invocation . . . . .                                | 85         |
| 4.7.2    | State Access . . . . .                                     | 85         |
| 4.7.3    | Object Creation . . . . .                                  | 87         |
| 4.8      | Restrictions . . . . .                                     | 87         |
| 4.8.1    | Automatic objects . . . . .                                | 87         |
| 4.8.2    | Arrays . . . . .   | 88         |
| 4.8.3    | Aliasing . . . . .   | 88         |
| 4.9      | Summary . . . . .  | 89         |
| <b>5</b> | <b>Reflective Programming with Iguana</b>                  | <b>91</b>  |
| 5.1      | Using Introspection . . . . .                              | 91         |
| 5.2      | Boundary Checks for Arrays . . . . .                       | 92         |
| 5.3      | Run-time Adaptation of Systems Software . . . . .          | 93         |
| 5.3.1    | The minimal Buffer Manager . . . . .                       | 94         |
| 5.3.2    | Adaptation using Design Patterns . . . . .                 | 94         |
| 5.3.3    | Adaptation using Reflection . . . . .                      | 95         |
| 5.3.4    | State Transfer . . . . .                                   | 97         |
| 5.3.5    | Discussion . . . . .                                       | 98         |
| 5.4      | A Meta-Type for Persistent Objects . . . . .               | 98         |
| 5.4.1    | Overview of Object Persistence . . . . .                   | 99         |
| 5.4.2    | Implementing Persistent Objects using Iguana/C++ . . . . . | 99         |
| 5.4.3    | Using Persistent Objects . . . . .                         | 101        |
| 5.4.4    | Adapting the Meta-Level . . . . .                          | 104        |
| 5.4.5    | Discussion . . . . .                                       | 105        |
| 5.5      | Summary . . . . .  | 106        |
| <b>6</b> | <b>Evaluation</b>  | <b>108</b> |

---

|          |   |            |
|----------|---|------------|
| 6.1      | Overhead, Where and Why . . . . .                               | 108        |
| 6.1.1    | Design Level . . . . .  | 109        |
| 6.1.2    | Implementation Level . . . . .                                  | 109        |
| 6.1.3    | Host-Language Level . . . . .                                   | 111        |
| 6.1.4    | Application Level . . . . .                                     | 112        |
| 6.2      | Discussion . . . . .  | 113        |
| 6.3      | Other Optimisation Techniques . . . . .                         | 114        |
| 6.3.1    | Partial Evaluation . . . . .                                    | 114        |
| 6.3.2    | Partial Evaluation of Iguana/C++ . . . . .                      | 116        |
| 6.3.3    | Elimination of Run-Time Checks . . . . .                        | 117        |
| 6.4      | Summary . . . . .   | 117        |
| <b>7</b> | <b>Conclusion and Future Work</b>                               | <b>118</b> |
| 7.1      | Understanding Reflective Programming . . . . .                  | 120        |
| 7.2      | Performance and Optimisation . . . . .                          | 121        |
| 7.3      | Future Work . . . . .   | 121        |
| 7.3.1    | Reflection and Design Patterns . . . . .                        | 121        |
| 7.3.2    | Composition of Meta-Types . . . . .                             | 121        |
| 7.3.3    | Compiler Support for Reflective Programming Languages . . . . . | 122        |
| 7.3.4    | Formalisation of Meta-Types . . . . .                           | 122        |
| 7.3.5    | Applying Reflection . . . . .                                   | 122        |

## Introduction

**REFLECTION, n.** An action of the mind whereby we obtain a clearer view of our relation to the things of yesterday and are able to avoid the perils that we shall not again encounter.      *Ambrose Bierce, The Devil's Dictionary.*

Modern software engineering has to keep pace with ever evolving runtime environments that place new demands on the development of computer applications. Since the advent and popularity of the Internet, for example, a completely different programming environment has emerged, introducing a demand for distributed and concurrent applications. Long-running systems such as operating systems are moreover faced with dynamic changes taking place at run-time, with components being added to or removed from the environment, requiring the system to adapt itself to the changed surroundings. As systems grow larger and become increasingly complex, the question arises as to what extent a programming language can assist developers in the task of developing such applications.

One approach is to provide direct support for some element of distributed computing from within the programming language. Examples of those languages include C\*\* [Tay93, VCdP93], Emerald [JLHB88] and Java [GJS96]. C\*\* is an extension of C++ and provides the application programmer with support for persistence, distribution, and transactions. Emerald provides object migration and an extended exception handling mechanism to recover from partial failures [Hut96]. Java on the other hand features built-in support for multi-threading and standard libraries for remote method invocation, making the development of concurrent, distributed applications easier.

These languages may provide a tailored solution for one individual programming environment but are limited in the sense that the combination of multiple facilities is not possible. The designer of a programming language is therefore faced with the dilemma of deciding which concepts are to be directly supported by the language and which are not. Each



supported feature usually requires a specific syntax and/or run-time support, making the language more complex and difficult to implement.

A different approach that addresses these problems is exemplified by component-based systems such as Sun's Enterprise Java Beans (EJB) [Mic01], the Common Object Request Broker Architecture (CORBA) [Gro95] and IBM's System Object Model (SOM) [Lau94]. These systems provide a platform-independent framework for building applications in a distributed environment and usually come with built-in support for distribution, persistence and transactions. Programmers simply select or deploy a pre-defined component into their application, subject to some programming conventions that one has to obey.

However, these systems are primarily targeted towards the development of business applications and can hardly be employed for the development of low-level, performance critical applications such as operating systems and embedded systems. Moreover, these systems are 'closed' systems, meaning that users do not have direct control over the implementation of the various facilities and can in general not provide customised services. Support for different operating systems, a major prerequisite for the development in a heterogeneous, distributed environment, might also not always be available. Interoperability between these systems does exist to some extent, for example, in the form of CORBA-COM bridges, but these come at the price of increased complexity and introduce compatibility problems.

Since the provider of a software component can in general not foresee all possible usage scenarios and system requirements, a new programming model is needed that is in some sense "open" and does not restrict its users by the decisions made by its designers.

## 1.1 Open Implementation

The traditional black-box model of software engineering is aimed at providing users with software components that have some well-defined behaviour while at the same time shielding them from complex or irrelevant implementation details. Kiczales proposed an alternative model of abstraction for software engineering known as the *open implementation* model [Kic91, KLM<sup>+</sup>93, KTW92, KL93]. The basis of the open implementation model is the argument that while the traditional black-box model of abstraction shields clients from having to know the details of how a particular component is implemented, it also prevents them from altering those details when desirable. An open implementation therefore exhibits some (but not all) internal details to its users, allowing them to customise components if required.

The open implementation approach should not be seen as a contradiction to the classical black box approach. Instead, it should be understood as an extension to the former, advocating a fairer trade-off between performance on the one hand and abstraction on the other. Take the Unix system call `advise` as an example. It allows programmers to 'advise' the underlying virtual memory manager about the usage pattern of a block of memory mapped into the application's address space. It therefore exposes some hidden functionality of the underlying operating system to the user, resulting in a more efficient implementation.

However, this example can merely be regarded as an ad hoc approach to open implementations and only allows very restricted access to the operating system's internals. What is therefore needed is a methodology that allows a more rigorous and structured access to a component's internals. Computational reflection, as described in the following sections, can serve as an underlying technology for building open implementations in that it equips a computational system with a separate representation of itself.

Inevitably, a number of issues arise, namely to what extent internal implementation details should or can be exposed to the user and how the erroneous or deliberate misuse of such information can be prevented. These issues will follow us through this thesis and will be dealt with more thoroughly as they arise.

## 1.2 Introduction to Reflection

This section introduces some of the main concepts and terminology used throughout the remainder of this thesis. In most cases the concepts and terminology presented here are not specific to the author's view and are widely used in the reflection community at large. Unfortunately, there is not complete consensus on the use of all of these terms even within the reflection community.

### 1.2.1 Computational Reflection

Computational reflection, or reflection for short, was described by [Mae87] as "the process of reasoning about and/or acting upon oneself". In a non-technical sense, humans "reflect" when they reason about their own state of mind, leading to insights about their existence, and inadvertently generating new knowledge that in turn might feed back and change their state of mind. When Descartes states "I think, therefore I am" [Des37], he reflects and captures his own state of mind ("I think"), leading to the realisation of his own existence ("therefore I am"). Although this example is only used in a metaphorical sense,

it illustrates some of the features that are characteristic of a reflective system, namely:

**Introspection:** Introspection is the process of observing one's internal state or structure. For a system to be able to introspect and reason about itself it needs a *self representation*.

**Self representation:** In the example above, 'I' represents Descartes himself, his notion of himself.

**Causal connection:** A computational system contains an internal representation of some part of the external world, its domain (for example in form of data and code). A system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other [Mae87]. A reflective system is one whose self-representation is causally connected to itself, or which is self-referential. Again, as Descartes deduces "therefore I am", he generates new knowledge which in turn feeds back into his self-representation in a causally connected way.

Reflection can be diverse and occurs in many different domains. So how does the above relate to object-oriented programming languages? An object-oriented programming language (OOPL) provides concepts such as classes, inheritance, method invocation and state access. Applications written in such a language conceptually consist of a set of objects exchanging messages and performing computations upon reception of messages. In a reflective programming language, introspection enables objects to query their own state, for example to find out about their class, the methods and attributes they provide, etc.<sup>1</sup> Moreover, the self representation may also provide a description of the object's behaviour, i.e., code that describes the object specific semantic of message sending/receiving, method dispatching, etc.

### 1.2.2 Reification

Any OOPL supports a number of features such as object creation, method invocation, state access and inheritance. A programming language therefore provides a syntax that is used by the programmer to express a particular concept as well as a semantic that describes how the concept is interpreted or executed. In traditional OOPLs the syntax/semantics relationship is fixed, meaning that there is no (easy) way for a programmer to modify the

---

<sup>1</sup>This information might not always be apparent, especially if the language is not typed or if it features dynamic typing.



syntax of the language or to modify the way a specific language construct is interpreted. *Reification* refers to the process of making the implementation of these features explicit within the language so that they can be reflected upon. In particular, reification allows the implementation of these features to be modified from within the language.

### 1.2.3 Structural versus Behavioural Reflection

Structural reflection is the ability of a language to provide a complete reification of both the program currently being executed as well as complete reification of its abstract data types (e.g. its classes) [DM95]. Behavioural reflection on the other hand is the ability of the language to provide a complete reification of its semantics. Behavioural reflection requires a mechanism to intervene in the execution of the current program in order to execute some reflective code that will analyse or modify the course of events.

### 1.2.4 Base–Meta Level Separation

In general we can distinguish between the base-level of a system and its meta-level. The base-level contains the code and data concerned with the application domain while the meta-level contains the code and data concerned with the base-level. We can talk about a system executing at the base-level when performing domain-specific computation and executing at the meta-level when carrying out computation about the base-level. Thus the (self-)representation of the base-level exists as the data of the meta-level and reflective computation is performed when executing at the meta-level. The transition from performing domain computation to performing reflective computation is characterised as a transition from base to meta-level execution.

### 1.2.5 Metaobjects and Metaobject Classes

In a reflective OOPL, a *metaobject* (also known as a *meta-level object*) is an object that holds information about the implementation and interpretation of some object [Mae87]. The set of metaobjects that represent a particular object together form its *meta-level*. The set of metaobjects representing all the base-level objects that make up an application together form the application's meta-level.

In general, we can distinguish between *structural* and *behavioural* metaobjects. Structural metaobjects are those whose only purpose is to store information about the objects that they represent. For example, structural metaobjects might be used to store the identity of an object's class, the list of types that it implements, the identifiers of its methods, or



even the code of its methods. On the other hand, behavioural metaobjects are used to implement various aspects of an object's behaviour. For example, behavioural metaobjects might be used to implement object creation or method invocation.

### 1.2.6 Metaobject Protocol

Metaobjects provide the necessary functionality to control and represent base-level objects. Communication between base and meta-level objects takes place through a well defined set of interface functions. The set of these interface functions is referred to as the *metaobject protocol* [KP96], or MOP for short.

### 1.2.7 Metaclasses

Some object-oriented programming environments, including Smalltalk and SOM, provide *metaclasses* as the natural base of their object model: metaclasses are classes whose instances are themselves classes. They define the common properties that all classes should provide, for example, how methods are dispatched. In Smalltalk there is a fixed one-to-one relationship between a class and its metaclass, every metaclass is defined by the Smalltalk run-time environment and cannot be altered by the application programmer. SOM on the other hand allows the development and composition of independent metaclasses [FDM94].

## 1.3 Reflective Programming Languages

Rather than supporting a fixed set of facilities for a specific programming environment, reflective programming languages are designed so that their implementation is more open, allowing adaptation of the language to varying, non-functional requirements. In a reflective programming language, programmers can alter the syntax and semantics of the language itself and thereby adjust the behaviour of the language constructs to build a programming environment that best fits their needs. For example, if the language does not have built-in support for, let's say, distribution, it can be extended to allow the invocation of remote methods by modifying the semantics of a normal method call.

Lisp for example is a functional programming language with reflective facilities: functions are first class entities that can be manipulated as can any other values. This is achieved by the quoting mechanism where a function is simply represented as a quoted list of instructions. The quoted expression can then be evaluated by an interpreter, the *eval* function.

## 1.4 Aims and Objectives

Generally speaking, we want to extend the existing object-model of a programming language in such a way that it is possible to introduce new object semantics to that language. This extension should (from the application programmer's point of view) be:

- **Transparent:** the reflective extension should be as transparent as possible, without the need for application programmers to explicitly invoke meta-level computation.
- **Composable:** it should be possible to combine and augment object behaviours in a meaningful way, so for example to allow the construction of both persistent and remotely accessible objects.
- **Dynamic:** the object model should be able to evolve over time, enabling objects to adapt themselves to a changing environment.
- **Extensible:** there should be no fixed set of object behaviours. It should be possible to develop new or modify existing object models.

[DM95] describes what an ideal reflective language should offer:

“Ideally, a reflective language should support a methodology of reflective computations giving its users as much flexibility as possible. It should be possible to modify the behaviour of some construct for the whole execution of a program, or for short period of (execution) time. It should also be possible to modify the behaviour of some construct for all the program or only for some of its subparts. None of the languages currently proposed achieves this level of flexibility”

Mainly because of efficiency reasons, providing all of the functionalities as described above may not be feasible. We therefore have to find a good compromise between functionality on the one hand and performance on the other hand.

## 1.5 Iguana

As an approach towards providing full support for reflective programming, previous research has resulted in the development of Iguana [Gow97], a reflective programming model for object-oriented languages. It was first developed as an outcome of research into adaptable operating system software and is targeted towards application domains that are faced

with evolving run-time environments, requiring the application to adapt itself to the new environment, such as operating systems and middleware.

A design principle that has been influencing the Iguana model to a great extent was the idea of a complete decomposition of the underlying object model, allowing a very fine grained control over the reification of language features. However, in practical terms this approach has been found as to be too complicated, leading to an overspecialisation of MOPS [Paw98]. The following is a critique that summarises the main deficiencies of the Iguana model in its previous version:

- **Complexity:** The model was too complex, confronting the user with a number of features whose semantics were not fully understood and/or practical. It also required a knowledge of the intrinsic mechanisms of the host programming language which defeated language independence and ease of use.
- **Safety and robustness:** Many issues concerning safety and robustness in reflective programming had not been addressed sufficiently. Opening up a language and allowing the programmer to intercede with the underlying object model requires a profound understanding of the language's intrinsic mechanisms. Users of a reflective system not only have to make sure that their base-level application is semantically correct, they must also ensure that the code at the meta-level is consistent. The thoughtless use of reflection can easily affect the entire system and makes debugging even harder as errors can occur at both the base and meta-level.
- **Transparency:** The previous version did not separate sufficiently between base and meta-level code and exposed too many implementation details to the application programmer. Again, this raises the issue as to what extent an implementation can be opened up without leading to an overload of accessible features. Moreover, it was not possible to use reflective and non-reflective objects interchangeably.
- **Implementation:** Only a partial implementation existed. The feasibility and applicability of the Iguana model in practical terms still had to be proven.

## 1.6 Contribution of this Thesis

Within the context of this thesis we undertook a substantial re-design and re-implementation of the Iguana model. The focus was on simplifying the model in order to make it more accessible to meta-level and application programmers alike.



The challenge was to maintain the flexibility originally envisaged and to find a user interface which is at the same time simple, robust and expressive. As an outcome of our work we introduce the concept of a *meta-type* as such an interface that serves our purpose. The meta-type of an object characterises its object model and as such its non-functional behaviour. For example, a meta-type **Persistent** might correspond to an object-model supporting object persistence while a different meta-type **Remote** might support remote method invocation. We also provide a semantic for combining multiple meta-types by a mechanism similar to class inheritance. As will become clearer later, meta-types differ from metaclasses as described above in that we allow meta-types to be dynamically selected by instances of potentially *any* class, subject to a few restrictions. Metaclasses on the other hand define the behaviour of a class and thereby the behaviour of *all* instances of that class.

By applying the Iguana model to a compiled language, in this case C++, we also demonstrated that compiled platforms can exhibit a flexibility comparable to interpreted languages, while at the same time introducing only little or no overhead when the reflective features are not used.

We have chosen C++ as a host language for a couple of reasons. First, it is widely used. Second, it allows low-level access and is highly performant, both crucial properties for the development of operating systems and embedded systems. The Apertos operating system as described in [YKL94, LYiI95] for example uses reflection as an underlying design principle. Efficiency has been recognised as a major concern. Supporting run-time reflection in a compiled language can advance the research in that area by providing a development platform for efficient and low-level programming.

It should be noted however that the programming model described in this thesis is intended to be language independent and not restricted to C++ in particular. In fact, current work is investigating the implementation of the Iguana model in Java [RC00]. More specifically, we addressed the following issues in our implementation of meta-types:

**Metaobject composition:** Iguana's modular design allows the composition of meta-level objects in order to combine two or more behaviours. As a new feature, we provide automatic metaobject composition. If the default semantics does not yield the desired behaviour, user defined composition rules can be applied easily.

**Dynamic meta-typing:** Dynamic meta-typing allows individual objects to dynamically switch between different meta-level representations. This feature imposes special requirements in terms of the compatibility, safety and robustness of such transitions. As

a significant improvement to earlier versions of Iguana, we now provide a safe transition between different meta-level configurations, accomplished by a combination of static sub-typing rules and run-time meta-type checking.

**Understanding meta-level programming:** In contrast to object-orientation, reflection remains a rather esoteric programming paradigm and has not found its way into existing compiler technologies. In a number of case studies we show how meta-level programming can provide a generic framework for building certain types of applications, for example, systems that need to dynamically adapt to changing requirements.

**Optimisation:** The flexibility gained by extending the language and giving the users the opportunity to adapt the language to their needs usually incurs substantial interpretative overhead. Although this issue has been addressed by a number of researchers, it has not sufficiently been solved in a way that it would have increased the acceptance of reflective programming. We identify profitable targets and techniques for optimisation, allowing a more efficient implementation of the reflective programming features. Practical experiences with using reflection as a tool for providing properties such as fault-tolerance and group-based distributed systems however have shown that the costs due to the use of a metaobject protocol are negligible with respect to the execution costs of the meta-functional properties [FP98].

## 1.7 Roadmap

After this introduction to reflection and its terminology we will in the next chapter examine and review a number of reflective extensions to object-oriented programming languages. Since this thesis is primarily concerned with C++ as a target language, we will only briefly review Java-based platforms. In chapter 3 we will then describe the Iguana reflective programming model and the rationale behind its design. Chapter 4 then describes a concrete implementation of the Iguana model for C++. In chapter 5 we present a number of programming examples in order to fully demonstrate the application of the various reflective features of Iguana. The performance of Iguana/C++ is then evaluated in chapter 6, while chapter 7 summarises the work presented in this thesis and outlines future directions in this area.

## Reflective, Object-Oriented Programming Languages

**REVIEW, v.t.** To set your wisdom (holding not a doubt of it.  
Although in truth there's neither bone nor skin to it)  
At work upon a book, and so read out of it  
The qualities that you have first read into it.

*Ambrose Bierce, The Devil's Dictionary.*

With traditional, object-oriented languages, the development of systems that are characterised by a strong interdependency of functional and non-functional requirements can result in code where parts dealing with the functional aspects of the application are to a large extent intertwined with parts that deal with the non-functional aspects. Take remote method invocations as an example. Implementing remote objects using an architecture such as CORBA requires that arguments to remote methods are wrapped in special wrapper objects<sup>1</sup>. Although much of the functionality underlying remote objects can be hidden by using normal class inheritance, the application now contains code to wrap and unwrap arguments and return values of remote methods.

Reflective programming languages provide an alternative approach in the development of such systems in that they allow the implementation of the host language's object model to be made more open, allowing different object models to be supported simultaneously within a single language. In a reflective architecture, code dealing with the functional requirements resides at the base-level whereas code dealing with non-functional requirements, such as persistence and distribution, resides at a separate meta-level. Commu-

---

<sup>1</sup>This is especially the case when using a strongly typed language such as Java since Java does not automatically perform a type-cast from native data types to their equivalent CORBA types.



nication between base and meta-level is achieved by interceding with the execution or interpretation of language operations.

However, despite its obvious advantages, reflective programming remains to be a rather neglected programming paradigm and hasn't found its way into mainstream software design and implementation techniques. This lack of popularity stems from the fact that with the additional functionality comes complexity, leading to the perception that meta-level programming is hard and complicated. In fact, our view is that meta-level programming is not trivial and requires a thorough understanding of the semantics of both the underlying object model and the non-functional requirements that are to be embedded into the language's object model. On the other hand, once this separation has been achieved, the less experienced base-level programmer can gain from the clean separation of concerns and should be able to apply the extended object behaviour without otherwise being concerned about the existence of the meta-level architecture.

What is therefore needed is a new level of abstraction that encapsulates most of the implementation details of reflective programming for both the base and meta-level programmer. To draw an analogy, the concept of inheritance in OOPLs provides a powerful tool for encapsulating and combining software components. Inheritance as a concept would not have been successful if programmers weren't sufficiently shielded from its implementation details, so for example if they had to explicitly access and initialise virtual function tables. Unfortunately, this is exactly what a number of reflective programming languages require, namely the direct and explicit access of meta-level information.

In order to address the problems described above, we advocate a strong separation of the roles of base-level (or application programmer) and meta-level programmer. The base-level programmer should primarily be concerned with the implementation of the functional requirements of an application and should to a greatest extent be unaware of the existence of the reflective language extension. The (more experienced) meta-level programmer should be able to provide the non-functional requirements independent of the actual base-level code. As an interface for both the application and meta-level programmer, we introduced the concept of a meta-type. From the application programmer point of view, meta-types encapsulate most of the functionality of reflective programming and constitute simply components that can be selected into an application. From the experienced meta-level programmer point of view, meta-types build a framework for designing, composing and implementing object models. Conceptually, meta-types should be transparent, extensible and efficient in their implementation.

In this chapter we review a number of reflective programming languages and examine to what extent they support our notion of meta-types.

## 2.1 General overview

Reflective facilities can be found to a greater or lesser extent in most object-oriented programming languages. C++ for example only provides rudimentary introspection in the form of run-time type information (RTTI) [Str91], whereas Java is equipped with a full introspection API [Mic99]. However, mainstream programming languages only provide basic reflective features, in general only structural reflection, which is not enough to support meta-types. Therefore, a number of object-oriented languages have been extended in order to provide a more rigorous support for reflection.

Providing full structural and behavioural reflection is a task that largely depends on whether the underlying programming language is interpreted or compiled. In an interpreted language for example, the interpreter already constructs a substantial amount of meta-level information about the program to be interpreted. Extending the language to be reflective only involves the exposition of this meta-level information and the interpretation mechanism to the application programmer.

In a typical compiled language (most notably C++) on the other hand little or no meta-level information is kept in the run-time image. Adding reflection is dominated by the problem of maintaining the structural information beyond the compilation process and of extending the code with the appropriate hooks to exploit the meta-level information.

Figure 2.1 depicts the taxonomy of reflective programming languages. Depending on the type of programming language, compiled or interpreted, we can distinguish between various types of reflection: compile-time, run-time and load-time reflection.

With compile-time reflection, the compilation process of a program is controlled by a corresponding meta-level program. It can be viewed as a kind of 'smart pre-processing' where the semantics of the application code are modified in a context sensitive way. Examples of these architectures are OpenC++ v2 [Chi96] and OpenJava [Chi95].

Run-time reflection allows the interpretation/execution of language constructs to be modified at run-time. This is in general achieved by interceding with the execution of language operations. Run-time reflection offers the highest degree of flexibility but usually incurs the most overhead.

With load-time reflection, code modifications are carried out while loading the bytecode into the interpreter or virtual machine. Examples of load-time reflection include Javassist [Chi00], Binary Code Adaptation (BCA) [Ral98] and OMOS [OLL95]. Binary Code Adaptation can be used to add methods to class files in order to achieve integration and evolution of components when the source code is not available. OMOS [OLL95] is an

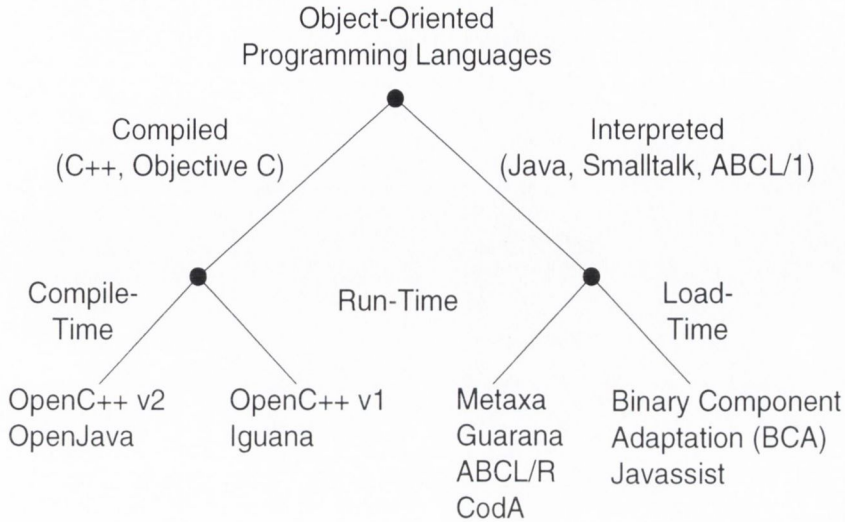


Figure 2.1: Taxonomy of reflective, object-oriented programming languages.

advanced linker/loader that inspects modules to be dynamically linked into applications. Taking application specific behaviour into account<sup>2</sup>, it is possible to load specialised implementations of system calls such as `fork` and `malloc`.

## Roadmap

In the remainder of this chapter we will first briefly review some of the more common object-oriented programming languages and discuss the inherent reflective features they provide. Then we will examine in turn a number of reflective extensions under the following aspects:

**Type of reflection:** Describes whether the platform provides compile-time, run-time or load-time reflection.

**Reflective facilities:** This section describes which features of the language are reified and how they can be accessed.

**Programming model:** This section describes the user interface provided by each language. More specifically, it describes how the programmer can define a new MOP implementation and how it can be applied to the application.

<sup>2</sup>Application-specific behaviour might include whether the program is multithreaded or not.



**Performance:** In this section we discuss the performance trade-offs which are incurred with the use of reflection.

**Known applications:** Describes concrete applications that have been written in the specific language.

**Support for meta-types:** Discusses to what extent the language supports the concept of meta-types. More specifically, we discuss whether the reflective language extension allows to compose and dynamically reselect meta-level configurations, how transparently the extension can be applied and whether type orthogonality is achieved.

## 2.2 Smalltalk

Smalltalk [GR83] was developed in the Xerox Palo Alto Research Center and was first released in 1980. It is an interpreted, fully object-oriented programming language featuring weak typing, multiple inheritance and automatic garbage collection).

### Inherent reflective features

Smalltalk contains many elements that can only be described as reflective. This can be seen from its consequent underlying programming paradigm: everything is an object. Classes in Smalltalk are first class entities, instances of a subclass of **MetaClass**. Every metaclass has exactly one instance, namely the class of the same name.

The class interface defines a number of methods that make it very easy to introspect on classes and methods, as the following example illustrates.

---

```
[01] listMethods: targetObject
[02]   | cl m |
[03]   cl := targetObject class.
[04]   m  := cl methodDictionary.
[05]   m inspect.
```

---

Figure 2.2: Introspection in Smalltalk: the function `listMethods` lists all methods of the object `targetObject`.

In the example above, function `listMethods` prints out a list of all methods for a given

object `targetObject`. First, the class (meta-)object is looked up and stored (line [03]). The method `methodDictionary` computes an ordered collection of all methods of that class, which is finally printed out in line [05].

[FJ89] discusses how further reflective features, such as customised method dispatching, could be incorporated into Smalltalk.

## 2.3 C++

C++ is an object-oriented extension of the C programming language and was designed by Bjarne Stroustrup. It was standardised by the ISO C++ standards committee in 1997. Because it originated from C, C++ is a hybrid language that supports the full range of low-level programming, procedural programming, object-oriented programming up to generic programming using templates.

### Inherent reflective features

C++ allows the redefinition or overloading of specific operators. The `new` operator for instance is a special method that is called whenever a new object is to be created on the heap. Its task is to allocate the appropriate amount of raw memory to contain the object's attributes. The `delete` operator on the other hand is invoked when an object is deleted in order to free the memory held by the object.

Overloading the `new/delete` operator can be viewed as reifying the memory management of objects and provides a means of interceding in the creation/deletion mechanism of objects. This can be useful to implement garbage collection or to allow objects to be located at specific addresses in memory. *Choices* [MKIC92] for example is a framework for constructing object-oriented operating systems that uses a reified heap manager which can be replaced in a running system. The code in figure 2.3 is an example of a customised `new` operator that prints out a message every time an object is created. `sz` contains the size (in bytes) of the object.

### RTTI

Run-time type information (RTTI) is an introspective feature of C++ that allows the querying of an object's type at run-time without entering it to perform an operation [Str91]. The type of an object is reified by instances of class `type_info`. The operator `typeid()` can be used to retrieve an object's type. Type information can be exploited

to perform optimised operations on objects, to realise object-oriented databases or for debugging purposes. In the example below, the serialisation of a data structure can be implemented in different ways if the actual type of the structure is known:

---

```

class list {
    ...      // implementation of a list
};

class linked_list : public list {
    ...      // implementation of a linked list
};

int serialize(list *ls, ostream *os) {      // dump a list to an output stream
    const type_info &ti = typeid(*ls);     // *ls can be of type list or linked_list
    if (ti == typeid(linked_list)){
        ... // do linked_list stuff
    } else {
        ... // do list stuff
    }
}

```

---

Exploiting type information as in the example above however should be avoided as it defeats transparency and inhibits software reuse. Instead, polymorphic member functions would be better used to implement type specific operations.

---

```

class Verbose {
    void *operator new(size_t sz){
        cout << "Calling Verbose::new(" << sz << ")" << endl;
        return malloc(sz);
    }
};

```

---

Figure 2.3: Overloading the `new`-operator in C++.



## 2.4 Objective-C

Objective-C [PW91] is another object-oriented extension to C and shares common characteristics with Smalltalk. It features dynamic typing and built in support for persistence (serialisation) and remote-method invocations.

### Inherent reflective features

Full type information (this includes name and type information of methods and instance variables and type information of method arguments) is available at run-time. User-defined classes in Objective-C are derived from a common base-class, class **Object**. This class contains a reference to the object's class definition and defines a number of methods to introspect its inheritance relationship and interface definition. Figure 2.4 shows the definition of class **Object** and its introspective member functions.

---

```
void listFields(id obj){
    Class cl = [obj class];
    int i;
    printf("Instance variables of class %s", cl->name);
    for (i=0; i< cl->ivars->ivar_count; i++){
        printf("Field: %s Type: %s", cl->ivars->ivar_list[i].ivar_name,
              cl->ivars->ivar_list[i].ivar_type);
    }
}
```

---

Figure 2.4: Introspection in Objective-C: the function `listFields` lists the names and types of the instance variables of a given object `obj`.

The code sequence in figure 2.4 illustrates how a list of instance variables together with a description of their types can be retrieved for a given object. In Objective-C, all objects are instances of a distinct data type `id`. `id` contains an instance variable that points to the object's actual class. By sending an object the `class` message it is possible to look-up an object's class definition.

```

@interface Object {
    Class isa; /* A pointer to the instance's class structure */
}
    /* Identifying classes */
- (Class)class;
- (Class)superClass;
- (MetaClass)metaClass;
- (const char *)name;

    /* Testing object type */
- (BOOL)isMetaClass;
- (BOOL)isClass;
- (BOOL)isInstance;

    /* Testing inheritance relationships */
- (BOOL)isKindOf:(Class)aClassObject;
- (BOOL)isMemberOf:(Class)aClassObject;
- (BOOL)isKindOfClassNamed:(const char *)aClassName;
- (BOOL)isMemberOfClassNamed:(const char *)aClassName;

    /* Testing class functionality */
+ (BOOL)instancesRespondTo:(SEL)aSel;
- (BOOL)respondsTo:(SEL)aSel;

    /* Introspection */
+ (IMP)instanceMethodFor:(SEL)aSel;
- (IMP)methodFor:(SEL)aSel;
+ (struct objc_method_description *)descriptionForInstanceMethod:(SEL)aSel;
- (struct objc_method_description *)descriptionForMethod:(SEL)aSel;
@end

```

Figure 2.5: Introspective member functions of class **Object** in Objective-C. A member function preceded by a plus sign specifies a function that can be used by class objects whereas a minus sign specifies a function that can only be used by instance objects.

## 2.5 Java

Java [GJS96] is an object-oriented programming language developed by Sun Microsystems and was first released in 1995. In common with Smalltalk it is byte-coded and interpreted as well as garbage collected. It employs a strong type checking and exception handling mechanism and provides built-in support for multi-threading.

### Inherent reflective features

With the Java Core Reflection API included with Sun's JDK versions 1.1 (1997) and 1.2 (1998) comes direct support for structural reflection from within a mainstream programming language. It enables Java code to discover information about the fields, methods and constructors of loaded classes. The majority of the methods provided by the Reflection API are for passive examination of the structure and attributes of a class, method or field. A small number of methods allow a new instance of a class to be created, a method to be invoked, or a field value to be altered. It is not possible at runtime to create a new instance of a method or field or to replace existing methods or fields.

The example shown in figure 2.6 illustrates the capabilities of the Java Reflection API to list all public methods for a given object.

---

```
void listMethods(Object targetObj) {
    String methClass, methName;
    Class targetClass = targetObj.getClass();
    System.out.println("Class = " + targetClass.getName());
    Method[] targetMethods = targetClass.getMethods();
    int i = targetMethods.length;
    for(int n = 0; n < i; n++) {
        methClass = targetMethods[n].getDeclaringClass().getName();
        methName = targetMethods[n].getName();
        System.out.println(methName + " " + methClass);
    }
}
```

---

Figure 2.6: Introspection in Java: the function `listMethods` lists all public methods (including inherited methods) of a given object.

## 2.6 OpenC++ v1

OpenC++ v1 is a simple run-time MOP that can be used to intercede with method calls and member variable access [Chi95], [Chi93]. It was developed by Shigeru Chiba at the Masuda Laboratory, University of Tokyo, Japan.

### 2.6.1 Type of Reflection

In OpenC++ v1, metaobjects exist during run time. However, declaring reflective methods/variables and associating a class with a metaobject class is done at compile time, which means objects cannot change their metaobject during run time.

### 2.6.2 Reflective Facilities

OpenC++ v1 only reifies a subset of the C++ object model, namely data member access, member function invocation and object creation/deletion. These aspects are represented by a single, predefined class called **MetaObj**. Figure 2.7 shows the specification of the OpenC++ MOP (excerpt).

```
class MetaObj {
  public:
    void Meta_MethodCall(Id method_id, Id category, ArgPac& args);
    void Meta_Assign(Id var_id, Id category, ArgPac& args);
    void Meta_Read(Id var_id, Id category, ArgPac& reply);
  protected:
    void Meta_HandleMethodCall(Id method_id, ArgPac& args, ArgPac& reply);
    void Meta_HandleAssign(Id var_id, ArgPac& args);
    void Meta_HandleRead(Id var_id, ArgPac& reply);
    void Meta_AssignValue(Id var_id, ArgPac& args);
    void Meta_ReadValue(Id var_id, ArgPac& reply);
    const char* Meta_GetClassName();
    const char* Meta_GetMethodName(Id method_id);
    const char* Meta_GetVarName(Id var_id);
    ...
};
```

Figure 2.7: The OpenC++ version 1 MOP. Class **MetaObj** defines the default behaviour of method invocation and state access. The behaviour can be adjusted by deriving from **MetaObj** and by redefining the handler methods.



### 2.6.3 Programming Model

**Defining a new MOP implementation** The methods defined in class `MetaObj` implement the default (C++) behaviour of method invocation, state access and object creation. `Meta_HandleMethodCall()` for instance is called in the event of a method invocation on a reflective object. This default behaviour can be adjusted by deriving subclasses of `MetaObj` and by redefining the handler methods.

**Selecting a MOP** At the base level, a class can be defined as either a normal C++ class or as a *reflective class*. A reflective class is one that is defined to have *reflect methods* and/or *reflect instance variables* and that has a metaobject class associated with it. The metaobject class contains the code that redefines how invocations of the reflect methods and accesses to the reflect instance variables in reflective instances of the base-level class are handled.

Modifying the behaviour of data member access and member function invocation is accomplished by deriving from that class and by associating a baselevel class with the new metaobject class. The code example shown in figure 2.8 illustrates the definition of a class with a reflective member function.

---

```

class Person {
    public:
        Person( char* name, int age);
        int Age();

        //MOP reflect:
        int IncAge(); /* This method is reflective */

    private:
        char* name;
        int age;
};
/* Associating class Person with a metaobject class */
//MOP reflect class Person: VerboseClass

```

---

Figure 2.8: Definition of a class with reflective member functions in OpenC++ v1.



The `//MOP reflect` directive declares the member function `IncAge()` as being reflective. Instances of class `Person` are associated with metaobjects of class `VerboseClass` using the `MOP reflect class` directive. `VerboseClass` has to be a subclass of the predefined class `MetaObj`.

The OpenC++ compiler then will generate a subclass of `Person`, named `refl_Person`. Instances of `refl_Person` are said to be *reflective objects*, as their behaviour is controlled by metaobjects. Thus it is still possible to create non-reflective objects by instantiating class `Person` directly.

Not necessarily all methods and member variables have to be reflective. Only those followed by the `//MOP reflect` directive will be controlled by the metaobject.

Whenever a reflective method of class `refl_Person` is called (or a reflective data member is accessed), the call is trapped and the flow of control is directed to the associated metaobject. The metaobject handles the actual argument list and return value of a method call. The argument list, return values and the values of variables are reified to be first-class entities at the meta level, a process similar to the marshaling of parameters in Remote Procedure Calls (RPC). The flow of control is depicted in figure 2.9.

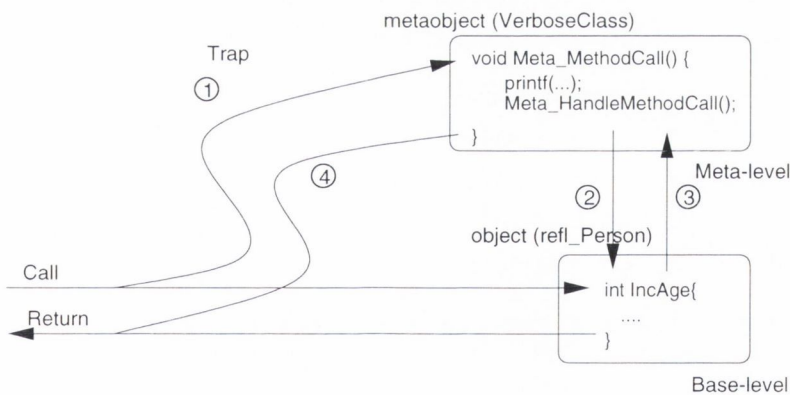


Figure 2.9: Flow of control of a reflective member function. The invocation of a reflective member function is trapped and diverted to the object's metaobject. The metaobject can perform additional tasks before handling the actual invocation.

The user can modify the behaviour of method invocation by redefining `Meta_MethodCall()` of class `MetaObj`.

The example program shown in figure 2.10 illustrates how the OpenC++ MOP can be used to achieve the task of retrieving a list of method descriptions for a given object.

---

```

void listMethod( MetaMsgReceptor* targetObject){
    int i=0;
    MetaObj* meta;
    meta = targetObject->Base_GetMetaObj();
    printf("Class is %s", meta->Meta_GetClassName());
    char**methods = targetObject->Base_GetMethodNameList();
    while (methods[i]){
        printf("method %d %s", i, methods[i]);
        i++;
    }
}

```

---

Figure 2.10: Introspection in OpenC++ v1: the function `listMethods` lists all methods of a given object.

#### 2.6.4 Known Applications

[SW95] describes the implementation of atomic data types using the OpenC++ MOP. The example is motivated by the fact that with traditional approaches application code is to a large extent intertwined with non-functional code which deals with synchronisation and recovery.

Atomic data types are associated with `AtomicMetaObj`, a metaobject class that provides operations for implementing local atomicity. A customised method dispatcher (a redefinition of method `Meta_MethodCall`) intercepts all method invocations and performs additional operations depending on the category of the target method. Subclasses of `AtomicMetaObj` can implement different concurrent control schemes, so for example optimistic or pessimistic concurrency control.

Base-level classes that are to support atomicity can to a large extent be written as usual, i.e. the functional code can be written without atomicity in mind. However, it requires the base-level programmer to categorise member functions as either being *read* or *write* operations<sup>3</sup>. This semantic distinction is necessary for the atomic MOP to acquire appropriate locks before each operation.

The inability of the OpenC++ MOP to dynamically associate objects with metaobjects has

---

<sup>3</sup>OpenC++ provides a mechanism that allows the annotation of member functions and data members with user defined categories.

been mentioned by the authors as a major limitation. On the positive side, OpenC++ allows a clean separation of concerns: different schemes can be introduced separately.

### 2.6.5 Performance

An evaluation of the OpenC++MOP presented in [CM93] reveals that a method call carried out via the meta-level is about 6–8 times slower than an C++ virtual function call. The overhead increases in proportion with the number of arguments since all arguments are copied separately via an `ArgPac` object. Compared to a non-virtual function call OpenC++ is about 10 times slower. Access to member variables is about 35 times slower.

### 2.6.6 Support for Meta-Types

Reflection in OpenC++ v1 is not transparent as the programmer has to explicitly create reflective objects. Although it is possible to create both reflective and non-reflective objects of the same type, they can not be treated interchangeably.

As mentioned above, it is also not possible to dynamically rebind an object to another metaobject at run time since this connection is established during compilation.

The combination of multiple, independently developed MOPs can indirectly be achieved by arranging metaobject classes in a reflective tower where each intermediate level propagates the operation to the immediate upper meta-level. This approach has been exercised in the development of a fault-tolerant MOP [FP98]. In this example, a three meta-level model was defined with separate levels implementing fault-tolerance, secure communication and group-based distribution.

## 2.7 OpenC++ v2

The OpenC++ v2 MOP provides control over the compilation process of application code. It was also written by Chigeru Chiba at the Masuda Laboratory, University of Tokyo, Japan.

### 2.7.1 Type of Reflection

Version 2 of the OpenC++ compiler is a compile-time architecture that is similar to a parse-tree translator: The parse-tree of the (base-level) source-code is generated, traversed and



transformed. The nodes of the parse-tree are metaobjects that can be customised in order to control the translation of, for instance, a method invocation or a class declaration.

### 2.7.2 Reflective Facilities

The OpenC++ MOP provides control over the compilation of class definition, member access, virtual function invocation and object creation [Chi95]. This is achieved by reifying the compiler's parse tree as a collection of objects. The source-to-source translation of the program as well as structural aspects, such as type information, are reified by either of the following meta-level classes:

- **Class** metaobjects: As well as representing class definitions, they control the source-to-source translation of the program.
- **Ptree** metaobjects: They represent the parse tree of a program. The parse tree is implemented as a nested-linked list. Methods of class **Ptree** are used to manipulate the list.
- **TypeInfo** metaobjects: They represent types that appear in the program.
- **Environment** metaobjects: They represent bindings between names and types and are also used to insert declarations in the translated program.

Version 2.3 introduces 2 more meta-level classes, namely

- **Member** metaobjects: they provide a more abstract view of member functions and allow the user to obtain information about a member of a class such as its signature, name or whether it is virtual or not.
- **Walker** metaobjects: can be used to traverse a parse tree and to call a user defined function each time a node object is visited.

Metaobjects of type **Class** play the key role in the MOP as they represent class definitions and control the source-to-source translation. An overview of the functionality provided by class **Class** is given in figure 2.11. Methods such as **TranslateMemberCall()** and **TranslateNew()** provide a way to translate expressions involving the class and thereby allow the user to change the behavioural aspects of method invocation and object creation. Introspection is supplied by methods such as **Name()** and **NthMemberName()** which return the name of a classes and its n-th member function respectively.

```

class Class {
  public:

    // Protocol for Introspection:
    Ptree* Name();
    Ptree* BaseClasses(); // Returns the base-classes field
    Ptree* Members();    // The body of the class declaration
    Ptree* Definition(); // Returns the whole clas definition
    Class* NthBaseClass(Environment *env, int n);
        // Returns the n-th base class
    Ptree* NthMemberName(int n);

    // Protocol for Translation:
    Ptree* TranslateClassName(Environment* env, Ptree* keyword, Ptree* name);
    Ptree* TranslateSelf(Environment* env);
    Ptree* TranslateMemberFunctionBody(...);
    Ptree* TranslateNew(...);
    Ptree* TranslateMemberCall(...);
    ...
};

```

Figure 2.11: The OpenC++ v2 MOP. Metaobjects of type `Class` represent class definitions and control the source-to-source translation.

### 2.7.3 Programming Model

**Implementing a new MOP** is accomplished by deriving from class `Class` and by redefining the appropriate member functions that control the source translation. The new MOP implementation is then compiled by the OpenC++ compiler to produce a modified version of itself. The new compiler is subsequently used to translate base-level programs.

#### Selecting a MOP

A base-level class can select a metaobject class either by a `metaclass` declaration or by registering a new keyword. Selecting a MOP is done on a per class basis, i.e. once a class is associated with a MOP, all instances will be modified by it.

The methodology for implementing a language extension consists of three steps:

1. Decide what the base-level program should look like.
2. Decide how it should be translated



3. Write a meta-level program to perform the translation and write the run-time support code, if required.

The following is a variant of the verbose methods example used to exercise the steps above:

---

```

metaclass Account : PrintMethodCalls; //metaclass declaration
class Account {
    protected:
        Money balance;

    public:
        Account_Nr getAccountNumber();
        void credit();
};

```

---

The `metaclass` declaration above associates the class `Account` with the metaclass `PrintMethodCalls`. The next step is to decide how the base-level program will be translated so that a message will be printed everytime an `Account` method is called. One way to do this is to translate every expression containing a method call on an `Account` object so that it prints out a message before actually calling the method. For example, if `myAccount` is an `Account` object, then the statements:

```

myAccount.getAccountNumber()
myAccount.credit()

```

can be translated into:

```

(puts("getAccountNumber()"),myAccount.getAccountNumber())
(puts("credit()"),myAccount.credit())

```

This example does not need any run-time support code, so the next step only involves writing the metaclass `PrintMethodCalls`. `PrintMethodCalls` is sub-classed from the default metaclass `Class` and overrides the default `TranslateMemberCall` method to translate expressions involving the `Account` class.

---

```

class PrintMethodCalls : public Class {
    public:
    PrintMethodCalls(Ptree* d, Ptree* m) : Class(d,m) {}
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*, Ptree*, Ptree*);
};

Ptree* PrintMethodCalls :: TranslateMemberCall(Environment* env,
        Ptree* object, Ptree* op, Ptree* member, Ptree* arglist) {
    return Ptree::Make("(puts("(%p())",%p)", member,
        Class::TranslateMemberCall(env, object, op, member, arglist));
}

```

---

The `TranslateMethodCall` takes an expression such as `myAccount.credit()` and returns the translated version. As mentioned above, it is actually parse trees that are perform the translation. Both the given expression and the translated one are represented as parse trees, `Ptree` is the data-type which represents a parse tree and `Ptree::Make()` is a method to construct a new parse tree.

The authors admit that writing a meta-level program using a compile time MOP is a difficult task, as the user has to deal with the internal workings of the parse tree, transform it and produce different code. However, version 2.3 of the OpenC++ compiler [Chi] is said to overcome some of the problems mentioned above by providing better support for introspection.

Figure 2.12 outlines the steps involved in writing an application in OpenC++. As mentioned above, the meta-level program written by the user is first translated by the OpenC++ compiler and linked with itself to generate an extended version. Compiling a base-level program in OpenC++ v2 consists of three stages: preprocessing, source-to-source translation from OpenC++ to C++ and the compiler backend.

#### 2.7.4 Performance

Implementing new object behaviour such as persistence or distribution is not involved with run-time overhead due to the MOP itself, as the metaobjects only insert additional code that is needed to support the desired feature.

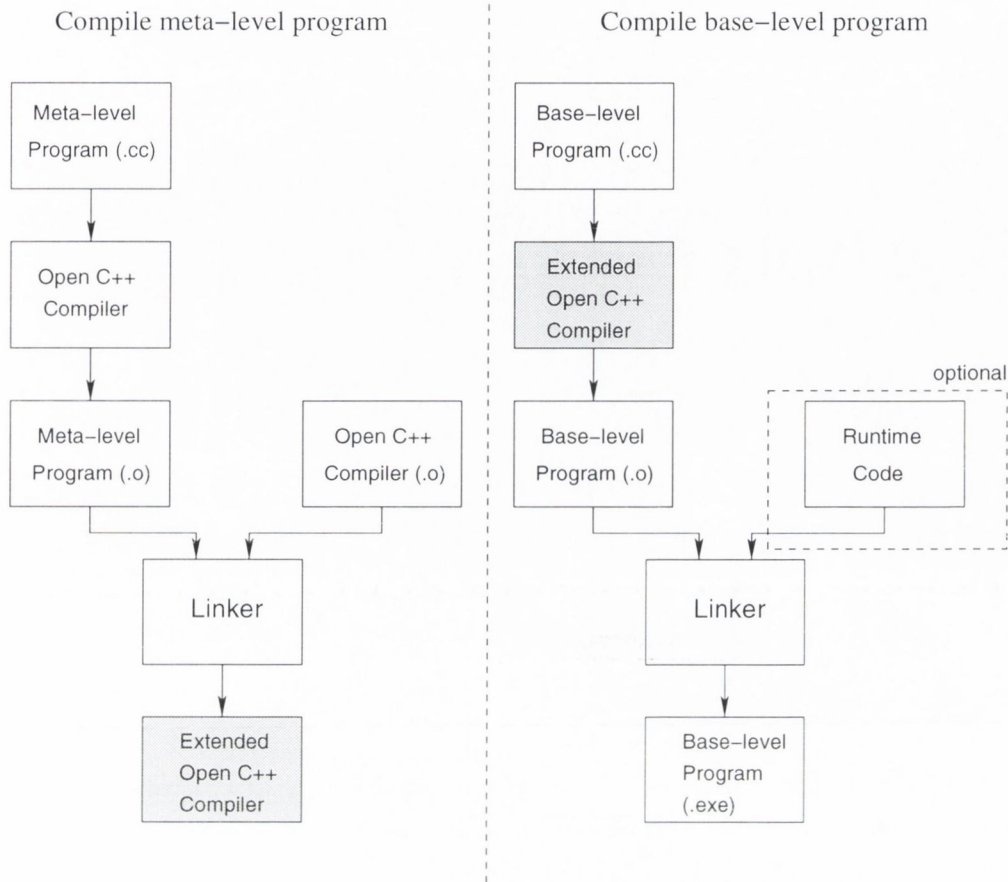


Figure 2.12: Writing an application in Open C++ v2.

### 2.7.5 Known Applications

As with version 1, useful applications include language extensions for C++ to support features such as persistence, distribution and concurrency. For example [KYK<sup>+</sup>99] describes a MOP-implementation using OpenC++ v2 supporting persistence in C++ and explains in great detail the linguistic issues involved in object checkpointing. As reflection only takes place at compile-time, it is difficult to achieve the flexibility offered by run-time architectures

As described in [KFRGC98], the OpenC++ compiler was used to implement a MOP supporting fault tolerance. The requirements for a fault tolerant object system using replication strategies include the control over

- object creation/deletion: objects have to be created in multiple copies over different nodes (replicas) and to subscribe to a communication group. When an object is

deleted, all replicas have to be removed from that group.

- object invocation: all invocations on a fault tolerant object have to be broadcasted to the replicas and the order of the messages has to be maintained in order to ensure replica consistency. Replicas must also be able to handle multiple copies of the same message and to synchronise themselves after the execution of the method.
- object state access: the evolution of an object's state has to be propagated to the system to ensure consistency among the copies. Under certain circumstances, objects have to be stored on stable storage periodically in order to retrieve their state after a system failure.

In order to trap all method invocations to an object, the OpenC++ compiler was used to rename all methods, including constructors and destructors. An invocation is trapped by an additional method with the original name that forwards the call to the metalevel. For example, a method `method1` is renamed into `real_method1` and a wrapper function, now called `method1` is added to the class definition. Figure 2.13 shows the modified code for a class declaration.

| Original Class   | Translated Class  |
|--|---|
| <pre>class Base {   Base();   ~Base();   void method1();   void method2(int); };</pre> | <pre>class Base' {   Base();           // Trap   ~Base();          // Trap   void method1();  // Trap   void method2(int); // Trap   // original methods:   void real_Startup();   void real_Cleanup();   void real_method1();   void real_method2(int); };</pre> |

Figure 2.13: Code modifications performed by the OpenC++ MOP in order to trap invocations on an object.

### 2.7.6 Support for Meta-Types

As a compile time architecture, OpenC++ v2 naturally lacks the flexibility of run time architectures. MOPs control the translation of whole class definitions, thus it is neither



possible to adjust the behaviour of individual instances of a class, nor to dynamically modify the behaviour of objects at run time.

## 2.8 CodA

The CodA meta-level architecture is based on an operational decomposition of meta-level behaviour into objects and the provision of a framework for managing the resultant components [McA95b].

### 2.8.1 Type of Reflection

CodA is a reflective extension of Smalltalk and as such is a run-time meta-level architecture. It provides behavioural reflection by interceding with a number of events triggered by the application objects, such as sending/receiving messages or accessing state. CodA uses a modified virtual machine in order to intercept events.

### 2.8.2 Reflective Facilities

In CodA, the meta-level is decomposed into seven so-called meta-components that reify different aspects of object behaviour such as sending/receiving a message, executing a method or accessing state. Figure 2.14 depicts the events and meta-components involved in the sending of a message M from object A to object B.

Object behaviour is modified by explicitly associating meta-components with an object. The role of the meta-components is as follows:

**Send** The main role of the Send meta-component is to manage the sending of a message to an object. This can involve supervision of the transmission of the message, synchronisation of the sender and the receiver, protocol negotiation and resource management.

**Accept** The Accept meta-component defines how the receiver of a message interacts with the message sender. It therefore also has to deal with synchronisation and protocol negotiation. It also determines if the message is valid and how the message should be handled, that is, whether the message should be queued for processing or whether it should be processed immediately.

**Queue** organises and holds messages which have been accepted but not yet received or processed.



In the Smalltalk implementation of CodA, each meta-component is represented by a class. The default classes provide the functionality associated with its role. For example, the `Send` component provides methods for sending a message to a base-level object, the `Queue` component allows the en- and dequeuing of messages etc ([McA95a]).

### 2.8.3 Programming Model

Defining and Selecting a MOP is achieved by re-defining individual classes that provide the intended behaviour and by replacing them with the default meta-components. New components can either be selected on a per object, per class or on a system-wide basis.

For example, if an object wants to replace its `Send` component, it first has to instantiate its meta level using the

```
Behaviour>>asExtendibleMetaFor:
```

method. This creates and installs the object's default metaobject. The metaobject is an instance of class `CodAMeta`. The class `CodAMeta` provides the necessary functionality to install and replace meta-components. In a second step, the object actually has to replace its `Send` component using

```
anObject meta componentAt: State put: myState for: anObject
```

This replaces the default state-component with the object `myState`.

Figure 2.15 depicts how the CodA meta-level architecture is embedded into Smalltalk objects. The `classField` slot is a hidden data member of every Smalltalk object which points to the object's class. The Smalltalk virtual machine uses this information for method dispatch: for every incoming message an appropriate method is searched in the object's class description. If the class object is replaced by an object which does not understand any messages (as a subclass of `nil` for example), every incoming message is trapped. Adding message handlers to the interceptor allows the individual selection of methods that are reified and those that are not (by default no methods are reified). Trapping a message causes the receiving object to re-invoke a reified send operation from the original sender.

### 2.8.4 Known Applications

CodA has been used to implement a variety of different object models, including concurrent, distributed and ported objects [McA95c]. The object models then have been



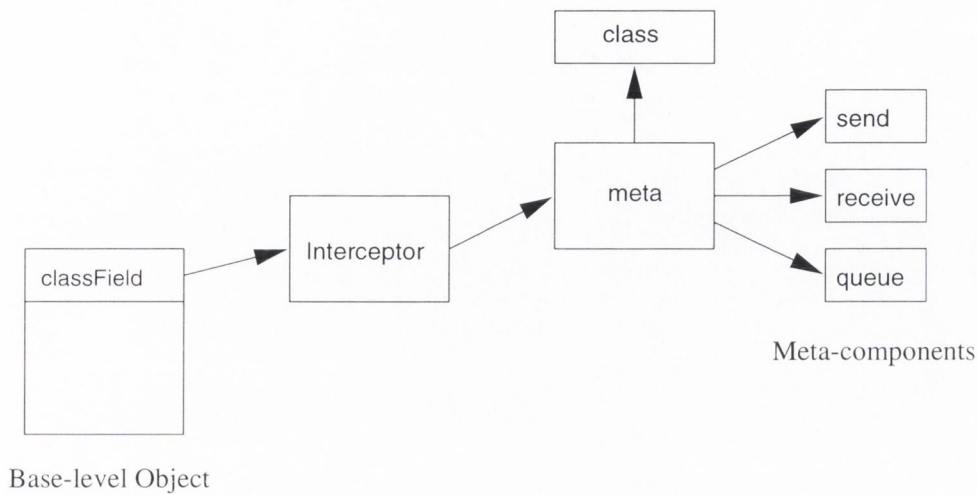


Figure 2.15: Object layout in CodA: the hidden `classField` slot that is part of every base-level object points to the interceptor (a subclass of `nil`). The interceptor in turn points to the object's set of meta-components.

applied to existing applications such as the 2D N-Body problem, an expert system and a monitoring and analysis system called *Vibes* [McA95b].

### 2.8.5 Performance

In CodA, a fully reified send/receive cycle is said to increase the execution time “about an order of magnitude” [McA95b]. McAffer claims that in real applications, where the performance depends on the amount and type of reified components, applications tend to be 3–5 times slower than in the absence of reflection. As an example, the implementation of the 2D N-Body problem where concurrency and distribution was added was four times slower.

### 2.8.6 Support for Meta-Types

Operations on reflective objects in CodA are by and large transparent to the user since the underlying virtual machine intercepts with the Smalltalk message passing mechanism. However, CodA does not provide an automated mechanism for instantiating and modifying meta-components. Instead, programmers have to explicitly instantiate an object's meta-level and insert those components that implement a particular behaviour. It is however possible to insert and replace individual meta-components at run time, thereby allowing objects to evolve in changing environments. Composition of multiple, independent meta-



components is not directly addressed in the framework.

## 2.9 ABCL/R

ABCL/R [WY88] is a reflective extension of the object-based concurrent programming language ABCL/1 [YBS86]. It was developed by Takuo Watanabe and Akinori Yonezawa at the Tokyo Institute of Technology and is based on Lisp.

### 2.9.1 Type of Reflection

ABCL/R is another example of a run-time architecture: the behaviour and state of base-level objects is controlled by exactly one metaobject.

### 2.9.2 Reflective Facilities

In ABCL/R, an object is an autonomous, individual information processing agent, similar to the actor model. It consists of a message queue, an evaluator and a set of state variables and scripts, equivalent to member variables and methods in the C++ terminology. Figure 2.16 depicts the object model in ABCL/R. Objects communicate via message passing. All message transmission is asynchronous. Sending and processing a message is associated with the following set of events:

**Arrival:** The message arrives at the receiver object. The receiver starts processing the message.

**Receiving:** The receiver object enqueues the arrived message in its message queue.

**Acceptance:** The receiver tries to find an appropriate script for the message by pattern-matching. If the receiver accepts the message, it starts executing the script for that message.

**Execution:** The script gets executed.

**End of Processing a Message:** The evaluation finishes and the next message is processed.

In the reflective extension of ABCL/1, an object is fully represented by a metaobject. The metaobject contains the representation of the message queue, the state memory, the set of scripts and the evaluator. Besides this structural aspect, the computational aspect of the

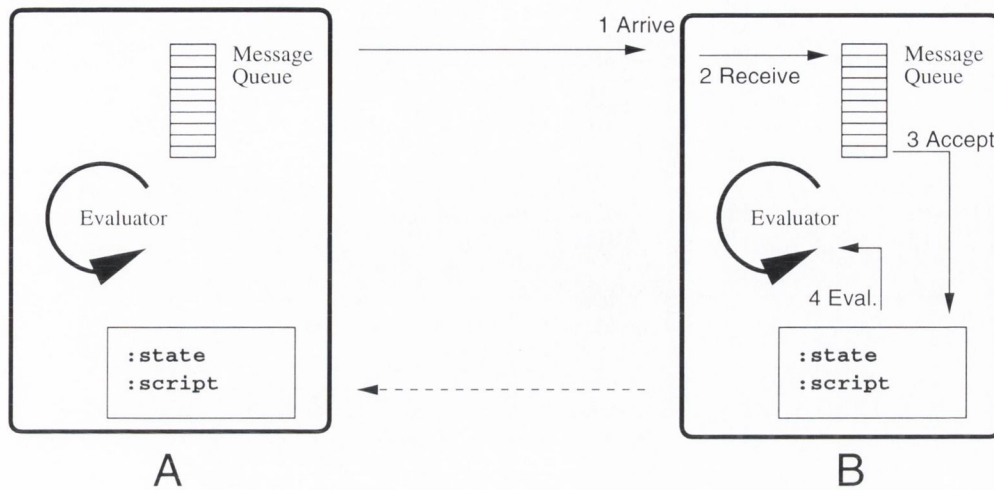


Figure 2.16: Object model in ABCL/R: an object consists of a message queue, an evaluator and a set of state variables and scripts.

object – arrival, receiving and acceptance of a message and the execution of scripts, are reified. The relationship between the base and the meta-level is depicted in figure 2.18.

The structure of object *A* is represented as the data in the state memory of the metaobject: `scriptset` defines the set of scripts, `state` the current contents of the state variables, `evaluator` and `queue` are objects defining the evaluator and the message queue resp. and `mode` represents an objects mode (either dormant or active). The behavioural aspects of an object – arrival, receiving and acceptance of messages as well as the execution of scripts – are described by the script-part (methods) of the metaobject.

### 2.9.3 Programming Model

The example program in figure 2.17 shows a simple object definition in ABCL/R. In this case, we instantiate the object `anObject` that contains two state variables ( $x$ ,  $y$ ) and scripts to set and retrieve the value of the state variables.

Objects are activated by message passing. For example, to set the value of the field variable  $x$  one can send the message `setx` to the object with an additional parameter as in `[anObject <== [:setx 23]]`.

Retrieving the metaobject of `anObject` is accomplished by evaluating the special form `[meta anObject]`. The metaobject contains the following scripts that allow the dynamic modification of its associated base-level object:

---

```
[object anObject
  (state [x := 0]
    [y := 0])
  (script
    (=> :getx !x)
    (=> :gety !y)
    (=> [:setx new_x]
      [x := new_x])
    (=> [:sety new_y]
      [y := new_y])
  )])
```

---

Figure 2.17: Example object definition in ABCL/R. Here we define an object with two state variables *x* and *y* and scripts to set and retrieve the value of the state variables.

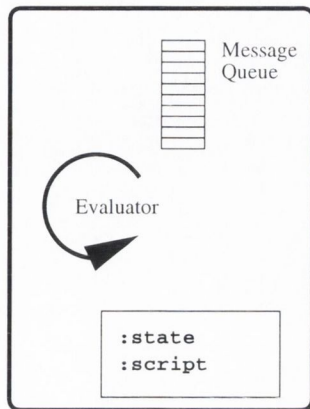
- `[:add-script s]` : Adds a new script which is defined in *s* to the object.
- `[:script m]` : Retrieves a script whose message matches *m*.
- `[:delete-script m]` : Deletes a script whose message matches *m*.
- `[:state]` : Returns a description of the object's state.

The interface described above can be used directly to look up scripts and to modify an object's state. For example, to add a new state variable to an object one can retrieve and alter the object's state as in:

```
[ st := [[meta anObject] <== :state]]
[ st <== [add-binding 'z 100]]
```

To modify the default behaviour of objects, one can replace the scripts that are executed during the various stages of message processing. For example, to monitor the beginning of a script execution, one can replace the `:begin-script` in the object's meta-level. To do so, an additional level of indirection has to be overcome: since every object is fully represented by its metaobject, modifying an object's metaobject can only be done via its meta-meta-object. ABCL/R features a fully reflective tower and (theoretically) allows the access to an infinite tower of meta-levels. This is achieved by creating metaobjects lazily, i.e. when they are first accessed. The following code extract shows how the `:begin-script` can be replaced:

## Base-Level-Object



## Meta-Level-Object

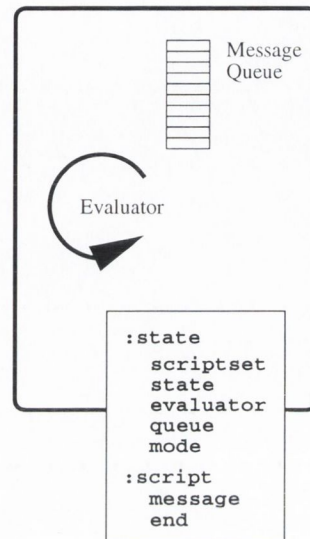


Figure 2.18: Relationship between a base and a metalevel object in ABCL/R: the baselevel object is fully represented by its metalevel object.

```
[mml := [meta [meta anObject]]] ; retrieve anObject's meta-meta-level
[mml <== [:add-script '(=> :begin ...)]] ; replace begin-script
```

#### 2.9.4 Performance

As reported in [MMAY95], the overhead introduced by meta-level computations when directly using an unoptimised interpreter is high, by a factor of 100 and more. In order to cope with this overhead, [MMAY95] describes the development of a compiler framework that employs partial evaluation as its primary optimisation technique. Optimised and compiled applications are reported to only exhibit an overhead of 10–30% compared to hand-crafted source-code optimisations in a non-reflective language. However, this restricts objects to select a meta-interpreter only at creation time, dynamic replacement or customisation of the meta-interpreter is no longer possible.



### 2.9.5 Known Applications

[WY88] and [MMWY92] outlines an implementation of the Time Warp synchronisation mechanism. The Time Warp mechanism is a synchronisation protocol that is targeted at distributed event simulation and distributed database concurrency control [Jef85]. In this scheme, processes communicate via messages, each message containing a timestamp. If conflicts are detected, i.e. messages arrive out of order according to the timestamp they contain, a rollback is performed restoring the state of the process to the time before the conflict appeared. Rollback is performed by sending *anti-messages*, each anti-message reverts the side effect of exactly one original message. The implementation of anti-messages and the rollback mechanism could in this case be completely separated from the application code. Baselevel objects select a customised meta-level interpreter that handles the processing of messages and captures the state of objects in order to allow to revert to earlier stages.

### 2.9.6 Support for Meta-Types

As an interpreted language, interceding with the execution of language operations is transparently carried out by the interpreter and meta-levels for objects are only created on demand. There is no direct support for composing the behaviour of multiple metaobjects in ABCR/R. However, since ABCL/R provides the (lazy) instantiation of a theoretically infinite reflective tower, composition can indirectly be achieved by building multiple layers of meta-levels, with each layer interceding with the level below, similar to the way as has been described for OpenC++ v1.

To our knowledge, there is no means for objects to deselect their meta-interpreter. In other words, once the meta-interpreter of an object has been instantiated, all following operations will be diverted and objects can not revert to a non-reflective configuration.

## 2.10 Discussion

In this chapter we examined and reviewed a number of reflective extensions to object-oriented programming languages. Compile-time reflection, while not introducing run-time penalties, offers the least flexibility as reflection is restricted to the compilation phase. The highest flexibility is provided by interpreted languages such as ABCL/R and CodA. This is possible because the underlying interpreter or virtual machine is able to intercede with the execution of the running program at any one time. However, interpreted languages do

per se not perform as efficiently as compiled languages and are hardly suitable for building performant, low-level applications such as operating systems and embedded systems.

Run-time reflection for compiled languages falls in between these two extremes. A major restriction of existing run-time architectures is their lack of flexibility compared to interpreted languages. OpenC++ v1 for example does not provide a mechanism for objects to dynamically re-select their meta-level representation. Moreover, type orthogonality is not achieved since reflective objects have to be created explicitly and can not be used in place of their non-reflective counterparts. We therefore conclude that none of the existing architectures for compiled languages can be used to implement meta-types.

In the Iguana reflective programming model as described in the following chapter, we aimed at overcoming the restrictions imposed by existing platforms while keeping the interpretative overhead low. We are specifically interested in providing reflection for compiled languages with a flexibility similar to that found in interpreted languages.

## The Iguana Reflective Programming Model

The iguana that jumped from the high iroko tree  
said he would praise himself if no one else did.

*Nigerian Proverb*

From our review of reflective programming languages in the previous chapter we have identified a number of shortcomings in current architectures. Existing architectures for compiled programming languages, as exemplified by OpenC++ v1, are too static and do not offer enough transparency since reflective and non-reflective objects have to be treated separately. Other, more flexible architectures such as CodA, require the programmer to explicitly create and access meta-level components, which again leads to a tangling of functional and non-functional code.

Iguana in its previous version [Gow97] suffered from similar problems in that it did not separate enough between the roles of base and meta-level programmer. As a consequence, applications written in Iguana contained a large amount of direct meta-level invocations, consisting of, for example, code for inserting or replacing metaobjects. Again, this defeats the transparent use of reflection and its claim for achieving a clean separation of concerns.

We therefore undertook a substantial re-design and re-implementation of the Iguana model. As explained before, our aim is to shield both the application and meta-level programmer from the actual implementation details of the reflective programming features. Although this thesis is primarily concerned with applying reflection to a compiled language, the concepts presented in this chapter are meant to capture various features commonly found in existing OOPs in order to make the model applicable to other programming languages as well, including Java [RC00].



## 3.1 General Overview

In Iguana, base-level objects are associated with a set of metaobjects, each representing or implementing a specific language construct. Programmers can substitute the default semantics of each of the language constructs individually by providing a customised implementation.

A design issue that distinguishes Iguana from comparable platforms, is that it is dynamic in nature: Iguana offers the dynamic and selective reification of language constructs at run-time, meaning that customised object models can evolve as the system runs. This is particularly important for building applications that are faced with changing non-functional requirements. Building those applications is inherently difficult as the designer cannot always foresee every possible usage scenario and/or run-time environment.

## 3.2 Reification Categories

In Iguana, *reification categories* represent language constructs whose implementation or interpretation can be modified by the programmer. Among these language constructs are structural categories such as information about classes, methods and attributes and behavioural categories that implement for example object creation, deletion or method invocation.

In contrast to earlier versions of Iguana, the number of reification categories has been reduced significantly, from originally 29 to now 12. For example, Iguana v1 provided 7 different reification categories concerning methods and method invocation, namely **Method**, **MethodName**, **MethodAddress**, **Invocation**, **MethodAccess**, **MethodBefore** and **MethodAfter**. The main motivation behind this approach was concerned with performance: the use of reflection should ideally only introduce overhead where the reflective features are explicitly used.

In practical terms, we found that the overload of reification categories was more confusing than beneficial, leading to an overlap of the semantics of the various categories. Moreover, the ideal of selectively reifying only those language constructs that are of actual interest for the meta-level programmer cannot always be met. For example, in order to reify method invocation, it is surely necessary that structural information about methods, such as their address, type and signature, is also present. This has led to a number of dependencies between different reification categories.

In the revised model, structural information about methods, including the type, name,



address and signature of methods, is combined into only one category. We believe that this reduction has led to a cleaner design and represents a more intuitive decomposition of object models. Although dependencies between different categories still persist (see section 3.2.4), they have been reduced significantly. Moreover, the reduction has not led to a loss of expressive power. For example, in order to provide the functionality of before/after methods, it is sufficient to only reify method invocation since the implementation of that reification category can contain any code that is executed before and after the actual method has been invoked.

A discussion of the available reification categories is presented below.

### 3.2.1 Structural Reification Categories

These represent structural aspects of the underlying object model, such as information about classes, member functions and data fields. Structural reification categories allow introspection and build the base for most of the behavioural reification categories described later in this section.

In the revised model, Iguana supports five structural reification categories, namely:

**Class:** contains information about a class, such as its name, its superclasses and a description of its methods and attributes.

**Method:** contains information about a method, such as its name, signature, type and address.

**Attribute:** describes a class's attribute (or data member): its name, type and size.

**Constructor** provides information about a constructor including its name, signature, and address.

**Array** provides information about an array, for example, the number of elements in the array.

### 3.2.2 Behavioural Reification Categories

Behavioural reification categories define the semantics of a specific language construct, for example how objects are created/deleted, how methods are invoked and how state variables are accessed. Currently, Iguana supports the following set of behavioural reification categories:

**Creation:** implements the process of object creation which usually involves memory allocation for the object's data members, initialising the object's virtual function table and calling its constructor.

**Deletion:** deleting an object usually involves calling its destructor and freeing up any memory that the object holds.

**Invocation:** implements the invocation of a method at the receiver side.

**StateRead:** implements reading from an object's data member.

**StateWrite:** implements writing to an object's data member.

**Send:** intercedes with a method invocation on the caller side, i.e. before it is dispatched by the receiving object.

**Dispatch:** reifies the dispatching of an invocation to the correct method.

### 3.2.3 Extent of Reflection

A relevant design issue that has been taken into account is concerned with the extent to which the reification categories should be applied. Should the reification of method invocation and state access be applied to individual methods/attributes of a class or should reification apply to the whole class? The former approach, as provided by OpenC++ v1, allows a more fine grained control over the reification of events and might result in more efficient implementations. However, this requires application programmers to individually select those attributes and methods that are subject to reflection, which leads to a greater dependency between base and meta-level code and defeats the transparent use of reflection. We therefore advocate the latter approach where all methods and attributes of a class are subject to reflection.

### 3.2.4 Dependencies between Reification Categories

In general, the behavioural reification categories can be applied independently of each other. However, most of the reification categories (i.e., both structural and behavioural) depend on one or more of the structural reification categories. For example, the behavioural reification category supporting method invocation requires that the structural reification category providing information about methods is also selected in any MOP in which it is included. Figure 3.1 summarises the Iguana reification categories and their dependencies.

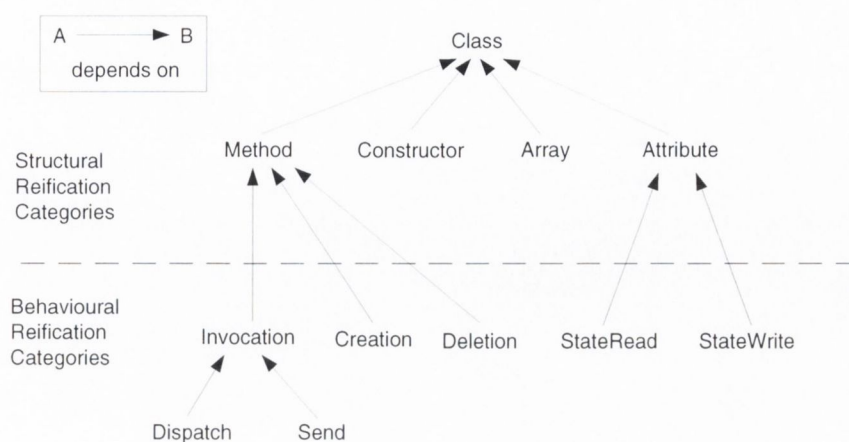


Figure 3.1: Dependencies between reification categories in Iguana.

### 3.2.5 Protocols and Protocol Selection

The Iguana model provides the concept of a *protocol* both as a means of defining a new MOP and of specifying the implementation of a MOP. In Iguana, the definition of a MOP specifies the set of reification categories to be applied to objects that select the MOP as well as the set of metaobject classes to be used to implement those reification categories. For example, the protocol definition

---

```

protocol DefaultProtocol {
  reify Class      : MClass;
  reify Method    : MMethod;
  reify Invocation: MInvocation;
};
  
```

---

specifies that structural information about a class and its methods is to be reified by instances of `MClass` and `MMethod` respectively. In addition, invocation is reified by instances of class `MInvocation`.

Every protocol defines a new *meta-type*. Objects that select a particular protocol can be said to *conform to* or *implement* the corresponding meta-type. Every object has an associated meta-type, its **current meta-type**, which can be changed dynamically. The process by which the meta-type of an object is specified is called *protocol selection*. Three forms of protocol selection are supported by Iguana:

- class protocol selection;
- default protocol selection; and
- instance protocol selection.

Each of these forms of protocol selection is discussed in turn below. In all cases, the meta-type is specified using the protocol selection operator `==>`.

### Class Protocol Selection

Class protocol selection allows the meta-type of all new instances of some class to be specified and also defines their **static meta-type**. For example, to specify that all new instances of class `Car`, a subclass of `Vehicle`, should have meta-type `Persistent`, the following class declaration is used:

```
class Car : public Vehicle ==> Persistent {
    // declaration of Car
};
```

Class protocol selections are inherited. Thus, subclasses of `Car` inherit its protocol selection. A subclass can also override the protocol selection of its superclass. However, in this case, the specified meta-type must be a sub-type of the meta-type specified for the superclass. For example,

```
class Taxi : public Car ==> Atomic {
    // declaration of Taxi
};
```

is only possible if `Atomic` is a sub-type of `Persistent`.

### Default Protocol Selection

Default protocol selection allows the meta-type of all new instances of a set of classes declared in a single source file to be specified. Logically, there is a default meta-type associated with each source file. The default meta-type may be changed with a default protocol selection declaration.

For example, to declare that all new instances of the classes `Car` and `Boat` should have meta-type `Persistent`, while new instances of `Plane` should have meta-type `Atomic`, where `Car`, `Boat`, and `Plane` are declared in the same file, we can write:



```
defaultProtocol ==> Persistent;
```

```
class Car : public Vehicle {  
    // declaration of Car  
};
```

```
class Boat : public Vehicle {  
    // declaration of Boat  
};
```

```
defaultProtocol ==> Atomic;
```

```
class Plane : public Vehicle {  
    // declaration of Plane  
};
```

---

### Instance Protocol Selection

Instance protocol selection allows the meta-type of a single object to be selected dynamically. In other words, it allows the **dynamic meta-type** of an object to be specified. For example, to specify that the object stored in variable `myHouse` is to have meta-type `Persistent` from now on, the programmer can write

```
House myHouse ==> Persistent;
```

The dynamic meta-type of an object must be a sub-type of its static meta-type. For example,

```
class Car : public Vehicle ==> Persistent {  
    // declaration of Car  
};  
...  
Car myMerc ==> Atomic;
```

is only possible if `Atomic` is a sub-type of `Persistent`.

**Protocol Inheritance** The set of active reification categories for a given object can increase and decrease over time by means of *protocol inheritance* and dynamic protocol selection. For instance, the following protocol definition

```
protocol AccessProtocol : DefaultProtocol {
  reify StateRead : DefaultRead;
  reify StateWrite: DefaultWrite;
};
```

is derived from `DefaultProtocol` and introduces two additional reification categories, namely `StateRead` and `StateWrite`. Where a protocol is derived from another protocol, the resulting protocol includes the full set of reification categories specified by both the base and derived protocol.

In the case where a reification category is repeated in a derived protocol, the resulting meta-type will include multiple implementations of that reification category. In other words, rather than overriding or replacing the metaobject class specification in the base protocol, derived protocols accumulate new metaobject class definitions which are combined with those of the base protocol. As a result, multiple metaobjects control the behaviour of a specific reification category.

Objects which are associated with `DefaultProtocol` can at run-time switch between the base and its derived protocol and thereby adapt their behaviour dynamically.

### 3.2.6 Shared Behaviour

Iguana defines two sharing modes for reification categories: *local* and *shared*. Declaring a reification category to be local implies that the metaobject implementing that particular category is local to the associated base-level object. A shared metaobject on the other hand is shared by all instances that have selected that MOP. This feature is particularly useful to achieve some sort of group behaviour where objects of different classes share a common feature or common information. Typically, structural reification categories would by default be considered as common to all instances of the same class and hence reified in a shared mode. In order to modify the behaviour of individual objects one would reify the corresponding behavioural reification category in a local mode. The following code example shows a possible protocol definition, this time with both local and shared reification categories.

---

```

protocol DefaultProtocol {
shared:
  reify Class      : MClass;
  reify Method    : MMethod;
local:
  reify Invocation : DefaultInvocation;
};

```

---

In this example, structural information about classes and methods is reified in a shared mode whereas invocation is reified in a local mode. The corresponding meta-level configuration is depicted in figure 3.2.

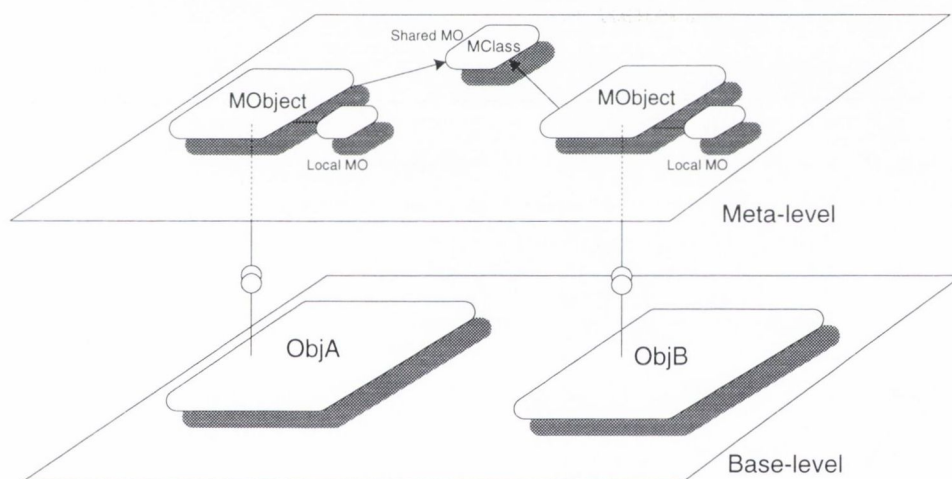


Figure 3.2: Metalevel configuration with local and shared metaobjects.

### 3.3 Metaobject Composition

Composition of metaobjects provides a means for partitioning more complex behaviours into separate “layers” where each layer is responsible for carrying out a specific task and then transfers the flow of control to the next layer in the hierarchy. Moreover, composition can be used to provide alternative implementations of a specific tasks, which can then be combined arbitrarily. For example, a set of metaobjects implementing remote method invocations could provide support for encryption. Conceptually, one layer of metaobjects

can perform the encryption of data before the next layer sends the data across the network. The network layer does not need to be aware of whether or how the data has been encrypted. Moreover, a variety of encryption algorithms could be provided which can then be selected into the application.

Several reflective languages support composition of metaobjects in order to combine two or more behavioural aspects. The Guaraná meta-level architecture [OB98] is a good example. Guaraná associates a single (or primary) metaobject with each base-level object. The primary metaobject acts as a *composer* that can delegate requests to its attached metaobjects (which can themselves be composers) and then combine their results to form its own result.

The need for composition has been recognised for a number of years ([Ber96], [NM94], [FDM94]), but also introduces a number of problems that need to be addressed. As has been noted in [MaPC95], combining different behaviours, or so-called *customisations*, is a hard problem because of the semantic interference that may occur between the combined behaviours.

For example, say we have two independently developed protocols supporting persistent and remotely invocable objects respectively (and which both make use of metaobject classes implementing method execution). In principle, we would like to be able to define a new protocol for objects that are both persistent and remotely invocable by means of multiple protocol inheritance. However, the logic of the respective method invocation metaobjects is most likely to work by attempting to locate the object, either in the persistent store or distributed system, and then invoking it or throwing an exception if the object can't be located. When the protocols are combined, the desired behaviour would be to search for the object as appropriate (depending on which metaobject is activated first), invoke it if found or otherwise give the other metaobject a chance to locate and invoke the object.

Iguana supports *automatic* meta-level composition by means of protocol inheritance: a protocol can be derived from one or more super protocols. The resultant protocol implementation consists of the union of the selected reification categories and their implementing metaobject classes. We intend metaobject composition to be a tool for the experienced meta-level programmer who wants to modularise complex object behaviours, but we do not advocate its use by the application programmer due to the difficulties outlined above.

### 3.3.1 Default Composition Semantics

With these considerations in mind, we provide a default semantic for composing two or more protocols as well as the ability to specify application-specific compositions. In



our implementation, multiple metaobjects implementing a single reification category are chained together in a linked list. Metaobjects are usually written to transfer control to their successors in the list. Thus, each metaobject has the chance to perform its own processing and then delegate the task to the next object in the list or to terminate the request. The final metaobject in the list is usually expected to be the default metaobject for the reification category that knows how to implement the basic operation.

The following example illustrates a possible use of multiple protocol inheritance. The code below defines three protocols, a default protocol that provides default behaviour, a verbose protocol that will print out a message every time a method on an object is executed and a persistent protocol whose task it is to retrieve the target object from the persistent store, if necessary, before invoking it.

---

```
protocol DefaultProtocol {
  reify Class      : MClass;
  reify Method     : MMethod;
  reify Invocation : DefaultInvocation;
};

protocol VerboseProtocol : DefaultProtocol {
  reify Invocation : VerboseInvocation;
};

protocol PersistentProtocol : DefaultProtocol {
  reify Invocation : PersistentInvocation;
};
```

---

As `VerboseProtocol` is derived from `DefaultProtocol`, an object that selects `VerboseProtocol` will have two metaobjects associated with it, one of class `VerboseInvocation` and one of class `DefaultInvocation`.

Implementing execution of a method on a persistent object on the other hand entails

- looking up the target object in the persistent store,
- reading its state into memory, and eventually
- invoking the method using the default execution mechanism.

If we want to combine the behaviour of the two protocols above in order to realise a protocol for verbose, persistent objects, this can very easily be achieved with

```
protocol VerbPersProtocol : VerboseProtocol, PersistentProtocol {};
```

An object selecting this protocol will have three metaobjects attached to it implementing verbose, persistent and default method execution. No further specifications are necessary, as the metaobjects are organised so that metaobjects from a super-protocol are put towards the end of the chain. Thus, the more specific behaviour (defined in the derived protocol) is executed in preference to the more general behaviour (defined in the base protocol). To prevent the same code from being executed twice, only one metaobject of a specific metaobject class is added to the chain, similar to the virtual inheritance mechanism in C++. Figure 3.3 shows the resulting metaobject configuration.

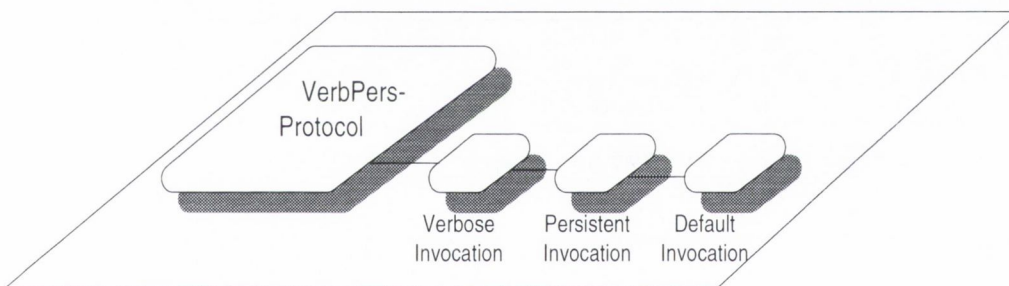


Figure 3.3: Composition of metaobjects in Iguana. By default, metaobjects are chained in a linked list. Each metaobject is responsible for delegating the call to the successor.

### 3.3.2 Modifying Composition

Under some circumstances the default behaviour as described above is not what is desired. Imagine that we want to achieve "exactly one" semantics where exactly one of a set of available metaobjects is invoked in the knowledge that it will perform the desired task. In this scenario the idea is to compose the two behaviours using an intermediate metaobject to delegate the call as appropriate. The necessary steps are outlined in the example below.

```
protocol DefaultProtocol {  
    reify Class      : MClass;  
    reify Method     : MMethod;  
    reify Invocation : DefaultInvocation;  
};
```

```
protocol RemoteProtocol {  
    reify Class      : MClass;  
    reify Method     : MMethod;  
    reify Invocation : RemoteInvocation;  
};
```

---

Here we define two protocols, a default protocol that provides the default behaviour for method invocation and a protocol that allows methods on remote objects to be invoked. Now consider that we want to combine the two protocols in order to implement a smart proxy policy on the client side: when the target object is cached locally there is no need to perform a remote method invocation and the default implementation can be used. In any other case, the remote invocation protocol contacts the remote host and arranges for the method to be invoked. This can be achieved using the following protocol definition:

```
protocol SmartProxyProtocol : DefaultProtocol, RemoteProtocol {  
    reify Invocation      : SmartProxyInvocation;  
};
```

In this case, we only want to delegate the method call to the remote host if the target object is not cached locally, otherwise we can use a normal method invocation. The **SmartProxyInvocation** metaobject now acts as a dispatcher that delegates the call as appropriate. A possible implementation is outlined in the following example:

```

class SmartProxyInvocation : public MInvocation {
    void *invoke(void *obj, MMethod method){
        // get MO for default invocation
        MInvocation *defaultInv = next;
        // get MO for remote invocation
        MInvocation *remoteInv = next->next;
        if (is_in_cache(obj))
            return defaultInv->invoke(obj, method);
        else
            return remoteInv->invoke(obj, method);
    }
};

```

Figure 3.4 shows the corresponding (logical) metaobject configuration.

As this example illustrates, multiple protocol inheritance can be used to resolve conflicts that arise from combining existing behaviours. However, it is the responsibility of the expert reflective programmer to meaningfully combine the existing behaviours. This in general requires an understanding of the desired semantics of the combined protocols and exploits knowledge about the organisation of the metaobject architecture. Although we would like to shield even the reflective programmer as far as possible from such details, this represents a common dilemma found in software reuse: combining existing components or class hierarchies, for example, can often only be achieved by writing some “glue code” in

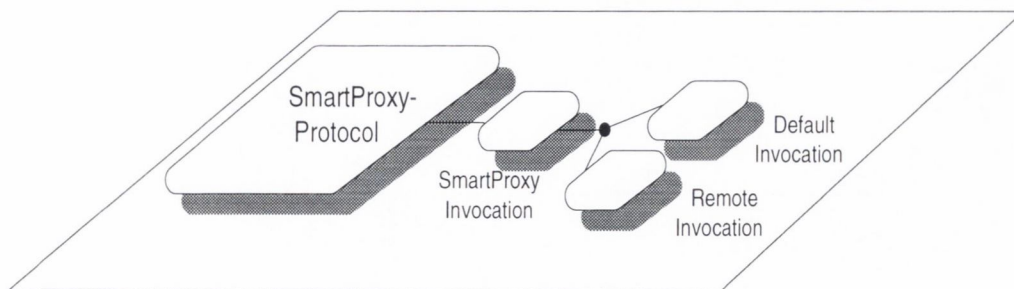


Figure 3.4: Modified composition of metaobjects in Iguana. Multiplexing between different metaobjects can be achieved by defining a subprotocol that combines multiple metaobjects in a meaningful way.



order to achieve the desired behaviour.

### 3.3.3 Discussion

The preceding description can be summarised by the following four rules:

- every object has a single meta-type;
- the meta-type of an object can be changed dynamically;
- the meta-type selected by a class must be a sub-type of that selected by its superclass;
- the dynamic meta-type of an object must be a sub-type of its static meta-type.

Of course, alternatives to each of these rules were considered in the design. In an earlier version of Iguana an object could be associated with multiple independent meta-types. From the application programmer's perspective, this represented a somewhat more complicated model. From the implementation perspective it required the ability to combine the metaobjects implementing the different meta-types in a meaningful way. Rules to combine metaobjects from meta-types that were written independently of each other are technically possible but unlikely to lead to meaningful combined behaviour when the meta-types are unaware of each other. Thus, the current design only allows an object to have a single meta-type. The meta-type in turn can be composed of multiple, but it is the meta-level programmer's responsibility to provide a meaningful composition semantic. Put differently, the experienced meta-level programmer uses protocol inheritance in order to compose meta-types, but from the application programmers point of view objects are at any one time controlled by a single meta-type.

Another design point is concerned with whether or not it should be possible to change the meta-type of an object dynamically. Clearly, static typing has proven to be extremely useful and hence it could be argued that static meta-typing is sufficient. Moreover, always knowing the meta-type of an object statically would certainly allow us to optimise our implementation significantly. However, our view has been that meta-types are primarily intended to be used to address non-functional requirements which are prone to change. Hence, it appears unnecessarily restrictive to prevent dynamic change of meta-types.

### 3.3.4 The Iguana Syntax

Iguana extends the C++ syntax with only a few constructs. The syntax extension is shown in figure 3.5. It mainly consists of a construct for declaring meta-object protocols and the

MOP selection operator ( $\Rightarrow$ ) that associates a MOP with objects.

### 3.4 Summary

This chapter described the Iguana reflective programming model. We introduced the notion of a meta-type that provides a common abstraction for the reflective features in Iguana. From the application programmer point of view, meta-types constitute components that can be selected into an application. From the experienced meta-level programmer point of view, meta-types build a framework for designing and implementing object models. The model addresses the issues of composing and selecting meta-types dynamically, thus both base and meta-level programmer are to a great extent shielded from the reflective features.

```

protocol-specifier:
    protocol-heading protocol-body

protocol-name:
    identifier

protocol-heading:
    protocol protocol-name protocol-base-specopt

protocol-base-spec:
    : protocol-base-list

protocol-base-list:
    protocol-name
    protocol-name , protocol-base-list

protocol-body:
    { protocol-member-listopt }

protocol-member-list:
    reification-list protocol-member-listopt
    mode-specifier : protocol-member-listopt

reification-list:
    reification-declarator
    reification-declarator reification-list

reification-declarator:
    reify reification-category metaobject-class-specifieropt ;

metaobject-class-specifier:
    : class-identifier

mode-specifier:
    local
    shared

assignment-expression:
    conditional-expression assignment-operator assignment-expressionopt
    identifier ==> protocol-name

```

Figure 3.5: The Iguana syntax definition.

## The Iguana/C++ Implementation

It is no disgrace to start all over. It is usually an opportunity.

*George Matthew Adams*

In this chapter we describe a concrete mapping of the Iguana model onto C++. Where appropriate, we highlight the impact of the host language on the implementation.

### 4.1 Applying the Model

There are two principal routes one can take in order to extend an existing programming language. A first approach would be to modify an existing compiler to incorporate the language extension. Alternatively, one can apply an additional pre-processing stage that translates a program in the extended language into the original language. The program is then compiled as normal with the standard compiler. Early C++ implementations for example took the latter approach in that they translated programs written in C++ back to C.

There are a couple of issues related to each of the approaches: an additional pre-processing stage complicates the development cycle and introduces dependencies between modules. Under some circumstances, pre-processing might not even be applicable if parts of the application are already compiled into object code and the source code is no longer available. Moreover, in order to find a mapping from the extended language to the original, one has to obey the typing rules of the native language, resulting in sometimes obfuscated and less efficient code. The advantages are that the pre-processor approach allows rapid prototyping and experimenting with different programming models.

Compilers on the other hand are complex pieces of software and, as such, difficult to modify. This is especially true in the case of C++ due to its origin from C. On the positive



side, having direct access to the compiler would result in a tighter coupling between the language extension and its translation into native machine code.

With these considerations in mind we decided to take the pre-processor approach, mainly in order to prove the feasibility of the model and to create an environment that allows experimentation with alternative designs. A first implementation using a pre-processor can subsequently serve as an example of how the Iguana model can be embedded into existing compiler technologies.

## 4.2 General Overview

Figure 4.1 gives a high-level view of the components that constitute the Iguana/C++ system. The C++ (base-level) code is augmented with meta-level directives and translated by the Iguana pre-processor into standard C++. In addition, the pre-processor generates code that contains run-time support that has to be linked with the final application. The run-time support mainly consists of code implementing dynamic meta-type selection.

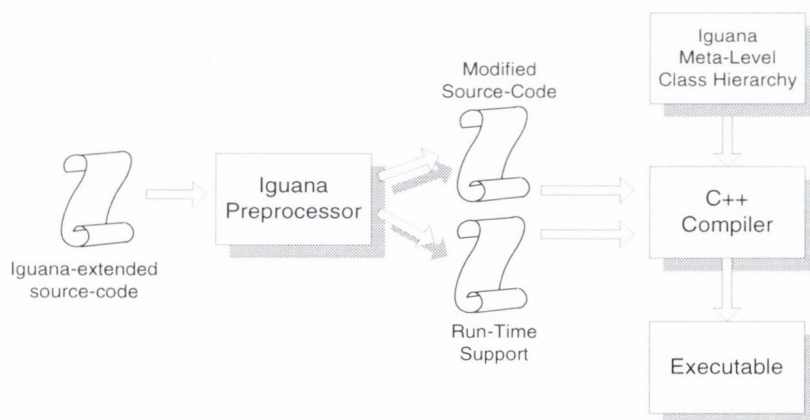


Figure 4.1: Conceptual overview of the Iguana meta-level architecture. The Iguana extended base-level code is translated by a pre-processor into standard C++ and additional run-time support. The final application is then compiled together with the Iguana meta-level class hierarchy and subclasses thereof.

Reified language features are represented by a set of classes, the Iguana meta-level class hierarchy. The classes of the meta-level class hierarchy constitute the self representation of the C++ object model and have to be subclassed by the programmer who wants to implement a customised language feature. As a starting point for developing MOP implementations we provide a default implementation, i.e., a set of concrete classes that provide

the default C++ semantics of all of the reifiable language features. The meta-level class hierarchy is described in section 4.3, the default implementation in section 4.7.

### Iguana Metalevel Class Hierarchy

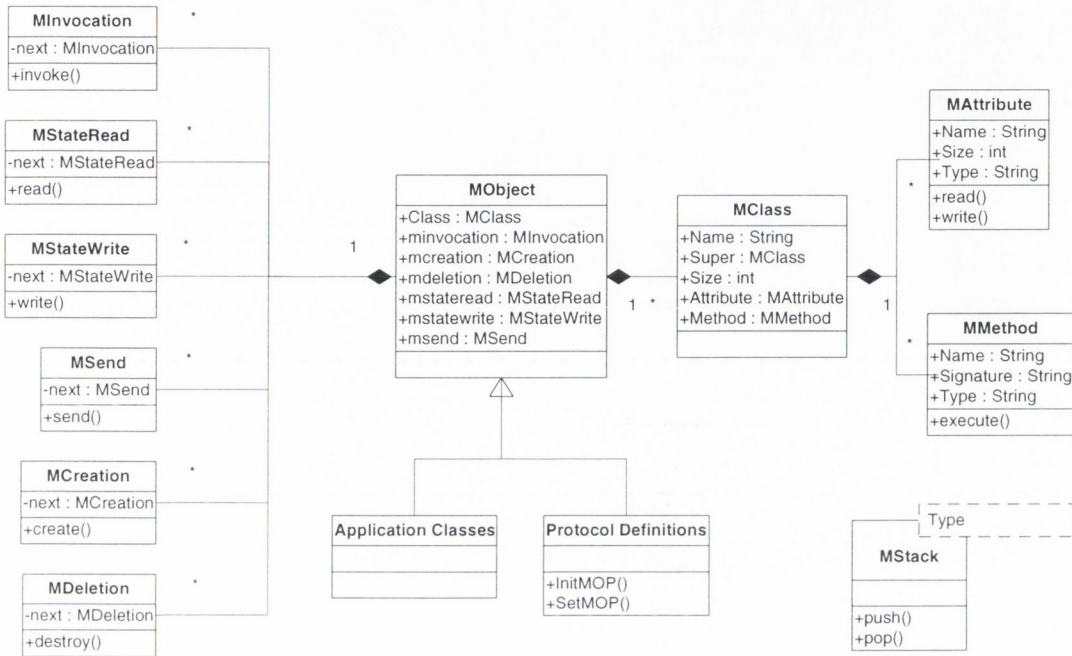


Figure 4.2: UML diagram of the Iguana meta-level class hierarchy. Each of the structural and behavioural reification categories is represented by a corresponding class.

## 4.3 The Iguana Meta-Level Class Hierarchy

The Iguana meta-level class hierarchy implements the self-representation of the C++ object model: each of the structural and behavioural reification categories is represented by a class. For example, structural information about methods is reified by objects of type MMethod, creation is reified by objects of type MCreation etc. Figure 4.2 shows the design of the class hierarchy in UML notation.

In our implementation, we did not explicitly reify **Array** and **Constructor**. This is due to the fact that C++ does not provide language support for arrays (arrays are implemented via pointers) and does not treat constructors differently from a normal method invocation. This is in contrast to, for example, Java where arrays are first class entities and constructors

are treated differently. We therefore implicitly reified arrays as simple attributes and constructors as normal methods.

Figure 4.3 shows an example meta-level configuration of a given object at run-time. MObject is the common base class of all reflective objects. It contains the necessary hooks to metalevel objects and provides the interface to access and invoke metaobjects. Behavioural reification categories can be represented by zero or more metaobjects. In case no metaobject is present for a particular reification category, this simply means that this feature is currently not reified and the native C++ mechanism is used instead. Multiple metaobjects for a particular reification category are chained together in a linked list so that a composition of different behaviours can be obtained. For more about metaobject composition see section 4.6

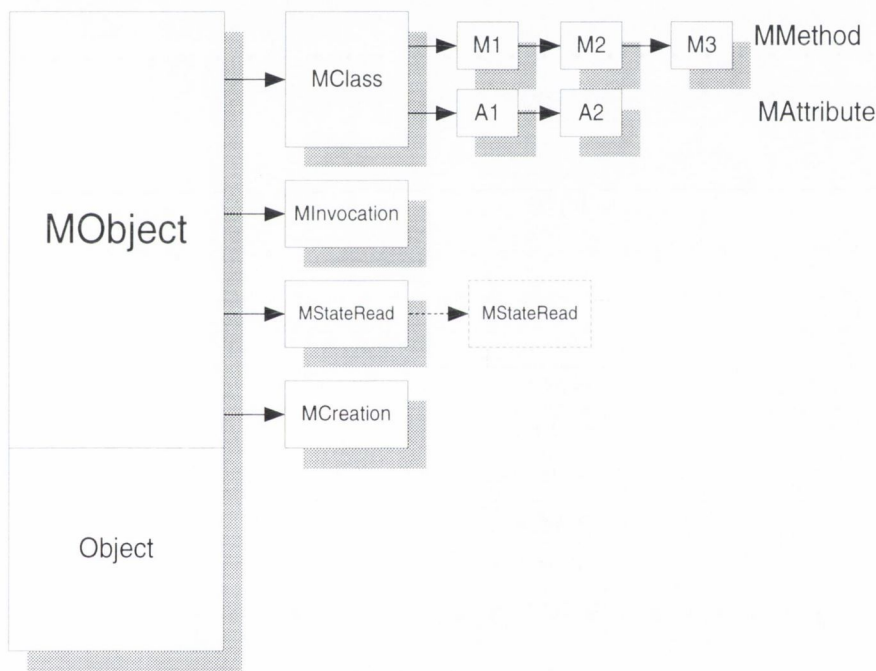


Figure 4.3: Object layout in Iguana. Reflective objects are associated with a number of metaobjects, each of which represents structural or behavioural aspects of the object. Multiple metaobjects representing the same aspect are chained in a linked list.

In contrast to the previous implementation of the Iguana meta-level class hierarchy, we took a different approach in binding objects to their meta-level representation. In the previous version, base-level objects contained a single pointer, the so-called *meta*-pointer, to their meta-level representation. The rationale behind this design was to allow a fast and easy transition from one meta-level representation to another by simply re-directing



the meta-pointer.

Mainly because of performance reasons we found the previous design was disadvantageous: in order to access metaobjects, an additional level of indirection has to be overcome. This overhead becomes significant when performing reified operations on objects such as state read and state write.

We believe that during the life-cycle of an object the number of base-level operations exceed the number of meta-level reconfigurations by far. In our design we therefore focused on allowing a fast transition from base to meta-level computations by minimising the number of indirections. Moreover, switching to a different meta-level representation by simply redirecting the meta-pointer has shown to be a very unsafe operation since some sort of compatibility between the old and new configuration is required. For example, it is not meaningful to replace an object's entire structural information with one that does not reflect the object's actual type.

## 4.4 The Pre-Processor

In the pre-processing phase the Iguana extended source-code is parsed and translated into standard C++. Parsing is traditionally carried out in various stages ([ASU86]):

1. Lexical analysis: the character stream of the source-code is scanned and grouped into tokens
2. Syntactical analysis: the token stream generated during the lexical analysis is parsed and grouped into grammatical phrases.
3. Abstract Syntax Tree generation: during the syntactical analysis an Abstract Syntax Tree (AST) is generated that represents the grammatical phrases of the source program.
4. Semantic analysis and AST modification: in a last step the AST generated during the syntactical analysis is walked and transformed by the parser to synthesise the translated source code.

A number of tools exist that automate the process of parser writing. We chose to use the PCCTS parser generator ([Par96]) as it is freely available and comes with a (fairly) complete C++ grammar. It generates a recursive descent parser from the grammar description. The grammar itself is written in EBNF-form and can be annotated to guide the automated AST-construction and AST-transformation.



## 4.5 Source-to-Source Translation

In this section we present a detailed description of the code modifications that are carried out by the Iguana pre-processor.

### 4.5.1 Protocol Definitions

Protocol definitions associate a set of reification categories with their implementing classes and define a new meta-type. During run-time, the set of metaobjects that constitute a particular meta-type are subject to change in the event an object selects a new meta-type dynamically. We therefore need to capture information about which set of metaobjects are needed to implement a particular meta-type.

Each protocol definition is translated into a corresponding class definition. The corresponding class contains functions to initialise and configure the meta-level that constitutes a particular meta-type. For example, given a protocol definition `MyProt`, the corresponding class definition would be:

---

```
class MyProt : public MObject {
public:
    static MObject *InitMOP(MObject *meta){...}
    static MObject *SetMOP(MObject *oldMOP,
        bool isTargetProt, void *param = NULL){...}
};
```

---

Class `MyProt` in the example above contains protocol specific code to create and initialise a new meta-level configuration (`InitMOP`) that adheres to the protocol definition. Function `SetMOP` on the other hand takes a compatible meta-level configuration and transforms it to the target protocol. Compatible meta-level configurations are those which are either defined in super or subprotocol definitions. Dynamic protocol selection and meta-level reconfiguration are described in more detail in section 4.6.

### 4.5.2 Adding Introspection

C++, as a compiled language, retains only little structural information in the run time image in the form of Run Time Type Information (RTTI). This information is not enough to inspect on actual methods or attributes. Adding introspection is therefore faced with the problem of maintaining structural information about classes, methods and attributes beyond the compilation process. For instance, the following class declaration

---

```
class MyClass {
public:
    int x;
    double Sqrt(double arg){
        return sqrt(arg);
    };
};
```

---

defines a simple class that contains one public attribute and one public member function. The kind of structural information we are interested in is

- The name and size of `MyClass`.
- A list of its super-classes.
- A list of all attributes of `MyClass`: their type, name, size and location (displacement) within objects.
- A list of all methods of `MyClass`: their type, name, signature and address.

Parts of this information, such as the name and type of attributes and methods, can be gathered during the pre-processing stage. Other parts, such as the size of attributes and the addresses of methods, are platform/compiler dependent or can only be determined at run-time.

The Iguana pre-processor augments class definitions with a static member function, *Init-MetaLevel*, which captures the necessary structural information, as shown in figure 4.4, line [08]. Run-time specific information is made available by the *sizeof*, *offsetof* and address of (&) operators. In line [15] the newly created class metaobject is stored in a class table. This allows applications to lookup class definitions given the class's name. In addition,

---

```

[01] class MyClass : public MObject {
[02] public:
[03]     int x;
[04]     double Sqrt (double arg ){
[05]         return sqrt (arg);
[06]     };
[07]     static MObject* MetaMyClass;
[08]     static void InitMetaLevel(){
[09]         MetaMyClass = new MObject();
[10]         MetaMyClass->Class= new MClass("MyClass", sizeof(MyClass), SHARED);
[11]         MetaMyClass->AddAttribute(new MAttribute("x","int", offsetof(MyClass, x),
[12]             sizeof(int), SHARED, 0));
[13]         MetaMyClass->AddMethod(new MMethod("Sqrt", (MAddress)&refl_Sqrt,
[14]             "(double)", PUBLIC, SHARED, "double"));
[15]         ClassTable.push_back(MetaMyClass->Class);
[16]     }
[17]     void refl_Sqrt(){
[18]         double arg;
[19]         Stack->pop(&arg);
[20]         double tmp = MyClass::Sqrt(arg);
[21]         Stack->push(&tmp, sizeof(tmp));
[22]     }
[23] };

```

---

Figure 4.4: Pre-processed class definition in Iguana/C++. Structural information about methods and attributes is created in function `InitMetaLevel`. In addition, the pre-processor has added a wrapper function `refl_Sqrt` that intercepts calls to the original method.

the pre-processor has added a wrapper function `refl_Sqrt`, line [17], to the class definition. Reified method invocations are carried out through this wrapper function.

### 4.5.3 Bootstrapping

At program start-up, a number of initialisation operations have to be carried out in order to create the reflective run-time environment. So, for example, class metaobjects for all user defined classes are created and stored in a class table. This is achieved by inserting code that invokes the `InitMetaLevel`-function for all user-defined classes. Storing class metaobjects in a table enables applications to lookup class definitions at run-time. In addition, the application stack is created. The application stack (an instance of class

MStack) is used by objects that perform a reified method invocation and is used to pass arguments to reflective methods.

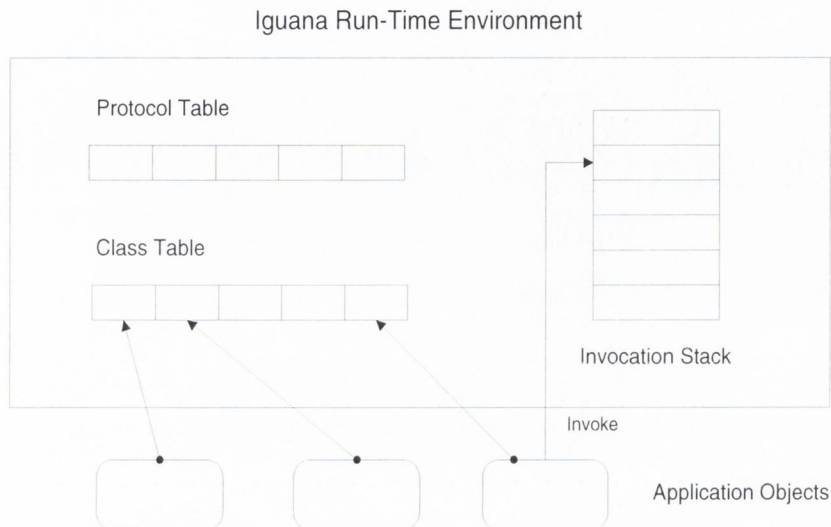


Figure 4.5: Iguana run-time environment, consisting of a class table that stores class definitions, a protocol table that contains supporting functions for dynamic protocol selection, and the invocation stack.

The protocol table contains a list of all protocol definitions and is accessible to the meta-level programmer in order to change an object's meta-type from the meta-level. In section 5.4 we describe how this functionality was used in the implementation of a persistent meta-type. Figure 4.5 depicts the Iguana/C++ run-time environment.

#### 4.5.4 Adding Intercession

The semantic analysis during the pre-processing stage identifies those parts in the application program that perform a reified language operation and replaces them with code that transfers control to the meta-level. The code translation is faced with the problem of replacing an expression with one that is semantically equivalent and type conformant. The following is a discussion of the code modifications for each of the behavioural reification categories. A summary of the code modifications can be found in table 4.1.



### Simple Method Invocation

Method invocations can be of different kinds. The simplest case occurs when a method of type `void` without any parameters is invoked. For example, the following expression

```
obj->method();
```

is translated into

```
obj->invoke(index);
```

where *index* denotes the index of the method in the method table.

### Method with Return Value

In case the method invoked returns a value, for example of type `int`, the translated code would be:

```
*(int*) obj->invoke(index);
```

The meta-level operation returns an untyped pointer to the result value which is then cast to its appropriate type.

### Method with Parameters

Arguments to methods are passed via a global stack object. A method invocation requiring arguments as in

```
obj->invoke(arg1, ..., argn);
```

is translated into

```
obj->invoke(index, (Stack->push(arg1), ... , Stack->push(argn)));
```

The comma expression `(Stack->push(arg1), ...)` is evaluated from the left to the right, thus pushing the arguments onto the stack. Stack operations are templated so that data objects of arbitrary type can be pushed onto the stack.

### State Read/Write

Example code modifications for read and write access to objects are given below. When writing to a field, the new value is passed via the reified stack, similar to method invocations.

|                               |  |   |
|-------------------------------|--|---|
| <code>v = obj-&gt;x;</code>   | $\xrightarrow{\text{translated into}}$ | <code>v = *(int*)obj-&gt;read(index<sub>x</sub>);</code>            |
| <code>obj-&gt;x = val;</code> | $\xrightarrow{\text{translated into}}$ | <code>obj-&gt;write(index<sub>x</sub>, Stack-&gt;push(val));</code> |

### Send

Sending a message to an object is basically a method invocation from within another method. The invocation is redirected to the send metaobject of the calling object, which is then responsible for forwarding the message to the target object.

Original code :

```
MyClass::Method(){
    TargetClass *targetObj = ...;
    targetObj-> m();
}
```

Modified code:

```
MyClass::Method(){
    TargetClass *targetObj = ...;
    this->send(targetObj, indexm);
}
```

Whether or not the sending of a message is actually diverted to the send-metaobject also depends on the target object: the target object is required to have invocation reified in order to be able to receive the message and to invoke the baselevel operation. Thus, two metaobjects have to be present: one for sending the message (at the caller side) and one for receiving the message (at the receiver side). In order to ensure the safe transition to meta-level operations, Iguana inserts run-time checks to the base-level code as described in section 4.5.6.

### Creation

Object creation is a relatively complicated task. In C++, it comprises of

- Allocating the appropriate amount of memory from the heap
- Initialising the virtual function table and pointers to virtual base classes (if any)

- Calling the constructor

Another issue that has to be addressed is the creation of dynamic arrays of objects. In this case C++ first allocates the memory for the entire array and then invokes the constructor for each object individually.

In Iguana, where metaobjects can be associated with individual objects, a fundamental problem arises when interceding with object creation: since the object is not yet existent, we first have to create the local metaobjects that will constitute the future object's meta-level. From that meta-level configuration the base-level object can be instantiated. Before the constructor of the newly created object can be invoked, the meta-level has to be bound to the base-level object since the constructor might rely on the existence of certain structural and/or behavioural metaobjects.

The Iguana run-time support provides the necessary functionality for constructing meta-level configurations (see also section 4.6). For example, given a protocol definition named `MyMOP`, the pre-processor generates a function `MyMOP::InitMOP` that creates and configures the behavioural metaobjects selected in the protocol definition. From that meta-level configuration, the base-level object can be created. So for example

```
obj = new MyClass ==> MyMOP;
```

is translated into

```
MyMOP::InitMOP(MyClass::MetaMyClass)->create( 0);
```

In addition, the Iguana preprocessor inserts into class definitions a factory function that handles both the construction of single objects and dynamic arrays. Taking our class declaration from 4.5.2, the generated factory function would be:

---

```
static void refl_MyClass(MObject *meta, int numObjs){
    MyClass *newobj;
    if (numObjs)           // if we create array of objects
        newobj = new MyClass[numObjs](meta);
    else newobj = new MyClass(meta);
    Stack->push(newobj);
};
```

---

The meta-level configuration that is in control of the creation of the object is passed as a parameter (meta) to the factory function. It is bound to the object in the constructor of `MObject`, the common base class of all reflective objects, and thus before the actual constructor is called. `numObjects` determines the number of objects that have to be created.

### Deletion

As with object creation, we have to distinguish between the deletion of single objects and entire arrays. Again, this is handled by a wrapper function inserted into the class definition. Given below are the code modifications carried out for object deletion, both for single objects and dynamic arrays. A flag to the destroy operation indicates whether or not a single object or an entire array has to be deleted.

|                            |  |                                     |
|----------------------------|--|-------------------------------------|
| <code>delete obj;</code>   | $\xrightarrow{\text{translated into}}$ | <code>obj-&gt;destroy();</code>     |
| <code>delete[] obj;</code> | $\xrightarrow{\text{translated into}}$ | <code>obj-&gt;destroy(true);</code> |

#### 4.5.5 Instance protocol selection

Instance protocol selection allows individual objects to select their dynamic meta-type. The Iguana run-time support takes care of restructuring the object's meta-level, as described in section 4.6. For example, the expression

$$\text{obj} ==> \text{dynMT};$$

selects a new meta-type for object `obj` and is translated into

$$\text{dynMT}::\text{SetMOP}(\text{obj});$$

where `SetMOP` is a generated function that implements the meta-level reconfiguration.

#### 4.5.6 Run-time Checks

Iguana allows the set of active reification categories for a given object to increase/decrease over time by means of dynamic protocol selection. Of course we only want to divert an operation if the target object has an appropriate metaobject that can handle the operation. We achieve this functionality by guarding every object access with a run-time check. If a specific reification category is not active, the native C++ mechanism is used, otherwise



the event is trapped and diverted to the meta-level. For example, method invocations are translated into:

```
META_INVOKE(obj) ?
    obj->invoke(index) : obj->method();
```

The macro `META_INVOKE()` evaluates to true if invocation is reified for a given object. If so, the reflective code is called. Otherwise, the native C++ invocation is executed.

Run-time checks have been introduced to the revised version of Iguana for a couple of reasons. First, they support type orthogonality: reflective and non-reflective objects can be used interchangeably. Second, they provide a convenient and efficient way of dynamically switching between different sets of active reification categories: in case an object deselects a particular reification category, it can revert to the native and efficient C++ mechanism.

As has been mentioned in section 4.5.4, a special case occurs for reifying sending of messages: both the sender and receiver need to be in possession of a metaobject to perform the operation. In this case two run-time checks have to be performed, one for checking whether the sending object has send reified and one for checking whether the receiver has invocation reified.

It is worth mentioning that run-time checks are only inserted where the semantic analysis cannot determine whether or not a particular reification category is selected by an object. For example, if the static meta-type associated with a class has invocation reified, the set of selected reification categories is only allowed to increase in derived protocols. Thus, all instances of that class will always have invocation reified and the run-time check guarding invocations can be omitted.

#### 4.5.7 Nested Expressions

Expressions in C++ can be complex and arbitrarily nested. Under these circumstances, nested expressions have to be broken down into simpler ones, each of which is then translated separately. In the following example,

```
AObj->a->b();
```

a method on a nested object is invoked. In a first step, this expression is broken down, introducing a temporary pointer to the nested object:

```
SomeClass *temp;
temp = AObj -> a;
```

```
temp -> b();
```

In a second step, the two expressions above are translated separately into:

```
SomeClass *temp;  
temp = (SomeClass*) AObj -> read(indexa);  
temp -> invoke(indexb);
```

As can be seen, the first expression is equivalent to a read access: the address of the embedded object is read and stored in a temporary pointer variable.

## 4.6 Dynamic Meta-Type Selection

Protocol inheritance as described in section 3.2.5 allows the (static) combination and augmentation of MOPS. A feature that distinguishes Iguana from other architectures is the ability to change an object's metalevel dynamically: at any one time, an object can select a meta-type that is a subtype of its static meta-type. Broadly speaking, from the implementation point of view, dynamically selecting a meta-type means restructuring the set of metaobjects that constitute the object's current meta-type at run-time. This section describes how dynamic meta-type selection is implemented.

### 4.6.1 Meta-level Reconfiguration

When migrating from one meta-type to another, we want to maintain the state of the object's current meta-level configuration, i.e. keep those metaobjects that are also part of the target meta-type. In fact, dynamically selecting a new meta-type is a quite complex task as it involves the automatic reconfiguration of an object's meta-level, with metaobjects being inserted to or deleted from the current meta-level configuration. The problem is to identify those objects that do and do not constitute the target meta-level. As the type of the metaobjects that build a particular meta-level configuration are protocol-specific, this requires some sort of run-time support in the application.

In general, we have to distinguish between the following cases:

- An object selects a meta-type that is derived from its current meta-type. This involves the creation and insertion of the additional metaobjects into the current meta-level configuration.

- An object selects a meta-type that is a super-type of its current meta-type. This involves the removal of superfluous metaobjects from the current meta-level configuration.
- An object selects a meta-type that is a sibling to its current meta-type, i.e., one that is derived from its static meta-type. This involves both the removal and insertion of metaobjects.

Consider the following example:

---

```

protocol MOP_A {
    reify Invocation : Invoke_A;
};

protocol MOP_B : MOP_A {
    reify Invocation : Invoke_B;
};

protocol MOP_C : MOP_A {
    reify Invocation : Invoke_C;
};

class XY ==> MOP_A {
...
};

```

---

Here we define three protocols, each of which declares **Invocation** to be reified. Objects of type **XY** now can at any one time select the meta-types defined by the protocols **MOP\_A**, **MOP\_B** and **MOP\_C**:

```

XY *obj = new XY(); // static meta-type is MOP_A
obj ==> MOP_C;     // switch to meta-type MOP_C
...
obj ==> MOP_B;     // switch to meta-type MOP_B

```

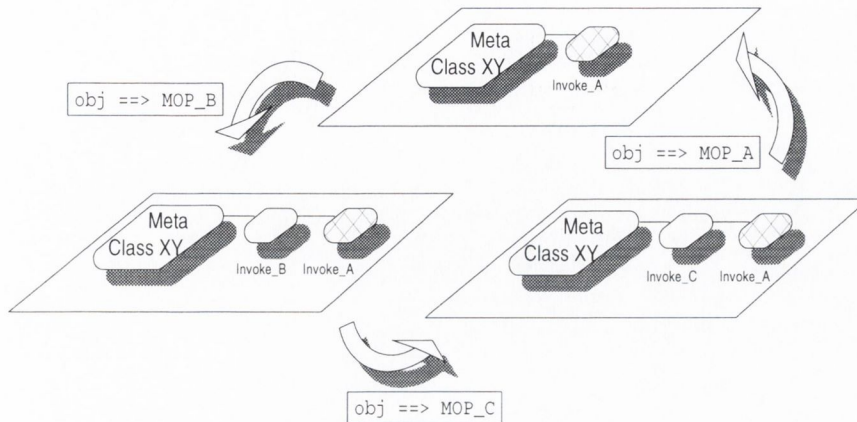


Figure 4.6: Dynamic meta-type selection in Iguana. Selecting a new meta-type at run-time involves the automatic creation, insertion and deletion of metaobjects.

Figure 4.6 illustrates the resulting meta-level configurations for the various meta-types. Switching from protocol A to B requires the creation of metaobjects of type `Invoke_B` since it is derived from protocol A. Switching to protocol C then requires the deletion of `Invoke_B` and the creation and insertion of `Invoke_C`.

As it cannot be known statically whether a newly selected meta-type is a super or a sub-type, the algorithm that performs the dynamic reconfiguration has to cater for both cases. The algorithm is outlined in figure 4.7. The Iguana pre-processor generates a function for each of the protocols, function `SetMOP`, that reconfigures a meta-level configuration so that it conforms to the destination protocol. This function iterates through the list of metaobjects for each of the selected reification categories and marks all those metaobjects that are part of its protocol definition (each protocol 'knows' the type of metaobjects that it contributes, line [4] and [5]). In order to do so, each metaobject contains the name of the protocol to which it belongs. If such an object can't be found, it creates the metaobject of the appropriate type and inserts it to the list (line [7]). This function is recursively called for all super-MOPs (each protocol 'knows' its direct super-protocols, lines [2] and [3]). The destination protocol finally iterates through the list again and deletes all metaobjects that have not been marked: these are the ones that do not constitute the target meta-level configuration (lines [9] and [10]).



```
[1] ProtocolX :: SetMOP(MObject * oldMOP) {  
[2]    $\forall p \in \text{Super}(\text{ProtocolX}) :$   
[3]     p :: SetMOP(oldMOP);  
[4]   if  $\exists m \in M(\text{ReificationCategory}) : m \in \text{ProtocolX}$   
[5]     mark(m)  
[6]   else  
[7]     create new m, insert it into M(ReificationCategory);  
[8]   if (ProtocolX is target protocol)  
[9]      $\forall m \in M(\text{ReificationCategory}) : \neg \text{mark}(m)$   
[10]    delete m  
[11] }
```

Figure 4.7: Algorithm for dynamic meta-type selection.  $\text{Super}(\text{ProtocolX})$  denotes the set of all direct super protocols,  $M(\text{ReificationCategory})$  is the set of metaobjects that implement a particular reification category.

## 4.7 The C++ Default Protocol

The Iguana/C++ default protocol is a concrete set of classes that provide the default behaviour of C++ objects. It can be used as a basis for developing MOP implementations. In this section we outline the implementation of the default protocol and highlight the language issues that arose when developing the C++ default behaviour.

### 4.7.1 Method Invocation

One of the problems faced when implementing reified method invocations is how to deal with user-defined data types as parameters and return values. OpenC++ v1 for example introduces a class `ArgType` that encapsulates arguments to functions. Application programmers are required to derive from this class for each user defined data type in order to provide specific marshalling and unmarshalling operations for each data type. This approach places much of the burden of meta-level programming on the base-level programmer and clearly defeats ease of use and transparency.

The previous version of Iguana used a similar approach in passing parameters to functions: arguments were passed via activation frame objects, appropriate subclasses for each user-defined data type could be generated automatically by the preprocessor. The problems with this approach were a) Scalability: this required for each method the generation of a new class that contained the arguments as its data members. b) Performance: creating and deleting activation frame objects incurred substantial run-time overhead.

In the revised version of Iguana, arguments are passed via a global stack object. Stack operations are templated so that push and pop operations for user-defined data types are generated automatically by the compiler back-end.

### 4.7.2 State Access

The default implementation of the behavioural reification categories `StateRead` and `StateWrite` respectively calculate the address of the data member whose state has to be read/written. Address calculation depends on whether the data member is a single field, an array of fixed or dynamic size, or a static data member (i.e., one whose data is shared between all instances of the class). Consider the following example:

---

```

class MyClass {
public:
    static int statInt;
    int someInt;
    int Array[10];
    int *dynArray;
    MyClass(){
        dynArray = new int[10];
    }
};

```

---

The default implementation of StateRead/StateWrite has to distinguish between the following cases:

```

obj->statInt = 456;
obj->someInt = 123;
obj->Array[index] = 234;
obj->dynArray[index] = 345;

```

Case 1 shows a write access to a static data member. The address of a static data member is the same for all objects of the class and can be determined by means of the `addressof` operator:

$$\text{address} = \text{addressof}(\text{MyClass}::\text{statInt}) \quad (4.1)$$

Case 2 shows a write access to a single field. The address of the data member is simply the address of the object plus the displacement between the object and the data member:

$$\text{address} = \text{addressof}(\text{obj}) + \text{offsetof}(\text{someInt}, \text{MyClass}) \quad (4.2)$$

Case 3 shows a write access to an array of fixed size. In this case we first have to calculate the beginning of the array within the object (as above) and then add the displacement within the array:

$$\text{address} = \text{addressof}(\text{obj}) + \text{offsetof}(\text{Array}, \text{MyClass}) + \text{index} * \text{sizeof}(\text{int}) \quad (4.3)$$

Case 4 shows a write access to a dynamic array of integers. Here, we first have to calculate the location of the pointer variable within the object. The address at that location points to the first element in the array, from where we can add the displacement within the array:

$$\begin{aligned} \text{address} = & \text{deref}(\text{addressof}(\text{obj}) + \text{offsetof}(\text{dynArray}, \text{MyClass})) \\ & + \text{index} * \text{sizeof}(\text{int}) \end{aligned} \quad (4.4)$$

### 4.7.3 Object Creation

As has been described above, object creation in C++ is a relatively complicated task. It involves the allocation of the appropriate amount of memory to contain the object's attributes, initialisation of the object's virtual function table and possibly the execution of a constructor method. When creating dynamic arrays of objects, these steps have to be repeated for all elements of the array.

As a matter of fact it is not possible in C++ to fully reify object creation for a number of reasons. First, we can only intercede with the explicit creation of objects, indicated by the use of the C++ `new`-operator. Second, certain aspects of the C++ object model are not fully specified and hence compiler dependent, for example, the layout of objects in memory and the implementation of the virtual function table. The default implementation for the creation of objects therefore invokes a factory function that has been added to the class definition by the pre-processor. The factory function in turn creates either single or dynamic arrays of objects using the default C++ `new`-operator.

## 4.8 Restrictions

Providing full support for reflection in a complex programming language such as C++ is a non-trivial task. Often we were confronted with the feature-richness and idiosyncracies of that language. The following is a summary of the restrictions that we encountered when applying the Iguana model to C++.

### 4.8.1 Automatic objects

C++ distinguishes between heap-allocated (or dynamic) and stack-allocated (or automatic) objects. Heap allocated objects are created explicitly by means of the C++ `new`-operator and persist until they are explicitly deleted by the `delete`-operator. Stack allocated objects on the other hand are created automatically when the object goes into scope.



Code for creating and deleting stack allocated objects is generated by the compiler and cannot be replaced by the pre-processor. It is thus not possible to intercede with the creation/deletion mechanism of stack allocated objects.

### 4.8.2 Arrays

In contrast to Java, C++ does not have direct language support for arrays. Instead, arrays are implemented using pointers. We therefore did not implement the reification category `Array`. Instead, arrays are reified as attribute metaobjects with additional information about the number of elements in the array.

### 4.8.3 Aliasing

Aliasing occurs in a program when two or more names exist for the same data object. In C++ aliasing comes in two shapes, pointers and references. In practice this means that under some circumstances, due to a loss of type information, not every operation on an object can be detected and transferred to the meta-level. Consider the following example:

```
class MyClass {
    public: int i;
};
MyClass *obj = new MyClass(); //create new object
int *pInt = (int*)obj;       // pInt is an alias for obj
*pInt = 23;                  // access member variable
```

Here we first create an object of type `MyClass` and then an alias to the object typed as a pointer to `int`. Subsequently dereferencing the pointer variable is effectively a state access to the object's fields that cannot be intercepted.

Although the above example is not considered as being a good programming style, it illustrates one of the many ways in which invocation via the meta-level can be circumvented. However, for a full reflective architecture it is necessary to detect all operations on application objects, otherwise full causal connection between base and meta-level cannot be guaranteed and inconsistencies are inevitable.

Problems of this kind can only be overcome by introducing an alias analysis for the application code. As a matter of fact, it has been shown that a complete alias analysis for pointer-induced aliasing becomes NP-hard and that no good approximation algorithms

exist [LR91].

## 4.9 Summary

This chapter described Iguana/C++, a concrete mapping of the Iguana model onto C++. Iguana/C++ is implemented by means of a pre-processor: the Iguana extended code is parsed and translated back into standard C++. A number of code modifications are carried out, most notably the insertion of code to capture structural information about the application and run-time checks to guard and divert operations on objects. Table 4.1 summarises the code modifications for the behavioural reification categories. For simplicity, the run-time checks have been omitted.

We also described the process of instance protocol selection and meta-level reconfiguration. This process allows individual objects to dynamically select a meta-type. Finally, we outlined the implementation of the C++ default MOP, a meta-type that provides the default semantics of C++.

| Reification Categories | Original Code                            | Translated Code   |
|------------------------|--|---|
| Invocation             | <code>obj-&gt;m();</code>                | <code>obj-&gt;invoke(<i>index<sub>m</sub></i>);</code>                      |
| Invocation + params    | <code>obj-&gt;m(arg);</code>             | <code>obj-&gt;invoke(<i>index<sub>m</sub></i>, Stack-&gt;push(arg));</code> |
| StateRead              | <code>x = obj-&gt;a;</code>              | <code>x = *(T*)obj-&gt;read(<i>index<sub>a</sub></i>);</code>               |
| StateRead(array)       | <code>x = obj-&gt;a[i];</code>           | <code>x = *(T*)obj-&gt;read(<i>index<sub>a</sub></i>, i);</code>            |
| StateWrite             | <code>obj-&gt;a = v;</code>              | <code>obj-&gt;write(<i>index<sub>a</sub></i>, Stack-&gt;push(v));</code>    |
| Send (inside method)   | <code>obj-&gt;m();</code>                | <code>this-&gt;send(obj, <i>index<sub>m</sub></i>);</code>                  |
| Creation               | <code>obj = new <i>Class</i>;</code>     | <code>obj = Prot::InitMOP(Class::MetaClass)<br/>-&gt;create(0);</code>      |
| Creation(array)        | <code>obj = new <i>Class</i>[sz];</code> | <code>obj = Prot::InitMOP(Class::MetaClass)<br/>-&gt;create(sz);</code>     |
| Deletion               | <code>delete obj;</code>                 | <code>obj-&gt;destroy();</code>   |
| Deletion(array)        | <code>delete[] obj;</code>               | <code>obj-&gt;destroy(true);</code>   |
| Protocol Selection     | <code>obj ==&gt; <i>Prot</i>;</code>     | <code><i>Prot</i>::SetMOP(obj);</code>                                      |

Table 4.1: Summary of code modifications for the various reification categories (run-time checks omitted).

## Reflective Programming with Iguana

Whoever admits that he is too busy to improve his methods has acknowledged himself to be at the end of his rope.

*J. Ogden Armour*

Object-oriented analysis and design are nowadays common practice and supported by a number of design tools that help to manage and automate the transition from the analysis to the design to the implementation phase. Reflective programming however lacks such a common methodology, which has only recently become a focus of research [CST00a]. In this chapter we will illustrate how the reflective features of Iguana can be used to extend the C++ object model in order to provide support for, for example, software evolution and persistence. We will see that meta-level programming with Iguana is not unlike conventional object-oriented design.

### 5.1 Using Introspection

Using introspection to look up the class definitions of objects is done in Iguana in similar ways to what has been described for other reflective architectures in chapter 2. Finding out about the structure of objects can for example be used to perform object serialisation or to find out about the interface offered by object whose type is not known until run-time. The example shown in figure 5.1 illustrates how introspection can be used in Iguana to look-up the name, type and signature of methods. Since not necessarily all objects contain class information (class information is only available if it is reified), we have to check whether the target object contains a valid class descriptor.



---

```

void listMethods(MObject *targetObject){
    int i=0;
    MClass *cl = targetObject->Class;
    if (cl){
        MMethod *m;
        while (m = cl->GetMethod(i)){
            printf("Method %d: %s %s %s",
                i, m->ReturnType, m->Name, m->Signature);
            i++;
        }
    }
}

```

---

Figure 5.1: Introspection in Iguana.

## 5.2 Boundary Checks for Arrays

By default C++ does not carry out boundary checks when accessing array elements, i.e. it is the programmer's responsibility to either ensure that array indices are never out of bounds or to perform run-time checks before each access. Using reflection, performing boundary checks on arrays can be implemented very easily. Moreover, using Iguana's protocol selection mechanism, it can be applied to classes and/or objects individually. Boundary checks can therefore be performed during the debugging phase and later simply switched off.

In order to implement boundary checks for arrays, we build a hierarchy of protocols, incrementally extending the functionality of the read/write operations as shown in figure 5.2. Starting with a simple protocol that only selects structural information to be reified, we subsequently add default semantics and finally boundary checks to the read/write operations.

The protocol **BoundTest** is derived from a protocol that implements the default semantics for state access. Metaobjects of type **BoundWrite** and **BoundRead** respectively will perform the boundary test before delegating the call to the default implementation. A simple implementation could print out debugging messages for out-of-bound accesses, a more sophisticated implementation could realise a "smart-pointer" policy and dynamically resize

```

protocol TypeInfo { // reify structural information
shared:
  reify Class      : MClass;
  reify Attribute  : MAttribute;
};

protocol DefaultMOP : TypeInfo { // provide default semantics
shared:
  reify StateWrite: DefaultWrite;
  reify StateRead : DefaultRead;
};

protocol BoundTest : DefaultMOP { // perform additional boundary checks
shared:
  reify StateWrite: BoundWrite;
  reify StateRead : BoundRead;
};

```

Figure 5.2: Protocol hierarchy implementing boundary checks for arrays. Starting with a protocol definition that reifies only structural information about classes and attributes, default semantics and boundary checks are added in subprotocols.

the array.

Since arrays in C++ are not first class entities, access to array elements can only be intercepted if the array is embedded into an object. Using dynamic meta-type selection we can switch on/off boundary checks for individual objects as the following code example illustrates:

```

class MyClass ==> TypeInfo { ... };

MyClass obj = new MyClass(); // creating object without boundary tests
obj ==> BoundTest;           // switching on boundary tests
obj->array[23] = 12;

```

### 5.3 Run-time Adaptation of Systems Software

One of the main motivations behind using reflection in software engineering is to build applications which are faced with changing requirements, either statically or dynamically.

[LYiI95] for instance describes how reflection is used in the Apertos operating system to support run-time adaptation. Other examples of reflection being used as a principle tool for supporting adaptability, especially in the area of middleware, include [Led97] and [KCR99]. In [DSC<sup>+</sup>00] we described how Iguana can be used to build flexible and adaptable systems software.

In this case study we examined how traditional, object-oriented design methodologies compare with the use of reflection. As an example we choose a memory allocator, hereinafter called a buffer manager, a shared resource that allocates and releases contiguous blocks of memory on behalf of its clients.

Ideally, the buffer manager should perform optimally for any sequence of allocate/release operations with both operations running in near constant time. However, performance is difficult to predict when subjected to an application-specific pattern of allocate and release operations. Different strategies provided by the buffer manager can allow clients to select an appropriate implementation which best fits their needs. Customised implementations could include a first-fit, best-fit or worst-fit strategy for the allocate and release operations.

In the following sections we will illustrate the steps involved with adding adaptation to a canonical, non-adaptive version of the buffer-manager, first by employing a traditional, object-oriented approach and then by using reflection. A more in-depth discussion of the various usage scenarios can be found in [DSC<sup>+</sup>00].

### 5.3.1 The minimal Buffer Manager

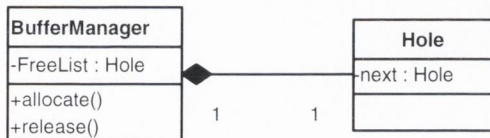
In its minimal form, the public interface of the buffer manager provides operations for allocating regions of memory. Internally, the buffer manager maintains a linked list of free block of memory. The actual policy employed by the buffer manager is embedded in the implementation of the `allocate/release` operations and cannot be changed, see figure 5.3.

### 5.3.2 Adaptation using Design Patterns

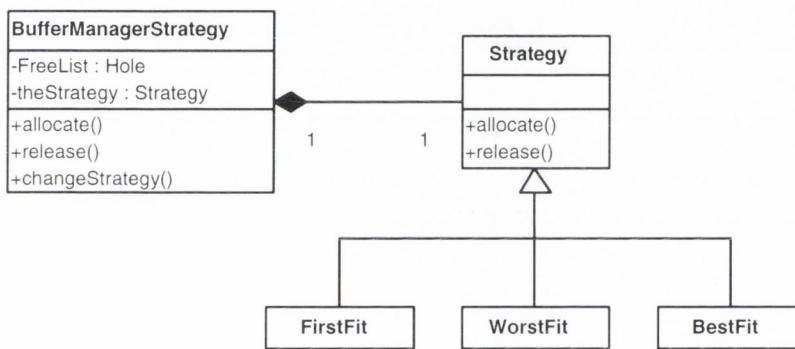
The strategy pattern is an object-oriented pattern [GHJV95] and has been used in the design of dynamically adaptable systems such as TAO [KCR99]. The design principle underlying the strategy pattern is to delegate the implementation of the exported operations of the buffer manager to a replacable strategy object. Multiple strategies can be provided by deriving from an abstract strategy class and compiling it into the system.

Figure 5.3 depicts the minimal buffer manager class and its extended version using the strategy pattern. The extended version specifies an interface that allows clients to request

a change of the implementation using the `changeStrategy`-operation. This operation represents a declarative meta-interface to the buffer manager [KLL<sup>+</sup>97]. At any time, a client can use knowledge of its own memory access patterns to select the most appropriate allocation strategy.



Minimal Buffer Manager Class



Buffer Manager with Strategy Pattern

Figure 5.3: UML diagram depicting the class hierarchies for a simple memory allocator (`BufferManager`) and its extended version using the strategy pattern.

### 5.3.3 Adaptation using Reflection

In the reflective version, adaptation is made available by reifying invocations on the original buffer manager class and by providing a meta-interface (a so-called *extension protocol*) that allows to rebind the code of the `allocate`/`release` operations. The code for the new strategies can for example be provided in the form of a dynamic link library (DLL).

The steps for implementing the dynamically adaptable buffer manager are outlined below. As in the previous example, we start with a simple protocol definition in order to reify



structural information about classes. Behavioural reflection can then always be added later in subprotocols. In a second step, we declare an extension protocol that provides the code for switching to a new strategy. The purpose of an extension protocol is to encapsulate and separate meta-level code from the actual MOP implementations, allowing the same extension protocol to be reused for multiple, compatible MOPs.

---

```

protocol TypeInfo {
shared:
    reify Class      : MClass;
    reify Attribute : MAttribute;
    reify Method    : MMethod;
};

class Hole ==> TypeInfo {...};
class BufferManager ==> TypeInfo {...};

class AdaptationProtocol {
public:
    void changePolicy(MObject *bufman, char *strategy);
};

```

---

When a client binds to a buffer manager object, it is provided with the strategy originally employed by the buffer manager class. As long as the client does not request a different strategy, invocation is not reified implying that the standard C++ invocation mechanism is used.

When adaptation is triggered by the application, invocation is reified allowing to divert the call to the new implementation. New strategies can be provided on the fly by subclassing the original buffer manager class, redefining the allocate/release methods and by compiling the code into a DLL.

The meta-level code for rebinding the implementation of the allocate/release methods performs the following tasks:

1. open a DLL as specified by the strategy parameter;
2. rebind the code of the allocate/release methods. This is done by modifying the method-metaobjects of the buffer manager class: each method metaobject contains

a function pointer to the method which can be replaced to point to the new code.

- reify invocation for the client: in order to divert all future invocations to the imported code, invocation on the client side has to be reified. Reifying invocation simply involves creating an invocation metaobject and inserting it to the buffer manager's meta-level configuration. Run-time checks ensure that all further invocations are redirected to the meta-level.

Figure 5.4 depicts the logical view of the adaptation mechanism. It is worth mentioning that the original interface of the buffer manager class has not been altered, the additional functionality to support adaptation is completely encapsulated in the extension protocol and is orthogonal to the base-level application.

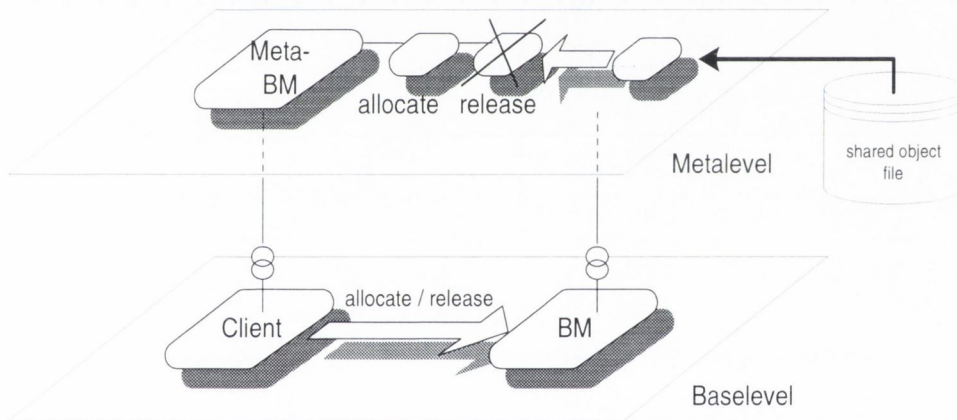


Figure 5.4: Conceptual view of the adaptation mechanism using reflection: clients communicate with the buffer manager through the allocate/release interface. Method metaobjects represent the implementation of these operations and can be replaced with code loaded from a DLL.

#### 5.3.4 State Transfer

A general problem arising out of the adaptation of software components has to deal with a possible transfer of the internal state of the component. In our example, switching to a new strategy may require the list of free memory blocks to be arranged in a different order, for example according to their size. This can only be done with a knowledge of the implementation details of the buffer manager.

The implementor of a new strategy is therefore required to provide a callback function that performs the transfer of state. Since a new strategy is provided by subclassing from the original buffer manager class, we can assume that the implementor of the new strategy has sufficient knowledge of how the list of free blocks is represented because the code that implements the allocate/release methods relies on their internal representation anyway. By using some sort of naming convention for the callback function, it is possible to link the new method to the meta-level representation of the buffer manager and to invoke it when adaptation has been triggered.

### 5.3.5 Discussion

The addition of the strategy pattern to the buffer manager to support dynamic adaptation necessitated the complete restructuring of the buffer manager class, thus leading to a tangling of code that implements dynamic adaptation and original code. Delegating the implementation of the buffer manager's methods to a strategy object also had the undesirable side-effect of having to make its state public. Apart from associating the buffer manager class with a protocol definition, the reflective version did not require any modifications to the original class.

This example has illustrated how reflection can provide a general infrastructure (consisting of MOP and extension protocol) for building applications that can dynamically adapt to changing requirements. The additional functionality to support adaptation can be completely encapsulated in meta-level code and is independent from the application's static class structure. It would be interesting to explore how other design patterns can be implemented using a reflective programming language.

## 5.4 A Meta-Type for Persistent Objects

Many software applications require data to be retained between consecutive executions. Programmers mainly use the file system or database management systems (DBMS) for storing such persistent data. In either case, the data model supported by the storage system is usually different from that of the programming language. This requires the program to convert persistent data into the format expected by the file or database system on storage and to re-convert it into the format normally used by the program on retrieval. Although these contemporary systems have strengths, the development of conversion code for every new application can be tedious, time-consuming and error-prone [CNTR97]. Moreover, it may be difficult to maintain pointer semantics and type checking between



the two data models.

In modern object-oriented programs, data is represented in form of objects. Making objects persistent is an efficient way of saving the program data into stable storage. In persistent object systems, the internal (in the language) and external (in the stable storage) data models are very similar to each other and the overhead of conversion is very small.

In this section we describe the design and implementation of a meta-type that extends objects with persistence, see also [HSC01]. We will first describe a more naive implementation that is later optimised for speed. From the base-level programmer's point of view there is no difference between the two implementations in the use of object persistence.

#### 5.4.1 Overview of Object Persistence

Because object references in C++ are implemented as memory addresses, they are essentially volatile, i.e. do not retain their meaning between program executions. Such references must therefore be replaced with persistent references when the objects in which they are contained are stored in non-volatile memory - a process known as reference swizzling [ACC82]. Likewise, persistent references must be replaced with the appropriate memory addresses when their containing objects are loaded into memory (unswizzling). Since the targets of these references may not actually be loaded in the address space, some means of handling attempts to access (i.e. by dereferencing a pointer to) such non-resident objects must be implemented so that the target objects can be loaded on demand. This process is usually referred to as object faulting.

In our case, non-resident objects are represented by proxy objects that occupy the same amount of memory space as the objects that they represent. When a reference to a non-resident object is unswizzled, its proxy is created, if necessary, using information in the persistent reference. The persistent reference is then replaced with the appropriate address within the space occupied by the proxy. Of course, subsequent attempts to use the reference must then be caught.

#### 5.4.2 Implementing Persistent Objects using Iguana/C++

Object persistence is implemented as a set of Iguana/C++ protocols as depicted in figure 5.5 and based on the Tigger object support framework [Cah99]. Any type can potentially persist, provided it is associated with the **Persistent** protocol. Pointer semantics between persistent objects are preserved. Classes can include primitive types, class, pointer and array attributes. The broad range of C++ language issues arising when designing a persistent



```

protocol TypeInfo {
shared:
  reify Class      : MClass;
  reify Attribute  : MAttribute;
  reify Method     : MMethod;
};

protocol Persistent : TypeInfo {
local:
  reify Class      : PersistentClass;
shared:
  reify Creation   : PersistentCreation;
  reify Deletion   : PersistentDeletion;
  reify StateRead  : PersistentRead;
  reify StateWrite : PersistentWrite;
  reify Invocation : PersistentInvocation;
};

```

Figure 5.5: Protocol hierarchy implementing persistent objects.

C++ extension can be found in [KYK+99].

In our design, we adopt persistence by reachability to determine which objects are to be retained. In other words, potentially persistent objects that are transitively reachable from a distinguished persistent root via references will persist between program executions. The states of these objects are stored in an underlying Persistent Object Store (POS). The implementation of the **Persistent** protocol interfaces to the underlying POS and initiates the loading and storing of objects in the POS as necessary as well as of detecting access to non-resident objects.

While the use of reflection is essentially transparent to the programmer, the use of persistence is not. For example, the application programmer must be aware of the specifics of the model of persistence provided by the **Persistent** protocol, the implications of persistence by reachability and the physical location of the POS files. When working with persistent objects, the application programmer may need to distinguish between the cases where a new object needs to be created and initialised for the first time versus the case where the object has been created by a previous execution of the program.

The role of each of the metaobject classes being used is outlined below:

**Class:** **PersistentClass** extends **MClass** to include an object header for each persistent ob-

ject. This header contains information about the object's persistent reference, state in memory (absent/present) and the number of references that it contains (including references inherited from parent classes). Both object faulting and reference swizzling and unswizzling are handled by `PersistentClass` when required.

**Attribute:** `MAttribute` metaobjects are used to store information about references (e.g. type, size, access modifier, offset within the object) for swizzling/unswizzling of references in persistent objects.

**Creation:** on object creation, `PersistentCreation` checks whether the object's class has been installed in the persistent class register. If not, it installs the class and then initiates the creation of the the persistent object with an appropriately initialised meta-level and object header.

**Deletion:** on deleting a persistent object, `PersistentDeletion` checks if the object has been recorded with a name in the name service. If so, it removes the entry and then it initiates the removal of the persistent object together with its meta-level and object header.

**Method Invocation:** when a method is invoked on an object that is present in memory, the default method invocation is carried out. When a method is invoked on a proxy, control is passed to the `PersistentClass` metaobject that handles the object fault. All persistent references in the object's state are then unswizzled.

**State Read and Write:** like method invocation, control is passed to the `PersistentClass` metaobject that is responsible for handling the object fault if the state of a proxy is accessed. Otherwise, the default state access is carried out.

### 5.4.3 Using Persistent Objects

Having outlined the design of a meta-type that equippes objects with persistent properties, we will now focus on how the base-level programmer uses persistence.

When working with persistent objects, the application programmer may need to distinguish between the cases where a new object needs to be created and initialised for the first time versus the case where the object has been created by a previous execution of the program.

To allow programs to refer to previously created objects, a simple name service is provided, which allows the association of symbolic names with persistent objects. The interface to the name service is provided as an Iguana/C++ extension protocol called PEP - the

Persistence Extension Protocol. Extension protocols are an Iguana/C++ concept used to provide a secure and structured interface to the meta-level code. In this case, the Extension Protocol simply constitutes an API for the base-level programmer who wants to make use of object persistence, similar APIs can be found in other architectures [SKW92]. The PEP is defined as:

---

```
class PEP {
public:
    static bool init(char* filename);
    static bool close();
    static bool record(char* name, char* type, void* object);
    static bool lookup(char* name, char* type, void*& object);
    static bool remove(char* name);
};
```

---

PEP::record allows a symbolic name to be assigned to a persistent object. Upon successful execution, this method makes the referred object a persistent root. PEP::lookup allows the recall of any previously recorded persistent root object based on its assigned symbolic name. The type argument is used to check whether the expected type and the type of the restored object match. Finally, PEP::remove allows the removal of symbolic name/object associations. PEP::init initialises the POS and a call to PEP::close results in storing all reachable persistent objects.

An application programmer who wants to avail of object persistence simply uses the default protocol selection to select the **Persistent** protocol. For example:

---

```
defaultProtocol ==> Persistent;

class Counter {
    ...           // Implementation of class counter
};
```

---

will result in all instances of class `Counter` being potentially persistent. Within the same source file, the programmer could define other (sub)classes that would also become potentially persistent. If the programmer uses multiple source files, each source file with class declaration(s) must include the same default protocol selection statement. In this case, extra care must be taken by the programmer as there is a danger that persistent object will hold references to non-persistent objects.

An example of an Iguana/C++ program that uses persistent objects is shown below. Note that the program is coded to be aware of whether it needs to create a new object or use a previously created one.

---

```

Counter *pc1;

PEP::init("/mypos");
if (firstTime) {
    // create object and record it in the name service
    pc1= new Counter(1);
    PEP::record("/this/counter", "Counter", pc1);
} else {
    // get reference to object from name service
    PEP::lookup("/this/counter", "Counter", pc1));
}

if(pc1) {
    // use object
    cout << "Value is: " << pc1->getValue() << endl;
}
PEP::close();

```

---

Of course, it is not necessary to record every persistent objects in the name service. If a persistent object contains references to other persistent objects, they will also persist and are loaded as and when required in subsequent program executions.



#### 5.4.4 Adapting the Meta-Level

A problem with the implementation described above is that all language operations on a potentially persistent object are trapped and carried out via the (slower) meta-level, even if the object is already present in memory and could be treated as a normal C++ object. To overcome this problem we will now present a refined version of the **Persistent** protocol that takes advantage of dynamic meta-type selection. The protocol hierarchy is defined as:

---

```
protocol Persistent : TypeInfo {
  shared:
    reify Creation      : PersistentCreation;
    reify Deletion     : PersistentDeletion;
};

protocol PersistentProxy : Persistent {
  shared:
    reify Class        : PersistentClass;
    reify StateRead   : PersistentRead;
    reify StateWrite  : PersistentWrite;
    reify Invocation  : PersistentInvocation;
};
```

---

The **Persistent** protocol only intercepts object creation/deletion and represents an object that is present in memory. The **PersistentProxy** protocol on the other hand intercedes with all other language operations and represents an object that is absent.

Figure 5.6 outlines the meta-level configuration for persistent objects that are present in memory. Important steps to point out are:

1. Potentially persistent objects are associated with the persistent protocols, hence only creation/deletion is reified.
2. Upon creation, the allocation of an object header is requested in the POS.
3. The POS completes the request.

4. Method invocation, state read or write is not reified, thus these operations are carried out without involving the meta-level.

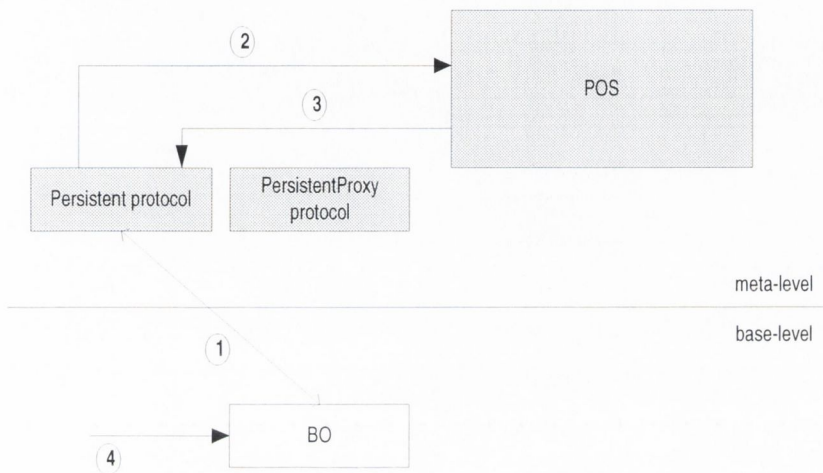


Figure 5.6: Meta-level configuration for present persistent objects. Operations are not intercepted.

Figure 5.7 depicts the meta-level configuration for persistent objects that are absent, i.e., are referenced by resident objects but not loaded from the POS yet. Absent objects are associated with the `PersistentProxy` protocol, all language operations are reified (1). When an operation is requested on a proxy base-level object (2), the operation is directed to the meta-level (3). The object fault is handled and the object is loaded from the POS (4). The POS completes the request and reference unswizzling is performed (5). The object is now present in memory and the meta-level configuration switches to the `Persistent` protocol, consequently all further operations are now carried out directly (6).

#### 5.4.5 Discussion

One of the goals of reflective programming is to provide a clean separation of concerns between the application logic and the meta-level representation. However, in this example full separation of concerns is not attainable: the logic of writing applications that make use of persistent objects requires two different execution paths to be provided by the programmer, one for a cold start when objects are created the first time and one for a warm start when objects are retrieved from the persistent store. Registering and retrieving persistent root objects also has to be done explicitly and cannot be shifted away to the

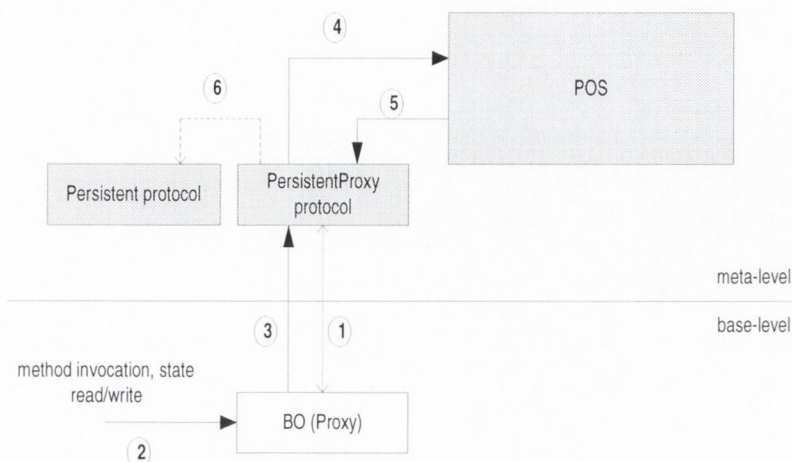


Figure 5.7: Meta-level configuration for absent persistent objects. Operations are intercepted and object faults are handled at the meta-level.

meta-level. As our experience has illustrated, the development of metaobject protocols can to a large extent be implemented separately from the base-level application.

From the base-level programmer's point of view, the steps involved with adding support for persistent objects to existing applications using Iguana consist of associating **Persistent** with classes that are to be made persistent, the insertion of code to register/lookup persistent objects and an additional pre-processing phase to apply the Iguana model. It should not be denied that due to the complexity of C++ it is not trivial to provide a fully reflective language extension. So for example it is only possible to make heap allocated objects persistent due to the inability to intercede with the creation of stack allocated objects, a restriction that also holds true for other systems.

## 5.5 Summary

The previous examples have illustrated how meta-level programming is done in Iguana/C++. We have shown that the design and implementation in Iguana is not unlike conventional object-oriented programming. Protocol definitions associate reified language operation with their implementing classes, protocol inheritance allows to incrementally design, extend and combine object behaviours.

When adding more complex object behaviours, such as persistence, a fundamental question

arises to what extent the semantics of an existing program can be maintained, or in other words, how transparently can persistence (or a similar functionality) be added to a language. Of course, working with persistent objects requires the base-level programmer to distinguish between the cases where an object is created the first time versus when it is retrieved from the object store. This semantic difference does however not arise due to the use of a meta-level architecture, but is inherent to the use of object persistence.

Composition of complex and non-trivial object behaviours raises a number of issues, so for example in which order metaobjects have to be processed and how conflicts are resolved should they arise. The default composition rules in Iguana are simple and generic enough to allow automatic and, if desired, user defined composition of metaobjects. As a major advantage compared to other architectures, including the previous version of Iguana, meta-level programmers do not need to explicitly access and restructure meta-level configurations. Combining and composing MOP implementations with a non-trivial semantic is the focus of ongoing research.



## Evaluation

It is an immutable law in business that words are words, explanations are explanations, promises are promises but only performance is reality.

*Harold S. Geneen (1910 -)*

The flexibility gained by opening up the language and allowing the programmer to customise language semantics comes with a price: reflection intrinsically incurs interpretative overhead. Although this problem has been addressed by a number of researchers ([Chi97], [MMWY92]), it has not been solved in a way that has increased the acceptance of reflective programming. To draw an analogy, in order to justify the choice for an object-oriented programming language over a simple procedural language, it has to be shown that the benefits of object-orientation outweigh the additional complexity of the language (in form of, for instance, single/multiple and virtual inheritance) and its decreased performance, incurred by, for example, method dispatching. In order to identify profitable targets and techniques for optimisation, it is crucial to understand where and why overhead occurs.

In this chapter we will present a detailed analysis of the overhead introduced by Iguana/C++ and show how further optimisation techniques could be incorporated into Iguana.

### 6.1 Overhead, Where and Why

Run-time penalties incurred by reflection are in general difficult to assess and depend on a number of factors. We have identified four levels at which overhead can be incurred and that are consequently a suitable target for optimisation:

1. Design level: How does the Iguana model address performance and efficiency? Where is overhead incurred due to the model?

2. Implementation level: Is the implementation optimal?
3. Host-language level: Does the host language, in this case C++, affect the performance? What is the overhead of the various reification categories? How efficient can an operation at the base level be diverted and executed at the meta-level?
4. Application level: What is reified? How much use of reflective features is made by the application?

We will examine the impact on performance under these four aspects in the following sections.

### 6.1.1 Design Level

Performance has always been a major concern in the development of the Iguana model. The idea of a fine-grained decomposition of the C++ object model and the selective reification of language operations ideally only introduces overhead where the reflective features are explicitly used. Unfortunately, following this approach scalability becomes a more important issue: implementing language operations separately can lead to an explosion of meta-level information since every base-level object is represented by a number of metaobjects. In the revised design we addressed this issue by reducing the number of reification categories significantly, leading to a more intuitive and logical decomposition.

Another design issue that has been crucial to the Iguana model is the ability for objects to individually select a meta-type at run-time. The examples presented in the previous chapter have demonstrated that this flexibility is useful in that objects can select an implementation that best represents their environment. However, with flexibility comes complexity. Allowing dynamic meta-typing implies that some amount of the execution time is spent in determining the run-time meta-type of objects.

We believe that the advantage of having dynamic meta-typing justifies this additional overhead. Our implementation provides a fair compromise between flexibility and performance. The overhead introduced by run-time checks is relatively low and only affects those classes that are potentially reflective. Moreover, objects can still revert to the native and more efficient C++ mechanism if the reflective features are no longer required.

### 6.1.2 Implementation Level

Source-to-source translation as a means of incorporating the Iguana model into C++ has a couple of disadvantages as far as performance is concerned: the translated code is

high-level and sub-optimal because it has to conform to the C++ typing system. Having direct support for reflective programming by a compiler could lead to a more efficient implementation since we could provide a tighter coupling between the language extension and its translation into machine code. Moreover, since the C++ object model is not fully specified<sup>1</sup>, compiler internal knowledge about, for example, the size and layout of data objects could be exploited.

Take argument passing as an example: passing arguments to reflective functions via the reified stack is a relatively expensive operation because the size of user-defined data types is in general not known until run-time (the layout of data objects is compiler/platform dependent). Knowing the size of data objects can result in more efficient stack operations since we could either employ a byte-, word- or doubleword-wise copy of data objects. In the current implementation we have to look-up the size of the data object at run-time (using the C++ `sizeof`-operator) and then perform the copy operation<sup>2</sup>. If the size of a data object is known in advance, we could provide specialised stack operations without the need to invoking the `sizeof`-operator.

**Overhead of Reified Operations** Implementing reified language operations efficiently is another crucial issue. This section presents an evaluation of the Iguana/C++ default protocol that provides the default semantics of C++. The results are summarised in table 6.1.

Rows 2 and 3 of table 6.1 show the relative overhead in the creation of objects that have structural information reified, i.e. **Class**, **Method** and **Attribute**. In this scenario, objects are created with the C++ `new`-operator together with additional structural metaobjects that are bound to the object's meta-level. The number of metaobjects to be created depends on the number of methods and attributes in the class: each method/attribute is represented by one metaobject. Reifying structural information in a local mode is therefore linear dependent on the number of methods and attributes in the class. In that case, we carry out a deep copy of the object tree representing the object's class structure. When reifying structural information in shared mode, objects are bound to preexisting metaobjects and the overhead can be regarded as constant. In that case, we only have to carry out a shallow copy of the object's class structure.

Significant overhead is involved with the creation of objects, see rows 4 and 5 of table 6.1. Again, we compared the creation of objects in both local and shared mode. Object

<sup>1</sup>The ANSI specification for C++ does for example not specify how objects are laid out in memory.

<sup>2</sup>Carrying out the additional look-up has shown to be more efficient than always performing a byte-wise copy of data objects.



| Reification Categories           | rel. overhead |
|----------------------------------|---------------|
| 1. plain C++                     | 1             |
| 2. shared structural MOs         | 13            |
| 3. local structural MOs          | 50            |
| 4. Creation, shared              | 27            |
| 5. Creation, local               | 70            |
| 6. Creation+Deletion, shared     | 31            |
| 7. Creation+Deletion, local      | 74            |
| 8. Invocation (null method call) | 12            |
| Invocation, (int)                | 18            |
| Invocation, (int, double)        | 20            |
| 9. Invocation + Send             | 15            |
| 10. StateRead                    | 9             |
| 11. StateWrite                   | 22            |

Table 6.1: Measurements showing the relative overhead of reified operations in Iguana/C++.

creation is complicated because we first have to instantiate the full object graph that constitutes the future object’s meta-level. Similarly, the deletion of objects entails the deletion of the entire object’s (local) meta-level.

Row 8 shows the overhead of a reified method invocation with varying number of arguments. In this scenario, arguments are passed via the reified stack, which introduces an overhead that is linear dependent to the size and number of arguments.

In lines 10–11 the overhead of the behavioural reification categories `StateRead` and `StateWrite` is shown. In C++, a write operation only consists of a few machine instructions and can be carried very efficiently. In the reflective version, the write operation is comparable to a method invocation where the new value of the data member is passed as a parameter via the reified stack, hence the overhead is relatively high.

### 6.1.3 Host-Language Level

By applying the Iguana model to C++, we were also confronted with the complexity and feature-richness of the language. As has been described in section 4.7.2, meta-level code to carry out a simple read or write operation is complicated because we have to distinguish between simple attributes, arrays of either fixed or dynamic size and static attributes. The creation and deletion of objects is also inherently complex since we have to distinguish between the creation/deletion of single objects and dynamic arrays of objects. Each



semantic introduces interpretative overhead at the meta-level, resulting in less efficient reflective code. An alternative implementation would be to provide separate reification categories – and hence separate implementations – for the different scenarios. Our experience with the previous version of Iguana has shown that this only overcomplicates the programming model and also defeats language independence.

#### 6.1.4 Application Level

Although the overhead of a reified operation at a first glance seems to be high, applications are in general not fully encumbered with the costs of reflection. Obviously, the amount of time spent for meta-level computations depends on the set of active reification categories and to which extent the application makes use of the reflective features.

In this section we measured the reflective overhead of 5 simple benchmark applications. The example programs were taken from an existing benchmark suite (Bench++, [Oro98]). We selected algorithms from that suite that make heavy use of object-oriented features in C++, including

1. Permutations: Calculates the number of permutations of a set of numbers. Makes heavy use of array access.
2. Towers of Hanoi: Algorithm that solves the Towers of Hanoi puzzle, highly recursive.
3. Eight Queens: Algorithm that solves the Eight Queens puzzle, highly recursive and extensive use of array access.
4. Quicksort: Implementation of the Quicksort sorting algorithm, highly recursive and extensive use of array access.
5. Bubblesort: Implementation of the Bubblesort sorting algorithm. Makes heavy use of array access.

For each of the algorithms, eight measurements were conducted, each of which reifying a different set of language constructs, namely

1. non-reflective: Plain C++ implementation.
2.  $n/r + \text{run-time check}$ : This measures the overhead of applying the Iguana model without behavioural reflection, i.e. the costs of the run-time checks guarding method invocations and state access.

| Reification Categories        | <i>App. 1</i> | <i>App. 2</i> | <i>App. 3</i> | <i>App. 4</i> | <i>App. 5</i> |
|-------------------------------|---------------|---------------|---------------|---------------|---------------|
| 1) plain C++                  | 1             | 1             | 1             | 1             | 1             |
| 2) plain C++ & run-time check | 1.60          | 1.33/1.92     | 1.60          | 1.25          | 1.62          |
| 3) Invocation                 | 15.00         | 9.19/9.19     | 2.39          | 1.98          | 1.62          |
| 4) Invocation + Send          | 16.50         | 10.59/10.59   | 2.56          | 2.08          | 1.62          |
| 5) StateRead                  | 4.42          | 3.71/5.63     | 4.09          | 5.00          | 9.79          |
| 6) StateWrite                 | 8.50          | 5.66/8.86     | 8.76          | 3.79          | 6.79          |
| 7) StateRead+Write            | 11.65         | 8.13/12.57    | 11.14         | 7.60          | 14.72         |
| 8) Reify all                  | 26.51         | 17.57/22.09   | 11.92         | 8.40          | 14.72         |

Table 6.2: Measurements showing the relative overhead of Iguana/C++ for 5 benchmark applications.

3. Invocation reified: Measures the overhead of reified method invocations.
4. Invocation + Send: Measures the overhead of both reified method invocation and send. Objects can also send messages to themselves, i.e. invoke their own methods. Since most of the algorithms above are implemented in a recursive fashion, almost every invocation is interceded with by the send-metaobject.
5. StateRead: Measures the overhead of reified read access to objects.
6. StateWrite: Measures the overhead of reified write access to objects.
7. StateRead + StateWrite: Measures the overhead of both reified read and write access.
8. reify all: Measures the overhead of all of the above reification categories.

For this set of benchmark applications we did not measure the overhead of object creation/deletion since all of the algorithms above only create one single instance of each class that implements the algorithm. The costs of object creation and deletion are therefore negligible. All applications were compiled with gcc version 2.96 and were run on a 1GHz Pentium PC under the Linux 2.4.1 operating system. The results are shown in table 6.2.

## 6.2 Discussion

As one would expect, the overhead introduced by Iguana grows in proportion to the number of selected reification categories and depends on the extent of which an applications makes

use of a reified language feature. Application 5 for example, an implementation of the bubble-sort algorithm, does not make use of method invocation but instead relies heavily on array access.

Application 2, the towers of Hanoi, contained a helper class for which we reified the operations in a separate measurement. The performance figures for this application hence shows 2 values. The helper class only contained attributes and did not define any methods, therefore reifying method invocation did not introduce additional overhead.

It is worth mentioning again that in case a particular reification category is not selected, the native C++ mechanism is used together with a run-time check that enables objects to enable that reification category at a later time. For the various applications this overhead lies between 25 and 90%.

The default implementations for the various reification categories do not provide any extra functionality and are only meant as a common base for the meta-level programmer who wants to define own object semantics. In a real application extra functionality in terms of meta-types implementing persistence or distribution would further diminish the costs of reflection.

## 6.3 Other Optimisation Techniques

Suitable optimisation techniques for reflective programming languages have been identified by a number of researchers, including [Chi97] and [MMWY92]. The common approach of these is to carry out meta-level computation as early as possible, ideally during compile-time, and reduce the amount of meta-level computation carried out during run-time. For the remainder of this chapter we will examine optimisation techniques with respect to the Iguana/C++ implementation.

### 6.3.1 Partial Evaluation

Partial evaluation, or program specialisation, is an optimisation technique which, when given some part of a program's input data, generates a specialised or so called *residual* program ([JGS93], [Jon96]). The residual program with the remaining input produces the same result as the original program with the entire input. More formally, given a program  $P(x, y)$ , partial evaluation of  $P$  with respect to the input  $x$  will generate a specialised version  $P_x(y) = PE(P, x)$  such that  $P_x(y) = P(x, y)$ .

Parts of the program which solely depend on the static input data can be evaluated



at compile time and/or optimised for performance. Consider the following C-function `pow(x,y)`. If we know the value of the exponent `y` prior to execution, for example 3, partial evaluation can generate a residual program which only relies on the dynamic input. In the specialised version shown below, the value for `y` has been propagated through the function and the loop has been unrolled.

---

|   |   |
|---|---|
| Original program:   | Residual program:   |
| <pre><b>int</b> pow(<b>int</b> x, <b>int</b> y){   <b>int</b> i;   <b>int</b> res = 1;   <b>for</b> (i =0; i&lt;y; i++){     res *= x;   }   <b>return</b> res; }</pre> | <pre><b>int</b> pow_3(<b>int</b> x){   <b>int</b> res = 1;   res *= x;   res *= x;   res *= x;   <b>return</b> res; }</pre> |

---

Tempo [C+96] and C-Mix [And94] are examples of partial evaluators for C programs. In Tempo, specialisation is carried out in two stages (so called *off-line* partial evaluation): in a first stage, the source code is analysed to gather and propagate information about known and unknown values throughout the program. In a second stage, actual specialisation values or invariants are provided by the user and the residual program is automatically generated. In Tempo, the specialisation phase can be performed either at compile-time or at run-time. In the latter case, the invariants to parts of the program are provided at run-time and the specialised version is generated by dynamic code generation. C-Mix on the other hand only supports specialisation at compile-time.

Partial evaluation has been applied in a number of case studies to optimise reflective programming languages. However, due to the nature of meta-level programming, only a restricted use of partial evaluation is possible.

A major practical restriction is that partial evaluation is naturally more applicable to functional languages and has to our knowledge not yet been applied to object-oriented programming languages. In [MMAY95] for example the reflective application code written in ABCL/R is first translated to *continuation passing style functions* (CPS functions) and then partially evaluated. Translating from an object-oriented language to a functional language does not only require an additional translation phase but also entails the loss of semantic information that can prevent a partial evaluator from identifying static program



parts.

A second restriction arises due to the flexibility offered by meta-level architectures. If it is possible for objects to dynamically alter and change their meta-interpreters at run-time, partial evaluation cannot eliminate the interpretative overhead, simply because the meta-level representation cannot be regarded as being static. Hence, in [MMAY95] partial evaluation was applied as if the meta-level was statically fixed: objects are only allowed to select their meta-interpreter at creation time. In Iguana on the other hand with its emphasis on dynamic meta-type selection, we cannot regard the meta-level as being static and hence cannot in general collapse base and meta-level to be flat.

Despite practical and theoretical shortcomings of partial evaluation of reflective programming languages and in order to assess possible performance gains, we will simulate how partial evaluation could be applied to Iguana/C++.

### 6.3.2 Partial Evaluation of Iguana/C++

In this section we will examine a possible application of partial evaluation in Iguana/C++. As has been explained above, due to the dynamic nature of Iguana we cannot in general fully collapse base and meta-level code. However, we can perform a faster transition from base to meta-level computations based on partial evaluation if we assume an object's structural meta-level (i.e. its class definition) as being static.

Once again, consider the code generated to shift from base to meta-level computation in the case of a state write operation:

$$\text{obj} \rightarrow a = v; \quad \xrightarrow{\text{translated into}} \quad \text{obj} \rightarrow \text{write}(\text{index}_a, \text{Stack} \rightarrow \text{push}(v));$$

$\text{index}_a$  in the example above denotes the index of the attribute in the attribute table. The meta-level object which carries out the write-operation in general requires a reference to the attribute metaobject. We therefore have to lookup the attribute metaobject in the attribute table prior to performing the write operation. If we assume the object's structural meta-level as being static, we can consider the attribute metaobject as being fixed and apply partial evaluation to generate a specialised write operation as in:

$$\text{obj} \rightarrow \text{write}_a(\text{Stack} \rightarrow \text{push}(v));$$

In the specialised write operation we can cache a reference to the attribute metaobject and thus speed up the lookup. Similar specialisations can be applied to all other reification categories. Table 6.3 summarises the overhead of the behavioural reification categories

| Reification Categories        | unoptimised | optimised |
|-------------------------------|-------------|-----------|
| Invocation (null method call) | 12          | 11        |
| StateRead                     | 9           | 7         |
| StateWrite                    | 22          | 20        |

Table 6.3: Measurements showing the relative overhead of reified language operations, optimised version.

in the optimised version. With a hand-crafted, specialised implementation we gained a moderate speed up of reified language operations: 10% for state write, 8% for method invocation and 25% for state read.

### 6.3.3 Elimination of Run-Time Checks

Run-time checks guarding method invocations and state accesses are a further target for optimisation. Run-time checks are always inserted where it cannot be determined statically whether or not an object has a particular operation reified. Under circumstances, this affects code which does not make use of the reflective features at all. Analysing techniques such as data-flow and control-flow analysis could determine more accurately the set of active reification categories for a given object. This approach is used in various optimising compilers (for example Vortex [DDG<sup>+</sup>96]) that try to reduce the cost of method dispatching: if the dynamic type of an object can be determined, a virtual function call can be replaced with a static function call without the need to indirect via a function table.

## 6.4 Summary

In this chapter we provided a detailed analysis of the overhead of the reflective programming features in Iguana/C++. We identified a number of inefficiencies in the current implementation that are mainly due to the complexity of the host language and the lack of compiler internal knowledge. We also investigated the application of high-level optimisation techniques such as partial evaluation and program specialisation. Due to the dynamic nature of the Iguana model, these optimisation techniques would only achieve a moderate speed-up for reflective applications.

## Conclusion and Future Work

How nature loves the incomplete. She knows  
If she drew a conclusion it would finish her.

*Christopher Fry*

In a distributed, heterogeneous environment applications are required to provide a number of services such as persistence, distribution and fault-tolerance independently from the actual functionality of the application. Providing these services is still a labour intensive task and requires programmers to be aware of the non-functional requirements throughout the design and implementation phase of their applications. Current approaches in form of component based systems allow a more structured and rapid development cycle but still lead to a tangling of functional and non-functional code. Moreover, these systems are 'closed' in the sense that users do not have any control over the implementation of the various facilities and can in general not provide customised or additional services.

The main motivation for the work described in this thesis therefore was to provide generic language support for distributed computing that allows the development of distributed applications to be more open and independent from the functional specifications. Reflective programming languages have been identified as a promising approach in that they provide a structured approach to extending the semantics of existing languages and provide a clean separation of concerns.

The goal of this thesis was to provide a programming environment so that the advantages of component-based programming are made available for the developers of operating systems, embedded systems and legacy systems alike. We provided such an environment in the form of a meta-level architecture for a compiled, object-oriented programming language. We introduced the concept of a meta-type as a common abstraction for the reflective features.



As has been mentioned in section 1.5, the Iguana reflective programming model in its previous version suffered from a number of shortcomings that we aimed to overcome in the work described in this thesis. In the re-designed and re-implemented version we simplified the model while at the same time maintaining most of its flexibility. More specifically, we addressed

## Complexity

Open implementations are faced with the dilemma of which and how many internal details should be exposed to the user. Clearly, exposing too many details will lead to an overspecification of some component and is potentially more harmful than beneficial. The previous version of Iguana suffered from that problem in that it tried to reify every single language construct separately, leading to an overlap of the semantic of the various reification categories. In the revised model we reduced the number of reification categories significantly in order to achieve a more intuitive and logical decomposition of the underlying object model.

Meta-types hide much of the underlying complexity of meta-level programming. From the application programmer point of view, meta-types constitute components that can be linked into existing applications, similar to an ordinary class library. From the meta-level programmer's point of view, meta-types build a framework for developing extended object models. The Iguana run-time support takes care of complex issues such as the meta-level reconfiguration in the event of a dynamic meta-type selection and thereby relieves also the meta-level programmer from knowing the implementation details of the reflective architecture.

## Safety and Robustness

Opening up a language and allowing the programmer to intercede with the execution of language operations is a potentially dangerous task and requires a profound understanding of the language's intrinsic mechanisms.

By defining inheritance rules for meta-types and by defining a semantic for composing and selecting meta-types we now provide a safe and automated mechanism to migrate between different meta-level configurations. Consistency is ensured by a combination of sub-typing rules and run-time meta-type checking. Application programmers are no longer required to access meta-level configurations directly, a fact which made the previous version too complex, error prone and exposed too many implementation specific details to the programmer.



## Transparency

In principle, the meta-type of an object is orthogonal to both its type and class. Objects of different types and classes might have the same meta-type, while objects of the same type or class might have different meta-types. In our implementation type orthogonality is achieved by run-time checks that are inserted into the application code in case the dynamic meta-type of an object cannot be determined otherwise. This is a major improvement to the previous version where reflective and non-reflective objects could not be used interchangeably.

## Implementation

Providing a full reflective extension to a complex language such as C++ is a non-trivial task. Since C++ is a compiled language, only little structural information is kept in the run-time image. Adding reflection is hence dominated by the problem of maintaining the structural information beyond the compilation process.

We developed a concrete mapping of Iguana onto C++, implemented by means of a pre-processor. The semantic analysis during the pre-processing stage identifies those language constructs that are to be replaced with code that transfers control to the meta-level. We also provided the C++ default MOP, i.e. a concrete set of metaobjects implementing the C++ object model.

## 7.1 Understanding Reflective Programming

Meta-level programming, compared to object-oriented programming, is still a rather neglected programming paradigm and hasn't found its way into mainstream programming languages. With the additional functionality comes complexity, leading to the perception that meta-level programming is hard.

The concepts introduced in Iguana, namely protocol definitions, protocol inheritance and meta-types are therefore designed to resemble those in traditional object-oriented models and should therefore lead to a flatter learning curve and wider acceptance. In a number of concrete meta-type implementations we demonstrated how reflective programming can be done in a fashion similar to conventional, object-oriented analysis and design. The methodology is to provide a default behaviour for objects that is equivalent to the one provided by the host language and to gradually augment, combine and specialise object behaviour in subprotocols.

## 7.2 Performance and Optimisation

For reflection to become a successful programming paradigm, it has to be shown that the benefits gained by reflection justify the run-time overhead and that these costs are minimal compared to the functionality provided. By understanding where and why overhead occurs, profitable optimisation techniques can be developed.

We identified a number of inefficiencies in the current implementation that are mainly due to the lack of compiler internal knowledge and the complexity of the underlying C++ programming language. In our implementation, both reflective and non-reflective code coexists in the application that allows to switch between native and reflective objects. This proved to be a safe and efficient way to achieve a flexibility that to date can only be found in interpreted languages.

## 7.3 Future Work

### 7.3.1 Reflection and Design Patterns

From our experience with using a reflective programming language we have seen that reflection can be useful to implement certain types of applications, so, for example, those that support some sort of adaptability and software evolution. With a conventional approach, adaptation can be achieved by using a design pattern, such as the strategy pattern as described in section 5.3. The advantage of using reflection is that it can be used as a tool that automates the development of this kind of applications. Future work could investigate if and how other design patterns could be incorporated into existing applications using a reflective programming language.

### 7.3.2 Composition of Meta-Types

Composition of meta-types in practical terms is a whole area of research that still needs to be explored. Some of the questions to be asked are what the semantics of composed meta-types should be, whether and how different meta-type implementations interfere and how these conflicts can be resolved.

For example, say we have two meta-types implementing object persistence and remotely accessible objects. How should a derived meta-type that combines the two behaviours perform? Is there a semantic difference between having remote-persistent and persistent-remote objects? Understanding these issues will help to improve the design and implemen-

tation of meta-types, leading to more robust and modular meta-level code. A medium term goal is to develop a library of meta-types that can be selected by base-level programmers independently.

### 7.3.3 Compiler Support for Reflective Programming Languages

As has been noted before, implementing a language extension by means of a pre-processor has a couple of disadvantages as far as performance and ease of use is concerned. An additional pre-processing stage introduces further complexity and module dependencies. Debugging of reflective programs for example has been found to be difficult because the pre-processed application code is complex and obfuscated. The long term goal therefore must be to incorporate the Iguana model into existing compilers. Direct compiler support for a reflective programming language would not only significantly increase the ease and acceptance of reflective programming by making the additional pre-processing phase redundant, it would also lead to a more efficient implementation by allowing a tighter translation into machine code.

### 7.3.4 Formalisation of Meta-Types

The notion of meta-types as described in this thesis arose from the practical need for encapsulating most of the functionality provided by the reflective programming model and give the programmer a powerful construct to define extended object semantics. However, for meta-types to become a successful concept, a formal specification is required. It is our believe that once the formal foundations of meta-types have been established, a further understanding of complex issues such as subtyping and type-safety will emerge.

### 7.3.5 Applying Reflection

So far the use of reflection has been limited to a few experimental platforms. Building large-scale, long-running systems that take full advantage of the dynamic adaptation features is the next logical step. Applying reflection in the area of operating systems and middleware is therefore the focus of ongoing research.



# Bibliography

- [ACC82] M. P. Atkinson, K. J. Chrisholm, and W. P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(1):24–31, 1982.
- [And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [Ber96] Lodewijk Bergmans. Composability: Why, What, and How? In *ECOOP '96 Workshop on Composability Issues in Object Orientation*. 1996.
- [C<sup>+</sup>96] C. Consel et al. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Berlin: Springer-Verlag, 1996.
- [Cah99] Vinny Cahill. Tigger: A framework supporting distributed and persistent objects. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, *Implementing Application Frameworks: Object-oriented Frameworks at Work*, pages 485–519. Wiley, 1999.
- [Chi] Shigeru Chiba. Open C++: Its design and applications. Submitted to ECOOP '98.
- [Chi93] Shigeru Chiba. Open C++ release 1.2 programmer's guide. Technical Report TR 93-3, Department of Information Science, University of Tokyo, 1993.
- [Chi95] Shigeru Chiba. A metaobject protocol for C++. In Wilpolt [Wil95], pages 285–299. Also SIGPLAN Notices 30(10), October 1995.
- [Chi96] Shigeru Chiba. *A Study of Compile-time Metaobject Protocol*. PhD thesis, Graduate School of Science, University of Tokyo, November 1996.



- [Chi97] S. Chiba. Implementing techniques for efficient reflective languages. Technical report, Department of Information Science, University of Tokyo, June 1997.
- [Chi00] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP*, pages 313–336, 2000.
- [CM93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the 7<sup>th</sup> European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 482–501. Springer-Verlag, 1993.
- [CNTR97] Vinny Cahill, Paddy Nixon, Brendan Tangney, and Fethi Rabhi. Object Models for Distributed or Persistent Programming. *Computer Journal*, 40(8):513–527, 1997.
- [CST00a] Walter Cazzola, Andrea Sosio, and Francesco Tisato. Shifting Up Reflection from the Implementation to the Analysis Level. In Cazzola et al. [CST00b], pages 1–20.
- [CST00b] Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors. *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2000.
- [DDG<sup>+</sup>96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, pages 83–100, 1996.
- [Des37] René Descartes. Discourse on the Method of Rightly Conducting the Reason, and Seeking Truth in the Sciences, 1637.
- [DM95] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a Short Comparative Study. *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.
- [DSC<sup>+</sup>00] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation. In Cazzola et al. [CST00b], pages 171–190.
- [FDM94] Ira Forman, Scott Danforth, and Hari Madduri. Composition of before/after metaclasses in SOM. In Carrie Wilpolt, editor, *Proceedings of*

- the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 427–439. ACM Special Interest Group on Programming Languages, ACM Press, October 1994. Also SIGPLAN Notices 29(10), October 1994.
- [FJ89] Brian Foote and Ralph Johnson. Reflective facilities in Smalltalk-80. In Norman Meyrowitz, editor, *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages and Applications*. Association for Computing Machinery, ACM Press, October 1989. Also SIGPLAN Notices 24(10), October 1989.
- [FP98] Jean-Charles Fabre and Tanguy Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [Gow97] Brendan Gowing. *A Reflective Programming Model and Language for Dynamically Modifying Compiled Software*. PhD thesis, Department of Computer Science, University of Dublin, Trinity College, 1997.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.
- [Gro95] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report Version 2.0, Object Management Group, 1995.
- [HSC01] Peter Haraszt, Tilman Schäfer, and Vinny Cahill. *The Iguana Experience: Meta-Level Programming in a Compiled Reflective Language*. Workshop on Experience with Reflective Systems, held in conjunction with The 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, 2001. <http://www.openjit.org/reflection2001/workshop-papers/haraszti.pdf>.
- [Hut96] Norman C. Hutchinson. An emerald primer. Technical Report TR-96-??, University of British Columbia, September 1996.

- [Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7, 1985.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. In *ACM Transactions on Computer Systems*, volume 6, pages 109–133, 1988.
- [Jon96] Neil D. Jones. *An Introduction to Partial Evaluation*, volume 28. September 1996.
- [KCR99] Fabio Kon, Roy Campbell, and Manuel Roman. Design and Implementation of Runtime Reflection in Communication Middleware: the dynamictao case. In *ICDS'99 Workshop on Middleware*, 1999.
- [KFRGC98] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia, and S. Chiba. A metaobject protocol for fault-tolerant CORBA applications. In *Proceedings of the 17<sup>th</sup> Symposium on Reliable Distributed Systems*, pages 127–134, September 1998.
- [Kic91] Gregor Kiczales. Towards a new model of abstraction in software engineering. In Luis-Felipe Cabrera, Vincent Russo, and Marc Shapiro, editors, *Proceedings of the 1<sup>st</sup> International Workshop on Object-Oriented in Operating Systems*, pages 127–128. IEEE Computer Society, IEEE Computer Society Press, October 1991.
- [KL93] Gregor Kiczales and John Lamping. Operating systems: Why object-oriented? In Luis-Felipe Cabrera and Norman C. Hutchinson, editors, *Proceedings of the 3<sup>rd</sup> International Workshop on Object-Oriented in Operating Systems*, pages 25–30. IEEE Computer Society, IEEE Computer Society Press, December 1993.
- [KLL<sup>+</sup>97] Gregor Kiczales, John Lamping, Cristina Lopes, Chris Maeda, and Anurag Mendhekar. Open Implementation Guidelines. In *19<sup>th</sup> International Conference on Software Engineering (ICSE)*. ACM Press, 1997.
- [KLM<sup>+</sup>93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The need for customisable operating systems. In *Proceedings of the 4<sup>rd</sup> Workshop on Workstation Operating Systems*, pages 165–169. IEEE Computer Society, IEEE Computer Society Press, October 1993.



- [KP96] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. 1996.
- [KTW92] Gregor Kiczales, Marvin Theimer, and Brent Welch. A new model of abstraction for operating system design. In Luis-Felipe Cabrera and Eric Jul, editors, *Proceedings of the 2<sup>nd</sup> International Workshop on Object-Orientation in Operating Systems*, pages 346–349. IEEE Computer Society, IEEE Computer Society Press, September 1992.
- [KYK<sup>+</sup>99] Mangesh Kasbekar, Shalini Yajnik, Reinhard Klemm, Yennun Huang, and Chita Das. Issues in the Design of a Reflective Library for Checkpointing C++ Objects. In *Proceedings of the 18<sup>th</sup> Symposium on Reliable Distributed Systems*, 1999.
- [Lau94] Christine Lau. *Object-Oriented Programming Using SOM and DSOM*. Wiley, 1994.
- [Led97] Thomas Ledoux. Implementing proxy objects in a reflective ORB. In *Proceedings of the ECOOP '97 Workshop on CORBA: Implementation, Use and Evaluation*, June 1997.
- [LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pages 93–103, January 1991.
- [LYi195] Rodger Lea, Yasuhiko Yokote, and Jun ichiro Itoh. Adaptive operating system design using reflection. In *Proceedings of the 5<sup>th</sup> Workshop on Hot Topics in Operating Systems*, pages 95–100. IEEE Computer Society, IEEE Computer Society Press, May 1995.
- [Mae87] Pattie Maes. Concepts and experiments in computational reflection. In Norman Meyrowitz, editor, *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155. Association for Computing Machinery, ACM Press, October 1987. Also SIGPLAN Notices 22(12), December 1987.
- [MaPC95] Philippe Mulet and Jacques Malenfant and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In Wilpolt [Wil95], pages 316–330. Also SIGPLAN Notices 30(10), October 1995.
- [McA95a] Jeff McAffer. Coda User's guide (Draft). Retrieved from <ftp://camille.is.s.u-tokyo.ac.jp/pup/members/jeff/docs/codauser.ps.gz>, 1995.



- [McA95b] Jeff McAffer. *A Meta-Level Architecture For Prototyping Object Systems*. PhD thesis, The Graduate School of The University of Tokyo, 1995.
- [McA95c] Jeff McAffer. Meta-level programming with CodA. In Olthoff [Olt95], pages 190–214.
- [Mic99] Sun Microsystems. Java<sup>tm</sup> 2 SDK Documentation, version 1.3, 1999.
- [Mic01] Sun Microsystems. Enterprise Javabeans Technology, 2001. <http://java.sun.com/products/ejb/index.html>.
- [MKIC92] Peter Madany, Panos Kougiouris, Nayeem Islam, and Roy H. Campbell. Practical examples of reification and reflection in C++. In Akinori Yonezawa and Brian Smith, editors, *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 76–81, November 1992.
- [MMAY95] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent relective languages using partial evaluation. In Wilpolt [Wil95], pages 300–315. Also SIGPLAN Notices 30(10), October 1995.
- [MMWY92] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanbe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In Andreas Paepcke, editor, *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 127–144. ACM Special Interest Group on Programming Languages, ACM Press, October 1992. Also SIGPLAN Notices 27(10), October 1992.
- [NM94] Oscar Nierstrasz and Theo Dirk Meijer. Requirements for a Composition Language. In *Proceedings of the ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, volume LNCS 924, pages 147–161. Springer Verlag, 1994.
- [OB98] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Brazil, 1998.
- [OLL95] Douglas Orr, Jay Lepreau, and Jeffrey Law. Program Specialization Using the OMOS System. Technical Report UUCS-95-016, University of Utah, Salt Lake City, 1995.

- [Olt95] Walter Olthoff, editor. *Proceedings of the 9<sup>th</sup> European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1995.
- [Oro98] Joseph Orost. The Bench++ Benchmark Suite. *Dr. Dobb's Journal*, page 58, October 1998. Retrieved from [http://www.research.att.com/~orost/bench\\_plus\\_plus.html](http://www.research.att.com/~orost/bench_plus_plus.html).
- [Par96] Terence Parr. *Language Translation Using PCCTS & C++*. Automata Publishing Company, 1996.
- [Paw98] Renaud Pawlak. Internship Report: Metaobject Protocols For Distributed Programming, 1998. Available at [citeseer.nj.nec.com/pawlak98metaobject.html](http://citeseer.nj.nec.com/pawlak98metaobject.html).
- [PW91] L. Pinson and R. Wiener. *Object-Oriented Programming Techniques*. Addison-Wesley, 1991.
- [Ral98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of the 12<sup>th</sup> European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at <http://www.disi.unige.it/RMA2000>.
- [SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient Portable Persistent Store. In *5th International Workshop on Persistent Object Systems*, September 1992.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
- [SW95] Robert J. Stroud and Zhixue Wu. Using metaobject protocols to implement atomic data types. In Olthoff [Olt95], pages 168–189.
- [Tay93] P. Taylor. Transactions for Amadeus. Master's thesis, Department of Computer Science, Trinity College, Dublin, August 1993.
- [VCdP93] N. Harris V. Cahill, R. Balter and X. Rousset de Pina. *The Commandos Distributed Application Platform*. Springer Verlag, 1993.

- [Wil95] Carrie Wilpolt, editor. *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Special Interest Group on Programming Languages, ACM Press, October 1995. Also SIGPLAN Notices 30(10), October 1995.
- [WY88] Takuo Watanbe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In Norman Meyrowitz, editor, *Proceedings of the 1988 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 306–315. Association for Computing Machinery, ACM Press, September 1988. Also SIGPLAN Notices 23(11), November 1988.
- [YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In Norman Meyrowitz, editor, *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 258–268. Association for Computing Machinery, Association for Computing Machinery, September 1986. Also SIGPLAN Notices 21(11), November 1986.
- [YKL94] Yasuhiko Yokote, Gregor Kiczales, and John Lamping. Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing. In *Proceedings of the 6<sup>th</sup> SIGOPS European Workshop*, pages 39–44. ACM Special Interest Group on Operating Systems, September 1994.