Towards Automatic and Dynamic Meta-Object Protocol Composition in a Compiled, Reflective Programming Language

Péter Haraszti

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

October 2003

THESIS
8065

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.

[illegible]

Péter Haraszti

Dated: 31 October, 2003

Permission to Lend and/or Copy

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

[illegible]

Péter Haraszti

Dated: 31 October, 2003

Acknowledgements

I could never have finished this thesis without the help and support of many people.

First and foremost, thanks are due to my supervisor Vinny Cahill for his great support and endless patience over the long years while he was supervising me.

Thanks to Tilman Schäfer, Barry Redmond, and Jim Dowling of the Distributed Systems Group (DSG) for working with me on the Coyote project.

A special thanks has to go to my former and current employers Broadcom Eireann Research Ltd. and CAPE Technologies Ltd., respectively, for their generous funding of my Ph.D. studies and for providing me with a flexible work arrangement.

To my loving wife, Mónika; without her encouragement and sacrifice I would never have completed my research and submitted this thesis.

I am grateful to my CAPE colleagues and friends Declan O'Shanahan and Michael Slevin for proof reading the thesis and suggesting improvements in readability.

Last but not least, I am very grateful to my children Petra and Áron for their understanding during the immeasurable time, that I spent working on my thesis and not playing with them. Now, I shall have much more time to play with them!

Péter Haraszti

University of Dublin, Trinity College

October 2003

Summary

This thesis describes a reflective programming model that provides a solution to the problem of automatic and dynamic metaobject composition in a compiled reflective programming language, IGUANA/C++.

Meta-object protocols (MOPs) reify elements of an object-oriented programming language and are used to alter the behaviour of the language's default object model in order to transparently implement common non-functional concerns such as object state persistence, synchronisation, remote invocation, authentication, encryption, fault-tolerance (replication), and logging at the meta-level. The use of MOPs in this manner facilitates the separation of concerns.

In many cases, applications have to address multiple non-functional requirements simultaneously. Thus, these MOPs and their constituent metaobjects need to be combined. Common approaches to MOP composition are static (composition happens at design time, requiring the programmer's intervention), or dynamic (composition happens at run-time). A dynamic solution may compose independently developed MOPs automatically. A few solutions exist to compose MOPs automatically, but they do not address semantic overlap between MOPs and provide little or no practical real-life examples.

The thesis attacks the problem of reusing already existing MOPs and combining their metaobjects meaningfully such that, where possible, interference in semantics is detected and taken care of. The main contribution of the thesis is the design and implementation of the automatic and dynamic metaobject composition algorithm. We define a reflective programming model called IGUANA, which supports automatic and dynamic MOP composition. Using this model, the application programmer at the base level may select one or many

MOPs from a library, which contains implementations addressing the above non-functional requirements.

The key to the automatic composition mechanism, implemented by a default composer metaobject, is that separate MOPs have descriptors containing information on the composability of the MOP. The descriptors define the non-functional concern area[1] in which the MOP is intended to be used (exclusively or in a shared manner), specify metaobject ordering requirements and constraints, indicate the explicit links between behavioural metaobjects and middleware components, and list base level modifications that constituent metaobjects in each behavioural reification category perform. The composer metaobject uses these MOP descriptors to automatically compose MOPs. In case the default composer metaobject cannot be used, the programmer can supply his/her own composer metaobject.

The thesis also defines a methodology for writing composable MOPs. The methodology is based on the distinction between local and remote-related MOPs, where remote-related MOPs conform to the proxy/server model. Metaobjects of remote-related MOPs are composed in such a way such that metaobjects in the client (server proxy) and server address spaces form a symmetric "protocol stack".

The new reflective programming model is derived from and retains many features of previous versions of the IGUANA model, but adds major new features such as a reified stack, that works in multi-threaded applications, support for dynamic multiple MOPs selection, extensible object references for reflective objects, an explicit middleware layer, which supports common MOP implementations, and a new composer metaobject with a default implementation that attempts to automatically combine behavioural metaobjects of several MOPs based on their semantic descriptors.

Similarly to its predecessors, the new version of IGUANA/C++ is implemented by a preprocessor, which parses the reflective C++ code and XML MOP descriptors and generates standard C++ code, which can be compiled using a standard C++ compiler.

We evaluate our model and implementation by designing, implementing, and dynamically combining a suite of MOPs that address some of the most common non-functional concerns

[1]We refer to these areas of concerns as "concern areas" in the text.

that can be found in real-life applications and use a common middleware. Our implementation provides a platform for the exploration of reflective software composition in the context of dynamic systems.

Publications Related to this Ph.D.

[1] 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 Walter Cazzola, Robert J. Srtoud, and Francesco Tisato, editors, Reflection and Software Engineering, volume 1826 of Lecture Notes in Computer Science, pages 171-190. Springer-Verlag, Heidelberg, June 2000.

[2] Peter Haraszti, Tilman Schäfer, Jim Dowling and Vinny Cahill: "The Iguana Experience: Meta-Level Programming in a Compiled Relfective Language", Presentation, Workshop on Experience with Reflective Systems, Reflection 2001 - The Third International Conference on Meta-Level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"The use of anthropomorphic terminology when dealing with computer systems is a symptom of professional immaturity." - Edsger Dijkstra

1.1 Preface

The complexity of system-level applications has increased many times over the last decade and is constantly increasing as new and higher-level requirements are being defined for them. A few years ago, application requirements such as logging (for traceability purposes or for debugging), distribution, state persistence, security, and fault-tolerance were specified only for a small set of highly specialised applications, examples being industrial process control and banking systems. The widespread, global use of the Internet and distributed computing has dramatically increased the need for security and availability. Application developers nowadays face a difficult challenge when meeting these ever-increasing requirements for flexible distributed deployment, security, persistence, fault-tolerance, and so on, which have become standard.

With the advent of object-oriented programming (OOP) technologies and application frameworks, support for these requirements, usually implemented as a hierarchy of application classes, has become more available for use by application programmers. The use of object-oriented frameworks has helped application programmers to focus more on their application's primary business logic, while using the appropriate (often specialised) components of the framework for implementing non-functional requirements. However, in order to meet new unanticipated and dynamically changing requirements, the framework may need to be extended and/or dynamically adapted at run-time.

The problem with frameworks is that, in most cases, the application programmer has to know the details of how different components of the framework implement their functionality and how they interact with each other, as well as what the required steps are to change the framework dynamically in order to address the changes in the environment. Also it requires a significant code refactoring effort to fit an existing (legacy) application into a new framework. Furthermore, a framework for persistence may not work well (or at all) with a framework for distribution.

The component-based architecture from Sun known as Enterprise Java Beans (EJB) [72] is widely considered as a move in the right direction when it comes to tackling application complexity, as it relieves the burden on application developers by providing a container for EJBs that implements and manages commonly found non-functional concerns such as persistence (in the case of Container Managed Persistence or CMP), distribution, security, and transaction management. Containers in application servers use structural reflection to find methods and members of EJBs. Containers also use intercession in order to carry out additional tasks such as transaction handling, authentication, encryption/decryption, and resource pooling, for example. This model insulates EJB developers from the details of the implementation of these non-functional concerns. Deployment descriptors allow convenient deployment of the same EJB in a container under different conditions. However, the implementation of the non-functional concerns is hidden from the EJB programmer and there is no control given to change/adapt these concerns in a running application. Microsoft's Component Object Model+ (COM+) [52] and OMG CORBA Components [55] support similar container-based models.

Computational reflection and meta-level architectures [67, 47, 46] on the other hand have been gaining popularity and are often used because of the ability to separate functional and non-functional concerns. The application functionality (i.e., its business logic) is implemented by base level objects, while meta-object protocols (MOPs) implement these non-functional concerns.

One of the benefits of using reflection is that once an experienced programmer has implemented and tested a recurring non-functional concern such as persistence of object state in the form of a MOP, it can easily be utilised by relatively inexperienced programmers in many applications. If an application has to persist its state between invocations, the base level or application programmer selects the MOP, that implements persistence for classes or objects that need it. Since meta-level programming is still regarded as a complex and difficult task [78], while using MOPs in base level applications is not as complicated, this distinction between the two programming roles in terms of programming skills required is very important. State persistence, fault-tolerance, and distribution (i.e., remote method invocations) for example can all be implemented as separate MOPs, with each MOP having its own set of metaobject (MO) classes. Base level and meta-level are causally connected through the process of MOP selection.

In a reflective programming model with multiple, fine-grained MOPs (e.g., IGUANA version 1 [30, 28] and version 2 [64]), each MOP consists of a set of structural and behavioural MO classes, which reify certain elements (e.g., method invocation or state read) of the object model of a typical object-oriented programming language such as C++ [70] or Java [27], and interact with each other in order to provide an implementation for the desired, alternative object model.

For system-level applications, that have to meet multiple non-functional requirements, the base level programmer should simply be able to select the corresponding MOPs[1] that implement them. Also, they should be able to deselect and reselect MOPs dynamically, at run-time, in order to adapt to changes in the requirements. This implies that these MOPs and their constituent MOs need to be combined.

The two common approaches to MO composition are static and dynamic. The static approach requires that the programmer manually combines MOPs when designing the application. Static composition is not useful for applications that have to respond (i.e., dynamically

[1]In our model, we consider the MOP as the unit of composition, although this is not a restriction.

adapt) to unanticipated changes in the operating environment. With the dynamic, run-time approach, the programmer relies on the meta-level architecture to meaningfully combine the MOs of the MOPs automatically, i.e., without prior knowledge of the set of MOPs to be combined. However, automatic and dynamic MOP composition in a compiled language is a difficult task.

1.2 The Problems of Metaobject Composition

A number of problems arises when one attempts to combine MOPs implementing different non-functional concerns:

1. Except in the simplest cases, such as, logging and persistence, non-functional concerns and hence MOPs may overlap semantically. For example, a synchronisation MOP overlaps semantically with a MOP for object persistence, because the former provides atomic read/write access to persistent objects, provided by the latter.

2. Different MOPs follow different implementation models or guidelines. For example, MOs of one MOP may be shared between instances of base-level objects, while in another MOP, each base-level object may have its own (local) set of MOs. A methodology for writing composable MOPs needs to be defined.

3. MOPs may not have been written with future composition in mind. This means that MO classes of a MOP may not participate in the composition model, which is most commonly the chain of responsibility model. In this model, MOs responsible for reifying a base level operation are ordered and inserted into a chain. The base level operation in question is intercepted at the meta-level, which activates the first MO in the chain. Each MO carries out its own *before* operation and is then responsible for calling the next MO in the chain. The last MO is supposed to reflect the operation at the base level and then return. This is followed by the execution of the *after* operations in a reverse order to that of the *before* operations. This chain may be broken by uncooperative MOs, that simply do not call the next MO in the chain.

4. A meta-level programmer may need the source code of a MOP, if he/she wants to combine it with another MOP or evolve it. However, as is often the case with system-level software, the source code of a MOP may not be available for other programmers to study and extend.

5. In many cases, MO classes of a MOP rely on standard or proprietary middleware components (e.g., the Common Object Request Broker Architecture (CORBA) [41] from the Object Management Group (OMG) [2] for object distribution or a Relational Database Management System (RDBMS) for object persistence). An example of using proprietary middleware can be found in [22]. However the relationship between MO classes and middleware components is often blurred. This can lead to difficulties when combining MOPs because it may require combining or reconfiguring middleware components, unless the MOPs are unrelated and they use non-overlapping sets of middleware components. The relationships between MOPs and middleware components are hidden in the implementation.

The next section gives a MOP composition example which demonstrates some of these problems.

1.2.1 An example of composing MOPs

Let us imagine that a programmer wants to implement secure remote method invocations, combined with the logging of events, whenever such secure and remote methods are invoked. Remote method invocations are carried out between client and server, that reside in different address spaces. By security in this example we mean that messages exchanged between client and server are authenticated.

There are four MOPs (Remote, RemoteProxy, Authentication, and Logging, respectively) needed to intercept method invocations at the client and server sides and carry out their additional computation before and/or after the actual method has been invoked on the server. For this reason, the above mentioned four MOPs reify method invocation (i.e., the Invocation

[2]CORBA is a registered trademark of OMG, see http://www.omg.org

reification category in IGUANA) and declare three MO classes: RemoteInvocation[3], AuthenticationInvocation, and LogInvocation, respectively. For a detailed description of these MOPs, see Chapter 4.

Let us take an implementation of these MOPs, which uses proxy objects at the client side: i.e., a proxy object represents a remote server object in the client object's address space. Note that care must be taken when Remote is involved in a MOP composition: a symmetric communication protocol stack must be formed between the proxy and the server, otherwise, similarly to a misaligned network protocol stack, communication would not be possible.

Ideally, the base level programmer would just select these MOPs for his/her application classes and objects, and the language run-time would automatically take care of composing the invocation MOs at the client and server sides such that it yields a system with the desired combined functionality. Assuming that the chain of responsibility model is used, we show why the ordering of MOs is important. We also show how the MO classes interact with middleware components.

First, let the order of the invocation MOs be: LogInvocation, AuthenticationInvocation, RemoteInvocation, and DefaultInvocation at the client side (see Figure 1.1). Upon invoking a method on the proxy object in the client, the meta-level intercepts this call and activates the first MO in the chain. The LogInvocation MO creates a temporary data record with the name of the operation being invoked, the identities of the client and server objects, and the current time. Then it passes control to the next MO AuthenticationInvocation, which creates the client's digital signature for authentication. The next MO in the chain is RemoteInvocation, which finds out whether the server object is remote or local. If the server is remote, RemoteInvocation MO marshalls the argument(s) together with the client's digital signature, and activates the communication middleware, which sends the method invocation request over the network to the server object. The last MO in the chain is DefaultInvocation by convention, but it will not be called for remote servers.

In order to make the server side interwork with the client side, the order of MOs must be the reverse to that of the client side: RemoteInvocation, AuthenticationInvocation, LogInvocation,

[3]The same MO class is used by both the Remote and RemoteProxy protocols

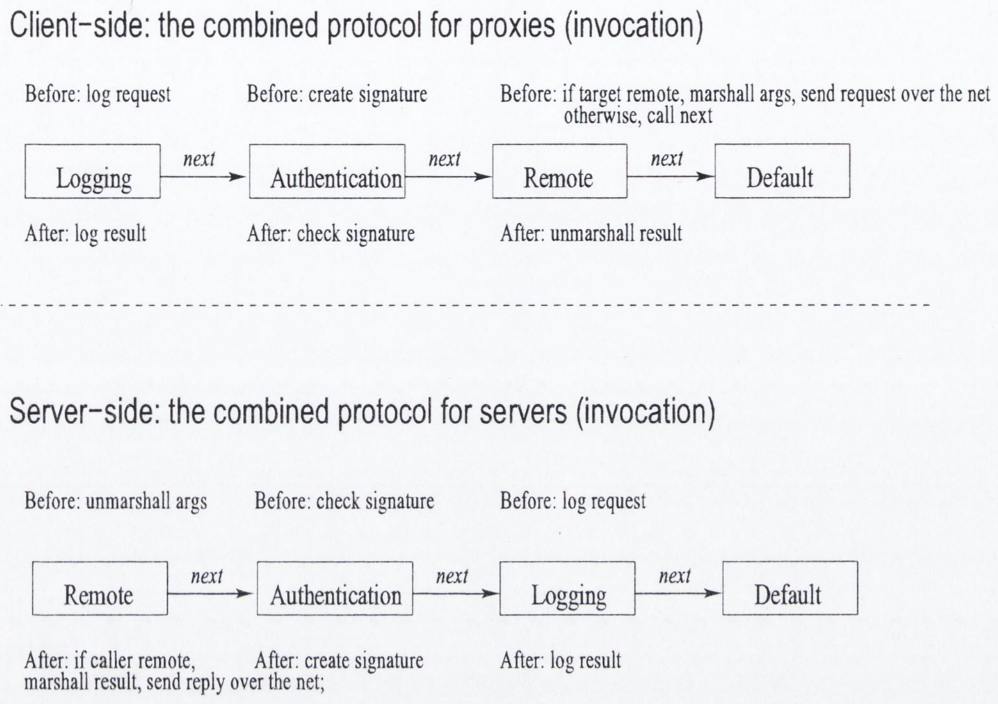

Fig. 1.1: Example of combining the invocation MOs of Logging, Authentication, and Remote MOPs at client and server sides.

and DefaultInvocation (see Figure 1.1).

At the server side, the communication middleware receives the remote method invocation request from the client, finds the server object, and passes control to the RemoteInvocation MO bound to the server. The RemoteInvocation MO unmarshalls the method arguments and the digital signature, and it calls the next MO in the chain. Thus, AuthenticationInvocation MO authenticates the remote caller object by checking its object identifier and digital signature with the help of an authentication server (part of the middleware) and, upon a successful authentication, it passes control to next MO, which is LogInvocation. The LogInvocation MO creates a log entry and records in it the name of the operation being invoked, the identities of the client and server objects, and the current time. It passes control to the last MO in the chain, which is DefaultInvocation (as recommmended in [64]). This MO reflects the base level operation.

After the base level method has been effectively executed, the LogInvocation MO records the result of the operation and writes the new log entry to the application's log. Control goes back through AuthenticationInvocation to the RemoteInvocation MO, which marshalls the return value of the method executed and returns control to the communication middleware, which in turn sends the result back to the remote client side.

When the reply arrives at the client side, the middleware passes control to the RemoteInvocation MO, which unmarshalls the result, and passes control back to AuthenticationInvocation, which in turn passes control to LogInvocation. LogInvocation logs the event at the client side and returns to the client.

The basic assumption throughout this example was that either the meta-level or a middleware component triggers the activation of the first MO in the chain: i.e., either a local client invokes a method on the proxy object, or a method invocation request arrives from a remote client over the network.

Now let us change the order of MOs at the server side from RemoteInvocation, AuthenticationInvocation, and LogInvocation to RemoteInvocation, LogInvocation, and AuthenticationInvocation. This implies that the order of MOs at the client side is the opposite, i.e., AuthenticationInvocation, LogInvocation, and RemoteInvocation. This MO combination would possibly log method arguments together with the client's digital signature and return values, which may not be of any use.

However, if we reorder the MOs by putting AuthenticationInvocation MO first at the server side, followed by RemoteInvocation and LogInvocation MOs, then we would break the composition because the AuthenticationInvocation MO cannot work until the RemoteInvocation MO has unmarshalled the method arguments.

This simple example shows that the order, in which the behavioural MOs are executed, is *vital* for correclty implementing any combined behaviour at both client and server sides. Different orderings may result in completely different and/or undesired behaviour. Our goal is to come up with a MO combination mechanism, that can calculate the "correct" order of MOs for a MOP combination at run-time, if possible.

1.3 Previous Work on Meta-Object Composition

A few reflective programming models (e.g., the IGUANA version 1 [28] and the model in [53]) support multiple MOP selection, which means that the base level programmer can select multiple MOPs for the base level classes or objects that comprise his/her application.

Other reflective models (e.g., Open C++ version 1 [10] and 2 [9] as well as IGUANA version 2 [64]) restrict the programmer to selecting only one MOP at any time for base level classes or objects. In this case, the single MOP has to combine and implement all of the selected non-functional concerns by means of multiple MOP inheritance: two or more MOPs, each implementing a certain non-functional concern, are manually or semi-automatically combined in the derived MOP. By manual combination, we mean that the meta-level prorgrammer has to write code (i.e., that of the MO classes) for the new, combined MOP, while in the semi-automatic combination case the meta-level programmer writes the MOP declaration only; the code from overlapping MO classes are combined at run-time, according to a fixed, pre-defined MO combination algorithm. In both cases, conflicts arising from the overlap must be resolved by the meta-level programmer.

Both IGUANA models support dynamic MOP re-selection, that is, the MOP associated with base level objects can be changed at run-time. In both models, easy composition of non-functional concerns, either at design-time or at run-time, needed to be properly investigated. Ideally, independently developed MOPs and their constituent MOs should automatically be composable, without requiring any additional effort (e.g., coding or code refactoring) from the programmer.

Some reflective programming languages support manual and/or automatic MOP composition. For example, the manual composition in Guaraná [57] requires the application programmer to write code (i.e., the composer metaobject) that explicitly combines the MOs of the selected MOPs. MOs that are being composed can be composers themselves. Although this approach is simple and performance efficient, it is inflexible as it requires rewriting the composer MO for each possible combination.

Reflective programming systems such as the Common Lisp Object System (CLOS) [36] MOP [2, 37], and IBM's System Object Model (SOM) with Before and After Meta-classes

[16, 25, 24] support some form of automatic composition. The CLOS MOP is a description of the CLOS system itself as an extensible CLOS program. Fundamental elements of CLOS such as classes, slot definitions, methods, generic functions, and method combination are reified as first-class entities and available to programmers as MOs. The standard method combination (implemented by the standard-method-combination) combines the primary, around, before, and after methods for each generic funtion invocation in order to form a single effective method. Although around, before, and after methods are not meta-methods, the concept has been adopted by many MOPs, including SOM and IGUANA. At run-time, SOM automatically determines an appropriate meta-class for an instance of a class, even if the class inherited from multiple parent classes and the parent classes specified different meta-classes. However the order of meta-classes depends on the class precedence list. Neither CLOS nor SOM deals with semantic overlap between meta-classes.

Many of the reflective programming languages (e.g., CLOS, IGUANA) use a variant of the simple chain of responsibility model, explained above. The chain of responsibility model for composition works well if the MOPs are orthogonal (i.e., their functionality is unrelated). But in practice, orthogonality is only an exception [53]. Even worse, the chain might be broken by some non-co-operative MO: e.g., a MO may conditionally break the chain by not passing control to the next MO.

It is clear that in most cases, automatic composition brings up problems with semantic interference between MOs. In this thesis we show that the interference can be detected and handled by describing the MOs semantics.

1.4 IGUANA version 1 and 2

In summary, IGUANA version 1 supports fine-grain, run-time MOPs, where a MOP can reify up to 29 aspects of the host object-oriented programming language, C++ for example. The programmer can select multiple MOPs for base-level classes and their instances. Through an explicit (i.e., using the meta pointer from base-level objects) or an implicit (i.e., using IGUANA *extension protocols*) mechanism, the base-level programmer can dynamically change/restructure the meta-level. Rudimentary support for combining MOs is provided in the form of the *chain of responsibility* model.

IGUANA version 2 has evolved the previous model by reducing the number of reification categories to 12, streamlining the run-time meta-level structure, and by introducing the concept of *meta-typing*, which constrains the base-level programmer such that he/she can select one single MOP for a base-level class or object. The single MOP selected for the object represents its meta-type. This meta-type can be dynamically changed (subject to certain conditions regarding the relationship between the old and the new meta-types), and the IGUANA run-time system automatically restructures the meta-level and combines behavioural MOs in the affected reification categories, following the same chain of responsibility model.

1.5 Our Approach

The IGUANA version 2 reflective programming model and its implementation addressed automatic MOP composition in a very limited way:

Single MOP selection: the base level programmer can select at most one MOP at a time for his or her classes or instances of classes. This implies that the selected single MOP must implement all of the desired non-functional concerns, for example, both object persistence and distribution. This can be done at design time only by means of multiple MOP inheritance. For example, the MOP PersistentRemote extends both Persistent and Remote MOPs. Meta-object classes in PersistentRemote are derived from the ones in Persistent and Remote, in that particular order. Should we need to combine a larger number of MOPs, we would need to write a new MOP for each desired combination of them. This is a design-time solution to MOP composition and as more and more non-functional concerns will be implemented as MOPs over time, it will lead to an exponential "explosion" in the number of MOPs.

Multiple MOP inheritance: IGUANA version 2 allows the use of multiple MOP inheritance. When the derived (metaobject) protocol's super-protocols reify the same language concept, the conflicting MOs are organised by default into a list (one list for each behavioural reification category), following the chain of responsibility model. The order of the MOs in the list is fixed and derived from the MOP precedence list [4]. The only simple way to change the order of MOs in the list is to change the MOP precedence list, i.e., by re-ordering the list of super-protocols. Even in this case, the new order may not be correct in all of the reification categories. For example, MOs for the Send and Receive reification categories in certain "remote"-related[5] MOs may have to be in reverse order of each other in order to form a symmetrical communication protocol stack between the remote client and server.

Ordering of the MOs: MOs reifying the same language concept are ordered according to the calculated MOP precedence list. If this order does not work for a particular new MOP, the IGUANA version 2 model allows the meta-level programmer to explore the MOs on the list in terms of next MO references. This solution works for single MOP selection only, where it is known at design time what MO the next references will refer to. With dynamic MOP selection, the selection or deselection of MOPs will result in a meta-level reconfiguration (i.e., MOs are inserted or removed). A meta-level reconfiguration can easily invalidate the next references, for instance they may refer to a wrong or null MO.

This thesis defines a new IGUANA model - we call it version 3 - supporting a flexible, automatic and dynamic MOP composition, that overcomes the above limitations.

The following design objectives are addressed in the new model:

Evolution: the new IGUANA model should address the problems and limitations of the previous two versions, instead of defining a radically new model.

Simplicity: the new IGUANA model should be intuitive to use for both base level and meta-level programmers.

Reusability: the new IGUANA model should facilitate MOP reusability through its framework for dynamic and automated MOP composition.

Our approach to automatic and dynamic MOP combination is based on the fact that the MOPs and their constituent MO classes are better specified (e.g., by clear separation of

[4]MOP precedence list is the MOP inheritance tree flattened out by a left pre-order walk.
[5]MOPs that are related to object distribution.

around, *before*, and *after* meta-computation in behavioural MOs and by the categorisation of MOs with respect to their relationships with each other and with the causally connected base level object), so that the MOP composer, which is itself a MO, can calculate the appropriate ordering of MOs in different reification categories at run-time, with certain limitations.

We also define a methodology for developing composable MOPs and MO classes by specifying simple design rules that meta-level programmers can and should follow. For example, only one MO in the chain should reflect the base level operation the way it is done by the default object behaviour (in IGUANA version 3, this MO is an instance of a MO class called e.g., DefaultCreation for creating a new instance of a reflective class, and should be the last MO in the chain).

The characterisation (description) of the MOPs and their MO classes with respect to their composability with other MOPs and their MO classes is key in solving the automatic and dynamic MOP composition problem. In IGUANA version 3, this information is captured as a set of MOP descriptor files, written in the eXtensible Markup Language (XML). We devise a Document Type Definition (DTD) for specifying the semantics of IGUANA MOP descriptors.

Moreover, in many cases MOs are inherently linked to components of the middleware that actually implement the desired non-functional behaviour. In the above example, RemoteInvocation uses distributed object middleware to communicate remote method invocation requests and replies over the network. Therefore we also express the MOPs and their links to components of standard or proprietary middleware in the MOP descriptor, which guides the automatic MOP composition and middleware (re)configuration.

Figure 1.2 shows the new proposed IGUANA model, which adds a second interface to the model to deal with the connections between MOs of IGUANA MOPs and the supporting middleware components. The new model also refines the existing interface (i.e., the first interface) between the base-level and the meta-level. See section 3 for a more detailed description of the new model and interfaces.

As our research interest is system-level programming, we implement IGUANA version 3 by extending a compiled object-oriented programming language, C++. Similarly to its predecessors, we implement our IGUANA/C++ as a preprocessor, which parses the reflective

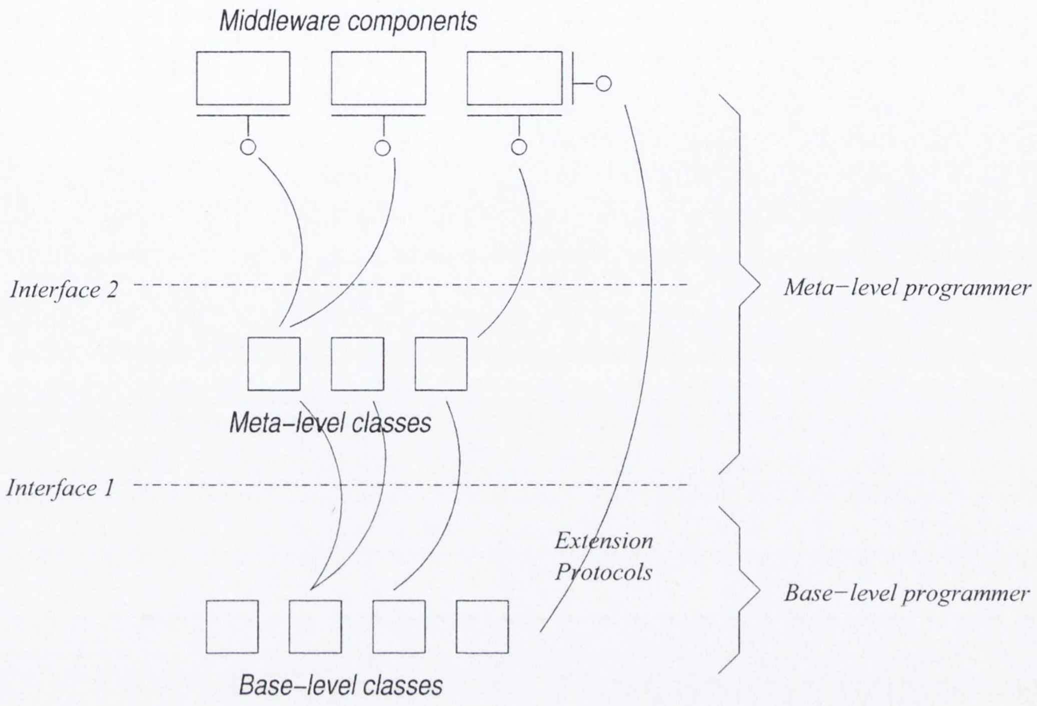

Fig. 1.2: The new IGUANA model with three layers

IGUANA/C++ code and XML-based MOP descriptors and generates standard C++ code, which is compiled using a standard C++ compiler.

Finally, we evaluate our thesis by designing, implementing, and dynamically combining a suite of MOPs that address the most common non-functional concerns and use a common proprietary middleware. Our implementation provides a platform for exploring dynamic software composition through reflection in a compiled proramming language, C++.

1.6 Roadmap to this Thesis

After this introduction to the problem of MOP composition and the outline of our approach to solving it, we review a number of reflective programming languages, meta-level architectures, and techniques to compose software components dynamically in Chapter 2. Within this Chapter, we describe IGUANA version 1 and 2 in more detail as our new model derives from them. In Chapter 3, we describe the design of IGUANA version 3, primarily focusing on the new extension that solves the automatic and dynamic MOP composition problem. We also define a methodology for writing composable MOPs and MO classes. Chapter 4 outlines the design of a number of automatically composable IGUANA MOPs that address the most common non-functional concerns. Chapter 5 describes the implementation of the IGUANA model for C++, called IGUANA/C++. Chapter 6 evaluates the model and composition mechanism through examples of combining the some of above MOPs by using the default and a customised Composer MO. Chapter 7 summarises the work presented in this thesis, draws conclusions, and specifies areas for future research.

Chapter 2

Related Work

"Whatever you do will be insignificant, but it is very important that you do it." - Mahatma Gandhi

2.1 Introduction

A number of reflective programming languages and extensions has been defined that address some form of metaobject (MO) composition. This chapter provides a brief overview of them and evaluates them according to the following criteria:

1. Compile-time or run-time MOP: when is the customisation of the reflective base-level program performed?

2. Fine-grain or coarse-grain MOP: what aspects of the language can be reified?

3. Single or multiple MOP selection: can multiple MOPs representing different behaviours be selected?

4. Support for manual or automatic MOP composition: are the MOs of MOPs combined by the programmer? Or are they combined by a composer object at run-time?

5. Static or dynamic composition: if multiple MOPs can be selected, is it possible to change them dynamically, at run-time?

6. Semantic aid for automatic composition: how does the composition deal with semantic overlap between MOPs?

7. Links between the meta-level and the middleware: in many cases, MOs of MOPs use a middleware to provide their intended functionality. What is the relationship between them?

This chapter also describes influentual reflective and non-reflective systems, that address composition in some form or are relevant to our thesis.

2.2 The Common Lisp Object System Meta-object Protocol

The Common Lisp Object System (CLOS) defines an object-oriented Lisp programming system. The CLOS Meta-object Protocol (MOP) [37] has been added to allow customisation of the CLOS system.

The CLOS MOP is a description of the CLOS system itself as an extensible CLOS program. Fundamental elements of the CLOS language (e.g., classes, slot definitions, methods, generic functions, and method combiners) are reified and available to programmers as first-class objects, called metaobject (MO) classes. The basic MO classes are thus: class, slot-definition, generic-function, method, and method-combination. A MO class is a subclass of exactly one of these classes. A MO is an instance of a MO class.

The behaviour of CLOS is implemented by these MOs. Each MO represents one program element and has information associated with it, which is required to serve its role. The MOs are interconnected. This interconnection means that the role of a MO is always based on the role of other MOs.

The essential features of CLOS are *classes* that define state and functionality and can inherit from one or more classes; *instances of classes* that are created, initialised, and used in programs; *generic functions*, whose behaviour depends on the classes of arguments supplied to them; and *methods* that specialise generic functions for class-specific behaviour. The programmer can qualify methods as *before*, *after*, or *around*: *before* and *after* methods are executed before and after the primary method execution takes place, respectively, while *around* methods are executed around *before*, *primary*, and *after* methods. A method with no qualifier is considered as *primary*.

In the remainder of the section we focus on the method combination aspect of the CLOS MOP. CLOS divides generic function invocation into three parts: determining the applicable methods (dictated by the method qualifiers such as *before*, *after*, and *around* and the classes of arguments); sorting the applicable methods into decreasing precedence order; and sequencing the execution of the sorted list of the applicable methods. This is often called the *effective method lookup*. CLOS has a standard method combination mechanism, which combines the *around*, *before*, *primary*, and *after* methods for each generic function invocation in order to form the single effective method.

The generic function invocation uses the *class precedence list*[1] for sorting the applicable methods, and it works as follows: the most specific *around* method is invoked first, which should call the call-next-method CLOS function to invoke the next most specific *around* method and so on. When there are no more *around* methods, the call-next-method runs the combination of the remaining *before*, *primary*, and *after* methods: the applicable *before* methods are executed first, from most specific to least specific. The most specific applicable *primary* method is executed next, followed by the applicable *after* methods, from least specific to most specific. The value(s) returned by the call to the generic function are the value(s) returned by the *primary* method. *Primary* methods may use the call-next-method if they want to invoke the next most applicable *primary* method. *Around* methods should always call the call-next-method from within the body of a user-defined method. *Primary* methods may call call-next-method. Finally, *before* and *after* methods must not call it.

The full CLOS MOP provides a mechanism for the user to replace the standard method combiner with a customised one.

Note that the *before*, *after*, and *around* methods are not meta-methods as such, i.e., they are base-level methods defined for application classes. However the concept of defining *before* and *after* operations at the meta-level has been adopted in and used by many reflective

[1]The class precedence list contains all of the direct or indirect superclasses of the class. The precedence list is calculated for each class such that it must satisfy the following two constraints: 1. a class precedes its superclasses; 2. superclasses have their order given in class definitions.

programming systems.

2.2.1 Evaluation and relevance to this thesis

The CLOS MOP can be considered as a fine-grain run-time MOP. MOP selection in CLOS is implicit: there is only one MOP which is the collection of the above MOs. Each MO class is a subclass of exactly one base MO class.
The effective method lookup mechanism and the explicit call-next-method has influenced the design of IGUANA version 2.

2.3 Open C++ version 1 and 2

Open C++ version 1 [10, 8], a reflective extension to the C++ language, supports run-time behavioural reflection. The Open C++ version 1 MOP allows the behaviour of method invocation, state read/write, and object creation to be reified and thus altered. The Open C++ MOP is defined by the MetaObj class. The methods defined in the MetaObj class implement the default C++ mechanisms for method invocation, state access, and object creation. This default behaviour can be altered by deriving subclasses of MetaObj, in which these methods are redefined. Each reflective instance of a reflective base-level class is controlled at run-time by an instance of its associated MO class, a MO. When a new reflective instance is created, a new MO is also created to control that object. The relationship between the base-level objects and MOs is one-to-one. The object-MO binding is created at object creation time and cannot be modified or undone later. Open C++ version 1 does not provide support for MO composition.

Open C++ version 2 [9, 11] uses a compile-time MOP to make the C++ programming language extensible. The MOP consists of a class hierarchy that reifies the compiler's parse tree and is used to generate a specialised version of a C++ compiler. The specialised compiler is subsequently used to perform code transformation to the base-level program. The Open C++ version 2 MOP provides control over the compilation of class definition, state access, virtual function invocation, and object creation. This is achieved by reifying the compiler's parse tree as a collection of objects. The MOP supports behavioural reflection in the sense that it can be used to change the objects' behaviour by replacing or inserting code in an application. The source-to-source translation of the program as well as structural aspects, such as type information, are reified by the following MO classes: Class, Ptree, TypeInfo, and Environment. MOs of type Class play a key role in the MOP as they represent class definitions and control the source-to-source translation. Implementing a new MOP is accomplished by subclassing Class and by redefining the appropriate member functions that control the source translation.

A base-level class can select a MO class either by a *metaclass* declaration or by registering a new keyword. Unlike in Open C++ version 1, in which non-reflective instances of a reflective class can still be created, MOP selection in version 2 is class-based, i.e., all instances of a reflective class are reflective. As the MOs exist during compile-time, the link between the base-level and meta-level objects is not maintained during run-time.

2.3.1 Evaluation and relevance to this thesis

Open C++ version 1 is a coarse-grain run-time MOP with 3 elements of C++ that can be reified. There is no support for MO composition in Open C++ version 1, apart from the ability to add a MO to a MO thus resulting in multiple meta-levels (e.g., this approach is used in FRIENDS, see section 2.15).

Open C++ version 2 is a coarse-grain compile-time MOP: 4 elements of C++ can reified. It is not known whether more than one meta-level class can be used in the generation of code. Therefore, it is assumed that Open C++ version 2 does not support dynamic MO composition. Interesting to note that a MOP for fault-tolerant Common Object Request Broker Architecture (CORBA) applications [41] uses Open C++ version 2 compile-time MOP to insert a run-time MOP for supporting fault-tolerance. An interesting application of Open C++ version 2 is implementing atomic data types [69].

2.4 Composition of Before/After Meta-classes in SOM

In IBM's System Object Model (SOM) [23], a class is a runtime object that defines the behaviour of its instances by managing an instance method table. During the initialisation of a class object, a method is invoked on it, that informs the class of its parent classes. This allows the class to build an initial instance method table.

Because classes are objects, their behaviour is defined by other classes, called *meta-classes*. SOMClass is the default meta-class, from which all other meta-classes are derived. Instances of classes are classes. For example, a Before and After meta-classes can be used to define the implementation of classes that arrange for each method invocation to be preceded by execution of a "before" method, and followed by an "after" method, respectively. [25] introduces and solves the problem of composing different Before/After Meta-classes in the SOM context.

Interfaces to SOM objects are written in IDL, an object interface definition language, defined by the Object Management Group (OMG) CORBA standard. SOM IDL extends IDL with the ability to include SOM class descriptions in addition to object interface definitions.

The SOMMBeforeAfter meta-class introduces two methods, BeforeMethod and AfterMethod. By default, these methods do nothing. The programmer derives from SOMMBeforeAfter and overrides its methods with the desired behaviour. Examples of the use of before/after methods are: method tracing, invariant checking, concurrency, persistence, and replication. Before/After meta-classes are not useful unless they compose, because if not, the use of one meta-class would preclude the use of others.

SOM encourages the definition and the use of explicit meta-classes. At the same time, however SOM relieves programmers of the responsibility for getting the meta-class right when defining a new class. SOM does this by introducing a concept called *derived meta-class*, which deals with upward (binary) compatibility: given an instance of a SOM class, which inherits from possibly multiple superclasses, how could someone ensure that the meta-class of the derived class meets the expectations of the meta-classes of the superclasses? For example, if class Y, and instance of MetaY and a subclass of X, which is in turn an instance of MetaX. There is an upward compatibility problem because MetaY does not inherit from MetaX.

At run-time, SOM automatically determines an appropriate meta-class that supports this compatibility. When necessary, SOM automatically derives by subclassing a new meta-class called a derived meta-class. Thus, following the above example, SOM creates a Derived meta-class, which inherits from both MetaA and MetaX. Thus, a SOM programmer never needs to consider the meta-classes of a newly defined class' ancestors. Instead, explicit meta-classes should only be used to add in desired behaviour for a new class. Anything else is performed by SOM automatically. Because class construction is a dynamic activity in SOM, the appropriate meta-class derivation is done at run-time.

In SOM, the composition of Before/After meta-classes has the associative property: the order of meta-classes depends on the search order, which is determined by the order of the parent meta-classes.

2.4.1 Evaluation and relevance to this thesis

SOM with its SOMBeforeAfter meta-classes represents a coarse-grain run-time MOP supporting only the reification of invoking (remote) methods. The single SOM MOP is implicit as it manifests in the derivation of the appropriate meta-class only. Nevertheless, the automatic and dynamic composition of SOMBeforeAfter meta-classes makes SOM relevant to our work.

The order of executing the before and after methods from the combined meta-classes is fixed. Also note that SOM may introduce semantic inconsistencies by automatically combining semantically overlapping metaclasses at run-time. Since SOM objects implement IDL interfaces, SOM itself is a middleware.

2.5 CodA

The CodA [51] meta-level architecture is based on an operational decomposition of meta-level behaviour into objects and the provision of a framework for managing the resulting components. In CodA, the meta-level is decomposed into seven so-called *meta-components*, that reify different aspects of object behaviour such as:

Send Manages the sending of a message to an object. This can include supervision of the message transmission, synchronisation of the sender and the receiver, protocol negotiation, and so on.

Accept Defines how the receiver of a message interacts with the message sender. Therefore it has to deal with synchronisation and protocol negotiation. It also determines whether the message should be placed in a queue or should be processed immediately.

Queue Organises and holds messages that have been accepted but not yet received or processed.

Receive Responsible for fetching the next message to be processed. This may involve the selection of a message from a queue.

Execution Specifiies how an object interacts with the system in order to execute one of its methods. For example, it determines whether the method should be executed in debug mode or not. It also controls where and when a method is executed, and is responsible for actually executing the method.

Protocol Responsible for mapping a message to be processed onto a method to be executed. This requires the specification of how messages and methods can be matched.

State Organises and maintains information on the object's state, that is, its instance variables. It defines what instance variables an object has, and how they can be accessed.

Object behaviour is modified by explicitly associating meta-components with an object. A certain combination of these seven meta-components makes an *object model*, which describes a particular behaviour for a base-level object. The first six of these meta-components (also called *roles*) provide the programmer with behavioural (intercessory) facilities, while the last meta-component (State) implements structural reflection (introspection).

Although the CodA model can be applied to other systems, it is closely linked to the SMALLTALK [26] language.

CodA programmers have to explicitly instantiate the meta-level for reflective objects and insert the meta-components they want to use in order to implement a particular object behaviour. Dynamic reflection is achieved by allowing any of these seven meta-components to be replaced at run-time.

The meta-level of a particular object is created by dynamically combining various object models. Objects begin life behaving according to the default object model. Users apply their own particular object models. When adding object models to a meta-level, the constraints (i.e., meta-components) specified by the models must be combined. Since meta-components are general objects and object models can be arbitrarily complex, their automatic composition is difficult. Combining non-overlapping object models (i.e., the intersection of the meta-components of the two object models is empty) is straightforward. The new model simply contains the union of the meta-components from the original object models.

Combining overlapping object models may require programmer intervention. CodA does not offer an automatic composition. Instead, it offers a simple property-based specification that helps programmers resolve such composition conflicts. For example, if object models X and Y both have their own Send meta-components (XSend and YSend, respectively), the programmer has to create a new XYSend, that has the properties of both XSend and YSend. Having resolved the conflict between XSend and YSend once by implementing XYSend, the new meta-component is reusable in future combinations of other object models that require XSend and YSend.

2.5.1 Evaluation and relevance to this thesis

CodA is a fine-grain run-time MOP with 7 meta-components (a MOP in CodA is called an object model). Multiple object models can be combined but there is no support for automatic composition of meta-components (e.g., Send). Thus, meta-components of overlapping object models have to be combined by the programmer. Individual meta-components can be replaced at run-time.

The fine-grain MOP of CodA has influenced the design of IGUANA version 1.

2.6 DASCo: Separation and composition of overlapping and interacting concerns

DASCo (Development of Distributed Applications with Separation of Concerns) [66] is an approach to developing object-oriented concurrent and distributed applications using a separation of concerns strategy.

Separation of Concerns (SoC) approaches deal with abstraction and integration. The former describes common concepts such as object synchronisation or concurrency separated from functional (application) objects, while the latter deals with integrating abstractions among themselves and with functional objects. The SoC approach integrates abstractions only in the final program.

DASCo is based on design patterns, composition patterns and object-oriented frameworks to, respectively, describe abstractions, describe composition of abstractions, and implement/integrate abstractions and their composition with functional objects.

A design pattern is defined for each abstraction. Abstraction composition is obtained from the composition of each concern's design patterns. The composition of design patterns also constitute an abstraction described by a design pattern, in which participants are built by composing each involved pattern's participants. In addition, the composition pattern's collaborations are built from the collaborations of each pattern participating in the composition.

There are two cases regarding concern composition: orthogonal (i.e., there is no semantic overlapping between concerns) and non-orthogonal. The DASCo experiment indicated that orthogonal composition is rather restrictive. In many situations, there is semantic overlapping between different abstractions. Moreover, they verified that some composition abstractions have their own policies that are not trivially inferred from each of the composed abstractions. In order to deal with semantic overlap, the composition pattern should describe the new policies that result from the composition. It should also describe what are the consistent combinations of overlapping parts and what are the restrictions to policy composition. A matrix can be used to identify new policies and the associated restrictions on policy combination.

Design and composition patterns are implemented in a three-layer object-oriented framework with separation of concerns. In this framework, classes implementing design and composition patterns are grouped within components. In the object-oriented framework's concrete case, components consist of classes implementing design and composition patterns, and are instantiated either through instantiation of the explicit interface's parameters or through specialisation of some of their classes.

The three layers in the framework are as follows:

Concern layer : contains classes implementing each of the design patterns grouped into concern components. Concern components provide an interface for composition with other concern components and for customisation of the concern's policies.

Composition layer : contains classes implementing composition patterns. They provide a minimum interface for integration with the application.

Application layer : in this layer, composition components are integrated in the final program and concern components are customised so that they provide the policies required by the application's functional objects.

2.6.1 Evaluation and relevance to this thesis

DASCo is based on design and composition patterns as well as on a framework. DASCo is a non-reflective architecture, thus it does not have a MOP. We included in our review because it addresses the combination of semantically overlapping non-functional concerns.

2.7 IGUANA version 1 and 2

This section summarises IGUANA, a reflective programming model developed in the Distributed Systems Group (DSG) of Trinity College Dublin.

2.7.1 Metaobject composition in IGUANA version 1

The original IGUANA (we refer to it as version 1 in the text) reflective programming model [29, 30, 28] supports multiple, fine-grained run-time MOPs with selective reification. The compile-time reification process is static[2] and performed during the MOP definition.

The behaviour described by an object model for a programming language can be quite complex. Consequently, a MOP that specifies the object model in terms of meta-level objects (classes), interfaces and interaction will be equally, if not more complex. Dividing an object model into fine-grained MOPs allows the implementation to be modular and benefit from reuse of MO classes. IGUANA version 1 introduced the concepts of reification categories and reification category lists. Each reification category reifies only a small part of the object model, for example, method send, receive, dispatch, and invocation, that can be represented by a MO. A reification category list groups a set of related reification categories.

There are 29 reification categories listed in IGUANA version 1 [28]. The 9 structural reification categories are: Class, Identity, InheritanceTree, MethodAddress, MethodName, StateAddress, StateName, Source, and Type. The 20 behavioural reification categories are: Creation, Deletion, Dispatch, Invocation, MethodAccess, MethodBefore, MethodAfter, Reception, StateBeforeRead, StateAccessRead, StateAfterRead, StateBeforeWrite, StateAccessWrite, StateAfterWrite, ActFrame, Inheritance, Method, State, TypeSoft, and TypeSoftPlus.

The benefit of this fine-grained approach is that only those parts of the object model, that are required to implement the derised object behaviour are reified. Thus selective reification reduces the reflective overhead placed on the object model.

An IGUANA version 1 MOP is a specification of the behaviour of an object model. It is also a first-class language entity in IGUANA: the protocol keyword can be used to specify a MOP. The MOP specification consists of four main parts:

- List of reification categories and their related MO classes;
- List of other MO classes that are not related to reification categories;

[2]The IGUANA MOP defines the aspects of the object model that are reified. MOP definitions are processed at compile-time by a preprocessor.

- List of parent protocols (i.e., MOPs);
- Sibling MOP dependencies.

The distinction between base-level and meta-level objects is less well-defined in IGUANA than in some other systems. The IGUANA MOP selection process causally connects base-level and meta-level objects and allows meta-level objects in turn to be causally connected to a further layer of meta-level objects to an arbitrary depth.

The IGUANA version 1 MOP supports *introspection*, *intercession*, and dynamic *meta-level reconfiguration*. A call to a meta-level object may be *implicit* or *explicit*. Implicit calls occur when a language component has been reified. Explicit calls can be made from normal base-level objects by using either an IGUANA *extension protocol* or calling public methods on MOs. Dynamic adaptation is supported via explicit invocations to the meta-level.

IGUANA MOPs are associated with base-level objects through procotol selection. A base-level object can select one or more MOPs. The selection process must be used somewhere in the implementation phase. Although causal connections between base-level and meta-level objects are made at compilation-time, this is not necessarily a limitation. It is a specific feature of the IGUANA version 1 meta-level architecture that meta-level objects can be replaced dynamically, at run-time.

MOP selection is either behavioural or augmentary. Behavioural selection is the most common and is used to describe or implement the behaviour of base-level objects. IGUANA version 1 supports three forms of behavioural MOP selection: *default*, *class*, and *instance*. They differ in their scope: default selection associates classes in a compilation unit (i.e., a C++ source file) with the MOP(s), while class selection selects the MOP(s) for all instances of that class. Instance selection binds the MOP(s) to that instance only, thus other instances of the same class may select different MOPs or may not be reflective at all. Augmentary selection is used to add to, or augment the behaviour of the base-level objects temporarily. There is one augmentary selection form in IGUANA, called the *expression* selection. The scope of an expression selection is that of a single expression.

The IGUANA version 1 model facilitates dynamic adaptation of the program in the following ways: the programmer can replace an entire MOP, or replace/modify individual meta-level and base-level objects.

IGUANA extension protocols provide controlled access to the meta-level from the base-level. They abstract away from the specific features of a meta-level object to provide an application-specific feature. They are written as standard C functions or C++ methods. The reasoning for this is given in [28] as follows: "The functionality encapsulated by an extension protocol can usually be encapsulated within a single function. If this is not the case, then a C++ class can be used as the implementation medium." Extension protocols provide the building blocks, from which secure adaptable systems can be constructed. Methods in an extension protocol simply group a number of calls to rearrange the meta-level in order to adapt it according to the application's changing needs. The base-level object invokes these calls on an extension protocol when it wants to adapt the application. Extension protocols thus can act as a coherent and concise abstract interface to a MOP.

For a given program or service, the number of extension protocols can be dynamically changed over time as appropriate. The IGUANA version 1 model can support multiple extension protocols even over a single MOP. For example, when a thread scheduling policy modification is requested by the client, an extension protocol can be dynamically put in place and executed in response to the event. All the necessary security validations can be implemented as a part of the extension protocol, thus separating the implementation (base-level), the description of the implementation (meta-level), and the adaptation/specialisation processing (interface between base-level and meta-levels) into three separate units.

Meta-level combination conflicts can arise as IGUANA allows more than one MOP to be causally connected to the base-level object. The IGUANA case of having multiple MOPs selected for a base-level object is quite complex, as these MOPs may have conflicting MOs declared and use multiple MOP inheritance. To address this problem, IGUANA version 1 relies on the programmer to specify the ordering of MOs in each of the reification categories. If no order is specified, IGUANA MOPs have a default behaviour for sequential ordering (i.e., this is the chain of responsibility model mentioned in section 1.2). This is implemented by the default *meta-combiner*. By inheriting from a meta-combiner interface, a programmer can write his/her own combiner algorithm. This can be used by specifying the name of the combiner class in the appropriate reify statement, which is part of the protocol definition.

For example, to use a special combiner class MyStateAccessCombiner for a distribution MOP, one would write the following:

```
protocol Distributed {
shared:
  reify StateRead (MyStateAccessCombiner) DistributedStateRead;
  reify StateWrite (MyStateAccessCombiner) DistributedStateWrite;
  [...]
};
```

Although IGUANA version 1 is a generic model that can be tied/applied to any (object-oriented) programming language, it has been implemented as IGUANA/C++, a reflective extension to the C++ language. IGUANA is a run-time MOP implemented as a source-to-source translator.

2.7.2 Metaobject composition in IGUANA version 2

The second version of the IGUANA model [21, 64] is a simplification and an evolution of the previous model. Simplifications include a reduction in the number of reification categories (from 29 to 12), and streamlining of the meta-level structures.

In IGUANA version 2, there are only 12 reification categories, which can be classified as *structural* and *behavioural*, namely:

Structural: Class, Method, Attribute, Constructor, and Array.

Behavioural: Creation, Deletion, Invocation, StateRead, StateWrite, Send, and Dispatch.

Dependencies between behavioural and structural reification categories have also been identified in the thesis. For example, the Invocation behavioural MO needs the Class and Method structural MOs to carry out its operations.

Similarly to IGUANA version 1, there are three ways of selecting a MOP: class, instance, and default. The expression selection has however been removed from IGUANA version 2.

IGUANA version 2 supports automatic and manual MO composition by means of multiple MOP inheritance. Behavioural MOs reifying the same language element (e.g., method invocation or state read) are organised in a chain: each MO carries out its operation and is expected to call the next MO in the chain. By convention, the last MO should be the default MO, which reflects the operation at the base-level. The order of MOs in the list is deduced from the MOP precedence list.

By default, control propagates through the list of MOs until it reaches the last MO, and then it goes back. Should not the default composition work for some MOP combinations, a MO positioned in the beginning of the chain can be coded as a combiner and it can either override the order of the remaining MOs or call them in a particular order (e.g., a combiner MO can implement the at-most-one semantics).

IGUANA version 2 introduced the concept of *meta-types*. The meta-type of a base-level object captures the functionality offered by the reflective language extension. An object's meta-type can be static or dynamic. The static meta-type is the MOP that has been selected for the object's class using class MOP selection. The dynamic meta-type is the MOP that has been (re)selected for the object, subject to the following rules:

1. Every object has a single meta-type.
2. The meta-type of an object can be changed dynamically.
3. A class inherits the meta-type of the superclass. The meta-type selected by a class must be the same or a sub-type of the meta-type selected by its superclass.
4. The dynamic meta-type of an object must be a sub-type of its static meta-type.

In contrast with IGUANA version 1, which allows objects to select multiple MOPs, here an object can have at most one MOP (meta-type) associated with it. The reason for this is given in [64]: "Rules to combine MOs from meta-types that are 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. This meta-type in turn can be composed [of multiple meta-types], but it is the meta-level programmer's responsibility to provide a meaningful composition order."

In both IGUANA versions, an IGUANA/C++ to standard C++ pre-processor is used to implement the model.

2.7.3 Evaluation and relevance to this thesis

Both IGUANA versions are fine-grained run-time MOPs, with 29 and 12 reification categories in version 1 and 2, respectively. The main difference between them is in the MOP selection: version 1 allows selecting multiple MOPs, while version 2 introduces the concept of meta-type, which translates to single MOP selection. There is a single meta-level in version 2 versus the meta-towers in version 1.

Both versions support changing the MOP(s) (the single meta-type in version 2) dynamically. Also both versions use the chain of responsibility model for combining MOs in a particular reificiation category. IGUANA version 1 has Combiner MOs, but it is difficult to see how multiple combiner MOs are combined. IGUANA version 2 supports dynamic composition of the MOs of the single meta-type.

Our work is an evolution of the IGUANA models, where we focus on solving the problem of automatic and dynamic composition of multiple, indepenendently developed, and semantically overlapping MOPs. We have identified the limitations of MOP composition in IGUANA version 1 and 2, refer to section 1.4.

2.8 Towards a Methodology for Metaobject Composition

A methodology aimed at the design of composable MOs has been proposed in [53]. The basic problem the paper attempts to solve is the reuse of existing MOPs: given two independently developed language customisations (in the form of two MOPs), is it possible to compose them into one combined customisation? Moreover, since the programmer wants to treat the MOPs as black boxes, what are the least condiditions one has to impose on the MOPs in order for this composition to work?

The paper points out that MOP composition in its full generality is a very hard problem because of the interferences between the combined behaviours. Composing semantics is a general problem, that can be handled for MOs by including some constraints leading to a specific methodology.

Practically, MOPs usually offer customisable functionalities as a set of elementary bricks, which have to be assembled together to define the complete semantics. Assembly is easy if the bricks do not have semantic overlap. However, assembling purely orthogonal functionalities constrains the range of combinations, since several customisations of the same brick cannot be composed unless an expert designer manually resolve conflicts between overlapping semantics. The issue the paper deals with is the composition of non-orthogonal semantics. It advocates the realisation of elementary MOs, having potentially overlapping semantics, and it attempts to reuse them afterwards by automatic composition.

The paper associates MOs in pairs in an ordered relation; it distinguishes between left-composed and right-composed MOs. A left-composed MO specialises or aggregates the right-composed one. Composing MOs only means having the left-composed MO requesting explicitly the functionality of the right-composed one, without knowing the internal details of it. The resulting co-operation depends closely on the type of the composition link, which is either specialisation (by means of inheritance or delegation) or aggregation (embedding). However, since specialisation and aggregation are not expected to compose MOs, the programmer has to design MOs that co-operate along the composition link.

Each composition solution (either specialisation or aggregation) induces particular constraints on the internal design of the MOs making use of it. In other words, MOs should be cooperative and written with future composition in mind.

The following three design rules must be respected:

Exclusivity: when a meta-computation is performed on an object, its respective MO must be activated to this aim, since it is assumed to be the only one capable of performing this task.

Encapsulation: a MO is like a black box, those external interface corresponds to the protocol specified by the MOP.

Independence: the implementation of a given MO must make no specific assumption on the implementation of the base-level object.

An application of this model to the MOOSTRAP language is given in the paper, which demonstrates composition with trace and delegation. The generalisation of the above methodology is then applied to the CLOS MOP. In the CLOS MOP context, a cooperative MO must foresee its future composition. Practically, FLAVORS inheritance already advocated the concept of cooperative components, in the form of *mix-ins*. A mix-in defines a particular feature of an object. A mix-in cannot be instantiated because it is not a complete description (there are base "flavours" that are complete). Considering that mix-ins are required not to interfere with other behaviour and their methods should explicitly invoke the call-next-method, the composition is achieved when defining a concrete complex MO, incorporating several semantics supported by mix-ins, and a final one, which is the base MO.

2.8.1 Evaluation and relevance to this thesis

We included this paper in the review because it addressed the composition of independently developed, semantically overlapping MOPs and provided a methodology for writing composable MOs. The generalisation of the methodology is applied to the CLOS MOP, which is described in section 2.2.

2.9 Guaraná

Guaraná [56] allows MOs to be combined through the use of *composers*. Composers are MOs that can be used to define arbitrary policies for delegating control to other MOs, including other composers. They provide the glue code to combine MOs, and to resolve conflicts between incompatible ones. The use of composers encourages the separation of the *structure* of the meta-level from the *implementation* of individual management aspects. The authors of Guaraná argue that the chain of responsibility model has some serious drawbacks:

- It is intrusive on the MO implementation, in the sense that a MO must forward operations to its successor in the chain.

- It forbids two MOs concurrently handling the same operation.
- It forces MOs to receive the operation results from the successor MO.
- The order of presentation of results is necessarily the reverse order of the reception of operations.
- It is impossible to mediate interactions between MOs and base-level objects with an adaptor capable of resolving conflicts that might arise when multiple MOs are put to work together.

These problems are solved in Guaraná by splitting the meta-level processing associated with a base-level object in the following steps:

1. The Guaraná kernel intercepts operations on base-level objects that have MOs associated with them.
2. A MO may produce the result for the operation. In this case, the meta-level processing terminates by unreifying the result as if it had been produced by the execution of the intercepted operation.
3. However, a MO is not required to reply with a result. The MO may reply with an operation to be delivered to the base-level. In addition, the MO may indicate that it is interested in receiving and/or modifying the result of the operation.
4. Finally, the operation is delivered to the base-level, and its result may or may not be presented to the MO, depending on its previous reply to the operation.

Replacement operations can be created in the meta-level using *operation factories*. Operation factories allow MOs to obtain access to the base-level objects they manage. Stand-alone operations can also be created with operation factories, and then performed, i.e., submitted for interception, meta-level processing, and potential delivery for base-level execution.

Composers separate operation handling from result handling, implemented in two distinct methods, namely, `handle operation` and `handle result`. A composer is a MO that delegates operations and results to multiple MOs and then composes their replies in its own replies. For example, a composer can implement the chain of responsibility model, but in a way that MOs on the chain do not need to keep track of their successors. Another composer implementation may delegate operations and/or results concurrently to mulitple MOs, or refrain from delegation if it knows that those MOs are not interested in that operation.

In Guaraná, there is at most one MO associated with a base-level object at any time. This MO is called *primary metaobject*. The primary MO can be a composer, and it can delegate operations/results to other composers, too.

Guaraná presents two additional features that enforce the separation of concerns between the base-level and the meta-level: the meta-configuration of an object is completely hidden from the base-level, and even from the meta-level itself; and the initial meta-configuration of an object is determined by the meta-configurations of its creator and of its class, a mechanism Guaraná calls *meta-configuration propagation*. In Guaraná, there is no way to find out the primary MO of an object. It is possible, however, to send arbitrary messages and reconfiguration requests to the components of the meta-configuration through the Guaraná kernel. The kernel supports the following operations:

Broadcast Messages can be used to extend the MOP, as they allow MOs to exchange information. MOs that do not understand a message are supposed to ignore it. Composer MOs are supposed to forward a message to their components.

Reconfigure A reconfiguration request message carries a pair of MOs, suggesting that the first MO should be replaced with the second MO in the object's meta-configuration. The MO reference *null* can be used to refer to the primary MO. It is up to the existing meta-configuration to decide whether the reconfiguration request is acceptable or not. If the base-level object is not reflective, an *InstanceReconfigure* message is broadcast to the meta-configurations of its class and superclasses. Their components can modify the suggested new meta-configuration.

Object creation In many object-oriented programming languages, object creation has two steps: first, storage is allocated for the new instance, then secondly, the constructor method is invoked. These steps are performed by the creator of the new object. In Guaraná, the creation of the meta-configuration for the new object takes place between these two steps. The primary MO of the creator is responsible for providing a meta-configuration for the new object. It may return `null`, a different MO or even itself (the same MO can be shared between different meta-configurations). A composer is expected to forward the creation request to the MOs returned by them. After meta-configuration propagation, the kernel broadcasts a *NewObject* message to the meta-configuration of the class of the new object, so that its MOs can try to reconfigure it. Finally, the object is constructed, and the constructor invocation can be intercepted by the kernel if the base-level object is reflective.

Proxy creation Guaraná provides a mechanism that allows proxy object creation from the meta-level, without invoking their constructor. When a proxy is created, the kernel broadcasts a *NewProxy* message to the meta-configuration of the new proxy object's class. A proxy will usually be given a meta-configuration that prevents operations from reaching it, but it may be transformed into a real object by its meta-configuration, through constructor invocation or direct initialisation.

Guaraná has been implemented using Kava, an open-source Java Virtual Machine (VM). MOLDS [58] is a library of MOs for Guaraná, which provides essential features for developing reliable distributed systems. Meta-level services include persistence, replication, distribution, caching, migration, logging, and atomic execution.

2.9.1 Evaluation and relevance to this thesis

Guaraná is a fine-grained run-time MOP with Array Read, Array Write, Field Read, Field Write, and Method Invocation Java operations reified. The MOs representing these reified concepts can be composed by explicit composer MOs. If composition of multiple MOs is required, a composer MO has to be instantiated. It is left to the (meta-level) programmer to design and implement a dynamic composition algorithm.

2.10 *X*-Kernel: an Architecture for Implementing Network Protocols

The *x*-kernel [33] is an operating system kernel that provides an explicit architecture for constructing and composing network protocols. It views a protocol as a specification of a communication abstraction, through which a collection of participants exchange a set of messages. Beyond this simple model, the *x*-kernel makes few assumptions about the semantics of protocols.

The *x*-kernel provides three primitive communication objects to support this model: *protocols*, *sessions*, and *messages*. These objects are classified as static or dynamic, and passive or active. Protocol objects are both static and passive. Each protocol object corresponds to a conventional network protocol (e.g., IP, UDP, TCP), where the relationships between protocols are defined when the kernel is configured. Protocol objects are created and initialised at kernel boot-time. Data global to the protocol is contained in the protocol state. Protocol objects serve two main functions: they create session objects and they demultiplex messages received from the network to one of their session objects.

Session objects are also passive, but they are dynamically created. A session object is an instance of a protocol object that contains a protocol interpreter and the data structures that represent the local state of some network connection. The two primary functions sessions support are push and pop for passing messages down and up, respectively. Because sessions represent connections, they are created and destroyed when connections are established and terminated. The session-specific state includes capabilities for other session and protocol objects as well as whatever state is necessary to implement the state machine associated with a connection.

Although the kernel is written in C, the infrastructure enforces a minimal object-oriented style on protocol and session objects, that is, each object supports a uniform set of operations.

Messages are active objects that move through the session and protocol objects in the kernel. The data contained in a message object corresponds to one or more protocol headers and user data. Messages either arrive at the bottom of the kernel (i.e., at a networking device) and travel upward to a user process, or they arrive at the top of the kernel (i.e., a user process generates them) and flow down to a networking device. While travelling downward, a message visits a series of sessions via their push operations. While flowing upward, a message visits alternatively a protocol via its demux operation and then a session in that protocol's class via its pop operation. As a message visits a session on the way down, headers are added, the message may fragment into multiple message objects, or the message may suspend itself while waiting for a reply message. As a message visits a session on the way up, headers are stripped, the message may suspend itself while waiting to re-assemble into a larger message, or the message may serialise (re-order) itself with sibling messages.

The *x*-kernel also provides the programmer with highly-tuned, general purpose utility routines. These routines include buffer, map and event management.

Usually a suite of protocols are configured into an instance of the *x*-kernel. When a protocol is initialised, it is given a capability for each protocol on which it depends, as defined by the protocol dependencies graph. The relationships between communication objects (i.e., the protocol dependencies) are defined using either a simple textual graph description language or an X-Windows-based graph editor. A composition tool reads this graph and generates C code that creates and initialises the protocols in bottom-up order.

The experience with *x*-kernel described in the paper shows that it is both general enough to accomodate a wide range of protocols, yet it is efficient enough to perform competitively with less structured operating systems.

2.10.1 Evaluation and relevance to this thesis

The *x*-kernel is a non-reflective operating system kernel. It has been included in this review because of its notion of describing communication protocols in terms of capabilities and requirements and its composition mechanism that reads this information and generates code for the instantiated protocol stack.

2.11 Secure Composition of Security Metaobjects

[63] outlines a MOP for secure composition of cryptography-aware MOs. Cryptography-based security services address confidentiality, authentication, data integrity, and non-repudiation, and are usually implemented on basic security services (encryption, digital signatures, and other types of electronic fingerprints). The composition of these mechanisms is often a means for satisfying more complex security requirements. However, special care is needed when combining these mechanisms, in order to avoid incorrect combinations.

Computational reflection can be used to implement security mechanisms in a transparent, non-intrusive manner, that is, without interfering with the application's original structure. MOs can implement these cryptographic mechanisms.

The cryptographic mechanisms for data integrity, authentication, and digital signature are mutually exclusive and relate to each other as follows:

- An authentication fingerprint supports both sender (origin) authentication and data integrity;
- A digital signature supports non-repudiation as well as sender authentication and data integrity;
- Encryption is orthonogal to other cryptographic mechanisms and can be combined with any of them.

The ways cryptographic-aware MOs are composed are limited by the above relationships. There are two kinds of composition: a simple composition permits orthogonal MOs to compose their behaviours seqeuentially. On the other hand, a selective composition allows composition of mutually exclusive MOs in such a way that when one is turned on, the others are turned off. When a cryptography-aware MO is asked for recofiguration, it can follow either a conservative or a non-conservative approach. In the first case, combinations resulting in weaker security are not allowed. Thus, the meta-configuration can either remain the same or stronger. In the non-conservative approach, weaker combinations are also allowed. The authors adopted the conservative approach for meta-level reconfiguration: e.g., a MO for digital signature

	Candidate configuration			
	Encryption	Auth. checksum	Integrity code	Digital signature
Encryption	1	3	3	3
Auth. checksum	3	1	1	2
Integrity code	3	2	1	2
Digital signature	3	1	1	1

Table 2.1: Summary of rules for reconfiguring security MOs

cannot be replaced by authentication or data integrity MO. Similarly, a data integrity MO cannot replace an authentication MO. Furthermore, MOs of the same type cannot replace each other. For example, two MOs for encryption based on different algorithms cannot replace each other. On the other hand, a single encryption MO can compose with any MO for signature, authentication or data integrity.

The MO reconfiguration policy can be presented as a table (see Table 2.1).

The numbers in the table indicate composition rules regarding the current and candidate meta configurations. They are as follows:

1 - the current meta configuration is not replaced;

2 - a selective composition of both current and candidate meta configurations replaces the current one;

3 - a composition of both current and candidate meta configurations replaces the current one;

4 - the candidate meta configuration replaces the current one.

Cryptographic services usually address non-functional requirements related to network communication and persistence requirements, but are orthogonal to these. The tower of MOs can be as high as the number of non-functional requirements. The decision concerning the position of cryptography in this tower is not simple. Aspects such as requirements composition or chaining must be considered very carefully. For example, because cryptography is orthogonal to persistence and communication, which can, in turn, be positioned at the meta-level, cryptography should be accomplished at the meta-level of these; that is at the meta-meta-level. However, if fault tolerance (replication) is another requirement, it can be accomplished either above or below encryption.

The *a priori* negotiation, concerning both usage of and agreement on cryptographic algorithms (e.g., generation, exchange and storage of keys) may or may not be handled at the meta-level. This decision depends on the degree of control over the cryptographic services the application programmer intends to have. For example, application programmers may be interested in the kinds of services are being used at a particular time, maintaining the ability of turning the communication security on or off.

2.11.1 Evaluation and relevance to this thesis

We could not find information on the implementation of this MOP. However its approach to composition of (security) MOs has influenced ours: we borrowed the notion of security concern area and broadened it to cover other areas of non-functional concern. The notion of exclusivity within a concern area is also drawn from here.

2.12 An Event-based Runtime Metaobject Protocol

Renaud Pawlak's CNAM internship report [62] introduces a new approach to run-time MO composition. The framework in this report is based on an *event-based run-time Metaobject Protocol* (EB-RT-MOP), which is in turn based on the Open C++ version 2 MOP.

In order to allow automatic MO composition, MOs first need to be classified. The classes are as follows:

Base-modificators: MOs that change the base-level object;

System-modificators: MOs that send messages to other objects;

Reflectors: MOs that reflect the base-level operation, but they do not change the base-level object's state and do not send messages to objects. Reflectors can be pure or conditional (as described below) or obligatory;

Pure reflectors: reflector MOs that always reflect the base-level operation;

Conditional reflectors: MOs that may (or may not) reflect the base-level operation;

Obligatory reflectors: reflector MOs that always have to be called;

Exclusive reflectors: reflector MOs that have to be called only if the base object is effectively reflected;

Obligatory conditional reflectors: conditional reflector MOs that always have to be called;

Exclusive conditional reflectors: conditional reflector MOs that have to be called only if the base object is effectively reflected.

Note that MOs may fall into more than one classes: e.g., a MO may be both base-level and system-modificator. Ordering of MOs is achieved as follows: base-level or system-modificators may need manual ordering performed by the meta-level programmer. For example, if two system-modificators send one message each to the base-object and the messages are not commutative, then the modificators have to be ordered in the way the programmer wants them to be. In practice, this is a difficult issue, as the meta-level programmer is not always aware of the kind of changes performed by all the modificator MOs and he/she may not be able to say if a modificator has to be called before or after another one. All the modificators could be regrouped into a family of orderable MOs. The number of obligatory conditional and exclusive conditional reflectors is fixed as one. Orderable MOs are assumed to be manually or automatically ordered thanks to an ordering key or an algorithm given by the meta-level programmer. However, we can assume that non-commutative operations are performed by the same MO so that modificator MOs do not need to be ordered. In this case, we do not need to distinguish modificators from reflectors anymore. Thus, the assumption is that modificators are programmed in a commutative way. Ordered modificators are defined as for future work. Therefore when discussing the algorithm for ordering MOs, we only speak about reflectors.

Next step of the classification is the definition of ordering relationships between MOs. We define four operators:

- $\mathbf{M}_a \sim \mathbf{M}_b$ means that the two MOs are equivalent; their order can be changed in the MOs list.
- $\mathbf{M}_a = \mathbf{M}_b$ means that the two MOs are equal; that is, of one is executed then the other one is executed, and conversely.
- $\mathbf{M}_a > \mathbf{M}_b$ means that M_a must be executed before M_b (M_a preceeds M_b in the MOs list).
- $\mathbf{M}_a < \mathbf{M}_b$ means that M_a must be executed after M_b (M_a suceeds M_b in the MOs list).

Operators $\sim$ and $=$ are symmetric, reflexive, and transitive. Operators $>$ and $<$ are transitive. The MOs attached to a base-level object can be split into the six sub-lists (some of them may be empty):

- **L(r)**: list of reflectors with n_r elements,
- **L(or)**: list of obligatory reflectors with n_{or} elements,
- **L(er)**: list of exclusive reflectors with n_{er} elements,
- **L(cr)**: list of conditional reflectors with n_{cr} elements,
- **L(ocr)**: list of obligatory conditional reflectors with n_{ocr} elements,
- **L(ecr)**: list of exclusive conditional reflectors with n_{ecr} elements.

The ordering relations between MOs of the same class can be written as:

- $\forall$ (i, j) $\in [1; n_r]^2 : r_i \sim r_j$
- $\forall$ (i, j) $\in [1; n_{or}]^2 : or_i \sim or_j$
- $\forall$ (i, j) $\in [1; n_{er}]^2 : er_i \sim er_j$
- $\forall$ (i, j) $\in [1; n_{cr}]^2 : cr_i \sim cr_j$
- $\forall$ (i, j) $\in [1; n_{ocr}]^2 : ocr_i = ocr_j$

- $\forall$ (i, j) $\in [1; n_{ecr}]^2 : ecr_i = ecr_j$

The two last cases are special. Because obligatory conditional reflectors must always be executed, they are related by the = relation. However, if one of their conditions is evaluated to be false, they break the chain: i.e., the following obligatory conditional reflection will not be called (because the base-level will not be reflected). Thus, the only possibility to respect obligatory conditional reflector properties is to fix their number to one. For exclusive conditional reflectors, it is quite the same problem, therefore there is only one MOs of this class allowed.

The final step in the classification algorithm is the ordering relations of MOs between any elements of the above sub-lists (inter-class relation). Because of the transitive property of the > relation, we can write:

$\forall$ (i, j, k, l) $\in [1; n_{or}] \times [1; n_r] \times [1; n_{cr}] \times [1; n_{er}]$:

$or_i > r_j > ocr_1 > cr_k > ecr_1 > er_l$

This ordering relation between any kind of reflector allows automatic ordering of MOs, thus when a new MO is added to the list, its position in the list can automatically be determined.

2.12.1 Evaluation and relevance to this thesis

The implementation of EB-RT-MOP is based on Open C++ version 2, see section 2.3. We find the analysis and classification of different types of MOs important in designing a composition mechanism. The limitation of this MOP is that the number of obligatory conditional reflectors and exclusive conditional reflectors allowed in a MO set is limited to one.
The classification of reflectors (MOs) has influenced the design of the IGUANA version 3 MOP descriptors.

2.13 APERTOS: a reflective operating system with metaspaces

APERTOS [80, 79, 43, 73] is an object-oriented, reflective operating system developed at Sony's Computer Science Laboratory. The most significant contribution of APERTOS is its use of *metaspaces*. A metaspace is a collection of MOs that provide the execution environment for a base-level object. MOs can describe features such as virtual memory management, communication, disk management, fault handling, and other typical functionality that can be found in operating systems.

Metaspaces exist in a hierarchy called a *meta-hierarchy*. Lower level metaspaces support the execution of higher-level ones. Metaspaces with alternative implementations and interfaces can freely co-exist in order to facilitate dynamic adaptation. For a base-level object to use an alternative metaspace, it has to migrate from the current metaspace to the new one.

The APERTOS MetaCore MO is at the bottom of the meta-hierarchy and can be considered as a micro-kernel (it has no metaspace). MetaCore offers the basic execution environment, upon which metaspaces can be built.

APERTOS allows an object to migrate between more than one version of an operating system feature (implemented as a metaspace). During an object's migration, a series of compatibility tests are performed. APERTOS does not allow the instantiation of new metaspaces with different features at run-time. New metaspaces can only be created at design-time and then need to be compiled.

2.13.1 Evaluation and relevance to this thesis

Apertos is a reflective operating system composed of a hierarchy of metaspaces. It is influential because it shows how a micro-kernel based operating system can be built using multiple meta-levels.

2.14 Rewriting semantics of metaobject and composable distributed services

In distributed systems and communications software, there is great interest in modular and dynamically composable approaches. According to [18], "services such as security and fault tolerance and services intended for boosting performance should be installed dynamically and selectively at run-time in those areas or domains of the distributed system where they are needed." Aspect weaving (see Section 2.17) is one mechanism that could be used to deal with this problem. The onion-skin, actor-based reflection [1] is another model.

Several of these approaches use reflection to achieve modularity and adaptability. But all of these approaches recognise that the goal of achieving truly modular and composable distributed systems, and ensuring good properties in compositions of such services is quite subtle. According to this paper, no satisfactory formal semantics of what is meant by composable distributed services has not yet been given, nor have the reflective aspects of such compositions been adequately formalised.

This paper proposes a semantics-based approach to make precise the reflective concept of composable service in a distributed system, and to reason about the properties of service compositions. This approach is based on the executable formal semantics for distributed object-oriented systems provided by *rewriting logic* and explicitly addresses the reflective properties that are essential for having a truly modular notion of service.

2.14.1 Evaluation and relevance to this thesis

This work represents one of the first approaches to apply formal methods to the use of reflection in composable distributed services.

2.15 FRIENDS: a Metaobject Architecture for Fault-Tolerant Distributed Systems

In FRIENDS[3] [35], MOs are used recursively to add new properties to distributed dependable applications. FRIENDS takes a multi meta-level approach in contrast to previous work [34] with a single meta-level.

On the actual implementation of fault-tolerant services, the paper notes: "MOPs are not the panacea and it is not claimed that they can be used on their own to build dependable services. Several basic services must be implemented at the system level (e.g., group communication, error detection, atomic multicast protocols, authentication and authorisation servers)."

[3]FRIENDS stands for Flexible and Reusable Implementation Environment for the Next Dependable System.

The FRIENDS architecture consists of kernel, system, and user layers. The system layer builds upon the (micro)kernel services and provides applications and MOs with the following services: security, group communication, and fault-tolerance.

The following three subsystems (middleware) comprise the system layer.

Fault-Tolerant Subsystem: implements error detection, failure suspectors, configuration, and replication domains management facilities, as well as stable storage support.

Secure Communication Subsystem: contains authentication, authorisation, directory, and audit servers.

Group Distribution Subsystem: provides group management and atomic multicast communication services.

FRIENDS uses the Open C++ version 1 MOP for its implementation. A base-level application object is bound to a MO for fault-tolerance, which is bound to a MO for security, which is in turn bound to a MO for group communication. The position of these MOs is fixed: the MO for group communication is always the last in the "protocol" stack. Depending on the actual application requirements, the first two MOs can be left out.

The application object and MOs together form a run-time object. The interaction between the application object and the MO is done through the MOP. FRIENDS uses a proxy server model for distribution and replication.

FRIENDS defines a hierarchy of MO classes. The Fault-Tolerance (FT) MOs library provides MO classes for various fault-tolerance strategies (based on stable storage and/or replication) with respect to physical faults, considering fail-silent nodes. Fail-silent means that errors are detected and dealt with by the underlying operating system. The Secure Communication MO library provides MO classes for secure communication protocols, including encryption and authentication. The Group-based Distribution MO library provide MO classes for handling remote object interaction, which can be implemented with groups.

2.15.1 Evaluation and relevance to this thesis

FRIENDS is a meta-level architecture providing libraries of MOs for fault-tolerance, secure communication for group-based distributed application. It uses Open C++ version 1 MOP, see section 2.3. As such, it provides a reflective language-based solution to address these non-functional concerns.

FRIENDS has greatly influenced the design of IGUANA version 3 thus they have many things in common:

- Proxy model for remote communication.
- MOs form a stack (although FRIENDS use different MOPs for proxy and server).
- Last MO is the one for group communication (e.g., the "remote" invocation MO in IGUANA).
- Subsystems for providing security, fault-tolerance, and group distribution service (IGUANA middleware components).
- Both FRIENDS and IGUANA are a language-based solution: the base-level programmer writes simple C++ classes and selects the required MO set (MOP set in IGUANA) for them.
- MOs are chained (multiple meta-levels in FRIENDS, while one meta-level in IGUANA version 3).

FRIENDS has recognised the importance of ordering the MOs: changing the order of MOs in the stack can lead to different application properties. The limitations of FRIENDS are mainly coming from that of Open C++ version 1:

1. The links between application objects and MOs are created at compilation time.
2. No support for application class inheritance.
3. No direct manipulation of the application object from the higher meta-levels.

Although FRIENDS defines a hierarchy of fault-tolerant and security MO classes, it does not address dynamic composition: it is only mentioned as part of current/future work.

2.16 Automatic Composition of Systems from Components

A mechanism has been proposed in [68], which can compose a system from components with anonymous dependencies: i.e., a component, which provides certain services, depends on other services in turn provided by some other components. Each software component has a descriptor with semantic-unaware properties. Clients use an application domain independent formalism for describing their configuration requests in terms of desired properties.

The composition algorithm has two phases: the optional first phase processes the application domain-specific requirements, which results in the translation of them into semantic-unaware domain independent properties. The second phase carries out the domain independent composition.

Current work on composition considers composing an application as a layered architecture: each layer encapsulates a primitive domain/feature. In this model, each layer is a component, that provides a specific set of services, which can be used by other layers on top of it. The composition process, which is intrinsically architecture-independent, expoits the layer property of incremental enchancement of services. If the current layer does not provide enough functionality for a given application, the service may be enhanced step-by-step by adding other layers.

Components have two ports, *UP* and *DOWN*. A component descriptor includes the following information:

- list of provided properties (property is a unique name so it ensures application domain independence);
- list of required properties (unique names) on the *UP* port;
- list of required properties (unique names) on the *DOWN* port;
- list of roles (optional).

Downward requirements are defined as the requirements imposed by a component on its *DOWN* port towards components that are in layers below. Similarly, *upward requirements* are defined as the requirements imposed by a component on its *UP* port towards components above it.

Requirements can be specified as *weak* or *strong* in terms of their strictness. A strong requirement must be fulfilled in order to yield a correct composition. A weak requirement should only not be contradicted by the composition solution. For example, if component C states property p as a weak upward requirement, then it is not allowed to have p provided by a component below C, but it is not necessary to have p provided by a component above C. A concrete example from the networking world would be a component that implements the Internet Protocol (IP), which may have a weak upward requirement called "transport". This means that there might be a transport layer in the protocol stack, but if it exists it has to be provided by a component above IP. Specifying *immediate requirements* is also possible, meaning that those requirements apply only to the adjacent layers, i.e., directly above or below.

The components may be complex building blocks that can have different functionality, depending on the context of their usage. In order to handle this situation, a component can be defined as being in a set of *roles*. Each role groups a list of related provided properties that impose its requirements towards the environment. At any time, a component may play a different role according to the context of its usage. There is one *basic role* for a component, while there can be a number of *alternative roles*. Only one alternative role can be active at a time. The component descriptors are stored in a component repository.

The composition algorithm tries to find a correct composition of components that fulfills a set of requirements. Essentially, the algorithm performs a top-down search by matching the requested properties with the provided ones. The match is only between current active roles of components: i.e., requirements in non-active roles are ignored. The compostion algorithm produces solutions as sequences of component descriptions. Assuming a set of client-specific configuration requirements R=R_1, R_2, ..., R_r, a succession of components C_1, C_2, ..., C_N (C_1 is in the top, C_N is in the bottom layer) represents a good composition, if:

1. All requirements R_i (1<=i<=r) are met, being present in the union of the provided properties list of all components C$_j$ in the sequence (1<=j<= N);

2. Each component C_i has its own downward requirements list DR_i accomplished by some components C_j below it (i<j and j<=N);

3. Each component C_i has its own upward requirements list UR_i accomplished by some components C_j above it (i>j and j>=1);

4. Additional imposed ordering restrictions are met;

5. There are no contradictions with respect to weak requirements.

The composition algorithm uses a propagation of requirements during its search: it delegates the responsibility for providing certain requirements posed on a component to other components.

2.16.1 Evaluation and relevance to this thesis

The composition algorithm reviewed in this section represents a generic non-reflective approach to combining components with dependencies on each other. The notions of strong, weak, and immediate UP and DOWN requirements are present in the MOP descriptors of IGUANA version 3.

2.17 Aspect-Oriented Programming and Reflection

Aspect-oriented programming (AOP) [40, 39, 17] consists of a component language to program functionality and one or more aspect languages to code concerns, and an aspect weaver, which combines them and generates the final code. Programs are separately built using different languages. AOP allows non-orthogonal composition (i.e., having overlapping concerns) and delegates the resolution of conflicts to the aspect weaver.

AOP can use reflection for aspect weaving [71]. On the other hand, reflection "meshes well with AOP in two main areas: first, reflective techniques appear amongst the most promising to build aspect weavers that would be both general and extensible. Second, AOP appears as a promising structuring tool for reflection as more and more aspects come into play in reflective descriptions of complex, distributed systems and programming languages." - from [48].

AspectS [32] for example, provides an environment to allow for experimental AOP. It is based on Squeak/SMALLTALK and draws on AspectJ [38] and MethodWrappers [6]. AspectS supports coordinated meta-level programming, addressing the "tangled code" [40] phenomenon by providing aspect-related modules. AspectS is realised without changing either the syntax or the virtual machine of SMALLTALK: it uses MO composition instead of code transformation. Aspects[4], implemented as regular classes in AspectS, are units of modularity that represent implementations of crosscutting concerns. Aspects associate code fragments with join points by the use of advice objects. These code fragments are executed whenever a join point is encountered. As a fundamental concept in SMALLTALK is message sending, an aspect in AspectS may refer to a set of receivers, senders or sender classes. These objects are added or removed by the client code and used by the woven/composed code at run-time.

Join points are well-defined points in the execution of the code. Join point descriptors in AspectS name a target class and a target selector, which is then used by the weaving process to apply computational changes to the base system. Join points of a pointcut can be enumerated statically, or collected at run-time by queries.

Advice objects associate code fragments (blocks) with pointcuts and their respective join points descriptors that describe targets for the weaving process to place these fragments into the system. An advice has to be qualified to state whether the woven/composed code will be receiver or sender, or class or instance aware. Call flow semantics can also be specified.

AspectS allows the execution of the following kinds of crosscutting concerns:

- before and after the execution of a method invocation;
- around method invocations (put in front of the actual compiled method);
- handle signalled exceptions (around a message send);

It is also possible to introduce new behaviour to the target clients.

[4]The AOP terminology used in this section is the one used by AspectS, not by the wider AOP community.

AspectS employs a run-time weaver process to transform the base system according to the aspects involved. The woven code is based on method wrappers and meta-programming. Method wrappers allow the introduction of code that is executed before, after, or instead (around) of an existing method. Method wrappers change the objects that the standard method lookup process in SMALLTALK returns. A method wrapper replaces an entry in the method dictionary of the class, adds behaviour to the method invocation, and eventually invokes the wrapped method itself. AspectS coordinates the placement of block method wrappers into method dictionaries of the receiver classes stated in various join points advised by the aspect. The weaving process runs every time an aspect instance is installed. An aspect can be installed by sending an install message to the aspect object. Similarly, an aspect can be uninstalled by sending an uninstall message to the corresponding aspect object. Thus, weaving and unweaving is completely dynamic in AspectS. Method wrappers are placed around a compiled method in such a way that their activation will happen in the following order [5] (considering the kinds of wrappers and the hierarchy of their originating aspects):

- Around advice/wrappers (most specific first);
- Before parts of before-after advice/wrappers (most specific first);
- Handler advice/wrappers (most specific first);
- Compiled base method;
- Handler advice/wrappers (least specific first);
- After parts of before-after advice/wrappers (least specific first);
- Around advice/wrappers (least specific first).

It is important to note that weaving/unweaving process runs only once, not during actual message sends. An advice is more specific than another if it is defined in an aspect that is more specific than the aspect the other advice defined in. If two advices are either defined in the same aspect, or if their aspects are not related directly or indirectly though class inheritance relationship, the specificity between them is undefined.

[5]This is similar to the effective method lookup in CLOS

2.17.1 Evaluation and relevance to this thesis

AOP presents a solution to the problem of dealing with cross-cutting non-functional concerns. We selected AspectS as a representative of AOP because it uses run-time dynamic weaving process based on Squeak/SMALLTALK.

However the paper does not specify how unrelated (in terms of aspect class inheritance) advices are woven. Thus, it does not deal with the composition of semantically overlapping aspects.

2.18 Reflective middleware

Fault-tolerant (FT) CORBA [41] is a specialised run-time MOP using compile-time reflection (Open C++ version 2). It provides a mean to dynamically attach fault-tolerance strategies to CORBA objects.

OpenCorba [44, 45] is a reflective open Object Request Broker (ORB), which enables users to adapt dynamically the representation and execution policies of the software bus. Reflection allows the extension of the initial OMG CORBA model with libraries of MOPs customising mechanisms of distributed programming. Thus, it is possible to introduce in a transparent way new semantics on the initial model such as concurrency, replication, security, etc., including semantics currently not thought of. OpenCorba's implementation is based on NeoClasstalk, which is in turn a result of applying a MOP to the Smalltalk language. NeoClasstalk presents an efficient solution for handling message sending and receiving as well as a way of achieving dynamic behaviour of a class.

[61] summarises the research being done in the University of Lancaster on a reflective component-based middleware, called OpenORB [3, 13, 4]. It uses OpenCOM, a component technology that is closely based on Microsoft's COM, but it is enhanced with richer reflective facilities. OpenCOM avoids dependencies on features of COM such as distribution, persistence, security, and transactions. OpenCOM is applied to both the application level and the middleware infrastructure. In order to address the need for adaptability, the middleware is reflective such that it helps facilitate and manage run-time changes in component configu-

rations. It incorporates structures representing aspects of the middleware itself and offers meta-interfaces for inspecting and adapting these reified aspects. *Provides* as well as *requires* relationships between OpenCOM components are made explicit. The IReceptacles interface is defined, through which required interface references can be passed, so that connections can be established by a third component. Each OpenCOM component has an IMetaEncapsulation interface providing meta-information about the interface types. This meta-information is used to support dynamic invocation of arbitrary interfaces. OpenCOM also supports *interception* at specified interfaces. In particular, components implementing IMetaEnvironment interface enable dynamic attachment/detachment of interceptors, that insert wrapping behaviour around method invocation.

The authors had found that designing these meta-interfaces was not easy. The approach, in which components of the middleware configuration would be represented as a graph, and manipulation of the graph would result in the reconfiguration of the middleware simply would not be robust enough. Instead, they proposed the use of *Component Frameworks* (CF), where a CF is a collection of rules and contracts that govern the interaction of a set of components. CFs typically address a specific and focused problem domain (e.g., buffer management, binding establishment), and thus many CFs may need to be integrated in a component system. While CFs are design-level entities, they also have an explicit run-time representation, called *Component Framework Representative* (CFR). Meta-interfaces exposed by the different CFs are typically implemented by CFRs, which maintain information about the current configuration and apply it to perform inspection and adaptation.

The CFs in the proposed architecture are organised in three layers, wherein components are only aware of interfaces/CFs defined at layers below themselves. The three layers are:

Binding layer: This layer contains a *binding CF*, which defines a set of interfaces, rules, and semantics that govern the collaboration between to-be-bound components, binder components, and the CFR itself. Binder components are responsible for marshalling and unmarshalling interface references, and producing the required proxy/stub/protocols infrastructure.

Communication layer: This layer contains components/CFs that are used by binders to establish required communication paths. Minimally, it contains the *protocol CF*, which defines an architecture for dynamically composing and reconfiguring protocol stacks using lightweight protocols. The communication layer may contain additional CFs depending on the needs of the binding layer.

Resource layer: This layer contains a collection of components/CFs that provide a uniform application programming interface for using and controlling low-level resources. This layer minimally contains the *buffer management CF* and the *transport management CF*. The associated CFRs provide base interfaces for buffer allocation and operating system-level transport services, respectively. An optional *thread management CF* can handle user-level threads.

COM+, Enterprise JavaBeans, and CORBA Components all support similar container-based models for building distributed applications. The significance of these architectures lies in that they achieve a separation of concerns between the functional aspects of the application and the non-functional aspects that are managed by the container (distribution, concurrency, and transactions). The drawback is that the configurability of the non-functional aspects is severely limited. The implementation of the container services is hidden and out of control of the application developer.

2.18.1 Evaluation and relevance to this thesis

We reviewed representatives of reflective middleware solutions. We believe that IGUANA version 3 is a strong candidate to be used for building reflective middleware.

2.19 Summary

This chapter described the related work on reflective programming languages and meta-level architectures, as well as it reviewed various non-reflective approaches to automatic and/or dynamic composition. We evaluated the reflective solutions according to a common set of criteria and stated how is IGUANA version 3 related to all of the reviewed reflective and non-reflective systems.

Chapter 3

The IGUANA Model for Automatic and Dynamic MOP Composition

"Only through hard work and perseverance can one truly suffer." - *source unknown*

3.1 Introduction

This chapter presents the design of our reflective programming model, which provides a solution to the problem of automatic and dynamic MOP composition in a compiled programming language.

Instead of defining a radically new reflective model for implementing automatic and dynamic MOP composition, the IGUANA version 3 model evolves the previous two IGUANA models, versions 1 and 2. In IGUANA version 3, multiple, independently-developed MOPs can be dynamically combined at run-time in order to provide reflective objects in a base-level application with a composed (non-default) behaviour, provided that the MOPs were developed according to the design rules (methodology) for composability, that we define shortly.

The motivation for IGUANA version 3 and a description of the previous two versions together with their shortcomings can be found in section 1.5, and section 2.7, respectively.

3.2 The Overall Approach

One objective for the design of the new IGUANA model was to come up with answers to questions that had arisen when we attempted to combine our Persistent [31, 65] and Remote MOP implementations, written in IGUANA/C++ version 2 (see section 2.7.2):

1. What if we do not want to "export" (i.e., to make them remotely accessible) certain objects but still want to avail of object persistence for them?

2. What if we want to turn a *persistent* object into *remote* one on the fly? And then back?

3. What if we decide to make a selection of *remote* objects *persistent* but not all of them?

4. What if we want to *mix in* other MOPs that are known to be composable with Persistent and Remote, for example, Synchronised?

5. What if we want to select other MOPs whose names are not known at compilation time?

IGUANA version 2 can answer some of these sample questions, but through its single MOP selection policy it implies that the meta-level programmer has to write a new MOP for every possible MOP combination, such as Persistent, Remote, PersistentRemote, PersistentSynchronised, PersistentRemoteSynchronised, and so on. In case of a large number of MOPs, this might lead to an explosion of MOPs; one for each possible combination. Thus, the IGUANA version 2 solution presents a static, design-time solution to combine MOPs. For system-level applications that must be capable of dynamically adapting to the changes in the environment, we really need a dynamic solution, which can compose MOs of a set of independently-developed but composable MOPs at run-time, when the actual content of the MOP set becomes known.

To answer the questions listed above, the new IGUANA model reintroduces[1] the ability to select multiple MOPs and it adds a mechanism that can automatically compose the MOs of the MOPs at run-time, whenever it is possible. The set of MOPs currently selected (which defines the current MOP *set*) can also be changed at run-time as MOPs can be selected and

[1] IGUANA version 1 had this feature, but version 2 opted for single MOP selection as a means of providing meta-typing.

deselected. Of course, certain conditions regarding what MOPs can dynamically be selected or deselected need to be identified at design-time and enforced (monitored) at run-time.

In order to develop a composable MOP, the MOP designer (i.e., the meta-level programmer) has to follow the design rules (methodology) that we outline in section 3.4.1. The MOP designer needs to position the new MOP in the global and extensible hierarchy of non-functional concerns, and he/she needs to know the set of middleware components the MO classes of the MOP use. Knowledge of design and implementation details (e.g., the MO classes and their inheritance relationships) of other MOPs are also required, but only in cases when the new MOP extends or derives from them.
We have adopted a working method, suggested in [53], to solve the general MOP composition problem:

1. Show simple example MOPs and their desired/meaningful combination(s). (See section 4 for details).
2. Propose a new composition model that works with the examples.
3. Study the inherent constraints of the model.
4. Elaborate a methodology for using it.

In summary, we take the following approach to constructing an algorithm for automatic and dynamic MOP composition: behavioural MOs that comprise a MOP are semantically described in MOP descriptor documents, that are available at run-time. The MOP descriptor document characterises the *before* and *after* meta-computations (e.g., what happens in the invoke method of an Invocation MO, or in the send method of a Send MO) for MOs that comprise a MOP.

The new Composer reification category is used to reify the composition itself: we define a new abstract metaobject class Composer, which defines the interface for MOP composition. Any concrete Composer metaobject class has to derive from the Composer class, and implement the methods defined for MOP composition.

We define an automatic MOP composition algorithm (see section 3.4.7), which we implement in the DefaultComposer class. If the application programmer decides to use the default, automatic composition, then an instance of the DefaultComposer class is created at run-time, when the application starts up. As part of the start-up code for an IGUANA application, this metaobject reads the MOP descriptor files of the MOPs that are present in the MOP sets of reflective classes, and makes decisions on how to combine the behavioural MOs of a particular MOP set. Upon changes to the MOP set for reflective (base-level) instances or classes, this metaobject may read additional MOP descriptor files, re-run the composition algorithm, and recompose the MOs. Unlike previous versions of IGUANA, the IGUANA version 3 run-time can also load compiled MO class code (from a shared library specified in the MOP descriptor file) dynamically by using the dynamic linking interface. MOP descriptors are written in the eXtensible Markup Language (XML) by a meta-level programmer, who is either the author of the MOP code or has reasonable knowledge about the MOP implementation details.

Automatic composition of metaobjects may not always be possible. In this case the model also allows manual composition by supporting the use of purpose-built Composer metaobjects, which can be used to replace the default, automatic composer. Composer metaobjects can dynamically be added at run-time to the meta-level configuration of reflective objects in order to switch metaobject composition from automatic to manual. They can also be replaced at run-time.

Behavioural MOs within the same reification category are organised into a list and invoked in a sequence (this is the chain of responsibility model, as described in section 1.2). In most of the practical MOP implementations, behavioural metaobjects are inherently linked to proprietary or standard middleware components that implement the non-functional behaviours, e.g., they provide a persistent object store, a name service, or a reference manager. The MOP descriptors have a section that specifies the links between the MOPs and the middleware components they use. This information is then used at run-time to (re)configure the middleware. Semantic interference between MOPs is detected when MOPs in the MOP set use common sets of middleware components.

3.3 Assumptions

The following assumptions were made, when we designed the new IGUANA model:

1. **Component-based middleware** The *middleware*, which is used by MOPs to implement their particular functionality (e.g., object distribution, transaction monitoring), is component-based. This means that the middleware is either componentised or constructed from components; each component has a well-defined software interface. We used our proprietary component-based middleware for this work. Part of our middleware is based on the Tigger Generic Run-time (GRT) framework [7] for distributed object services. Design of an extensive IGUANA MOP suite and the outline of the supporting middleware can be found in Chapter 4.

2. **Design rules for composable** MOPs The meta-level programmer adheres to the rules governing the design of new MOPs and their MOs; i.e., he/she writes a new MOP according to the programming methodology outlined in section 3.4.1, which can ensure composability with existing and future MOPs. For example, the source code of composable MO classes of a MOP shall not refer explicitly to MOs of other MOPs. Instead, the generic next reference should be used when referring to the next MO in the chain of MOs within a particular reification category.

3. **Accurate** MOP **descriptors** The meta-level programmer also writes accurate descriptors for his/her MOPs that will be used by the composition algorithm described below. The definition of the MOP descriptor files can be found in section 3.4.6.

4. **Seamlessly combining Extension Protocols** IGUANA *extension protocols*, which provide base-level programmers with an application programming interface (API) to work with the functionality provided by MOPs, are expected to compose seamlessly: i.e., although the extension protocols may not be orthogonal, they complement each other and do not clash. Although this might be a strong assumption, we have made it in order to simplify the problem, and to allow us to focus our attention on the MOP composition. As an illustrutation to this issue, both Persistent and Remote extension protocols offer a Name Service interface, in which names can be recorded/looked up for persistent and remote objects, respectively. For persistent *and* remote objects, the base-level programmer should use one single extension protocol (implemented by the Iguana class in our

implementation), that unifies the two extension protocols for working with persistent and/or remote reflective objects.

5. **Proxy objects in remote clients** MOPs related to object distribution (e.g., Remote, Encryption, Authentication) use proxy objects in the client's address space to represent remote servers. Proxy objects interact with the server object over a communication network. We provide two concrete MOPs, RemoteProxy and Remote for proxy and server objects to select, but the meta-level programmer is free to provide his/her own implementation. Both of our MOPs use the same set of MO classes, but have different MOP descriptors to indicate to the Composer the proxy/server role they play in the implementing remote communication. One has to notice that in order for the proxy and server objects to communicate succesfully, the Composer has to build a "symmetric" stack of MOs between the two sides.

   Naturally, there are other MOPs, whose functionality is not related to object distribution. We call them "local" MOPs.

6. **One single meta-level** The meta-level of an object consists of one single layer only, in order to limit complexity. However, more than one MOP can be part of the single meta-level. This model is different from the usual "meta-tower" [75, 50, 74, 49] or "onion-skin" [1] model of reflection, where meta-layers can be built on top of other meta-layers.

   Whether an IGUANA MO *class can select a* MOP *is an interesting question that is for further study.*

7. **Composition of middleware components is out of focus** A similar composition algorithm could be designed to compose middleware components, however it is out of scope for this thesis. Research has been done on component-oriented middleware and architectural reflection, refer to [5, 14, 61].

3.4 Description of IGUANA version 3

This section describes the design of the IGUANA version 3 reflective programming model in detail and shows examples of how the base-level and meta-level programmer can use it.

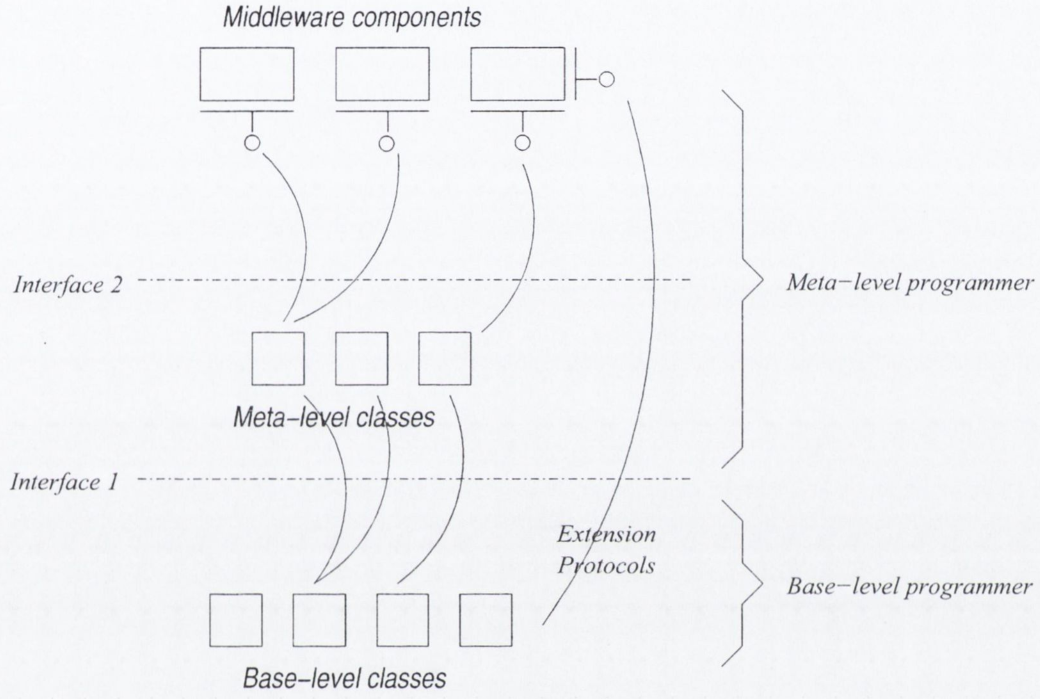

Fig. 3.1: The new IGUANA model with three layers (repeated)

The main features of the new IGUANA model can be summarised as:

Three-layer architecture with an explicit middleware layer The IGUANA version 2 model provides only a single meta-level [2], i.e., there are no meta-towers in IGUANA/C++. Although the new IGUANA model retains this restriction, it also adds an explicit middleware layer and an interface to interact with it from the base- and meta-levels. MOPs specify links to components of the middleware in order to identify semantic overlap between their metaobjects. The interface to the middleware layer is used to initiate reconfiguration of the middleware components. The new IGUANA architecture can be

[2]This was an implementation restriction only.

seen in Figure 3.1.

New reification categories Although the 29 reification categories of IGUANA version 1 were certainly an "overkill" [62] for the meta-level programmer, the reification category list of 12 (Class, Method, Attribute, Creation, Deletion, Send, Receive, Invocation, StateRead, StateWrite, Constructor, and Array) in IGUANA version 2 needed to be expanded. We have added two new reification categories to the model: ObjectReference and Composer.

The new ObjectReference is a structural metaobject class which reifies language pointers. This allows a MOP programmer to augment volatile language references to objects with more complex MOP-specific references that store additional referential information on reflective objects. For example, PersistentObjectReference used in our Persistent implementation contains a class identifier, an object number[3], and an offset value for embedded persistent objects. Thus, PersistentObjectReference can uniquely identify a persistent object regardless of its current location; i.e., whether it is in memory or in the user's Persistent Objects Store (POS). ObjectReference MOs are the only local (i.e., per-object) metaobjects; all the other MOs are shared between reflective instances of a class. Typically, MOs in the Creation and Deletion reification categories are responsible for creating and deleting ObjectReference MOs, respectively. In case of multiple MOP selection, where a number of MOPs reify ObjectReference, the base-level reflective object can have multiple ObjectReference MOs attached to it. The IGUANA run-time (the Composer MO more precisely) automatically adds or removes ObjectReference MOs when MOP selection or deselection happens, respectively.

The new Composer reification category reifies a meta-level concept: MOP composition itself. It is interesting to note that the other reification categories reify base-level concepts. Composer metaobjects allow the programmer to intercede in and control the MOP selection or deselection operation. Custom Composer MOs can be used to replace the default, automatic metaobject composition mechanism, implemented by the DefaultComposer class.

[3]A sequence number generated by the Persistent Objects Store.

Description (classification) of behavioural metaobject classes The behavioural MOs (i.e., Send, Receive, Invocation, Creation, Deletion, StateRead, and StateWrite) have been refined with the ability to describe what they do in their "before" and "after" meta-computation. A particular "before" meta-computation might cause a problem with the composition, for example, it may not call the next metaobject in the chain. Such conditional metaobjects are difficult to compose, because they can break the chain of responsibility model for composition, which is used in this work.

MOP **selection** The new model allows the programmer to select multiple composable MOPs for base-level classes and instances. See section 3.4.1 for guidelines on writing composable MOPs. For example, the following code excerpt shows class MOP selection:

```
class A ==> Remote, Persistent {...};
```

This code defines the initial MOP set for reflective class A and results in instances of class A having the combination of Remote and Persistent MOPs selected. These MOPs have been defined as composable and their constituent MOs can be composed automatically. The names of the MOPs are known at compilation time.

The multiple MOP selection syntax in this example is identical to that of the IGUANA version 1 model. Similarly to the previous IGUANA versions, programmers can select new protocol(s) for instances of reflective classes. This is called instance protocol selection. Section 3.4.3 gives more examples of default, class, and instance protocol selection.

Dynamic MOP **selection** The MOP set for a reflective class or instance can be enhanced at run-time by selecting additional MOPs. For example, the following code excerpt:

```
A* a= new A() ==> ++OptimisticSync;
```

adds OptimisticSync MOP (which implements optimistic concurrency control) to the MOP set of a new instance of class A, which already contained MOPs Remote and Persistent.

Dynamic MOP deselection Previously selected (default, instance, or class) MOPs can also be deselected. Attempting to deselect a MOP that has not been selected will result in a null operation (i.e., there will be no exception raised). The MOP deselection syntax is identical to that of IGUANA version 1. For example, the following code excerpt:

```
B* b ==> --Logging;
```

deselects the previously selected Logging MOP for a heap-allocated instance b of class B, previously created.

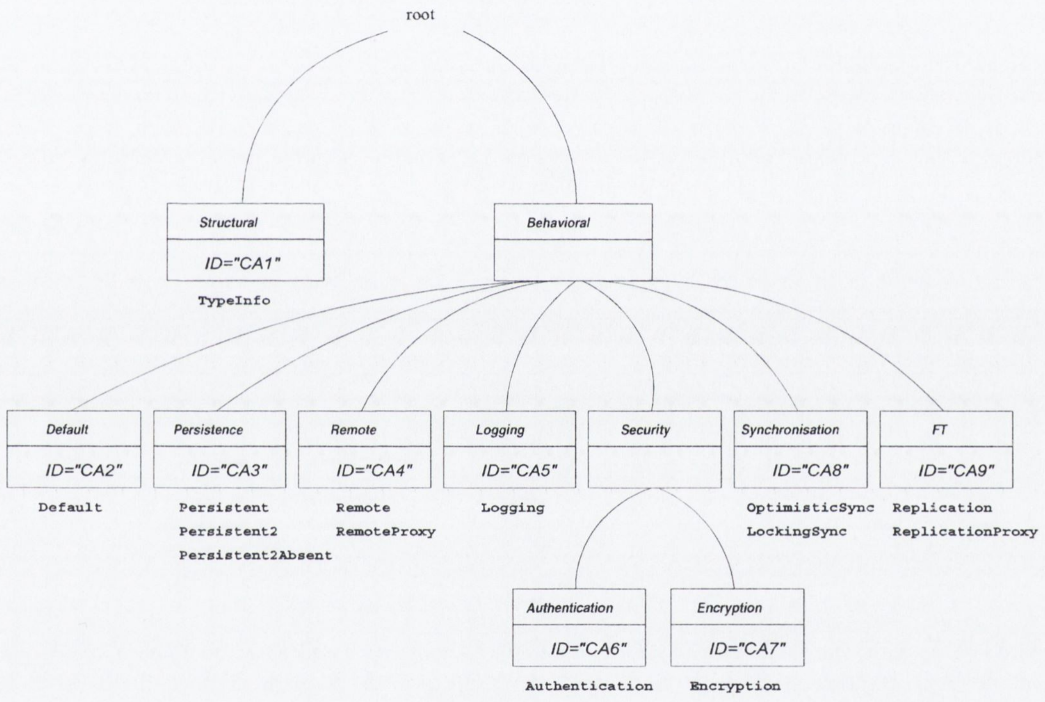

Fig. 3.2: The hierarchy of non-functional concerns grouped into areas

MOP descriptors The automatic composition, which is implemented by the default DefaultComposer metaobject, is guided by the MOP descriptors. MOP descriptors are written in XML, and explained in Section 3.4.6.

Non-functional concern hierarchy We place our MOPs implementing common non-functional concerns in a hierarchy, see Figure 3.2. Placing any new MOP in this extensible structure is part of the methodology for writing composable MOPs. Figure 3.2 shows the position of the MOPs from Chapter 4. For example, both OptimisticSync and LockingSync are in the Synchronisation concern area, which is underneath the Behavioural concern area, which is in turn underneath the root of the hierarchy. The hierarchy is extensible, i.e., new child concern areas can be added for new MOPs as needed.

3.4.1 Methodology for designing composable MOPs

This section describes the basic composition framework and design rules, which have to be followed in order to write composable MOPs.

MOPs are implemented by a set of related MO classes and possibly other auxiliary (helper) classes. Each MO class represents one of the 14 IGUANA reification categories. The MO classes can be divided into two groups: structural and behavioural. The structural MO classes are as follows:

- MClass maintains information on a base-level class such as its name, super-classes (if any), size, virtual function table, attributes, constructor(s), and other methods.
- MAttribute maintains information on an attribute of a base-level object such as the name, type, address, size, "staticness", and accessibility (i.e., public, protected or private).
- MMethod maintains information on the name, signature, return type, address, "staticness", and accessibility of a method.
- MConstructor maintains information on the signature, address, and accessibility of a constructor. Unlike IGUANA version 2, we support non-default constructors.
- MArray maintains information on the type and size of elements, size, and address of an array.
- MObjectReference this class forms the basis, on which MOP-specific per-object referential information can be maintained in subclasses. MObjectReference instances provide an extensible object header, in which referential information is stored. Instances of the MObjectReference class hold a pointer back to the reflective base-level object as well as

the name of the related MOP. Reflective objects can have zero or many MObjectReference MOs associated with them. The Composer metaobjects are responsible for inserting or removing object references, created by the relevant Creation MOs, when a MOP is added or removed, respectively. For example, a base-level object that selected both Persistent and Remote MOPs, will have two object references, namely PersistentObjectReference and RemoteObjectReference.

The behavioural MO classes are all abstract classes that are used to provide an interface through which concrete subclasses can define how the operation should be performed at the base-level. As our focus is the automatic and dynamic composition of the behavioural MOs at run-time, each MO class has a pointer (next) to the next MO of the same reification category. Composer metaobjects are responsible for combining MOs in each behavioural reification category and setting their next pointers accordingly. The following behavioural MO classes can be used by MOP designers:

- MCreation: concrete subclasses can define the way an instance of a reflective class is created.
- MDeletion: concrete subclasses can define how a reflective object is deleted.
- MInvocation: concrete subclasses can define how methods on a reflective object are invoked.
- MStateRead: concrete subclasses can define how attributes of a reflecive object are read.
- MStateWrite: concrete subclasses can define how attributes of a reflective object are written.
- MSend: concrete subsclasses can define how method invocation on an object (which is not necessarily reflective) is initiated from within a method of a reflective class.
- MReceive: concrete subclasses can define the way an incoming method invocation is dispatched to the right method (interesting for virtual methods).
- MComposer: concrete subclasses can define the way MOPs are composed. We provide a default implementation (i.e., the DefaultComposer MO class), which supports automatic and dynamic MOP composition.

Chapter 4 shows the design of an IGUANA MOP suite and illustrates how these MO classes can be used to implement common non-functional concerns.

The automatic and dynamic MOP composition works only if the meta-level programmer who implements a new MOP adheres to the following design rules:

MOP **XML descriptor:** this document should faithfully reflect the design of the MOP and its MO classes. For example, the MOP designer must describe the composition characteristics of each behavioural MO class with respect to whether it calls the next MO in the chain (if any) or calls a middleware component.

Concern aera: the new MOP, which is being designed to implement a new non-functional concern, has to fit into the global hierarchy of non-functional concern areas (see Figure 3.2). This means that either the new MOP has to fit into an existing concern area, or a new concern area should be created and then placed somewhere in the global and extensible hierarchy. The hierarchy is managed by meta-level programmers who design MOPs. The current hierarchy in the Figure shows how the IGUANA MOPs described in Chapter 4 are positioned.

Concern area exclusiveness: depending on the concern area, the new MOP should be declared as either Singleton or Multi. The former means that this MOP is the "exclusive" (sole) implementor of this non-functional concern, while the latter allows multiple implementations of the same non-functional concern to co-exist.

Permitted base-level modifications only: Through the inherent links between MO classes, a behavioural MO can in theory modify any aspects of the base-level objects. For example, an Invocation MO could alter the way base-level objects are created. But in order to make automatic composition work, we require that a behavioural MO performs only the base-level modifications permitted for its reification category (see Table 3.1).

Chain of responsibilites model: composable behavioural MO classes must have their "before" and "after" logic separated. Within their code, explicit references to instances to other MO classes must not be used. Instead, the generic next reference should be used.

If a specific MO has to be the next MO in the chain of MOs, the MOP XML descriptor should be used to specify the next MO as requirements on the "after" or "before" MOs (see BeforeRequirements and AfterRequirements elements of the DTD in Section 3.4.6).

"Remote" MOPs: the meta-level programmer should decide whether the new MOP is related to object distribution or not (i.e., is local). If the MOP is "remote"-related, its design has to fit in to the model of object distribution, which follows the proxy design pattern.

Middleware links: the meta-level programmer should (re)use as much of the IGUANA middleware component library as he/she can. Dependencies on middleware components should be listed in the MOP XML descriptor.

Extensible object references: subclasses of the MObjectReference class should be used to hold MOP- specific per-object referential information. For example, our RemoteObjectReference carries sufficient information to identify the remote server object: object identifier, creation time, IP address and TCP port number of the server application.

3.4.2 Automatic and dynamic MOP composition: Composer metaobjects

The automatic and dynamic MOP composition algorithm, as it is implemented by the DefaultComposer class, is used for MOPs that are declared as automatically composable. The composition algorithm is described in Section 3.4.7.

Figure 3.3 shows how the default, automatic and dynamic composer works: it reads the MOP descriptor files, runs the composition algorithm, which arranges the MOs in each reification category in a list, assuming that the selected MOPs are all automatically composable. The intercepted (reified) base-level operations are directed towards the first MO in the chain, which in its "before" operation may call related middleware component(s) and the next MO in the chain. For example, the RemoteInvocation MO in the client calls the CommunicationManager middleware component in the "before" part of the invoke method in order to send a method invocation operation over the network.

By convention, the last MO in the chain is always the Default MO, which reflects the base-level operation. For example, DefaultCreation::create method creates a new heap-allocated instance of the reflective class. This convention was adopted from IGUANA version 2. Please note that the Default MO may not be called.

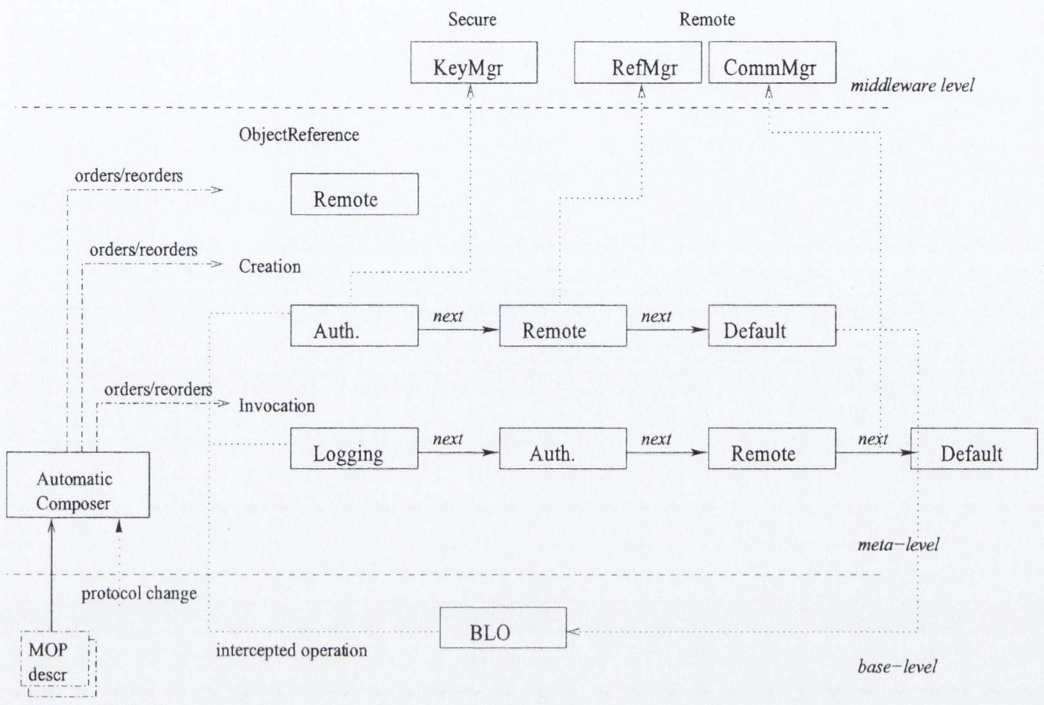

Fig. 3.3: Automatic MO composition example for combining Logging, Authentication, and Remote at the client side.

The meta-level programmer is also free to write his/her own Composer metaobject, that can be used by the base-level programmer in cases when the selected MOPs require manual composition or the base-level programmer wants to achieve a non-default composition semantics (e.g., concurrent invocation of MOs instead of the default sequential invocation).

The Composer MO is responsible for composing MOs from the (dynamically) selected MOPs. The IGUANA run-time calls the Composer MO at run-time, whenever MOP selection or deselection occurs at the base-level. The Composer MO is given information on the selected MOPs (i.e., their MOP descriptors), and it is expected to arrange the behavioural MOs in chains (one chain for each behavioural reification category) and set their next references according to their order in the chain. It should also insert or remove MObjectReference MOs according to the MOPs selected or deselected, respectively. In case the MOs cannot be composed, the Composer MO is expected to throw an exception (i.e., a CompositionException) back to the base-level application.

In some cases, the next reference can refer to the user-defined Composer MO itself. Here we give an example of a manual Composer MO implemented by the ManualComposer class, which also acts as a "super" behavioural MO; i.e., the ManualComposer class not only inherits from the MComposer abstract class, but it also inherits from MCreation and MInvocation MO classes, virtually. Thus, an instance of ManualComposer can also insert itself (similarly to Guaraná) as the first MO in the chains for the Creation and Invocation reification categories.

Unlike the DefaultComposer MO, which is active during MOP composition only, the above ManualComposer MO is involved in the intercepted creation and invocation operations. This ManualComposer is also free to combine the results of these operations returned from the Creation and Invocation MOs. This model is similar to Guaraná composition with one main difference: in IGUANA, there can be at most one Composer MO in operation for any base-level object.

Figure 3.4 illustrates how the above ManualComposer MO may decide on an alternative arrangement of MOs, resetting their next reference, and calling them in an order that is different from the default one, calculated by the DefaultComposer MO.

A Composer MO can be replaced at run-time by calling the MObject::replaceComposer method on the reflective object as each reflective class in IGUANA inherits from the MObject class. State transfer between the old and new Composer needs to be coded by the meta-level programmer. This can be done by providing an implementation of the MComposer::getState and MComposer::setState pure virtual methods.

For example, the following code excerpt creates a new MyNewComposer MO, which then replaces the DefaultComposer MO for a reflective class A. State transfer is arranged between the two Composer MOs. The base-level programmer then selects MOP1 and MOP2 for this class.

```
MyNewComposer* newComposer= new MyNewComposer();
newComposer-> setState(A::metaA-> mcomposer-> getState());
```

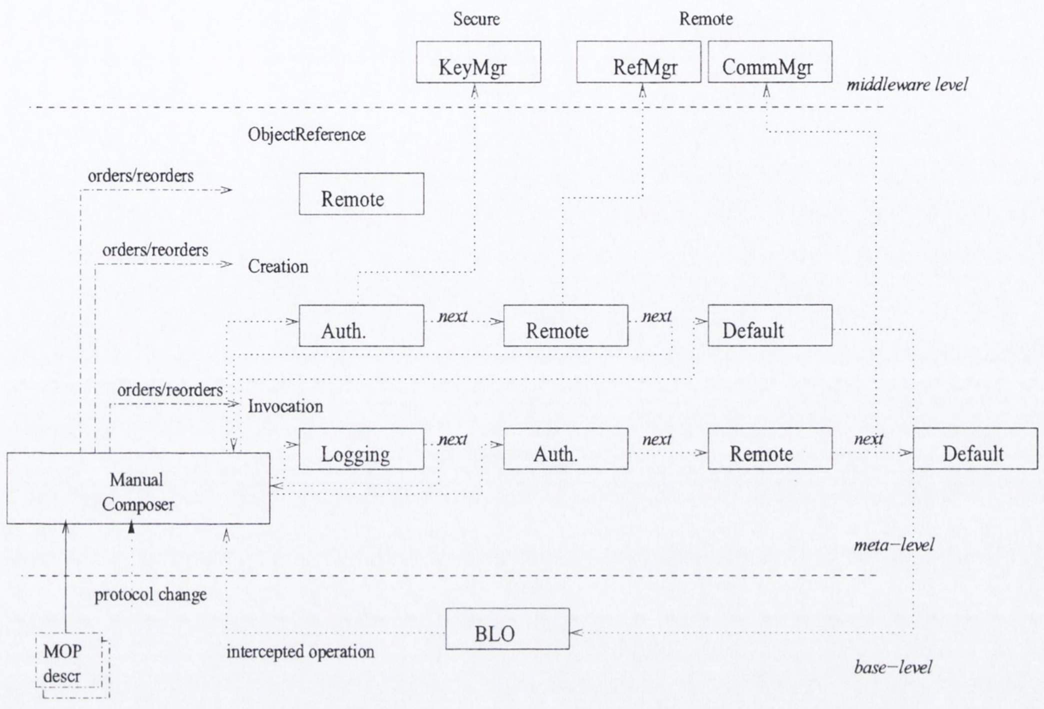

Fig. 3.4: Manual metaobject Composer in operation for combining Logging, Secure, and Remote at the proxy side.

```
A::metaA-> mcomposer= newComposer;
class A ==> MOP1, MOP2;
```

If only the Composer MO needs to be changed, the programmer can write the following statement in the source code:

```
YourNewComposer* yourComposer= new YourNewComposer();
a-> replaceComposer(yourComposer);
```

The "YourNewComposer" MO will replace the current Composer MO (represented by the mcomposer field of MObject) for object referenced by a: this time without the explicit state transfer.

IGUANA version 3 also allows a non-default Composer MO class, whose name is not known at compilation time, to be loaded dynamically. The non-default Composer implementation does not have to be statically or dynamically linked with the application. This is done by utilising the dynamic linking and loading interface provided by the UNIX operating system (see dlopen(3ld), dlsym(3ld), and dlclose(3ld) manual pages). The dynamic loading of a new composer from a shared library is implemented by the Meta::loadComposer method. This method loads the Composer MO class based on the name of its class, and the name of the shared library. We assume, that the shared library can be found on the LD_LIBRARY_PATH. If the composer is no longer needed, the Meta::unloadComposer method can be used to explicitly unload it. This feature is useful for dynamically adapting and extending long-running applications.

We make the MO composition an atomic operation: i.e., the reflective (base-level) object is temporarily locked during the MO composition. This is needed to avoid dynamically rearranging the behavioural MOs while the reflective object is being used (e.g., methods are being invoked on it).

3.4.3 Rules for MOP selection and deselection

This section revisits the MOP inheritance and selection rules of IGUANA version 2, and defines rules for multiple MOP selection/reselection for IGUANA version 3. Issues that are not addressed by IGUANA version 2 are put in to boxes.
The following rules govern MOP selection in IGUANA version 2:

§1. Every MOP defines a new meta-type. An object that selects a MOP is said to conform to or implement the meta-type.

§2. Every object has an associated meta-type (called its *current* meta-type), which can be changed dynamically.

§3. There are three ways of selecting a MOP: class, instance, and default.

§4. Class MOP selection results in a *static* meta-type. It means that all instances of the class implement this meta-type.

§5. Class MOP selections are inherited: a subclass inherits the meta-type of its parent class.

§6. Subclasses can override the MOP selection of its parent class. However, in this case, the specified meta-type must be the sub-type of the meta-type specified for the superclass.

Problem 1: what happens when we use multiple class inheritance and the superclasses select different MOPs?

§7. Default MOP selection allows the meta-type of all new instances of a set of classes declared in a single source file to be selected. The default MOP selection can be changed within the source file by repeating the

```
default ==> P1;
```

statement with a different MOP name, e.g., P2.

§8. Instance MOP selection: the meta-type of a single object can be changed dynamically. The new meta-type is called the object's *dynamic* meta-type. The dynamic meta-type of an object must be a sub-type of its static meta-type.

Problem 2: what happens if the object does not have a static meta-type (i.e., it is not reflective)?

§9. Protocol inheritance: the derived MOP includes the full set of reification categories specified by both the base and the derived protocols.

§10. There are two sharing modes for MOs in reification categories: *local* and *shared*.

Problem 3: there are no specific rules covering the effect of dynamic meta-type change on local/shared metaobjects. For example, what happens if the reflective object has a mixture of local and shared MOs?

§11. IGUANA version 2 "supports automatic meta-level composition by means of multiple MOP inheritance". The chain of responsilities model is used. If there are multiple MOs within the same reification category, MOs from the super-protocols are put towards the end of the chain. Thus, the more specific behaviour (defined in the derived protocols) is executed in preference to the more general behaviour (defined in the super-protocol).

Problem 4: the exact way in which local/shared MOs are composed is not addressed in the thesis.

§12. To prevent the same code executing multiple times, only one MO of a specific reification category is added to the chain. This is similar to the C++ mechanism for handling virtual multiple inheritance.

§13. Manual/explicit composition: the meta-level programmer can manipulate the next MO references in metaobject classes in order to combine MOs explicitly.

§14. A MO composition algorithm is executed at run-time every time the meta-type for an object changes. This algorithm automatically creates and/or deletes MOs according to the old and new MOPs.

In IGUANA version 3, the following rules control MOP selection and deselection.

§1. There is no strong meta-typing in IGUANA version 3. In other words, multiple meta-types are allowed simultaneously. Thus, a reflective object can be associated with multiple MOPs, if and only if these MOPs were developed according to the *composability* rules defined in section 3.4.1. MOPs must be accurately described by the meta-level programmer in separate MOP descriptors.

§2. Every object has an associated MOP *set*, which can be changed dynamically. Non-reflective objects have an empty MOP set.

§3. IGUANA version 3 retains all three forms of MOP selection: default, class, and instance.

§4. Class MOP selection results in an *initial* MOP *set*. For example the following statement:

```
class A ==> Remote, Persistent {...};
```

results in instances of class A having the combination of Remote and Persistent MOPs selected, assuming that these two MOPs are composable. If there is a default MOP selection, the above class MOP selection will override it.

The following example shows how a programmer would define the default MOP set such that it contains Persistent and Remote and how the default MOP set could be extended with a third composable MOP Authentication for instances of class A:

```
default ==> Persistent, Remote;
class A ==> ++Authentication {...};
```

Furthermore, class MOP selection can also be changed: composable MOPs can be added to, or previously selected MOPs can be deselected from the current MOP set of the class. The scope for this change is the running application, i.e., the change in the MOP set will only affect new instances of the class, created in the running application. The reason for this restriction is that otherwise the IGUANA run-time would need to keep track of all previously created instances of a reflective class. However the base-level programmer is free to keep a list of reflective instances of a particular class and change the MOP set in a loop iterating through the list. Furthermore, if object creation is reified in a MOP that is being added to the MOP set, the Creation MO would not be called for existing objects as they had already been created. Thus, it is problematic to re-initialise existing objects (e.g., calling related middleware components, or extending their ObjectReference). To overcome this problem, we added support the creation of ObjectReference MO, without the creation of the base-level object, see §8.

§5. A class inherits its MOP set from its super-class(es). Therefore all MOPs in the MOP set must be composable. If MOPs are not composable, the Composer MO at run-time will raise a Composition exception, that should be handled at the base-level.

§6. Subclasses can override the initial MOP set: composable MOPs can be added, and removed.

§7. Default MOP selection allows the initial MOP set for all new instances of a set of classes defined in a single source file to be defined. In the following example, classes of A and B will have MOP1 and MOP2 selected:

```
default ==> MOP1, MOP2;
class A {...};
class B : public A {...};
```

The default MOP set can be changed in the same source file, e.g., instances of class D in the following code sample will have MOP1, MOP2, and MOP3 selected:

```
default ==> MOP1, MOP2;
class A {...};
class B : public A {...};
...
default ==> ++MOP3;
class D {...};
...
default ==> MOP3;
class E {...};
class F : public A {...};
```

Instances of class E will have only MOP3 selected. However, instances of class F will have all of the three MOPs selected because a class inherits its MOP set from the parent class(es). In this case, all members of the MOP set have to be composable.

§8. Instance MOP selection: the current MOP set of a single object can be changed dynamically. The change can be the addition, removal or replacement of MOPs. The condition of a successful change to the current MOP set is that the MOPs in the new MOP set must be composable with each other. This means that any pair of MOPs must be composable.

The following code excerpt creates a persistent instance of class B and dynamically extends the meta-type for the instance by adding the composable Remote MOP to the MOP set. Later in the code, the same MOP is removed from the MOP set.

```
class B ==> Persistent {...};
B* b= new B();
B* b ==> ++Remote;
...
B* b ==> --Remote;
```

A problem is that by the time we add the Remote MOP to the reflective object b, the object has already been created, thus it is difficult to extend its persistent object reference with remote object reference without calling the RemoteCreation MO, which is expected to initialise the RemoteObjectReference. We need a mechanism that allows the "late" insertion of object references to solve this problem. We overcome this problem by adding a new MCreation::lateCreate method to the MCreation abstract class, which is intended to create and insert the object reference. Furthermore, we need a second mechanism, which allows the removal of RemoteObjectReference when the Remote MOP is removed. This is done by the composer MO removing the object reference for the reflective instance b.

§9. Multiple MOP inheritance is allowed. Like in IGUANA version 2, the derived MOP includes the full set of reification categories specified by both the base and derived protocols. Unlike in IGUANA version 2, the order of super-protocols is irrelevant. However, it is the meta-level programmer's responsibility to ensure that the MOPs are either automatically-composable or there is a user-defined manual Composer MO provided.

§10. All MOs instances are shared bewteen reflective instances of the same class, except ObjectReference and Composer MOs. ObjectReference MOs are always local. Composer MOs are either shared or local, depending on the selection context. Composer MOs are usually shared between reflective classes and instances of reflective classes. However, composer selection "inherits" the context of the MOP selection/deselection: e.g., if the scope of the MOP selection was a particular instance, then the same instance will get its local Composer MO associated and initialised. Thus, there is no need to specify locality (i.e., whether a MO is local or shared) in the MOP definition.

Note that the notion of *local* and *shared* metaobjects found in IGUANA version 1 and 2 has been eliminated for two reasons: firstly, the explicit links between the MOs and the components of the supporting middleware made the need for local MOs redundant (i.e., it is assumed that per-object information is stored in the middleware, not in MOs, with MObjectReference MOs being the only exception) and secondly, the resulting model simplified the MO composition problem.

§11. Support for automatic meta-level composition is provided by means of writing composable MOPs and relying on the default Composer metaobject. If the MOPs are not automatically composable, a user-defined Composer MO can be used to compose the MOs. Rules regarding the writing of composable MOPs are given in Section 3.4.1.

§12. Similarly to the mechanism in IGUANA version 2, which prevents multiple instances of the same MO class to be added to the list of MOs in a particular reification category, only one MO of a particular MO class in a reification category can be added to the list.

§13. Manual/explicit composition through manipulating next MO references in metaobject classes is not recommended. Instead, explicit requirements on ordering of MOs is supported in the MOP descriptor document. A manual Composer MO can also be supplied by the meta-level programmer.

§14. Either the default (automatic) or the user-defined Composer MO is called every time the MOP set of an object changes. The Composer MO is responsible for creating/deleting and ordering MOs in their respective (behavioural) reification categories. Similarly, ObjectReference structural MOs are added/removed by the Composer MO.

There are additional rules in IGUANA version 3 covering manual MO composition:

§15. In case of manual MOP composition, the meta-level programmer (the author of a MOP) has to delegate a Composer MO. This can be set in the Composer attribute of the MOP element in the MOP descriptor XML document. When multiple manually-composable MOPs are in the MOP set, it is the base-level programmer's responsibility to ensure that all of them have the same manual Composer MO specified in their MOP descriptors. In other words, we do not attempt to compose Composer MOs.

§16. In the case where multiple MOP inheritance is combined with manual MOP composition, the most specific MOP "delegates" the manual Composer MO.

§17. Unlike the previous versions of IGUANA, IGUANA version 3 allows the addition of a MOP to the MOP set, whose name and definition is not known at compilation time. Furthermore, the new MOP implementation does not have to be statically or dynamically linked with the application. This is done by utilising the dynamic linking and loading interface.

3.4.4 Multi-threaded reifed stack

The IGUANA version 2 introduced the concept of a reified stack implemented by the MStack class, which was used during reified method sends/invocations as well as state read and write access operations. The implementation of the stack allows pushing and popping arbitrary elements on to and from the stack.

There is one single stack instance created as one of the first steps in the auto-generated meta-level initialisation code inside the main method of every reflective IGUANA/C++ application.

However, this single stack is ultimately thread-unsafe: IGUANA version 2 does not provide any mechanism to synchronise access to the stack from multiple threads. Therefore, in a multi-threaded application (which is the norm for any serious system-level application) multiple threads sharing the same single stack without synchronisation can easily cause the application to fail or crash: e.g., one thread might pop the stack, while another thread is pushing an argument on to it.

Therefore we redesigned the reified stack in the new model and we have made it thread-aware and thread-safe. We also introduced exception handling: the templated push and pop methods throw exceptions to signal that the stack is over and underflown, respectively.

See section 5.5 for more information.

3.4.5 Base-level modifications

Table 3.1 summarises the base-level modification that can potentially be performed by different IGUANA behavioural MOs upon intercepting the operation at the base-level object.

Reification category	Reifies	Modifies
Invocation	Method execution at receiver	method address, arguments (order, value, number), method return value
Send	Method execution at caller	method address, arguments (order, value, number), method return value
Dispatch	Selecting the method at receiver	method address, arguments (order, value, number), method return value
StateRead	Member state read	member (offset, subscript, value)
StateWrite	Member state write	member (offset, subscript, in-value), member (out-value)
Creation	object creation	memory allocation, VFT init, constructor call, object address
Deletion	object deletion	destructor call, memory release
Composer	overwrites automatic composition	ordering of metaobjects

Table 3.1: Base-level modifications performed by IGUANA behavioural metaobjects

3.4.6 MOP descriptors

In the new IGUANA model, MOPs and their MO classes must be described by the meta-level programmer, who created and coded them according to the new MOP design rules/methodology (see section 3.4.1).

The MOP descriptors are used by the IGUANA Composer metaobject to dynamically compose metaobjects. They are also used by the IGUANA pre-processor, to generate standard C++ code for the reflective IGUANA/C++ application annotated (enhanced) with MOP selection, deselection, and other IGUANA statements.

In IGUANA version 2, a MOP definition contains the following information:

- the name of the MOP;
- the list of super-MOPs (multiple MOP inheritance is allowed);
- the list of the reification categories used together with the name of the reifying MO class (if it is different from the default), and the sharing mode of the MO (i.e., whether it is

local or shared);

The IGUANA version 3 model retains the previous MOP definition detail (except the distinction between local and shared) as well as adding the following items:

- user-defined Composer MO can be delagated. If not specified, the Composer MO for the MOP is the automatic DefaultComposer MO.
- dependencies on other MOPs(if any), in terms of non-functional concern areas. As was explained in section 3.4.1, each MOP fits into a concern area, which is part of a global and extensible hierarchy (see Figure 3.2). Note that the IGUANA version 1 model had the notion of dependencies as references to other MOPs;
- reference to extension protocol(s), if any;
- reference to the shared library, that contains the MOP implementation. IGUANA version 3 supports dynamic loading and initialisation of MOPs, whose name and implementation is not available when the base-level reflective application is compiled;
- links to proprietary (i.e., custom-developed) middleware components;
- ability to specify ordering constraints for before and/or after MOs. This lets the programmer specify different ordering in different behavioural reification categories.

The IGUANA version 3 MOP descriptor is XML (eXtensible Markup Language) based, and it contains all of this information. It is required that the meta-level programmer specifies the new MOP in XML. Figure 3.5 shows the XML Data Type Definition (DTD) defined for MOP descriptors. The new IGUANA/C++ pre-processor reads the XML-based MOP descriptors and generates standard C++ code for a reflective application.

We assume that the MOP descriptors used by reflective IGUANA applications are stored in one particular directory, e.g., $HOME/.iguana/mops. For simplicity, we also assume that the name of the MOP file reflects the name of the MOP: e.g., DefaultMOP.xml file defines the Default MOP.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Document Type Definition for Iguana version 3 MOP Descriptors -->
<!ELEMENT MOP (SuperMOPs?, ConcernArea, MetaObject+, ExtensionProtocol?, MiddlewareLinks?)>
<!ATTLIST MOP
Name ID #REQUIRED
Composability (Manual | Auto) #REQUIRED
Composer CDATA #IMPLIED
SharedLibrary CDATA #IMPLIED
Distribution (Local | Distributed | Common) #REQUIRED
Locality (Proxy | Server | Common | NotApplicable) #REQUIRED
>
<!ELEMENT SuperMOPs (MOPRef+)>
<!ELEMENT MOPRef EMPTY>
<!ATTLIST MOPRef
Name IDREF #REQUIRED
>
<!ELEMENT ConcernArea (ConcernAreaElement+)>
<!ATTLIST ConcernArea
Id ID #REQUIRED
Exclusiveness (Singleton | Multi) #REQUIRED
>
<!ELEMENT ConcernAreaElement (#PCDATA)>
<!ELEMENT MetaObject (BeforeBehaviour?, AfterBehaviour?, BeforeRequirements?,
                    AfterRequirements?,MiddlewareComponentRef*)>
<!ATTLIST MetaObject
ClassName ID #REQUIRED
ReificationCategory (Class | ObjectReference | Attribute | Method | Constructor | Send |
                         Invocation | Dispatch | Creation | Deletion | StateRead | StateWrite |
                         Array) #REQUIRED
>
<!ELEMENT BeforeRequirements (ConcernAreaRef+)>
<!ELEMENT AfterRequirements (ConcernAreaRef+)>
<!ELEMENT ConcernAreaRef EMPTY>
<!ATTLIST ConcernAreaRef
Id CDATA #REQUIRED
Strength (Strong | Weak | Immediate) #REQUIRED
Locality (Proxy | Server | Common | NotApplicable) #REQUIRED
>
<!ELEMENT ExtensionProtocol EMPTY>
<!ATTLIST ExtensionProtocol
ClassName ID #REQUIRED
>
<!ELEMENT MiddlewareLinks (MiddlewareComponent+)>
<!ELEMENT MiddlewareComponent EMPTY>
<!ATTLIST MiddlewareComponent
ClassName ID #REQUIRED
>
<!ELEMENT BeforeBehaviour ((InvocationMods | StateReadMods | StateWriteMods | CreationMods |
                           DeletionMods)?)>
<!ATTLIST BeforeBehaviour
CallNext (UnconditionalNext | ConditionalNext | ActivatesMiddleware) #REQUIRED
CalledBy (Middleware | Metalevel) #REQUIRED
>
<!ELEMENT AfterBehaviour ((InvocationMods | StateReadMods | StateWriteMods | CreationMods |
                          DeletionMods)?)>
<!ATTLIST AfterBehaviour
CallNext (UnconditionalNext | ConditionalNext | ActivatesMiddleware) #REQUIRED
CalledBy (Middleware | Metalevel) #REQUIRED
>
```

```
<!ELEMENT MiddlewareComponentRef EMPTY>
<!ATTLIST MiddlewareComponentRef
Name IDREF #REQUIED
>
<!ELEMENT InvocationMods (InvocationMod*)>
<!ELEMENT InvocationMod EMPTY>
<!ATTLIST InvocationMod
Mod (ArgumentOrder | ArgumentValue | ArgumentNo | ArgumentType | ArgumentSize | ReturnValue |
        ReturnType) #REQUIRED
>
<!ELEMENT SendMods (SendMod*)>
<!ELEMENT SendMod EMPTY>
<!ATTLIST SendMod
Mod (ArgumentOrder | ArgumentValue | ArgumentNo | ArgumentType | ArgumentSize | ReturnValue |
        ReturnType) #REQUIRED
>
<!ELEMENT StateReadMods (StateReadMod*)>
<!ELEMENT StateReadMod EMPTY>
<!ATTLIST StateReadMod
Mod (AttributeOffset | AttributeValue | AttributeType | AttributeSize |
        AttributeSubscript) #REQUIRED
>
<!ELEMENT StateWriteMods (StateWriteMod*)>
<!ELEMENT StateWriteMod EMPTY>
<!ATTLIST StateWriteMod
Mod (AttributeOffset | AttributeValueIn | AttributeValueOut | AttributeType | AttributeSize |
        AttributeSubscript) #REQUIRED
>
<!ELEMENT CreationMods (CreationMod*)>
<!ELEMENT CreationMod EMPTY>
<!ATTLIST CreationMod
Mod (MemoryAllocationType | MemoryAllocationSize | MemoryAllocationLocation | VFTLocation |
        VFTEntries | ConstructorArgumentOrder | ConstructorArgumentValue | ConstructorArgumentNo |
        ConstructorArgumentType | ConstructorArgumentSize | ConstructorReturnValue) #REQUIRED
>
<!ELEMENT DeletionMods (DeletionMod*)>
<!ELEMENT DeletionMod EMPTY>
<!ATTLIST DeletionMod
Mod (ObjectValue | DestructorArgumentValue | DestructorArgumentType | DestructorArgumentSize |
        MemoryReleaseSize | MemoryReleaseLocation) #REQUIRED
>
```

A metaobject can have "before" and/or "after" meta-computation.

The base-level modifications specified in the DTD are the ones that can be performed by a behavioural metaobject in its reification category on the causally-connected base-level object during its "before" and "after" meta-operations (also refer to Table 3.1). However, through the current referential links (i.e., pointers) between different MO classes, a behavioural MO can acquire any information about the base-level object, which can be used to change additional aspects of the base-level object that is not intended for that behavioural metaobject class. Currently, there is no security mechanism in place to prevent a MO in one reification category from performing additional changes to the base-level object, i.e., other than the ones listed above. Instead, in order to make the automatic and dynamic MO composition work, we rely on the meta-level programmer to respect the design rules. If additional base-level modifications are required for a MO class, the meta-level programmer should declare a new MO class within the appropriate reification category.

Figure 3.5 shows attributes and containment relationships between elements in the MOP descriptor DTD.

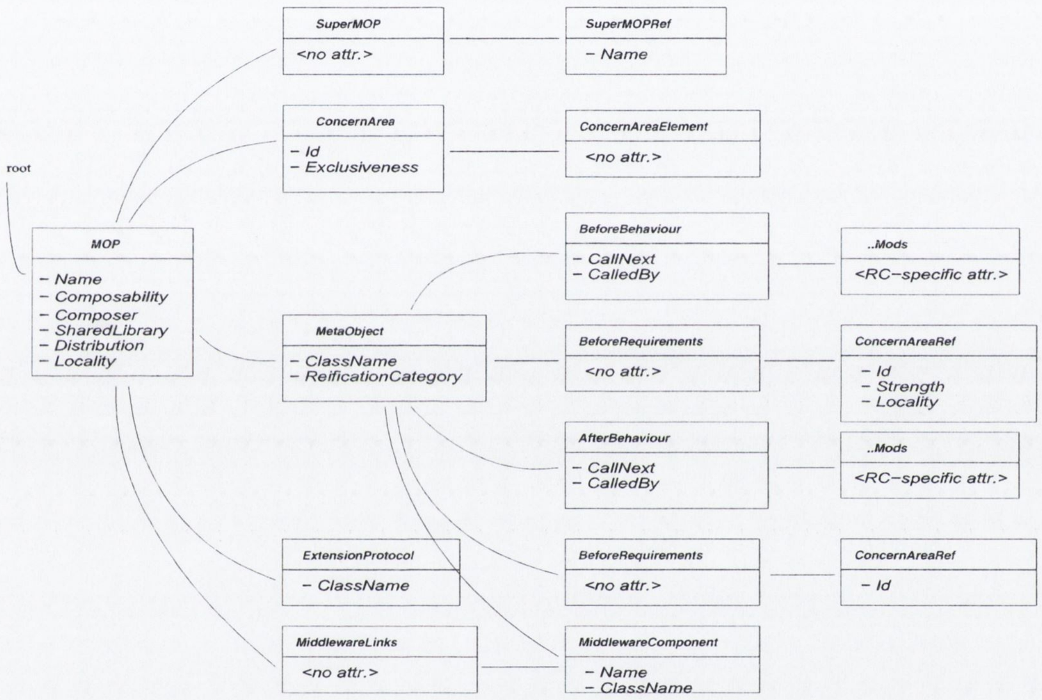

Fig. 3.5: Elements, attributes, containment relationships in the MOP descriptor XML DTD.

The MOP descriptor DTD defines the following elements:

MOP **element:** This element is the root in an XML-based MOP description. Note that each MOP must be accompanied by a separate MOP description document, which must have one and only one MOP element in it. This element has the following mandatory attributes:

Name: the unique name of this MOP. Type is ID.

Composability: indicates whether the MOP can be composed by the default, automatic DefaultComposer MO. The value is set to either Manual or Auto. Manually composable MOPs must declare a manual Composer MO, that is capable of combining the MOs of this MOP with other (manually or automatically composable) MOPs.

Composer: If the value of the Composability attribute is Manual, this attribute specifies the user-defined Composer MO.

SharedLibrary: the name of the UNIX shared library that contains the implementation of this MOP. For example, the libFTMOP.so file contains the implementation of a MOP which provides a form of fault-tolerance. Note that there is no specific requirement to link the IGUANA base-level application dynamically with this library: the IGUANA run-time can load this shared library dynamically and initialise the MOP.

Distribution: this attribute (value is one of Local, Distributed or Both) indicates whether the MOP is related to object distribution or not. In the Local case, the MOP is intended to perform local operations only (i.e., within a single address space). In other words, MOs of the MOP work without the presence of a "remote" MOP (e.g., Remote). The Distributed option means that a "remote" MOP must be present. In this case, the IGUANA run-time in different address spaces needs to co-ordinate meta-level activities such as dynamic MOP selection and deselection in order to maintain a balanced (i.e., symmetric) communication stack between servers and their proxies in address spaces of remote clients. Finally, the Both option indicates the MOP can work with or without the presence of a "remote" MOP.

Locality: this attribute indicates the locality of the MOP, with respect to object distribution. The value has to be one of the following: Proxy, Server, Common, or NotApplicable. It should be set to NotApplicable in descriptors of MOPs, that are intended to be local (i.e., the Distribution attribute described above is set to Local). For MOPs related to object distribution, this attribute suggests the side, at which the MOP is designed to be used: Proxy for the client side, Server for the server side.

The Common value can be used for MOPs that have been designed to work at both sides.

The MOP element contains an optional SuperMOPs element, one ConcernArea element, one or more MetaObject, ExtensionProtocols, and MiddlewareLinks elements.

SuperMOPs **element:** This element is used to refer to one or more parent MOPs, from which this MOP is derived. Only the direct parent MOPs need to be listed here, as the default composition algorithm will find and load the MOP descriptors of indirect parent MOPs as needed, There are no attributes defined in this element. This element contains one or more SuperMOPRef elements.

SuperMOPRef **element:** this element is used to refer to parent MOPs. It has one attribute called Name, which refers to the parent MOP. Its type is ID. This element does not contain other elements.

ConcernArea **element:** this element defines a particular non-functional concern area that this MOP implements. As explained in section 3.4, non-functional concerns are organised into a global and extensible hierarchy. There is a "root" concern area with child concern areas such as *Structural* and *Behavioural* positioned underneath. Furthermore, concern areas such as *Persistence*, *Remote*, *Security* are positioned underneath the *Behavioural* concern area. As an example, MOPs related to object persistence should be arranged in the *Persistence* concern area. A concern area can be further divided into smaller areas. For instance, the broad *Security* concern area can be divided into sub-areas such as *Encryption* and *Authentication.* It is important to note that each MOP should preferably fit into an existing concern area in the hierarchy. If this is not possible, then the MOP designer needs to extend the hierarchy by creating a new concern area.

There are two attributes defined in the ConcernArea element:

Id**:** the unique identifier of the concern area with which this MOP deals. Its type is ID.

Exclusiveness**:** this attribute (value is either Singleton or Multi) indicates whether this MOP is a singleton: it is the single MOP allowed to implement/deal with this particular concern area or not. The latter case means that multiple MOPs can share the responsibility of implementing this concern area.

The ConcernArea element contains one or more ConcernAreaElement elements.

ConcernAreaElement **element:** this element contains the name of the concern area element as character data (PCDATA).

MetaObject **element:** this element describes a metaobject class, which falls into one of the 14 IGUANA reification categories (see Section 3.4.1 for the complete list of them). There are two attributes defined:

ClassName**:** the unique name of this metaobject class. The type is ID.

ReificationCategory**:** the name of one of the 14 IGUANA reification categories.

The MetaObject element contains the following optional elements: BeforeBehaviour, BeforeRequirements, AfterBehaviour, AfterRequirements, and MiddlewareComponentRef. Structural MOs do not contain these four elements.

BeforeBehaviour **and** AfterBehaviour **elements:** these elements are used for describing behavioural MO classes, i.e., characterising what they do in their "before" and "after" meta-computations. There are two attributes defined for these elements (both are mandatory):

CallNext**:** this attribute (value is one of UnconditionalNext, ConditionalNext, or ActivatesMiddleware) reflects the chaining behaviour of the before/after meta-operation, i.e., whether this MO always or conditionally calls the next MO in the chain or it calls a middleware component instead, and by doing so breaks the chain. This classification criterium is similar to Pawlak's model [62], which characterised MOs as excplicit, conditional, etc.

CalledBy**:** this attribute (value is either Middleware or MetaLevel) reflects the caller of this MO, i.e., it is either called by a middleware component, or another metaobject in the chain, respectively.

The BeforeBehaviour and AfterBehaviour elements can optionally contain one of the six elements, that describe the actual base-level modification(s) performed by the MO class. For instance, if an Invocation MO class modifies the state of the causally linked base-level object(s) in both the "before" and "after" meta-operations, then it must have two InvocationMods elements describing these modifications, specified in its BeforeBehaviour and AfterBehaviour elements, respectively.

BeforeRequirements **and** AfterRequirements **elements:** these elements contained optionally in MetaObject elements of behavioural MO classes are used for listing optional ordering requirements (contraints) on "before" and "after" composition in terms of non-functional concern areas. This model is similar to what is described in the "Automatic composition of software components" paper [68] with requirements on UP and DOWN ports. There are no attributes defined for these elements. These elements contain one or more ConcernAreaRef elements.

ConcernAreaRef **element:** This empty element has three attributes defined:

Id**:** it refers to the concern area, for which the representative MO must be present before or after this MO. Type is ID.

Strength**:** this attribute (value is one of Strong, Weak, or Immediate) indicates the strength of the requirement: strong requirements must be satisfied, i.e., a representative MO of a MOP in the referenced concern area must be present before or after this MO. Weak requirements do not have to be met, but they shall not be violated, e.g., if there is a weak before requirement for a particular concern area, then a representative MO of a MOP in that concern area shall not be placed after this MO. Finally, immediate requirements imply that the representative MO must be placed directly before or after this MO.

Locality**:** this attribute is set to one of the following values: Proxy, Server, Common, or NotApplicable and it indicates the applicability of the requirements with respect to object distribution. For example, depending on the intended side of this MOP, there can be two different before requirements specified; one for the proxy side and another for the server side.

Base-level modification (e.g. InvocationMods**) elements:** these elements describe the base-level modificiations performed by the MO. See Table 3.1 for more information about the permitted base-level modifications.

ExtensionProtocol **element:** This empty element has only one attribute defined:

ClassName: it refers to the class that implements the extension protocol. We expect that the implementation of the extension protocol is contained either within the shared library of the MOP, or it is dynamically linked with the IGUANA base-level application.

MiddlewareLinks **element:** this element has no attributes defined. It contains one or more MiddlewareComponent elements.

MiddlewareComponent **element:** this empty element has two attributes defined:

Name: the unique name of the middleware component. Type is ID.

ClassName: it refers to class that implements the middleware component interface.

MiddlewareComponentRef **element:** this empty element is used in MetaObject elements to refer to components of the middleware (MiddlewareComponent element defined above) that are actually used by the MO class. This element has only one attribute

Name: the unique name of the middleware component. Type is IDREF.

3.4.7 Algorithm for the DefaultComposer MO

This section describes the way default, automatic and dynamic MO composition works.

The default MOP composition algorithm is implemented by the DefaultComposer MO class. An instance of this composer MO is created when an IGUANA application starts. This MO is then invoked to compose MOs of the MOPs specified in the MOP sets of reflective base-level classes.

The algorithm is based on processing all the MOP descriptor files of the selected MOPs, and making decisions on how to compose the MOs of MOPs in each behavioural reification category. Please note that depending on the before and after requirements, the calculated order of MOs can differ in different reification categories. This is in contrast with IGUANA version 2, where the same order was imposed on MOs in all the behavioural reification categories. The default composer MO also inserts/removes MOP-specific object references.
Before defining precisely (algorithmically) how the default Composer MO works, we describe the way it solves some interesting composition problems.

1. *Problem: how does the default* Composer *build a symmetric protocol stack between remote clients and servers?*
   Solution: The Composer MO detects that the MOPs selected are at the client or the server side. As we described in section 3.3, we assume that the proxy design pattern is used for implementing object distribution at the meta-level.

   This is done by selecting a remote proxy MOP (e.g., the RemoteProxy from our MOP suite) directly or indirectly (i.e., by selecting a MOP that depends on it) at the client side. Our RemoteProxy MOP is different from our Remote only in that it specifies different locality i.e., Proxy and Server, respectively. Otherwise, they use exactly the same MO classes and middleware components.

   The MObject has a callback method onProtocolSetChange which is called by the DefaultComposer MO upon changes to the MOP set of a particular instance. The MOP descriptor for Remote may specify that the IGUANA run-time running in the client's address space will need notifications from its peer running in the server's address space in order to maintain the symmetric stack of non-local MOPs (represented by MOs of the MOPs), every time the current MOP set changes at either side.

   The DefaultComposer MO detects "remote"-related MOP in the MOP set and uses the locality information to compose them differently depending on the side i.e., whether the MOPs are for the proxy or the server.

The composition algorithm defined below is deterministic in the sense that it selects "remote"-related MOs for the proxy side in an order that is the reverse that of the MOs for the server side. This ensures that these MOPs form a symmetric protocol stack.

The following code excerpt shows how the client can create a proxy:

```
// Select the remote proxy protocol
A* proxy= new A()==> RemoteProxy;
// Lookup the server in the name service
Iguana:: lookup(``MyServer'', ``A'', &proxy);
// Invoke the setValue method
proxy-> setValue(10);
```

Code excerpt for the server side is as follows:

```
// Select the remote protocol and create a server object
A* server= new A() ==> Remote;
// Record the server in the name service
Iguana:: record(``MyServer'', ``A'', server);
// Wait for client requests
Iguana:: wait();
```

2. *Problem: how does the base-level programmer select an additional* MOP*(s)?*

   Solution: the assumption is that the code at server side initiates the change to the MOP set and the negotiations (regarding what MOPs should be added or removed at both sides) between the two sides happen out of band; i.e., the client and the server communicate directly with each other. The server side can receive a notification event from the Composer MO if it subscribed to them. It can then contact the connected proxies to tell them of the changes in the MOP set.

   In the following code excerpt, the code at the server adds a new Encryption MOP to the current set of protocols:

```
// Add a secure protocol (e.g. encryption) to the server
A* server ==> ++Encryption;
// Initialise the cypher through the Security extension protocol
Security:: init(server, ''theSECRETkey'');
```

Let us assume that the server communicated with the connected clients the name of the MOPs such that the client code can add the same Encryption MOP to the MOP set of the reflective proxy object, as shown below:

```
char* name= ''Encryption'';
char* sharedKey= ''theSECRETkey'';
// Get the Encryption protocol
MProtocol* protocol= Meta:: getProtocol(name);
// Add a secure protocol for encryption
proxy-> mcomposer-> addProtocols(protocol);
// Initialise the cypher through the Security extension protocol
Security:: init(proxy, sharedKey);
```

First, the MOP is looked up then the DefaultComposer MO of the proxy object reruns the composition algorithm and rearranges the meta-level configuration and inserts the Encryption object reference and behavioural MOs in their respective lists according to the MOP descriptors.

3. *Problem: how does the composition cater for possible co-location of client and server objects?*
   Solution: MOs in MOPs like Remote and Secure have to be written keeping possible co-location of client and server in mind. For example, an encrypted invocation MO should not encrypt the arguments if the proxy and server objects are co-located. Any "remote"-related MO can test what side the MO is on: i.e., whether it is in a proxy or a server object's meta-configuration. This test simplifies MOP programming because it is possible to use one MO class (e.g., EncryptedInvocation) for doing argument encryption

or decryption in the "before" meta-level operation and exactly the other way around in the "after" meta-level operation, depending on the outcome of the co-location test. Convenient macros provided by the IGUANA/C++ run-time library can be used to check for proxy/server side and possible co-location:

IS_PROXY(o) this macro returns 1 (defined by the PROXY constant) if the object o is at the proxy side and has selected a remote-related MOPİt returns 2 (SERVER) if the object o is the remote server. Otherwise, it returns 0 (LOCAL).

IS_COLOCATED(o) this macro returns 1 (defined by TRUE) if object o specified as an argument is in this address space and it is not a proxy. Otherwise it returns 0 (FALSE).

4. *Problem: how does the default* Composer *check if a* MOP *is intended to be used with object distribution (i.e., related to "remote")?*
Solution: the MOP descriptor indicates that a MOP is related to object distribution or not. The Distribution attribute in the MOP element can be set to one of Distributed, Local and Common. Also, the BeforeRequirements or the AfterRequirements element should refer to Remote concern area for MOPs that are related to object distribution. The Locality attribute (possible values are: Proxy, Server, Common, and NotApplicable) indicates the "side" on which the MOP is intended to be used. For example, RemoteProxy has the Locality attribute set to Proxy.

The default MO composition algorithm, implemented by the DefaultComposer MO works as described in the following steps:

1. The Composer MO reads in and parses all the MOP descriptor files. It may additionally read in the XML descriptors of the super MOPs (if any). Note that the XML descriptors are assumed to be in the same directory for simplicity.

2. It checks if all of the MOPs are auto-composable: that is, if the MOP::Composability attribute in the XML documents is set to Auto. If not, the default composer MO stops and raises an exception.

3. It checks whether the non-functional concern areas for the selected MOPs overlap. This is indicated by the ConcernArea elements. If there is more than one MOP in an exclusive concern area (i.e., the ConcernArea::Exclusivity is set to Exclusive), the Composer MO stops and raises an exception.

4. It forms two groups of MOPs: local and remote. Remote MOPs are related to object distribution and must conform to our proxy MOP design pattern. "Local" or "remote-related" MOs are handled differently: the Composer MO distinguishes between local and remote-related MOPs because remote-related MOs need to form a symmetric stack between the client (more precisely the proxy object in the client address space) and the server objects, while local MOs can be inserted in the list more freely, where it is appropriate (see below).

   An interesting example is the task of combining MOs of a MOP set, that contains Logging, Replication, Authentication, and Encryption: Logging MOP is the only local MOP. In this example, a LogInvocation MO could be placed before both authentication (performed by AuthInvocation) and encryption (performed by EncryptInvocation) take place at the proxy side. This would result in logging unencrypted (clear-text) arguments. However, if the LogInvocation MO is positioned after AuthInvocation and EncryptInvocation MOs, it would log the client's digital signature as well as the encrypted arguments. The conclusion is that the meta-level programmer, who also writes the XML MOP descriptor, should specify the intention of how his/her MOP should be used in combination with other MOPs. This can be done by using the BeforeRequirements or AfterRequirements elements and specifying Weak, Strong, or Immediate ordering requirements (relative to concern areas) for the behavioural MOs. For example, there could be two Logging MOP derivatives using exactly the same MO classes for implementation, but having two different MOP descriptors (e.g., CleartextLogging and CryptoLogging) with two alternative ordering constraints.

5. For each behavioural reification category, the default composer MO places the Default MO at the end of the chain. Then it orders the other related MOs according to the following

rules:

- Declared behavior of the before/after meta-computation of MOs with respect to chaining with other MOs or the middleware (CallNext attribute of the BeforeBehaviour and AfterBehaviour elements in the MOP descriptor): the DefaultComposer MO can combine MOs that unconditionally call the next MO in the chain and there can be at most one MO that breaks the chain by calling a middleware component instead of the next MO. For example, the RemoteInvocation MO of the RemoteProxy MOP is a MO that breaks the chain. If there are more than one such MOs, then the composer MO raises an exception. MOs that call the next MO conditionally are also difficult to combine automatically as they might break the chain. Thus, the default composer MO raises an exception if it encounters conditional MOs.
- Base-level modification(s) performed by the MO may conflict, in which case the composition may fail: the DefaultComposer MO raises an exception.
- Declared ordering requirements (for both "before" and "after" meta-computation) relative to other non-functional concerns. Similarly to the solution in [68], where REQUIRES/PROVIDES rules can be described for UP/DOWN ports, the BeforeRequirements and AfterRequirements elements contain information that imply the order in which required MOs can be positioned relative to each other. As explained above in section 3.4.6, the requirements can be strong, weak, or immediate. If the requirements cannot be met or are violated, the DefaultComposer raises an exception.

  It is possible that there is more than one combination that meets these requirements. In this case, the DefaultComposer MO will pick one "good" combination according to the following logic: if the MOP set contains "local" MOPs only, the DefaultComposer MO picks the first combination. If the MOP set contains a "remote" MOP, e.g., RemoteProxy, or Remote, the DefaultComposer MO picks symmetric combinations of MOs for the proxy and server sides.

6. If no exception was thrown in the previous steps, the DefaultComposer MO builds the meta-layer for the reflective object(s).

We give composition examples in Chapter 6, where we evaluate our thesis.

3.5 Summary

This chapter defined the design IGUANA version 3 reflective programming model.
The main contribution of the new model can be summarised as follows: the IGUANA version 3 supports:

1 *Automatic and dynamic* MOP *composition, which can combine* MOs *of a composable* MOP *set, where some* MOPs *may overlap semantically.* By *automatic* and *dynamic* composition we mean that independently developed MOPs that were designed according to our composability rules can be dynamically selected for reflective instances of classes, and the selected MOPs will continue to provide their intended non-functional behaviour. The actual MOP set selected for any reflective object may not be known until run-time: composable MOPs can be added to the meta-level at any time facilitating dynamic adaptation. Similarly, MOPs can be removed from the MOP set at run-time. Automatic and dynamic composition can be used to deal with unanticipated changes in the requirements of applications.

2 *Interlinking* MOPs *and component-based middleware.* We have found that metaobject classes that comprise a MOP often use one or more middleware components (e.g., a NameService component for distributed objects) to implement their intended functionality. These links between MO classes of the MOP and components of the middleware are explicitly stated in the MOP descriptor, which helps the composition mechanism detect semantic overlap between MOPs that are to be combined.

3 *Supporting practical, real-world meta-level programming via a programming methodology.* We have defined a methodology for meta-level programmers, which if followed leads to the design of composable MOPs. These rules were derived from our experience of combining MOPs that implement common non-functional behaviours, such as remote

method invocation, object persistence, synchronisation, security, and fault-tolerance. Within our programming methodology, these and similar MOPs can be composed and/or adapted to meet the needs of changing application requirements.

Chapter 4

Design of an IGUANA MOP Suite

"Nothing ever becomes real until it is exprerienced." - John Keats

4.1 Introduction

This chapter describes the design of a number of automatically composable IGUANA MOPs that will be used to evaluate IGUANA version 3.

These MOPs range from the simple TypeInfo MOP for structural reflection through MOPs for object distribution to MOPs for implementing fault-tolerance at the meta-level. Due to their volume, the MOP descriptor XML files have been moved to the Appendix A.

We have implemented the following MOPs together with their extension protocols and middleware components: TypeInfo, Default, Logging, Persistent, Persistent2, Persistent2Absent, Remote, and RemoteProxy. We also outlined the design of MOPs for Synchronisation (atomic persistent objects with optimistic and pessimistic concurrency control) and Replication (fault-tolerance through leader and follower replicas).

4.2 Structural reflection: the TypeInfo MOP

The TypeInfo MOP defines the following MO classes:

- MClass

- MAttribute
- MMethod

Selecting the TypeInfo MOP allows structural information to be retained about classes. This information includes a description of the class, its attributes, and its methods in the form of the above MOs. The roles of these structural MOs are:

Class: the MClass MO maintains information on a class: name of the class, size of an instance of the class, reference to the class' table of virtual function pointers, references to MOs that describe parent classes (if any), attributes, methods (including constructor and destructor) declared by the class.

Attribute: the MAttribute MO stores information about an attribute of a class such as the name, type and size of the attribute, its offset within the object (for non-static attribute), or its address (for static attribute), its accessibility (private, protected, or public), and whether or not it is static.

Method: the MMethod MO stores information about a method such as the name of the method, its signature, and return type, its address, accessibility (private, protected, or public), and whether or not it is static.

The TypeInfo MO is used by the Default MOP and other MOPs described in the subsequent sections as a base MOP.

4.3 Design of the Default MOP

The Default MOP inherits from the TypeInfo MOP and defines the following MO classes:

- DefaultCreation
- DefaultDeletion
- DefaultInvocation

- DefaultStateRead
- DefaultStateWrite
- DefaultSend

The Default MOP implements the default C++ object model for object creation, deletion, member state read and write, method invocation, and method send. The main purpose of the Default MOP is that it reflects the reified operation at the base-level, therefore its MOs can be used in MOP composition as the last MO in chains for different behavioural reification categories.

The Default MOP inherits from the TypeInfo MOP, which retains structural information on class, attribute, and method definitions. Note that the ObjectReference has not been reified because by default the host language C++ uses pointers to refer to reflective objects. We outline the role of each MO class being used:

Creation: the DefaultCreation MO reflects the object creation operation by allocating memory for the base-level instance or array of instances and executing its constructor method, which might possibly take arguments.

Deletion: the DefaultDeletion MO reflects the object deletion operation by executing the destructor method for the reflective instance or array of instances and releasing the allocated memory.

Invocation: the DefaultInvocation MO reflects the method invocation operation by executing the selected method and returning the results.

StateRead and StateWrite: the DefaultStateRead and DefaultStateWrite MOs reflect the member state read and write operations at the base-level.

Send: the DefaultSend MO reflects sending method invocations to other reflective objects.

There are no extension protocol and middleware component classes defined for the Default MOP.

4.4 Design of the Logging MOP

The Logging MOP inherits from the TypeInfo MOP and defines the following MO classes:

- LogCreation
- LogDeletion
- LogInvocation
- LogStateRead
- LogStateWrite
- LogSend

The Logging MOP enhances the default C++ object model for object creation, deletion, member state read and write, method invocation, and method send by adding logging audit information whenever these events happen. The Logging MOP is a "local" MOP, i.e., it does not require the presence of a "remote" MOP. Thus, MOs of this MOP can more freely be combined with MOs of other MOPs. As we have already pointed out in section 1.2.1, depending on the requirements on what information should be logged, the meta-level programmer should specify ordering constraints to indicate the required position of logging MOs in the chains of behavioural MOs. This is done in the MOP descriptor file for Logging. We outline the role of each MO class being used:

Creation: the LogCreation MO logs the object creation event.

Deletion: the LogDeletion MO logs the object deletion event.

Invocation: the LogInvocation MO logs the method invocation event.

StateRead and StateWrite: the LogStateRead and LogStateWrite MOs log the state read and write events, respectively.

Send: the LogSend MO logs the event of sending method invocations to other reflective objects.

The Logging MOP implementation uses the logging extension protocol (LEP) and a LogManager middleware component to log these events to a file locally. Other implementations could log events to a central log server.

4.5 Design of the Persistent MOP

The Persistent MOPsinherits from the Default MOP and defines the following MO classes:

- PersistentClass
- PersistentObjectReference
- PersistentCreation
- PersistentDeletion
- PersistentInvocation
- PersistentStateRead
- PersistentStateWrite

The Persistent MOP extends the behaviour of the default C++ object model for object creation, deletion, member state read and write, and method invocation. It uses its own structural MO classes for object references and classes. We outline the role of each MO class being used:

Class: PersistentClass MO class extends MClass to include methods for reference swizzling and unswizzling (i.e., switching between volatile language pointers and persistent object references) and counting the number of references a class contains (including references inherited from parent classes).

ObjectReference: the local (per-object) PersistentObjectReference MO is used as a persistent object reference. The persistent object reference consists of an object number, a class number, and an offset value, which is used when referencing persistent objects embedded in other persistent objects.

Creation: when creating a persistent object, the PersistentCreation checks whether the object's class has been installed in the persistent class register (implemented as part of the proprietary IGUANA middleware for object persistence). If not, it installs the class. Then it calls the next creation MO. It is expected required that eventually the base-level operation will be reflected by the DefaultCreation MO at the end of the chain. The PersistentCreation MO then allocates a persistent object reference for the newly created object.

Deletion: when deleting a persistent object, the PersistentDeletion checks if the object has been recorded in the persistent name service. If so, it removes the name service entry, and passes control to the next MO. It is required that the DefaultDeletion at the end of the chain will reflect the base-level operation. The PersistentDeletion MO then removes the allocated persistent object reference from the reference manager middleware component.

Invocation: when a method is invoked on a proxy object, control is passed to the shared PersistentClass MO, which handles the object fault and loads the object from the Persistent Object Store (POS) (part of the IGUANA middleware). All persistent object references in the object's state are then unswizzled. Control is passed to the next MO. It is required that the DefaultInvocation MO at the end of the chain will reflect the base-level method invocation.

StateRead and StateWrite: like method invocation, control is passed to the shared PersistentClass MO, which is responsible for handling the object fault, and loads the object from POS, if it was not already loaded. State read and write are expected to be carried out by the last MO in the chains, DefaultStateRead and DefaultStateWrite, respectively.

Figure 4.1 shows the meta-level configuration for the Persistent MOP with the middleware components. The IGUANA extension protocol is called Common and provides an application programming interface for initialising the POS, recording, looking up, and deleting persistent root objects, and closing down the POS.

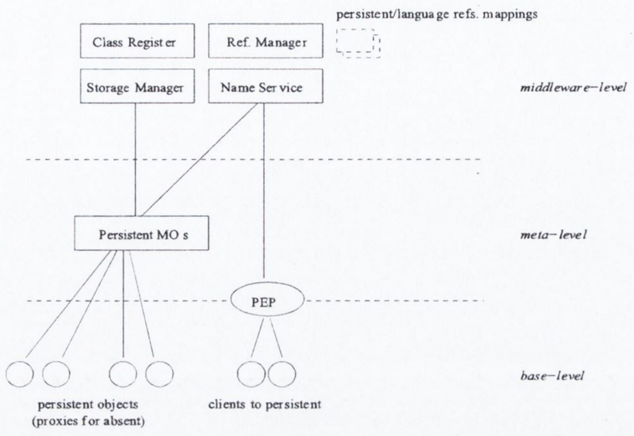

Fig. 4.1: Meta-level configuration for the Persistent MOP

4.5.1 Adaptation of Persistent

A problem with the implementation outlined above is that all language operations on potentially persistent objects are intercepted and carried out via the (slower) meta-level, even if the object is already present in memory and could be treated as an ordinary C++ object. To overcome this problem, the Persistent MOP can be refined in a way, that takes advantage of dynamic MOP selection. The refined Persistent MOP hierarchy is defined as follows: the Persistent2 MOP inherits from the Default MOP, while the Persistent2Absent MOP in turn inherits from the Persistent2 MOP. The definition of the MO classes they use remains the same as in the Persistent MOP above.

The Persistent2 MOP uses the following MO classes:

- PersistentObjectReference
- PersistentCreation
- PersistentDeletion

The Persistent2Absent in addition uses the following MO classes:

- PersistentClass
- PersistentStateRead

- PersistentStateWrite
- PersistentInvocation

As in the design of the Persistent MOP, the base-level programmer selects the above Persistent2 for classes or objects he/she wants to retain after the application has terminated. This MOP reifies only object references, creation, and deletion. The roles of these MOs are the same as above. The Persistent2Absent MOP inherits from the Persistent MOP and reifies class, state read/write, and method invocation. This MOP is dynamically selected by the Persistent2 implementation for proxy objects, that represent absent objects (i.e., those that have not been loaded from the POS yet) and for which object faults need to be detected. After an absent object is loaded from POS and the persistent references it may contain are unswizzled, the meta-level re-selects the Persistent MOP, which results in the state access and method invocations by-passing the meta-level thus improving the performance. The IGUANA extension protocol and middleware components are the same as in the Persistent MOP.

4.6 Design of the Remote and RemoteProxy MOPs

The Remote and RemoteProxy MOPs implement remote access to (public) member state and method invocations. Both MOPs inherit from the Default MOP. The two MOPs define and use the same MO classes as follows:

- RemoteObjectReference
- RemoteStateRead
- RemoteStateWrite
- RemoteInvocation
- RemoteCreation
- RemoteDeletion

These MOPs override the behaviour of the default C++ object model (represented by the Default MOP) for creation, deletion, state read and write, and method invocation. They use their own structural MO class for object references. The implemenation uses proxy model: proxy server objects that represent remote server objects in the client's address space.
The role of each MO class is outlined below:

ObjectReference: the per-object RemoteObjectReference MO is used as a global remote object reference. It contains sufficient information to find and communicate with the remote server object. The remote object reference consists of the IP address[1] address and the TCP port number of the server object, the unique object identifier, class number, and a timestamp, which indicates the time the server process was started.

Creation: when creating a remote object, the RemoteCreation MO checks whether the object's class has been installed in the class register (implemented as part of the IGUANA middleware). If not, it installs the class. It calls the next creation MO in the chain. It is required by our model that the DefaultCreation MO at the end of the chain will reflect the base-level operation. After this, RemoteCreation allocates a remote object reference for the newly created object. The ReferenceManager component of the middleware maintains the mappings between remote and language object references.

Deletion: when deleting a remote object, the RemoteDeletion checks if the object has been recorded in the name service component. If so, it removes the name service entry, and passes control to the next MO. It is required that the DefaultDeletion MO at the end of the chain will reflect the base-level operation. Then it removes the allocated remote object reference and the mapping between the remote/language object references, which is managed by the ReferenceManager component.

Invocation: when a method is invoked on a proxy object, the RemoteInvocation MO creates a remote method invocation request and marshalls the method arguments. The request is passed to the Communication component of the middleware, which handles the network communication between the client (proxy) and the server. Communication blocks until

[1]We support IP version 4 only.

it receives a reply from the peer Communication component. Upon receiving a reply, RemoteInvocation unmarhals the return value of the remote method call (if any).

StateRead : similarly to method invocation, the RemoteStateRead MO builds a remote state read request and passes it to the Communication component, which interacts with the remote peer. The Communication component blocks until it receives the reply to the state read request. Upon receiving a reply, control goes back to RemoteStateRead MO, which unmarshalls the value.

StateWrite: similarly to method invocation, the RemoteStateWrite MO builds a remote state write request, marhsalls the new value and passes the request to the Communication component. The Communication component blocks until it receives an acknowledgement to the state write request. Upon receiving an acknowledgement, control goes back to the RemoteStateWrite MO.

The current implementation supports synchronuous communication only. We use the Common IGUANA extension protocol for initialising the communication middleware, waiting for incoming client requests, recording, looking up and deleting remote objects in the name service, and closing the supporting middleware components. Figure 4.2 shows the meta-level configuration for Remote.

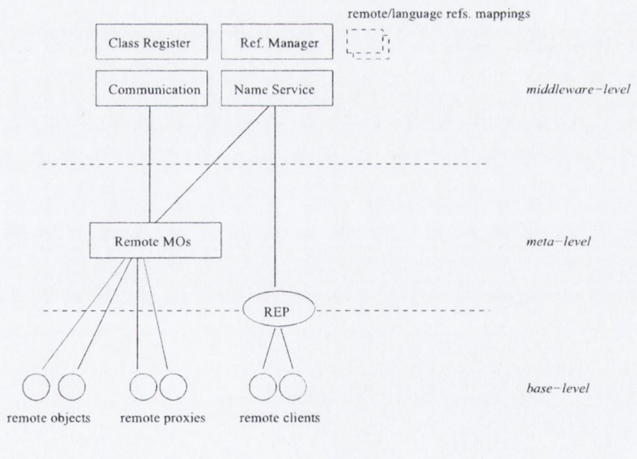

Fig. 4.2: Meta-level configuration for the Remote MOP

4.7 Design of MOPs for atomic objects

The Synchronisation MOP enhances the object state persistence (e.g., the Persistent MOP is in this concern area) with transaction support. When the Synchronisation MOP is selected, one MOP in the Persistence concern area must also be selected. The Synchronisation MOP inherits from the Default MOP and defines the following MO classes:

- SyncObjectReference
- SyncStateRead
- SyncStateWrite
- SyncInvocation
- SyncCreation
- SyncDeletion

Objects that select this MOP will become *atomic*: i.e., atomic objects provide "all or nothing" execution for *concurrent transactions* on persistent objects. A transaction groups client calls on atomic objects. A transaction is executed as a whole unit; if the transaction commits, then all of its changes are recorded in persistent storage or if it aborts, none of its changes are visible. Transactions have the "ACID" properties: atomicity, consistency, isolation, and durability.

The base-level programmer uses the synchronisation extension protocol (SEP) as the programming interface to work with transactions. The SEP provides the programmer with three methods to begin, commit and abort a transaction. The SEP and the shared Synchronisation MOs together co-ordinate the transactions and implement (local) concurrency control. In case of distributed transactions (there are servers distributed at a number of nodes involved in a transaction), the SEP co-ordinates transactions between the client and the distributed transactional servers.

In the following two sections, we outline how one would implement the Synchronisation MOP using two concurrency control mechanisms: an optimistic and a pessimistic solution.

4.7.1 Optimistic concurrency control

This section describes how the optimistic concurrency control algorithm [42] with backward validation could be used for an Synchronisation MOP implementation. The 2-phase Commit Crotocol [15] is used for co-ordinating distributed transactions. For atomic objects, state read/write and method invocation are reified. State write and method invocations are considered as read-write, while state read operations are considered as read-only operations on an atomic object. The first time an atomic object is accessed within a transaction's context, the relevant state read/write or invocation MO registers the atomic object as a participant in the transaction and creates a shadow copy of the atomic object. All subsequent operations (including the first one) within the same transaction context are performed on the shadow copy and will be validated against operations of other concurrent transactions when the transaction wants to commit. Each transaction operates on its own set of shadow copies of the atomic objects that are involved in the transaction. For each transaction, the meta-level records the transaction identifier together with a read and write sets, which contain the identifier of the object the operation was invoked on. This information will be used later to validate tha transaction: i.e., to check if the transaction performed an operation that violates the rules for executing concurrent transactions in a serially equivalent order.

The transaction's operations invoked on atomic objects within one process are called the *transaction component.* The execution of a transaction component consists of two or possibly three phases: a read phase, a validation phase, and possibly a write phase. During the read phase, read or write operations are executed on the atomic object's shadow copy. In order to commit a transaction, the base-level programmer calls the SEP::commit method. The SEP plays the transaction co-ordinator role. The SEP requests the meta-level (one of the shared Synchronisation MOs) to validate the transaction being committed. This is the *validation* phase.

If the atomic object is accessible only locally (i.e., a "remote" MOP such as our Remote MOP has not been selected), this shared MO validates the transaction and depending on the result, commits or rolls back the transaction. Figure 4.3 shows the meta-level configuration for Synchronisation MOP.

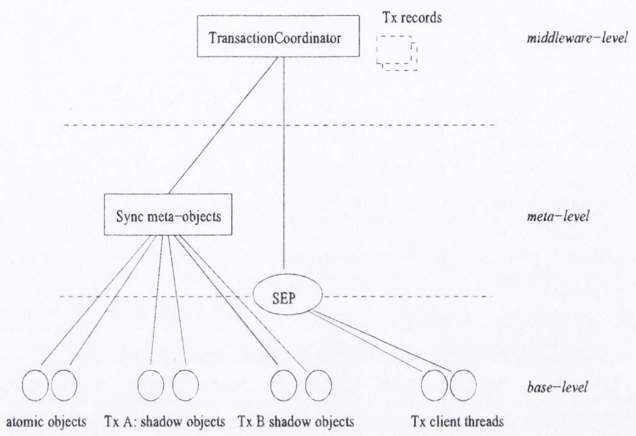

Fig. 4.3: Meta-level configuration for Synchronisation MOP with an optimistic concurrency control

There are two well-known algorithms for validation: *backward* or *forward*. We would implement the simpler backward validation. The backward validation compares the read set of the transaction being validated with the write sets of the previous transactions that have not committed before the validating transaction started. If the validation succeeds, the shadow copies of the participating atomic objects are written back to the persistent store (write phase) and the shadow copies are destroyed. If the validation fails, the shadow copies are simply discarded (no write phase).

When the set of selected MOPs include both a "remote" (e.g., our Remote MOP) and Synchronisation, one of the shared Synchronisation MO (e.g. SyncInvocation) becomes the local transaction co-ordinator (one within each process) while the SEP acts as the global transaction co-ordinator, executing the 2-PC protocol to reach a global decision regarding the outcome of the transaction at different participating nodes.

The transaction identifier is implicitly (via the remote call) passed from client to the server: the calling thread identifier (e.g., result of the pthread_self call) can be used to identify a transaction.

4.7.2 Using strict 2-phase locking for concurrency control

The second implementation sketch for a Synchronisation MOP is that of the well-known strict 2-phase Locking Protocol [15]. There are read and write locks on atomic objects, and a transaction must acquire the appropriate lock for an object before it can perform a read or write operation on it.

For atomic objects, state read/write and method invocation are reified. State write and method invocations are considered as read-write, while state read operations are considered as read-only operations on an atomic object. The first time an atomic object is accessed within a transaction's context, the relevant state read/write or invocation MO registers the atomic object as a participant in the transaction and requests a read or write lock on the atomic object. The Synchronisation MOs share a singleton (i.e., one instance per process) lock manager object, which enforces the above lock compatibility rules. Thus, in case of locking conflicts, the transaction requesting a lock will have to wait.

All subsequent operations (including the first one) within the same transaction context can be carried out only if the transaction has acquired the required lock on the atomic object. Locks are released when the transaction client calls the SEP::commit or SEP::abort method.

As locking might lead to deadlocks, a deadlock detection mechanism is needed. If the atomic object is accessible only locally (i.e., the "remote" MOP such as our Remote has not been selected), the singleton lock manager is used to detect and resolve deadlocks. It maintains a "wait-for" graph and regularly checks the graph for cycles that represent deadlocks. When it finds a cycle, it makes a decision on what transaction it should abort in order to resolve the deadlock. The SEP plays the transaction co-ordinator role.

Figure 4.4 shows the meta-level configuration for the Synchronisation.

When the set of selected MOPs include both Remote and Synchronisation, the SEP plays the transaction co-ordinator as well as the global deadlock detector roles. Each lock manager sends a copy of their local "wait-for" graph to the global deadlock detector, which amalgamates them in order to construct a global "wait-for" graph. Then the global deadlock detector checks for cycles in the graph. When detecting a cycle, it makes a decision on how to resolve the situation and it informs the lock managers about the transaction to be aborted.

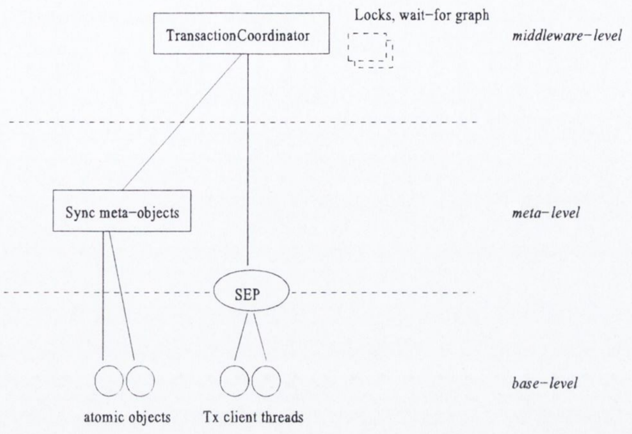

Fig. 4.4: Meta-level configuration for Synchronisation MOP with strict 2-phase locking

Similarly to the optimistic concurrency control solution, the transaction identifier is implicitly passed from client to the server: the calling thread identifier (e.g., result of the pthread_self call) is used to identify a transaction.

4.8 Design of a Replication MOP

Similarly to the approach of FRIENDS (see section 2.15), IGUANA MOPs could be used to implement fault-tolerance at the meta-level. In this section we outline how one would implement a Replication MOP. The Replication MOP is "remote"-related: i.e., when the base-level programmer selects this MOP, a remote MOP in the Distribution concern area must also be selected. The Replication MOP inherits from the Default MOP and defines the following MO classes:

- ReplicationObjectReference
- ReplicationStateRead
- ReplicationStateWrite
- ReplicationInvocation
- ReplicationCreation

- ReplicationDeletion

Objects that select this MOP will become *fault-tolerant* to some extent: the remote server objects are replicated at a different node. The semi-active replication mechanism is used in this design: there is one leader server object that sends every client request to one or more follower server replicas running on separate nodes.

The base-level programmer uses the Fault-Tolerance IGUANA extension protocol (FTEP) as the programming interface to work with replicated objects. The FTEP provides the programmer with methods to initialise the error detector, define the group of nodes where replicas can be launched (implemented as separate components).

Replication is transparent to the remote clients. Both leader and follower servers execute the request, return the result, but only the leader returns to the client. Should the leader or any of the followers fail, an error detection mechanism will detect it and create a new leader or follower server. The client communicates with the leader server, and the leader server selects the inter-replica MOP (see the IRP MOP below) to synchronise state read/write and method invocation requests with the follower replicas.

The IRP MOP also requires the present of a remote MOP (e.g., our Remote MOP) and it inherits from the Default MOP. The IRP MOP defines the following MO classes:

- ReplicationObjectReference
- IRPStateRead
- IRPStateWrite
- IRPInvocation

The ErrorDetector middleware component detects leader/follower server errors by regularly "pinging" them. Upon a leader server crash, it invokes the recover method on one of the alive followers, which becomes the new leader. It also creates a new follower on a free node within the replication domain.

Figure 4.5 shows the meta-level configuration for the Replication and IRP MOPs.

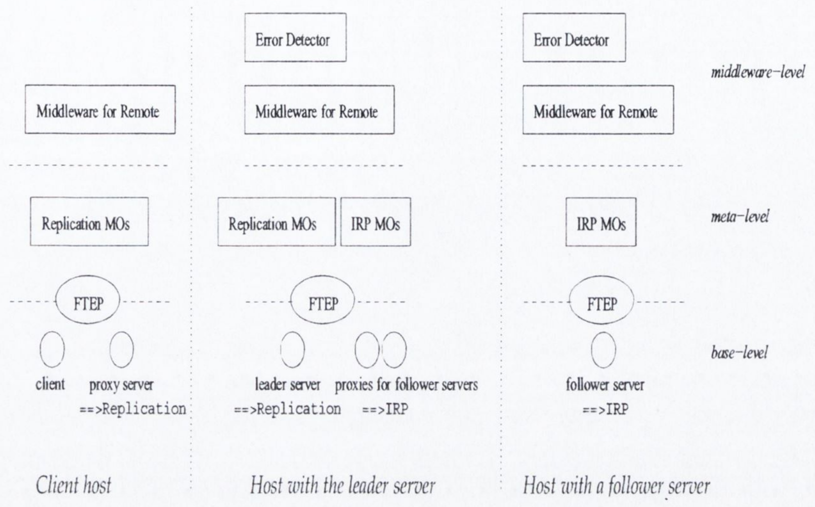

Fig. 4.5: Meta-level configuration for Replication and IRP MOPs implementing leader/follower replication

4.9 Summary

This chapter described the design of a MOP suite, which contains automatically composable IGUANA/C++ MOPs. These MOPs implements some of the most common requirements for real-world applications.

The implementation of these MOPs follow the methodology for writing composable MOPs, which was defined in section 3.4.1. We use some of the implemented MOPs to help us evaluate the IGUANA version 3 model and its implementation. The MOP descriptors for the implemented MOPs can be found in Appendix A.

Chapter 5

Implementation

"Keep it simple: as simple as possible, but no simpler." - Albert Einstein

5.1 Introduction

This chapter describes the implementation of the IGUANA version 3 reflective programming model as it is applied to C++.

We have implemented IGUANA/C++ on a Pentium III PC running RedHat Linux 7.0 with a 2.2 kernel. We used the open source GNU C++ (g++) compiler version 2.96.

5.2 Implementing the new IGUANA meta-level

Similarly to its predeccessors, we implemented IGUANA/C++ as a source-to-source preprocessor, which translates IGUANA-extended C++ code to standard C++. By extended, we mean that the base-level application is augmented with meta-level directives (e.g., MOP selection). Meta-level programmers implement MOPs by using and extending the IGUANA meta-level class and middleware library.

5.2.1 The meta-level class library

The IGUANA meta-level and middleware classes are written in C++ and some of these classes in turn use reflection (e.g., the NameService middleware component selects the Persistent and Remote MOPs).

Classes in the IGUANA meta-level library reify elements of the C++ language and have to be subclassed by the meta-level programmer who wants to implement a new MOP. Each of the structural and behavioural reification categories are represented by a class. Figure 5.1 shows the main C++ meta-level classes expressed in the Unified Modelling Language (UML). Method attributes, return types and exceptions thrown are omitted for simplicity.

As in the IGUANA version 2 implementation, we did not explicitly reify Array and Constructor. However they are supported implicitly: they are part of MAttribute and MMethod. Altough the reification category Receive is part of the model, we did not use it in our implementation of the IGUANA MOP suite described in Chapter 4.

Class MObject serves as the common base class for every reflective class. It provides the interface to the meta level as well as containing the methods for handling intercepted calls from the base level. Template class Deque[1] in Figure 5.1 is used in many IGUANA classes to hold multiple ordered pointers to MOs. For example, pointers to MObjectReference and behavioural MOs are stored in them. The order in which behavioural MOs are stored is decided by the delegated MComposer MO (e.g., the DefaultComposer MO). Whenever the MOP set changes for a reflective object, the delegated MComposer MO is called and it rearranges the above Deques in the MObject base of the reflective object: it may add or remove MObjectReference and behavioural (e.g. MInvocation) MOs. We further reduced the overhead due to reflection compared to IGUANA version 2. This is mainly because we eliminated local MOs with MObjectReference MOs being the only exception. We evaluate the performance overhead in Chapter 6.

Please note that the SingleStack and MultiStack MO classes are not shown in Figure 5.1 because they are discussed separately in Section 5.5.

[1]Deque is similar to the vector class in the C++ standard template library, but much simpler.

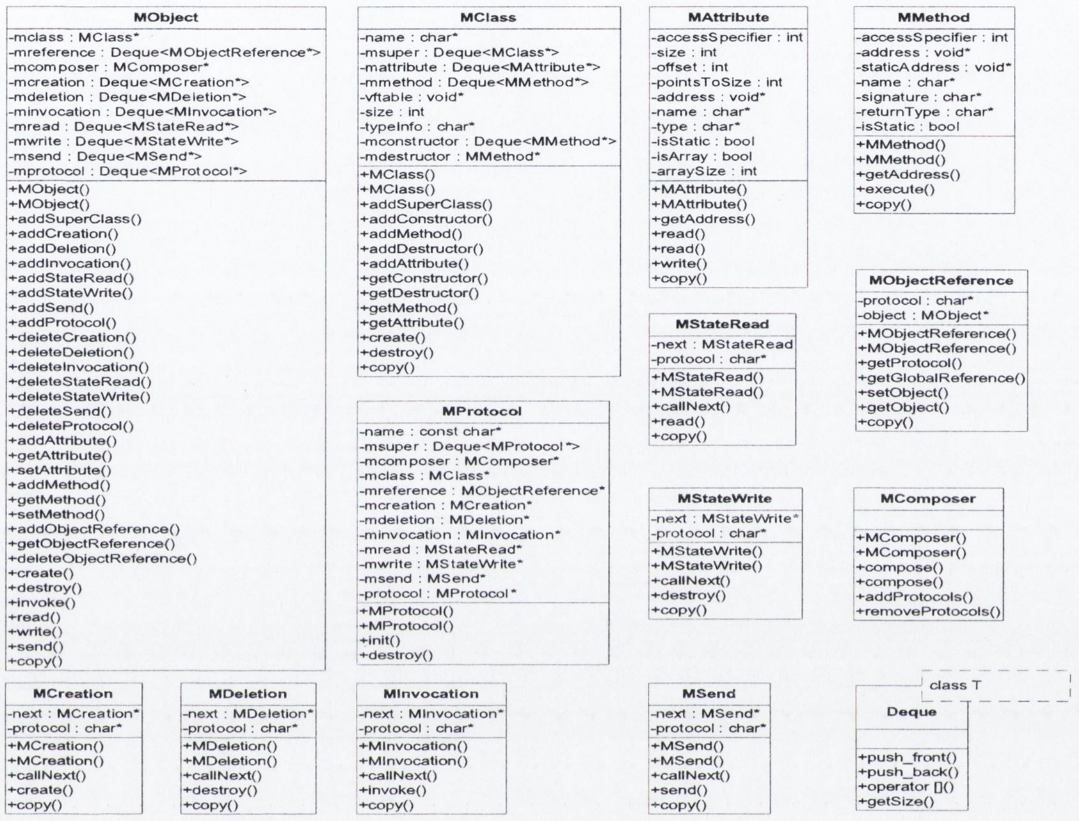

Fig. 5.1: Iguana meta-object classes

5.2.2 Multiple MOP inheritance

The IGUANA version 3 model retains multiple MOP inheritance as a meta-level programming feature. This section specifies the rules governing MOP inheritance and analyses its impact on structural and behavioural MO classes primarily, as well as on concern areas and middleware components.

Since multiple MOP inheritance is a design-time concept, it is the meta-level programmer's responsibility to ensure that the parent MOPs are composable.

As defined in Section 3.4.6, each MOP definition includes the parent MOP(s) (optional), its concern area, MO classes with compositional constraints, an optional extension protocol, and middleware components. For a derived MOP, the meta-level programmer may specify multiple parent MOPs. There is no restriction on the concern area specification of the derived MOP: it can be in the same or different concern area as its parent MOP(s). Because concern area specification is mandatory, the concern area specification of the derived MOP, if different, overrides that of the parent MOP(s).

Regarding MetaObject specifications in the MOP descriptor, which are there to declare the MO classes for reification categories, the following rules apply:

1. If the derived MOP specifies its own MO class for a given reification category then MO classes of the parent MOP(s) in the same reification category will not be used by the DefaultComposer MO. For example, AMOP has an AInvocation MO class and its derived MOP BMOP has BInvocation MO class specified, then BInvocation will be used in combining MOPs in a set that contains BMOP. However it is recommended but not required that the meta-level programmer use C++ class inheritance when he/she defines BInvocation; i.e., class BInvocation inherits from AInvocation. The DefaultComposer MO uses the Before/AfterBehaviour and Before/AfterRequirements specifications from the descriptor of the derived MOP.

2. If the derived MOP does not specify a certain MetaObject in its MOP descriptor, but one or more parent MOPs do, then those MO classes will be combined by the DefaultComposer MO. For example, if AMOP and BMOP have an ACreation and BCreation MO classes defined, but CMOP, which derives from both AMOP and BMOP does not have a Creation MO, then ACreation and BCreation MOs will be combined whenever a MOP set contains CMOP. In case the same MO class is specified in parent MOPs more than once, the DefaultComposer MO uses it only once.

The first rule above is different from the route taken by IGUANA version 2, which specifies the following: "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."

This difference is explained by our understanding of the role of MOP inheritance: it is to override MO class behaviour where necessary.

5.2.3 The DefaultComposer MO

The DefaultComposer MO inherits from the MComposer abstract class. Class MComposer is defined as follows:

```
class MComposer {
public:
  MComposer();
  MComposer(MComposer* src);
  virtual ~MComposer();
  virtual bool compose(MObject* object, MProtocol* inSet[], \
                                MProtocol* outSet[]) throw(CompositionException)= 0;
  virtual bool compose(MObject* object, MProtocol* newSet[]) \
                                                     throw(CompositionException)= 0;
  virtual bool addProtocols(MObject* object, MProtocol* inSet[]) \
                                                     throw(CompositionException)= 0;
  virtual bool removeProtocols(MObject* object, MProtocol* outSet[]) \
                                                     throw(CompositionException)= 0;
};
```

Thus DefaultComposer provides an implementation for the above four composition methods. The first argument of each is always a pointer to the base-level object for which dynamic composition is requested.

The four composition methods can be described as follows:

1. compose **method:** This method is used when there is a request for modification to the

MOP set with incoming and outgoing MOPs. Object headers and behavioural MOs of incoming and outgoing MOPs need to be added and removed respectively once the order of behavioural MOs has been recalculated.

2. compose **method (overloaded):** This method is used when the new MOP set is intended to replace the current one.

3. addProtocols **method:** This method is used when there are new additions to the current MOP set of the object.

4. removeProtocols **method:** This method is used when MOPs are to be removed from the current MOP set of the object.

Upon receiving one of the four method invocations, the DefaultComposer MO recalculates the order of behavioural MOs in the involved reification categories and rearranges them in the corresponding Deque in the MObject base of the reflective object. It also adds/removes MObjectReference instances in the mreference Deque of the reflective object.

The DefaultComposer MO may also request loading/unloading of IGUANA middleware objects that are specified in the MOP descriptor.

Note that it is the base-level programmer's responsibility to initialise and shut down the IGUANA middleware, which is usually done through IGUANA extension protocols.

The implementation of DefaultComposer MO class uses libxml++ developed by Ari Johnson, an open source XML Document Object Model (DOM) parser library for C++.

5.2.4 On building a symmetric protocol stack

This section describes how the DefaultComposer MO combines behavioural MOs of "remote"-related MOPs in an attempt to build a symmetric "stack" of MOs between proxy and server, similarly to a stack of network protocols between clients and servers.

- MO of the Default MOP (if present) is always the last MO.
- MO of the "remote" MOP (i.e., the concern area is called "Distribution") will be the first MO on the server side and the one before the last on the proxy side. This is needed

to place the "remote" MO (e.g., the RemoteInvocation MO from our RemoteProxy and Remote MOPs) at the bottom of the stack[2].

- The remaining "remote"-related MOs are ordered. First, the DefaultComposer MO orders the behavioural MOs according to the locality of the "remote" MOP. If there is more than one "good" composition, then it "swaps" sides temporarily (i.e., it switches the locality from Server to Proxy, or vice versa) and re-runs the composition in an attempt to find two "good" compositions (one for the real side, and the other for the "anticipated" peer side) of MOs where the order of the "remote"-related MOs is reverse to that of each other.

  For example, AInvocation and BInvocation MOs are followed by RemoteInvocation and DefaultInvocation MOs at the proxy, and RemoteInvocation, BInvocation, and AInvocation MOs followed by DefaultInvocation MO at the server side.

  In order to faciliate a deterministic outcome from selecting the same "good" composition for a particular side, the DefaultComposer MO preorders the behavioural MOs derived from the input MOP set according to the alphabets: it uses ascending order for the Proxy, and descending order for the Server side. Then this preordered set of MOs is presented to the composition, which analyses the MOP descriptors and orders the MOs according to the contraints specified in the descriptors.

  The implementation of steps 1, 2, and 3 of the composition algorithm (see section 3.4.7) is straigthforward. Steps 4, 5, and 6 are implemented as a variant of the recursive algorithm of permutating the input MOs. The recursive newCompose (not shown in Figure 5.1) method takes the initial preordered set of MOs together with their before and after requirements as well as two sets: front and back. When starting, the front set is empty while the back contains the same set of MOs as the initial set of MOs. Inside the newCompose method, the code selects the first candidate from the back set that satisfies the strong after requirements of the MOs in the front set. Before proceeding, it also checks whether the candidate's strong and/or immediate before requirements are

[2]Note that in IGUANA version 3 the RemoteInvocation MO on the proxy side calls the CommunicationService middleware component, which calls its peer on the server side, which in turn calls the RemoteInvocation MO.

met by the MOs in the front set. Immediate requirements have to be met by the last MO in the front set. Similarly, the last MO in the front set may also have an immediate after requirement that the candidate must satisfy. The candidate should not violate any weak before requirements of the MOs in the front set. If the candidate is OK for all the above criteria, the newCompose method is called again with the candidate MO moved from the back to the front set. Otherwise, the next candidate is examined from the back set. The recursion "bottoms out" when the front contains all of the initial MOs with all of the before and after requirements (including Weak, Strong, and Immediate requirements) satisfied. The satisfying combination is added to a list of "good" combinations and the newCompose method returns to allow the search for other "good" combinations (only when a "remote" MOP is in the set of MOPs).

Let us see an example with two "remote"-related MOPs AMOP and BMOP with Locality set to Common. AMOP is in concern area CA-a, while BMOP is in CA-b. Both MOPs have an Invocation MO AInvocation and BInvocation, respectively. The ordering requirements for AInvocation specifies that if a MOP in concern area Ca-b is present (weak requirement), then its Invocation MO should be placed before AInvocation at the proxy side, and after AInvocation at the server side. The following is the excerpt from the MOP descriptor of AMOP:

```
<BeforeRequirements>
  <ConcernAreaRef Id=''CA-b'' Strength=''Weak'' Locality=''Proxy''/>
</BeforeRequirements>
<AfterRequirements>
  <ConcernAreaRef Id=''CA-b'' Strength=''Weak'' Locality=''Server''/>
</AfterRequirements>
```

Although the above attempt to find two "good" combinations works in most cases, it might not always be fruitful: for example, the MOP set for the proxy and server may not be identical. For this reason the out-of-band communication (e.g., driven by the RemoteInvocation MO) between the client and the server is still needed to ensure that the two sides can communicate successfully with all the behavioural MOs in place.

5.3 Dynamically loadable MOPs

With the help of dlopen, dlsym, and dlclose system calls on most UNIX systems, it is possible to dynamically load C++ class code from a shared library. These functions provide access to the dynamic linker. The full description of these functions can be found in the appropriate man pages. Briefly, dlopen can be used to open a shared object file, such that the symbols defined in that file can be accessed by dlsym. dlsym returns the address of the symbol (which can be NULL). If a symbol cannot be resolved, dlerror can be used to get the error message. Finally, dlclose closes an open shared library.

Although dynamic class loading is not new (see [54]), and our reflective BufferManager example [21] has demonstrated its use in IGUANA version 2, its direct support at the IGUANA version 3 meta-level is extremely useful as it enables full dynamic adaptation, because the C++ implementation of a MOP (containing MO classes and middleware component classes) not known at compilation-time can be dynamically loaded into the running application.

Because handling of C++ symbols (constructors and class methods) is not straightforward compared to C methods, we show that through a simple C factory interface and the IGUANA MProtocol C++ class it is possible to do.

Each IGUANA version 3 MOP has a corresponding C++ class, which is derived from the MProtocol abstract class. The IGUANA preprocessor automatically generates source code for each MOP based on its XML MOP descriptor file. Note that this code should not be mistaken for the code of the MO classes which implement the MOP. For example, a MOP called DistributedLogging has a corresponding DistributedLoggingProtocol class, for which an instance is created at run-time.

The MProtocol abstract class is defined as follows:

```
class MProtocol {
protected:
  const char*  name;
public:
  virtual bool init(MComposer* composer, NVPair* params[]) = 0 ;
```

```
    virtual bool destroy() = 0;
  };
```

The init pure virtual method allows to pass in a pointer to the delegated MComposer MO as well as an array of generic name-value pairs (simple string pairs), which can be interpreted accordingly by the derived concrete MOP.

In order to overcome the problems of C++ name mangling and calling the constructor method of a derived MProtocol class that would not be known in advance, we require that the meta-level programmer defines the following C-style factory methods in each shared library (therefore each shared library should contain only one MOP implementation):

```
  extern "C" {
    MProtocol* create();
    void destroy(MProtocol* object);
  }
```

Simply, the implementation of the create method should create and return an instance of the derived MProtocol (e.g., DistributedLoggingProtocol). This MProtocol instance can be initialised by calling the above init method on it with a pointer to the MComposer MO, and the array of pointers to NVPair objects. When a MOP is no longer needed, its implementation classes can be unloaded from the running application by calling the above destroy method.

The IGUANA/C++ Meta class provides the base-level programmer with methods for loading and unloading of a named MOP, whose implementation has previously been compiled into a shared object, with the above C-style factory methods defined.

```
  class Meta {
    ...
  public:
    static MProtocol* loadProtocol(const char* name, const char* libraryPath)
      throw (LoadingException);
    static void unloadProtocol(MProtocol* protocol)
```

```
      throw (UnLoadingException);
  };
```

The Meta::loadProtocol method can be used to dynamically load a MOP from a shared library. If the named MOP cannot be loaded from the named file, a LoadingException is raised. Once the MOP is loaded, it can be initialised by calling the init method on it. If the MOP is no longer needed, it can be destroyed and unloaded by calling the destroy and unloadProtocol methods, respectively, on it.

5.4 The new IGUANA preprocessor

The IGUANA version 3 preprocessor is implemented by reusing much of the version 2 implementation (see Chapter 6 in [64]). Its implementation is based on the ANTLR and DLG, components of the public domain Purdue Compiler Construction Tool Set (PCCTS) version 1.33. DLG provides DFA (Deterministic Finite-state Automata) Lexical Analyser Generation, while ANTLR (Another Tool for Language Recognition) is a compiler compiler that we use for building up and modifying the Abstract Syntax Tree (AST) of an IGUANA/C++ program.

The major differences between IGUANA/C++ preprocessor version 3 and 2 can be listed as:

Handling of MOP descriptors (XML): The new IGUANA/C++ preprocessor handles XML files validated to our Composition Document Type Definition (DTD), which is defined in Section 3.4.6. In contrast, IGUANA version 2 used IGUANA/C++-specific MOP syntax (resembling the C++ class definition syntax) to define protocols.

Multiple compilation units: Multiple compilation units are better handled by IGUANA version 3. IGUANA version 2 relies on the standard C++ preprocessor to include all the C and C++ header files (including relfective ones) in each compilation unit, then it runs the IGUANA/C++ preprocessor to transform the code to standard C++. Our approach is more subtle: see Section 5.4.3 below.

Exception handling: because automatic and dynamic composition of independently developed MOPs may not result in a good combination, IGUANA version 3 adds exception handling. The meta-level code defines a number of exceptions and various methods in IGUANA meta-level classes raise exceptions. The IGUANA/C++ preprocessor retains the base-level programmer's `try/catch` blocks around the transformed MOP selection statements.

5.4.1 The IGUANA/C++ syntax

The way IGUANA/C++ extends the standard C++ language is minimal: only the MOP selection syntax is really new, and a few other rules have been extended to allow MOP selection.

The following lexical tokens are defined (only the MOPSELECT token is new, the others are from standard C++):

- `MOPSELECT` = `==>`
- `COMMA` = `,`
- `PLUSPLUS` = `++`
- `MINUSMINUS` = `--`
- `DEFAULT` = `default`
- `SEMICOLON` = `;`
- `LCURLYBRACE` = `{`
- `RCURLYBRACE` = `{`
- `COLONCOLON` = `::`
- `NEW` = `new`

The new IGUANA/C++ syntax in Extended Backus-Naur Form (EBNF) can be seen below. The scope of MOP selection can be default, class, and expression. Square brackets indicate optional constructs (lexical tokens or grammar rules). Lines starting with pipe symbol indicate a valid alternative in expanding the grammar rule. Finally, the backslash (\) breaks a long line.

```
mop-selection:
    MOPSELECT mop-set

mop-set:
    mop-expr COMMA mop-set
  | mop-expr

mop-expr:
    [PLUSPLUS | MINUSMINUS] mop-name

mop-name:
    identifier

default-mop-selection-statement:
    DEFAULT mop-selection SEMICOLON

class-specifier:
    class-head [mop-selection] [LCURLYBRACE [member-specification] RCURLYBRACE]

assignment-expression:
    conditional-expression
  | logical-or-expression assignment-operator assignment-expression
  | throw-exception
  | identifier mop-selection

new-expression:
```

```
[COLONCOLON] NEW [new-placement] new-type-id [new-initialiser] \
[mop-selection]
```

The rules identifier, class-head, member-specification, conditional-expression, logical-or-expression, assignment-operator, throw-exception are that of the standard C++ grammar definition as listed in Appendix A of the C++ book [70]. We have modified the assignment-expression, class-specifier, and new-expression grammar rules to accomodate class and instance MOP selection.

5.4.2 Source code transformations by the Iguana/C++ preprocessor

The IGUANA/C++ preprocessor can parse programs written in IGUANA-extended C++. After successfully parsing an IGUANA program, the preprocessor performs AST transformations in order to implement reification in the form of causal connection between reflective base-level objects and their meta-objects. The preprocessor analyses the source code and modifies the AST in order to intercept (direct) reified operations to the IGUANA meta-level.
The full list of AST code transformation rules is listed in Table 5.2 and Table 5.1 below.

Before we start elaborating on the code transformation rules, it is important to know that each reflective class in IGUANA/C++ inherits publicly from the MObject class, which provides the base-level programmer with methods to access and manipulate the meta-level.

Default MOP selection: The scope for default MOP selection is the translation unit in which it is defined. It defines the default MOP set for classes that are declared after this statement, however a class MOP selection can override it. A subsequent default MOP selection changes the default MOP set: it either overrides (when there is no ++ or - - specified in front of MOP identifiers) it or it adds/removes particular MOPs from it (when using ++ or - - in front of the MOP identifiers). The actual MOP set for the class is reflected in the transformed initMetaLevel method definition (see below).

Class MOP selection: The programmer can specify the MOP set for a class by using the class MOP selection syntax and/or having a default MOP selection. The actual MOP set for the class is reflected in the transformed initMetaLevel method definition (see below).

	Code	
Case	IGUANA/C++	Standard C++

Default MOP selection	default==> AMOP;
	class A;
	class B;
		class A : public MObject;
		class B : public MObject;

Class MOP selection	class A==> AMOP;
	class B==> BMOP;
		class A : public MObject;
		class B : public MObject;

Instance MOP selection (before creation)	A* a=
	new A()==> AMOP;
		A* a= 0;
		{ MProtocol* inSet[2]=
		{ AMOP::protocol, 0 };
		A* tmp= (A*) A::metaA-> copy();
		tmp-> mcomposer->
		compose(tmp, inSet);
		a= (A*) tmp-> create(2, 0, 0);
		delete tmp;
		}

Instance MOP selection (after creation)	a==> ++BMOP;	{
		MProtocol* inSet[2]=
		{ BMOP::protocol, 0 };
		a-> mcomposer->
		compose(a, inSet, 0);
		}

Multiple class MOP selection	class A==>
	AMOP, BMOP
		class A : public MObject;
		void A::initMetaLevel(MComposer* c) {
		metaA-> mcomposer= c;
		MProtocol* set[3]=
		{ AMOP::protocol,
		BMOP::protocol, 0 };
		metaA-> mcomposer->
		compose(metaA, set);
		...
		}

Table 5.1: Summary of the IGUANA code transformations for MOP selections

Instance MOP selection when creating a new instance: The MOP set defined for a class can be modified when creating a new instance of the class. For example, the following code:

```
A* a = new A()==> AMOP;
```

is transformed into the following (the use of a code block prevents a potential identifier name clash):

```
A* a= 0;
{
  MProtocol* inSet[2] = { AMOP::protocol, 0 };
  A* tmp = (A*) A::metaA->copy();
  tmp->mcomposer->compose(tmp, inSet);
  a = (A*) tmp->create(2, 0, 0);
  delete tmp;
}
```

First, a temporary instance with the initial MOP set for class A (e.g., Default) is created, then the delegated MComposer MO replaces the MOP set, and then an instance of class A is created. Finally, the temporary instance is deleted. This new instance will have only AMOP in its MOP set.

Instance MOP selection after creating an instance: The actual MOP set for an instance of a class can be dynamically modified even after the instance has been created. For example, the following code:

```
a==> ++BMOP;
```

is transformed into the following:

```
{
  MProtocol* inSet[2] = { BMOP::protocol, 0 };
```

```
    a->mcomposer->compose(a, inSet, 0);
  }
```

The transformed code adds BMOP to the MOP set of instance a. If the delegated MComposer MO cannot compose BMOP then it raises a CompositionException. It is recommended that the base-level programmer uses try/catch statement in his/her code in order to handle exceptions raised at the meta-level.

Generating the initMetaLevel method of a reflective class: For each reflective class the preprocessor inserts an additional initMetaLevel method, which is then called once when the main function of the application is called (see bootstrapping below). The structure of the generated initMetaLevel method is as follows in pseudo-code:

1. Create the static (shared) MObject* X::metaX object, which will be used in creating other instances of class X.
2. Set the X::metaX->mcomposer reference to the delegated MComposer MO.
3. Create the shared MClass MO.
4. Add MClass MOs of super-classes (if any).
5. Create the shared MAttribute MOs.
6. Create the shared MMethod MOs.
7. Add the listed MOPs to the MOP set.
8. Call the delegated MComposer MO to compose the MOP set and initialise the MOs.

The following example shows the generated initMetaLevel method for a class A with a default constructor, an int member called a, and a getA() method, which returns with an int. The class A selected AMOP only:

```
void A:: initMetaLevel(MComposer* composer) throw(CompositionException) {
  metaA = new MObject();
  metaA->mcomposer = composer;
```

```
    metaA->mclass = new MClass("A", sizeof(A), typeid(A).name(), 0);
    metaA->addAttribute(new MAttribute(Protected, "a", "int", offsetof(A, A), \
                                         sizeof(int), 0, 0));
    MMethod* constructor = new MMethod(Public, "A", (MSAddress) &A:: refl_A,
                                       "()", "A*");
    metaA->addMethod(constructor); // 0
    metaA->addConstructor(constructor);
    metaA->addMethod(new MMethod(Public, "getA", (MAddress) &A:: refl_getA,
                                 "()", "int")); // 1
    // Compose the protocol(s) selected for this class
    MProtocol* inSet[2] = { AMOP:: protocol, 0 };
    metaA->mcomposer->compose(metaA, inSet, 0);
  }
```

Wrapper methods around reflective methods: In addition to the initMetaLevel method, the preprocessor adds wrapper methods that intercept calls to the original method in the reflective class.

For example, the wrapper methods refl_A and refl_getA are defined as:

```
void A:: refl_A(MObject *meta, int numObjs) {
  A* newobj;
  if(numObjs)
    newobj= new A[numObjs](meta);
  else
    newobj= new A(meta);
  stack-> push(newobj);
}

voif A:: refl_getA() {
  int tmp= A:: getA();
  stack-> push(tmp);
}
```

Initialising the meta-level (bootstrapping): The preprocessor generates code inside the main function in order to initialise the meta-level. The structure of the generated code is as follows in pseudo-code:

1. A pointer to the reified multi-threaded stack MultiStack is declared as a global variable outside of the main function.
2. For each MComposer MO delegated in MOP descriptors of reflective classes (defined in this translation unit or in included header files), there is a global variable (pointer). In most cases, the delegated composer is the DefaultComposer MO.
3. Inside the main function create the reified stack.
4. Create the delegated MComposer MO(s).
5. Initialise the MOPs with their delegated MComposer.
6. Initialise the meta-level of the reflective classes.

Let us see a concrete example, in which two base-level classes A and B selected TypeInfo and Default MOPs, respectively:

```
MultiStack* stack= 0;
MComposer*  composer= 0;

int main() {
  // Auto-generated by the Iguana version 3.0 pre-processor
  stack= new MultiStack(1024);
  composer= new DefaultComposer();
  TypeInfo:: init(composer);
  Default:: init(composer);
  A:: initMetaLevel(composer);
  B:: initMetaLevel(composer);
  ...
```

Note that it is the base-level programmer's responsibility to initialise the IGUANA middleware, which is usually done through IGUANA extension protocols.

Shutting down the meta-level: The preprocessor also inserts code at the end of the main function in order to shut down the meta-level. The structure of the generated code is as follows in pseudo-code:

1. Delete the static (shared) MObject* X::metaX objects, which were used in creating other instances of class X.
2. Destroy the MOP(s) in reverse order.
3. Delete the MComposer MO(s).
4. Delete the reified stack.

Let us see the continuation of the above concrete example, in which two base-level classes A and B selected MOPs TypeInfo and Default, respectively:

```
  ...
  // Clean-up code auto-generated by the pre-processor
  delete B:: metaB;
  delete A:: metaA;
  Default:: destroy();
  TypeInfo:: destroy();
  delete composer;
  delete stack;
}
```

Note that it is the base-level programmer's responsibility to shut down the IGUANA middleware, which is usually done through IGUANA extension protocols.

		Code	
Feature	Case	IGUANA/C++	Standard C++
Creation	Single object, default constructor	class A ==> MOP1; A* a= new A();	A* a= (A*) A::metaA-> create(0, 0, 0);
	Single object, non-default constructor	class A ==> MOP1; A* a= new A(x, y);	A* a= (A*) A::metaA-> create(0, 1, (stack->push(x), stack->push(y)));
	Array, default constructor	class A ==> MOP1; A* a= new A[2];	A* a= (A*) A::metaA-> create(2, 0, 0);
	Array, non-default constructor	class A ==> MOP1; A* a= new A[2](x, y);	A* a= (A*) A::metaA-> create(2, 0, (stack->push(x), stack->push(y)));
Deletion	Single object	delete a;	a-> destroy();
	Array	delete [ ] a;	a-> destroy(true);
State read	Non-array element	int i= a-> i;	int i= *(int*) a-> read(0);
	Array element	int i= a-> j[2];	int i= *(int*) a-> read(1, 2);
State write	Non-array element	a-> i= 2;	a-> write(0, (stack->push(2)));
	Array element	a-> j[2]= 2;	a-> write(1, (stack->push(2)), 2);
Invocation	No arguments, no return value	a-> m();	a-> invoke(1);
	Arguments, no return value	a-> m(12);	a-> invoke(2, (stack->push(12)));
	No arguments, return value	int i= a-> n();	int i= *(int*) a-> invoke(3);
Send [3]	No arguments, no return value	a-> m();	a-> send(1);
	Arguments, no return value	a-> m(12);	a-> send(2, (stack->push(12)));
	No arguments, return value	int i= a-> n();	int i= *(int*) a-> send(3);

Table 5.2: Summary of the IGUANA reflective code transformations

Table 5.2 summarises how intercession is added to the reflective IGUANA/C++ code. The preprocessor analyses the AST of a reflective program and identifies parts that perform a reified C++ operation and transforms them with code that transfer control to the IGUANA meta-level. The code transformation in IGUANA version 3 is only marginally different from that of version 2 (see Chapter 4 of [64]). The difference is in the way that version 3 supports the creation of reflective objects using non-default constructors which version 2 did not support.

5.4.3 Working with Iguana/C++ code

Figure 5.2 shows the process of compiling IGUANA-extended source files.

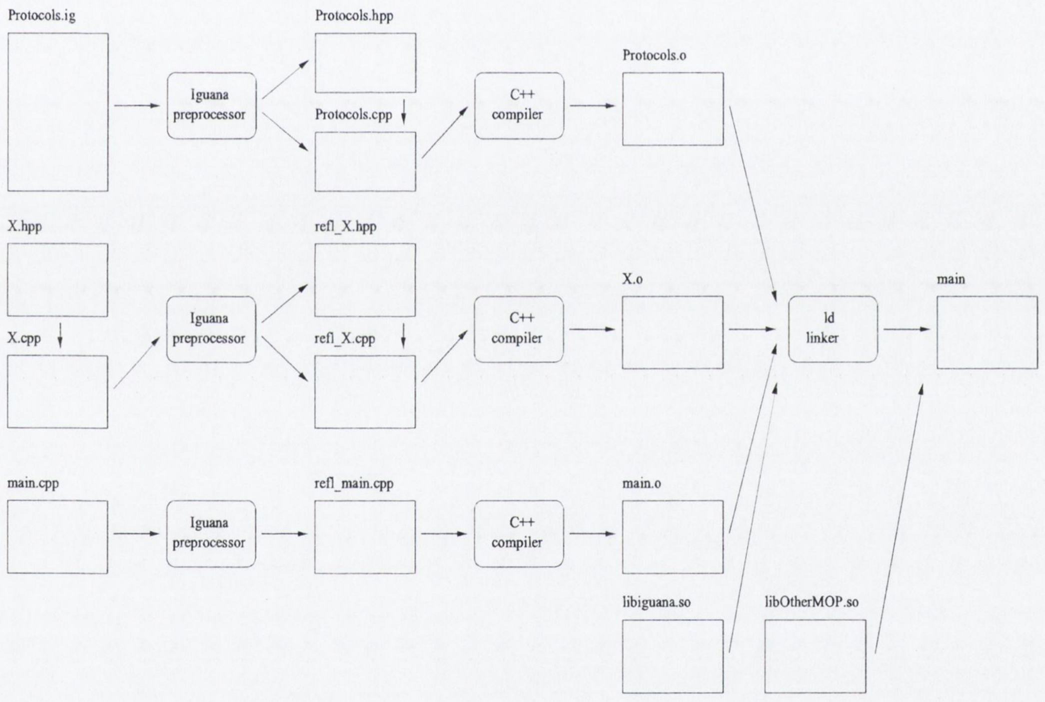

Fig. 5.2: Working with Iguana/C++ files: the compilation process from Iguana-extended source to executable

First, the file Protocols.ig lists the IGUANA MOPs that the base-level programmer wants to use in his/her application. This list does not have to be exhaustive, as unlisted MOPs compiled into a shared library can be loaded into, and used in a running IGUANA application. As a convention, for each name on this list there should be a corresponding MOP descriptor XML file. The IGUANA/C++ preprocessor parses the XML files of the corresponding MOPs and creates Protocols.hpp and Protocols.cpp; two standard C++ files, that contain the declaration and the definition of the MOPs (derived from the MProtocol class, see above).

Reflective base-level class definition files (i.e., X.cpp file in Figure 5.2 - which include X.hpp and Protocols.ig header files) have to be run through the IGUANA/C++ preprocessor, too. The preprocessor reads in the XML MOP descriptors as it processes the MOP selection statements and translates the IGUANA/C++ source file into a standard C++ files (i.e., refl_X.cpp and refl_X.hpp). If the preprocessor encounters an included header file for a reflective class or classes, it will substitute it with its refl_ equivalent.

Every base-level application has a translation unit with a main function (i.e., main.cpp in Figure 5.2). The preprocessor parses this main.cpp file and generates a refl_main.cpp file, which is a standard C++ file. This file includes the initialisation and finalisation code inserted by the preprocessor.

From this point, a standard C++ compiler and linker is used to compile the above C++ files into objects and link them with the IGUANA/C++ shared library (i.e., libiguana.so which contains the implementation of the IGUANA meta-level library, extension protocols, and middleware components) and possibly others depending on the application.

5.5 Multi-threaded Reified Stack

The IGUANA version 2 design introduced the concept of a reified stack in form of the MStack class, which is used during reified base-level operations e.g., method sends/invocations and state read/write accesses. The implementation of the stack allows pushing and popping of arbitrary elements, as it uses C++ push and pop template methods. There is one single stack instance created as one of the first steps in the main method of the reflective application.

However, this single stack is ultimately thread-unsafe: IGUANA version 2 did not provide any mechanism to synchronise access to the stack from multiple threads.

Therefore we redesigned the reified stack in the new model: we use the facade and singleton design patterns, in which we create a singleton in the form of an instance of the SingleStack class for pushing elements on to and popping elements from multiple stacks. The SingleStack manages instances of the MultiStack class: each thread is associated with one MultiStack. Our implementation is based on POSIX threads, where the pthread_self call returns the current thread identifier, which is used as a key in SingleStack to find the associated MultiStack instance.

We also introduced exception handling: the templated push and pop methods throw exceptions to signal that the stack is over or underflown, respectively.

```
class MultiStack {
public:
  ...
  template<class T>
    void push(T data) throw(StackOverflowException);
  template<class T>
    void pop(T& data) throw(StackUnderflowException);
};
```

5.6 Summary

The chapter described the implementation of IGUANA/C++, which is a mapping of our IGUANA model to the C++ language.

Similarly to its predessesors, IGUANA/C++ has been implemented as a preprocessor. IGUANA/C++ is an non-obstrusive extension to the standard C++ language: only one additional token (MOP selection symbol: ==>) has been added, and the IGUANA/C++ syntax extends the current C++ grammar with default, class, and instance MOP selection rules.

The preprocessor parses the IGUANA-extended code as well as the specified MOP descriptor XML files, and translates them into standard C++ source files. The Abstract Syntax Tree of the parsed IGUANA source code is transformed to include structural information as well as the causal connection for reified operations at the base-level, but only for reflective objects.

The source code of non-reflective classes is not transformed.

We also described the operation of the multi-threaded reified stack and dynamic MOP loading, which is an essential reflective feature in supporting unanticipated changes.

Chapter 6

Evaluation

"Nothing succeeds like success." - Alexandre Dumas

6.1 Introduction

This chapter demonstrates meta-level and base-level programming with IGUANA version 3 and evaluates its implementation through three real-world programming examples. These examples show the expressive power of the IGUANA version 3 reflective model.

The emphasis is on the separation of concerns, the distinction between the roles of simple base-level programming and more complex meta-level and middleware programming, and the automatic and dynamic composition of MOPs that have been written following our methodology. Once a more experienced programmer has implemented a MOP, possibly adding new middleware components, relatively inexperienced programmers can compose these MOPs in their applications in order to meet challenging non-functional requirements. The separately available MOP descriptors define the MOP in terms of MOP inheritance, positioning in a concern area, structural and behavioural MOs, extension protocol[1], and middleware component classes that together implement it. Furthermore, the specification of behavioural MO classes in terms of before and after compositional behaviour, base-level modifications performed by the MO, constraints on the presence of other MOPs in a different concern area, and links to

[1]Extension protocols are an IGUANA concept used to provide a structured and secured interface for manipulating the meta-level

middleware components. These MOP descriptors guide the automatic and dynamic composition in finding semantic overlap between MOPs to be composed and making decisions on the ordering of the behavioural MOs in each reification category as well as inserting and removing MOP-specific object references (object headers). The automatic and dynamic MOP composition is implemented by the DefaultComposer MO.

The MOP descriptors also help base- and meta-level programmers who want to understand and extend MOPs: they can immediately see the names of implementation classes for MOs, extension protocols, and middleware components without looking at the source code.

IGUANA version 3 facilitates dynamic adaptation: composable MOPs whose implementation may not be available when the reflective application was designed and implemented can be dynamically added to a running application.

In case an alternative MOP composition is needed (for example, to implement a composition different from the chain of responsibility model used by the IGUANA version 3 reflective model), the DefaultComposer MO can be replaced by another suitable MComposer MO. We also show an example of implementing a Guaraná-style composer MO.

However the separation of concerns through reflection comes with a price: it introduces a performance overhead that reflective applications must carry. Selective reification and shared MOs help reduce the burden. In section 6.5 we quantify and analyse the overhead introduced by IGUANA/C++.

6.2 Combining Persistent and Remote MOPs

This section shows how the Persistent and Remote MOPs can be combined.

As described in sections 4.5 and 4.6, object state persistence and distribution have been implemented as IGUANA MOPs. Both MOPs inherit from the Default MOP, which in turn inherits from the TypeInfo MOP (see sections 4.3 and 4.2).

Base-level application classes, whose instances the programmer wants to make persistent and remotely accessible, must select the Persistent and Remote MOPs as shown in the following code excerpt for two simple classes:

```
class Person ==> Persistent, Remote {
protected:
  char name[MAX_NAME_LENGTH];
  int  age;
public:
  Person();
  Person(char* name, int age);
  virtual ~Person();
  void setName(char* name);
  char* getName();
  void setAge(int age);
  int getAge();
};

class Parent : public Person {
protected:
  Parent* spouse;
  Person* children[MAX_CHILDREN];
public:
  Parent();
  Parent(char* name, int age);
  Parent(Parent* spouse, Person* children []);
  virtual ~Parent();
  void setSpouse(Parent* spouse);
  Parent* getSpouse();
  void setChildren(Person* children []);
  Person** getChildren();
};
```

Note that the child class Parent implicitly selects the Persistent and Remote MOPs because it inherits the MOP set from the Person superclass.

Instances of class Person and Parent can be used to model families. Let us see how reflective instances of these classes can be created at the client and server sides. The notion of persistence is applicable only to the remotely accessible servers, while the clients communicate with the servers using proxies: in the client-side code, the Person and Parent classes have to select RemoteProxy instead of Remote and Persistent.

Because full separation of concerns is rarely attainable (i.e., there are always some configuration issues related to the non-functional concerns), the main method for the server side code must take a few arguments:

- coldstart flag indicates whether persistent objects should be created from scratch and stored in the Persistent Object Store (POS) upon terminating the application or should be recreated by retrieving them from the POS. Our Persistent MOP implementation provides persistence for heap-allocated objects that are transitively reachable (via pointers) from name service recorded persistent objects (i.e., persistent roots).
- directory defines the directory in which the POS files are located (which defaults to the user's `HOME` directory).
- hostname specifies the host name or IP address to be used. For the server, it is relevant only if the computer is multi-homed (i.e., it has many network interface cards configured).
- port specifies the TCP port number. In order to initialise remote communication, the IGUANA middleware needs to know the host name or IP address and the TCP port number to be used when listening to client requests.

These arguments are presented to the IGUANA extension protocol Common, which in turn uses them to initialise the relevant components of the IGUANA middleware. The following excerpt shows the server mainline code (for shortness, we omit the processing of the above command line arguments):

```
int main(int argc, void** argv) {
  ...
  try {
    // Use the extension protocol to initialise common middleware
    Common::init(directory, hostname, port);
    if(coldstart) {
      // Create a family
      Person* child = new Person(``Eve'', 4);
      Parent* father = new Parent(``John'', 25);
      Person* kids [] = { child, 0 };
      Parent* mother = new Parent(father, kids);
      mother->setName(``Marie'');
      mother->setAge(24);
      father->setSpouse(mother, kids);
      // Record ``John'' in the name service
      Common::record(``John'', ``Parent'', father);
    }
    else {
      Parent* father = 0;
      // Look up ``John'' in the name service
      Common::lookup(``John'', ``Parent'', (void**) &father);
    }
    // Wait for client requests
    Common::wait();
    Common::close();
  }
  catch(...) {
  }
  return 0;
}
```

First, through the Common IGUANA extension protocol we initialise the IGUANA middleware, such that it creates/opens the POS files, launches the communication service, the class register, the name service, and the reference manager. The name service component is both remote and persistent, using the very same MOPs. If the coldstart flag is set, it creates an exemplary family which is the couple and one child and records one of the parents as a remote and persistent object. Otherwise, it looks up a previously created persistent Parent object in the name service. Note that this operation results in a persistent proxy being created, on which the meta-level detects object faults, loads the state from the POS, swizzle references, see 4.5. Finally, the server waits for incoming client requests and upon receiving a TERM signal saves persistent objects in the POS and closes the POS and communication middleware.

As persistence is required at the server side only, the main method for the client side code takes even less arguments:

- hostname specifies the host name or IP address for the server we want to use.
- port specifies the TCP port number the server is listening on.

In the client side implementation, the Person and Parent classes select the RemoteProxy MOP only. The client mainline code is shown below (again, we omit the handling of command line arguments):

```
int main(int argc, void** argv) {
  ...
  class Person ==> RemoteProxy;
  class Parent ==> RemoteProxy;
  try {
    // Use the extension protocol to initialise remote middleware
    Common::init();
    // Create a URL
    sprintf(url, ``iguana:%s:%u/John'', hostname, port);
    Parent* father = 0;
    // Look up ''John'' in the remote name service
    Common::lookup(url, ''Parent'', (void**) &father);
    cout << ''Father's age='' << father->getAge() << endl;
    cout << ''Mother's age='' << father->getSpouse()->getAge() << endl;
    Person** kids = father->getChildren();
    while(*kids) {
      cout << ''Child's age= '' << (*kids)->getAge() << endl;
      kids++;
    }
    Common::close();
  }
  catch(...) {
  }
  return 0;
}
```

First we initialise the IGUANA middleware for remote communication using the Common extension protocol. We use an IGUANA-style Unified Resource Locator (URL) to look up one of the Parent remote objects in the remote name service, then we invoke some of its methods remotely to explore the attributes of the family. Finally, we close the communication middleware and terminate the client application.

Because the Persistent, Remote, and RemoteProxy MOPs are automatically composable, the DefaultComposer MO composes the behavioural MOs in the Creation, Deletion, Invocation, StateRead, and StateWrite reification categories as well as inserting the ObjectReference per-object MOs. For example, the order of Invocation MOs at the server side is: RemoteInvocation, PersistentInvocation, and DefaultInvocation. The order of Invocation MOs at the client (proxy) side is: RemoteInvocation and DefaultInvocation. RemoteInvocation uses the IGUANA middleware to transport method and state access requests from the client to the server and back.

6.3 Applying a Logging MOP to a running application

In this section, we show how a logging MOP can be dynamically loaded and added to a running application.

Suppose we want to log remote access to our mother and father objects (both are instances of class Parent) on the server side. In order to initiate the dynamic adaptation of the server program, the base-level programmer has to implement an internal or external communication mechanism that can be used to request the change to the MOP set for certain reflective objects.

We have modified the above server code to include a new class Adaptation, for which we select the Remote MOP. Thus, when an external client invokes the Adaptation::adapt method, which takes the name of the MOP and the shared library to be loaded as well as its initialisation parameters, the definition of this method uses the IGUANA Meta class to load the named MOP dynamically from the shared library, and apply it to the two previously created instances of the Parent class. Subsequent remote requests (invocation and state read/write) addressed to these objects will be logged to the named file on the server host.

The definition of the Adaptation class is as follows:

```
class Adaptation ==> Remote {
public:
  Adaptation() {}
  ~Adaptation() {}
  void adapt(char* name, char* library, NVPair* params []) {
    try {
      Parent* parent1 = 0;
      Parent* parent2 = 0;
      Common::lookup(``John'', ``Parent'', (void**) &parent1);
      parent2 = parent1->getSpouse();
      MProtocol* mop = Meta::loadProtocol(name, library);
      // Initialise the named MOP using the parents' composer MO
      mop->init(parent1->mcomposer, params);
      // Add the named MOP to the MOP sets
      MProtocol* mops[] = { mop, 0 };
      parent1->mcomposer->addProtocols(parent1, mops);
      parent2->mcomposer->addProtocols(parent2, mops);
    }
    catch(const MetaException& e) {
      // Handle loading and composition exceptions
      ...
    }
  }
};
```

The server mainline code differs from the one shown in the previous section only in that it creates an instance of class Adaptation, which is then recorded in the name service:

```
int main(int argc, void** argv) {
  ...
  try {
    // Use the extension protocol to initialise common middleware
    Common::init(directory, hostname, port);
    Adaptation* adaptation = new Adaptation();
    Common::record(''Adapt!'', ''Adaptation'', adaptation);
    if(coldstart) {
      // Create a family
      Person* child = new Person(''Eve'', 4);
      Parent* father = new Parent(''John'', 25);
      Person* kids [] = { child, 0 };
      Parent* mother = new Parent(father, kids);
      mother->setName(''Marie'');
      mother->setAge(24);
      father->setSpouse(mother, kids);
      // Record ``John'' in the name service
      Common::record(``John'', ``Parent'', father);
    }
    else {
      Parent* father = 0;
      // Look up ``John'' in the name service
      Common::lookup(``John'', ``Parent'', (void**) &father);
    }
    // Wait for client requests
    Common::wait();
    Common::close();
  }
  catch(...) {
  }
  return 0;
}
```

We use a modified version of the client code (shown in the previous section) to remotely trigger dynamic adaptation. Note that ideally, dynamic adaptation should be initiated from the meta-level as a result of some fault detection or changes in some measured software metrics. The client mainline code is shown below (again, we omit the handling of the command line arguments):

```
int main(int argc, void** argv) {
  ...
  class Person ==> RemoteProxy;
  class Parent ==> RemoteProxy;
  class NVPair ==> TypeInfo;
  class Adaptation ==> RemoteProxy;
  try {
    // Use the extension protocol to initialise remote middleware
    Common::init();
    // Create a URL for ''John'' (a parent)
    sprintf(url1, ''iguana:%s:%u/John'', hostname, port);
    // Create a second URL for adaptation
    sprintf(url2, ''iguana:%s:%u/Adapt!'', hostname, port);
    Adaptation* adaptation = 0;
    // Look up ''Adapt!'' in the remote name service
    Common::lookup(url2, ''Adaptation'', (void**) &adaptation);
    // Initiate dynamic adaptation (add Logging MOP)
    NVPair* param= new NVPair(''logfile'', ''iguana.log'');
    NVPair* params [] = { param, 0};
    adaptation->adapt(''Logging'', ''libLoggingMOP.so'', params);
    Parent* father = 0;
    // Look up ''John'' in the remote name service
    Common::lookup(url, ''Parent'', (void**) &father);
    cout << ''Father's age='' << father->getAge() << endl;
    cout << ''Mother's age='' << father->getSpouse()->getAge() << endl;
    Person** kids = father->getChildren();
    while(*kids) {
      cout << ''Child's age= '' << (*kids)->getAge() << endl;
      kids++;
    }
    Common::close();
  }
  catch(...) {
  }
  return 0;
}
```

As a result of calling the adapt method from the client, the two instances of the persistent and remote Parent class become logged too: invocation, state read/write and delete operations will be logged to a local text file on the server machine. Note that although the Logging MOP reifies object creation, the LogCreation MO will be combined with the other three Creation MOs, but it will not actually be used for the two Parent objects as they have already been created.

Because Logging is an automatically composable MOP, the DefaultComposer MO combines its behavioural MOs with the existing MOs. For example, the order of Invocation MOs is: RemoteInvocation, LogInvocation, PersistentInvocation, and DefaultInvocation.

6.4 Using a manual Composer MO

The DefaultComposer MO implements the automatic and dynamic MOP composition algorithm described in Section 3.4.7. The DeafultComposer MO reads in MOP descriptor files written in XML and uses them to make decisions on combining the MOs in different behavioural reification categories. The composition is triggered by a change in the MOP set of reflective classes and their instances. After the DefaultComposer has finished arranging the behavioural meta-objects, it is not involved in the intercession of reified operations at the base-level (e.g., invocation or state read).

Because composing semantically overlapping MOPs is difficult, there may be a need to use an alternative Composer MO. In IGUANA version 3, the meta-level programmer has the freedom to write his/her own Composer MO in order to implement an alternative composition algorithm. For example, an alternative Composer can act as a "switchboard" and be involved in the intercession as well as in the composition. This is in line with the composition mechanism of Guaraná [56, 57, 59], in which Composer MOs delegate the intercepted base-level calls to different (behavioural) MOs, possibly including other Composer MOs.

In this section, we show the implementation of such a Guaraná-style manual Composer MO. We define the GuaranaComposer MO class as follows:

```
class GuaranaComposer : public MComposer, MCreation, MDeletion,
                        MInvocation, MStateRead, MStateWrite {
```

```
public:
  GuaranaComposer();
  GuaranaComposer(MComposer* src);
  virtual ~GuaranaComposer();
  virtual bool compose(MObject* object, MProtocol* inSet[], MProtocol* outSet[])
               throw(ComposerException);
  virtual bool compose(MObject* object, MProtocol* newSet[])
               throw(ComposerException);
  virtual bool addProtocols(MObject* object, MProtocol* inSet[])
               throw(ComposerException);
  virtual bool removeProtocols(MObject* object, MProtocol* outSet[])
               throw(ComposerException);
  virtual void* create(MObject* meta, int index= 0, int numObjects= 0);
  virtual void  destroy(MObject* object, bool isArray= 0);
  virtual void* invoke(MObject* object, MMethod* mmethod, MClass* mclass,
                       int index= 0);
  virtual void* read(MObject* object, MAttribute* mattribute, MClass* mclass,
                     int subscript= -1);
  virtual void* write(MObject* object, MAttribute* mattribute, MClass* mclass,
                      int subscript= -1);
  ...
};
```

Through multiple inheritance, the GuaranaComposer MO is able to act as a MComposer MO as well as a behavioural MO for object creation, deletion, invocation, and state access. It acts as a primary MO and such it has the task to delegate operations to other behavioural MOs or indeed to other composer MOs. Our simple implementation uses a separate thread to handle reified operations for logging.

6.5 The computational overhead in IGUANA version 3

The IGUANA/C++ preprocessor transforms IGUANA-extended source code to standard C++. It inserts the neccessary code required to initialise and maintain the meta-level as well as intercept reified operations at the base-level and divert them to the meta-level. This incurs an overhead in form of MOs the base-level objects are associated with and a performance penalty paid for intercession.

In the IGUANA version 3 model and its C++ implementation we have revised the version 2 model and eliminated local behavioural MOs as during our experiment with implementing and combining the Persistent and Remote MOPs we found that most of the per-object MOP-specific state can be separated out and maintained in the newly introduced ObjectReference MO and/or in new explicit IGUANA middleware components (e.g., the ReferenceManager object we use for Persistent and Remote). Thus, behavioural MOs are shared between instances of reflective classes.

The DefaultComposer MO is also shared between instances of reflective classes.

The exact overhead due to run-time reflection is difficult to quantify because it depends on a number of factors:

Number of reification categories used in a MOP: selective reification helps reduce the overhead by allowing the meta-level programmer to reify the necessary language features only. Features that have not been reified will not incur any additional overhead, apart from a run-time check, implemented as a macro (see below).

Efficient implementation of the MOP: multiple MOP selection allow the base-level programmer selecting the MOPs for reflective classes and their instances that actually need them. Dynamic MOP deselection of one, more, or all of the MOPs can reduce the overhead. The support for dynamic loading and selection of a MOP means that a MOP implementation in a particular concern area can be replaced by another MOP with a more efficient implementation.

Frequency and type of using the reified operations: if a reified operation (e.g., method invocation) is used more frequently than an operation, which has not been reified (e.g.,

Operation	Relative overhead in version 3 (version 2)	Absolute time in version 3
Plain C++	1 (1)	n/a
Creation	17 (27)	10.2
Deletion	10 (n/a)	6.6
Creation + Deletion	14 (31)	13.1
Invocation (no arguments)	21 (12)	0.54
Invocation (1 int argument)	30 (18)	0.84
StateRead	12 (9)	0.12
StateWrite	48 (22)	0.56

Table 6.1: Measurements indicating the relative and absolute overhead of reified C++ language operations using the Default MOP

state read) then it incurs a higher run-time penalty. Furthermore the number of arguments to a reified method send or invocation makes a difference: the reified stack operations are significantly slower than the native C++ ones.

We took measurements to quantify the performance overhead of reified operations implemented by the Default MOP as compared to the plain C++ operations with and without run-time checks.

Table 6.1 shows the relative overhead for IGUANA/C++ version 3 and 2 (in parentheses) as well as the actual time taken in microseconds (version 3 only). We carried out the measurements on an Intel Mobile Pentium III 500 MHz PC with 192 MB of physical memory, running RedHat 7.0 operating system with Linux 2.2 kernel and the GNU C++ (g++) compiler version 2.96.

Reified creation and deletion shows an improved performance, mainly due to the shared-only behavioural MOs in version 3. The reified invocation, state read and write operations however are slower in IGUANA/C++ version 3 because the additional level of indirection in the new multi-threaded stack implementation (i.e., the MultiStack class).

Other measurements (see table 6.2) quantify the time in microseconds taken when executing the compose method on the DefaultComposer MO with six different MOP sets. Compared

MOP set	Absolute time (microseconds)
TypeInfo	1700
Default	3200
Persistent	8100
Remote	7800
Persistent and Remote	10700
Persistent, Remote, and Logging	14000

Table 6.2: Measurements indicating the absolute overhead of automatic and dynamic MOP composition by the DefaultComposer MO

to the absolute figures of reified operations, the results show that composition is ten thousand times more expensive on average. This can mainly be attributed to the cost of processing the XML MOP descriptor files and the complexity of the composition algorithm.

Once the MOP descriptor files have been parsed, the additional time taken to compose the MOPs depends on a number of factors:

The number of MOPs in the MOP set: the composition algorithm is based on testing different permutations of behavioural MOs in search of a good combination which satisfies all the constraints (see the description of the newCompose method in section 5.2.4).

The behavioural reification categories used: the composition algorithm combines the behavioural MOs separately in each behavioural reification category.

Whether MOPs are "remote"-related: the composition algorithm uses heuristic (see section 5.2.4) to build a symmetric MO "stack" for the proxy and server sides, if a "remote" MOP is present in the MOP set.

The number of before and after requirements: ordering constraints can significantly reduce the number of possible permutations to test in the recursive newCompose method implementation.

However, automatic and dynamic MOP composition does not occur very frequently in a carefully designed IGUANA application. We believe that the benefits of our approach (flexibility, separation of concerns, and ease of use) outweights the price of performance decrease.

6.6 Summary

This chapter evaluated the IGUANA version 3 model and its implementation.

Composition of complex MOPs is handled by the DefaultComposer MO, provided that the MOPs were written according to our methodology. As a major advantage compared to other architectures, including Guaraná, base-level programmers do not need to deal with composing MOPs and their related MOs. However if they want, they can replace the DefaultComposer MO with their own Composer MO. IGUANA version 3 was designed to further simplify the meta-level by eliminating the notion of local and shared MOs. The performance overhead introduced by the IGUANA/C++ version 3 model is comparable to that of version 2.

Chapter 7

Summary and Conclusions

"An optimist is a guy that has never had much experience." - *Don Marquis*

7.1 Introduction

This chapter summarises the thesis, draws conclusions, and suggests areas for future work.

7.2 Summary and Conclusions

In this thesis, we have explored computational reflection as an effective means of dealing with rapidly increasing application requirements and, as a consequence, increased software complexity. Reflection is favoured because of its property of separating of functional (business logic of the application) and non-functional (requirements independent from the business logic) concerns: business logic is implemented at the base level of the application while the non-functional requirements are implemented at the meta level, in the form of meta-object protocols (MOPs), where each MOP addresses a particular non-functional concern (e.g., object state persistence, or remote object invocation).

As programmers of today have to meet multiple recurring non-functional requirements for most of their applications, where some of these requirements may change dynamically at run-time, the automatic and dynamic composition of non-functional concerns is an important issue to address.

However, automatic and dynamic MOP composition is a difficult task. We identified the main problems hindering MOP composition as:

1. Non-functional concerns may overlap.
2. Lack of methodology for writing composable MOPs.
3. MOPs may not have been written with composition in mind.
4. The source code of a MOP may not be available for use when writing another composable MOP or evolving it.
5. MOP source code may have hidden links to middleware components.

We have designed and implemented a solution that addresses the above problems. Our solution is based on having MOP descriptors (XML files) that are available to the MOP composition at run-time in order to make decisions on ordering/arranging MOs for reflective objects. These MOP descriptors describe the MO classes that comprise a MOP together with their semantics and their links to a component-based IGUANA middleware that supports common non-functional concerns. MOP descriptors are also used to express constraints on the composition. MOPs are positioned in to a hierarchy of concern areas, where each area specifies a particular area of non-functional concern.

The MOP composition itself is reified and implemented as a MO class. We provided a default implementation (i.e., the DefaultComposer MO), which can combine composable MOPs that have been written according to the methodology that we defined. The DefaultComposer MO parses the MOP descriptor files in order to make decisions regarding the composition of behavioural MOs. Naturally, the programmer is free to use his/her non-default MComposer MO in order to implement an alternative MOP composition. In case the source code of particular MOPs is not available to a meta-level programmer, the MOP descriptor file gives useful information about the semantics of the MOP, thus semantic overlap between MOPs can be detected. We have made the links between MO classes and middleware components explicit, which helps the meta-level programmer reason about dependencies and semantic overlap between MOPs.

7.3 Contribution of this thesis

The main contributions of this thesis are:

A mechanism for automatic and dynamic MOP composition: composition of MOPs with overlapping semantics is a complex task. Guided by MOP descriptors, the DefaultComposer MO in IGUANA version 3 can detect and deal with semantic overlaps. Automatic and dynamic composition can be used to deal with unanticipated changes in the requirements of applications.

Interlinking MOPs and middleware components: most reflective architectures use middleware to implement a particular non-functional concern. These links between MO classes of the MOP and components of the middleware are explicitly stated in the MOP descriptor, which helps the composition mechanism detect semantic overlap between MOPs that are to be combined.

Methodology for writing automatically composable MOPs: the methodology presented in this thesis, if followed, leads to the design of automatically composable MOPs.

Real-world examples: we described a suite of IGUANA version 3 MOPs that implements the some of the most commonly found non-functional concerns faced by application programmers of today, and we show how they can be automatically combined.

7.4 Future work

IGUANA version 3 is a result of ongoing research in the Distributed Systems Group. Although this thesis has described the solution to the problem of automatic and dynamic composition of MOPs, there are a number of open issues that remain:

1. Not all of the language features can be reified. Currently, there is no support for reifying C-style functions and C++ templates.

2. All or nothing reification of class methods and attributes: currently, if Invocation is reified then invocations of any method of the reflective class are intercepted.

3. This thesis used a proprietary middleware for implementing some aspects of the non-functional concerns. Experiments with standard middleware would be desirable.

Areas that remain for future work include:

1. Research the unification/combination of architectural (e.g., K-Component architecture [20, 19]) and behavioural (i.e., IGUANA version 3) reflection.

2. Analyse the source code of MOPs, so MOP descriptors could be (partially or in whole) generated.

3. A Graphical User Interface (GUI) that helps with composing large number of protocols. This GUI could show how the DefaultComposer MO would compose MOs of MOPs in a particular MOP set under examination. This could be used to verify if certain MOP sets defined at run-time could achieve the overall effect the base- and meta-level programmers intended.

4. In IGUANA version 3, the adaptation is initiated from the base-level, preferably through extension protocols. Ideally, a meta-meta-level should be constructed to monitor the base- and meta-level behaviours and initiate adaptation.

Bibliography

[1] Gul A. Agha. Abstracting interaction patterns: A programming paradigm for open distribute systems, 1997.

[2] G. Attardi, C. Bonini, and M. Boscotrecase. Metalevel programming in CLOS. In Stephen Cook, editor, *Proceedings of the* 3^{rd}*European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 243–256. Cambridge University Press, July 1989.

[3] Gordon Blair, Fabio Costa, Geoff Coulson, Fabien Delpiano, Hector Duran, Bruno Dumant, Francois Horn, Nikos Parlavantzas, and Jean-Bernard Stefani. The design of a resource-aware reflective middleware architecture. In Cointe [12], pages 115–134.

[4] Gordon Blair, Geoff Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of Middleware '98.*

[5] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[6] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue, 1998.

[7] 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.

[8] Shigeru Chiba. A metaobject protocol for C++. In Wilpolt [77], pages 285–299. Also SIGPLAN Notices 30(10), October 1995.

[9] Shigeru Chiba. Open C++ programmer's guide for version 2. Technical Report SPL-96-024, XEROX Palo Alto Research Center, 1996.

[10] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the* 7^{th}*European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 482–501. Springer-Verlag, 1993.

[11] Shigeru Chiba and Takashi Masuda. Open C++ and its optimization (Extended abstract). In *Proceedings of the OOPSLA '93 Workshop on Reflection*, September 1993.

[12] Pierre Cointe, editor. *Reflection '99 – Proceedings of the* 2^{nd} *International Conference on Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1999.

[13] Fabio Costa, Gordon Blair, and Geoff Coulson. Experiments with reflective middleware. In *Proceedings of the ECOOP '98 Workshop on Reflective Object-Oriented Programming and Systems*.

[14] Fabio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon S. Blair, and Geoff Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In *OORaSE*, pages 79–98, 1999.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, third edition edition, 2001.

[16] Scott Danforth and Ira Forman. Reflections on metaclass programming in SOM. In Wilpolt [76], pages 440–452. Also SIGPLAN Notices 29(10), October 1994.

[17] John Dempsey and Vinny Cahill. Aspects of System Support for Distributed Computing. In *ECOOP '97 Workshop on Aspect-Oriented Programming*, 1997.

[18] G. Denker, Meseguer J, and C. Talcott. Rewriting semantics of meta-objects and composable distributed services, 1999.

[19] Jim Dowling and Vinny Cahill. Dynamic software evolution and the k-component model.

[20] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Reflection 2001 - The Third International Conference on Meta-Level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan*, pages 81–88, 2001.

[21] 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 Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 171–190. Springer-Verlag, Heidelberg, Germany, June 2000.

[22] Jean-Charles Fabre, Tanguy Pérennou, and Laurent Blain. Meta-object protocols for implementing reliable and secure distributed applications. Laas report 95037, Laboratoire d'Analyse et d'Architecture des Systemes (LAAS), 1995.

[23] I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming.* Addison-Wesley, October 1998.

[24] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work.* Addison-Wesley, 1998. A new dimension in object-oriented programming.

[25] Ira R. Forman, Scott H. Danforth, and Hari Madduri. Composition of before/after metaclasses in SOM. In Wilpolt [76], pages 427–439. Also SIGPLAN Notices 29(10), October 1994.

[26] A. Goldberg and D. Robson. *Smalltalk-80, the language and its implementation.* Addison-Wesley, 1983.

[27] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1st edition, 1996.

[28] Brendan Gowing. *A Reflective Programming Model and Language for Dynamically Modifying Compiled Software.* PhD thesis, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, University of Dublin, 1997.

[29] Brendan Gowing and Vinny Cahill. Making meta-object protocols practical for operating systems. In Luis-Felipe Cabrera and Marvin Theimer, editors, *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, pages 52–55. IEEE Computer Society, IEEE Computer Society Press, August 1995. Also technical report TCD-CS-95-21, Dept. of Computer Science, Trinity College Dublin.

[30] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. In *Proceedings of Reflection '96*, pages 137–152. XEROX Palo Alto Research Center, April 1996.

[31] Peter Haraszti, Tilman Schäfer, Jim Dowling, and Vinny Cahill. The iguana experience: Meta-level programming in a compiled relfective language. Presentation, Workshop on Experience with Reflective Systems, Reflection 2001 The Third International Conference on Meta-Level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[32] Robert Hirschfeld. Aspect-oriented programming with aspects, 2002.

[33] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[34] Jean-Charles Fabre and and V. Nicomette and Tanguy Pérennou and Robert J. Stroud and Zhixue Wu. Implementing fault-tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[35] Jean-Charles Fabre and Tanguy Pérennou. FRIENDS - a flexible architecture for implementing fault-tolerant and secure distributed applications. In *Proceedings of the 1996 European Dependable Computing Conference*, October 1996.

[36] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989. ISBN 0-201-17589-4.

[37] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[38] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[39] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. Unpublished, 1997.

[40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the* 11^{th} *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[41] Marc-Olivier Killijian, Jean-Charles Fabre, J. C. Ruiz-Garcia, and Shigeru Chiba. A metaobject protocol for fault-tolerant CORBA applications. In *Proceedings of the* 17^{th} *Symposium on Reliable Distributed Systems*, pages 127–134, September 1998.

[42] H. T. Kung and J. T. Robinson. Optimisitic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[43] Rodger Lea, Yasuhiko Yokote, and Jun ichiro Itoh. Adaptive operating system design using reflection. In *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems*, pages 95–100. IEEE Computer Society, IEEE Computer Society Press, May 1995.

[44] Thomas Ledoux. Implementing proxy objects in a reflective ORB. In *Proceedings of the ECOOP '97 Workshop on CORBA: Implementation, Use and Evaluation*, June 1997.

[45] Thomas Ledoux. OpenCorba a reflective open broker. In Cointe [12], pages 197–214.

[46] Pattie Maes. Computational reflection. Technical Report 87.2, Vrije Universiteit Brussels, Artificial Intelligence Laboratory, January 1987.

[47] 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.

[48] Jacques Malenfant and Pierre Conte. Aspect-oriented programming versus reflection: a first draft, 1997.

[49] 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 [77], pages 300–315. Also SIGPLAN Notices 30(10), October 1995.

[50] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanbe, and Akinori Yonezawa. Object-oriented concurrent relective languages can be implemented efficiently. In Paepcke [60], pages 127–144. Also SIGPLAN Notices 27(10), October 1992.

[51] Jeff McAffer. *A Meta-Level Architecture For Prototyping Object Systems.* PhD thesis, The Graduate School of The University of Tokyo, 1995.

[52] Microsoft. Com home page on the internet. URL, August 2002.

[53] Philippe Mulet and Jacques Malenfant amd Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In Wilpolt [77], pages 316–330. Also SIGPLAN Notices 30(10), October 1995.

[54] James Norton. Dynamic class loading for c++ on linux. Linux, May 2000.

[55] Object Management Group. *CORBA Components*, February 1999.

[56] Alexandre Oliva and Luiz Eduardo Buzato. Composition of meta-objects in Guaraná. Technical report, Institudo de Computacão, Universidade Estadual de Campinas, Brazil, 1998.

[57] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical report, Instituto de Computacão, Universidade Estadual de Campinas, Brazil, 1998.

[58] Alexandre Oliva and Luiz Eduardo Buzato. An overview of molds: a meta-object library for distributed systems, 1998.

[59] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflective architecture of Guaraná. Technical report, Institudo de Computacão, Universidade Estadual de Campinas, Brazil, 1998.

[60] Andreas Paepcke, editor. *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Special Interest Group on Programming Languages, ACM Press, October 1992. Also SIGPLAN Notices 27(10), October 1992.

[61] N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a reflective component based middleware architecture, 2000.

[62] Renaud Pawlak. Internship report: Metaobject protocols for distributed programming. Technical report, Laboratories CNAM-CEDRIC, 1998.

[63] Alexandre Braga Ricardo. A meta-object protocol for secure composition of security mechanisms, 2000.

[64] Tilman Schaefer. *Supporting Meta-Types in a Compiled, Reflective Programming Language.* PhD thesis, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, University of Dublin, 2001.

[65] Tilman Schäfer, Peter Haraszti, and Vinny Cahill. Experiences with meta-level programming in a compiled reflectivelanguage: Implementing object persistence. Rejected paper from Reflection 2001 - The Third International Conference on Meta-Level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[66] A. R. Silva. Separation and composition of overlapping and interacting concerns. In *Proceedings of the ACM?*, pages 215–225, 1999.

[67] Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of theSymposium on Principles of Programming Languages*, pages 23–35. Association for Computing Machinery, 1984.

[68] Iona Sora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies specified by semantics-unaware properties. In *Proceedings of TOOLS Eastern Europe*, March 2002.

[69] Robert J. Stroud and Zhixue Wu. Using metaobject protocols to implement atomic data types. In Walter Olthoff, editor, *Proceedings of the* 9^{th}*European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 168–189. Springer-Verlag, August 1995.

[70] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition edition, 1997.

[71] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[72] Sun Microsystems. *Enterprise Java Beans Specifications Version 2.1*, August 2002. Proposed Final Draft.

[73] Takao Tenma, Yasuhiko Yokote, and Mario Tokoro. Implementing persistent objects in the Apertos operating system. In Luis-Felipe Cabrera and Eric Jul, editors, *Proceedings of the* 2^{nd}*International Workshop on Object-Orientation in Operating Systems*, pages 66–79. IEEE Computer Society, IEEE Computer Society Press, September 1992.

[74] Takuo Watanbe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *In Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL).*

[75] 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.

[76] Carrie Wilpolt, editor. *Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Special Interest Group on Programming Languages, ACM Press, October 1994. Also SIGPLAN Notices 29(10), October 1994.

[77] 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.

[78] Zhixue Wu and Robert J. Stroud. Using meta-objects to provide flexible system software. In *Proceedings of the OOPSLA '94 Workshop on Flexibility in System Software*, October 1994.

[79] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In Paepcke [60], pages 414–434. Also SIGPLAN Notices 27(10), October 1992.

[80] Yasuhiko Yokote, Gregor Kiczales, and John Lamping. Separation of concerns and operating systems for highly heterogeneous distributed computing. In *Proceedings of the 6th SIGOPS European Workshop*, pages 39–44. ACM Special Interest Group on Operating Systems, September 1994.

Appendix A

XML MOP descriptors

A.1 MOP descriptor for the TypeInfo MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This MOP retains type information about the classes that select it. -->
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="TypeInfo" Composability="Auto" Distribution="Common" Locality="Common">
  <ConcernArea Id="CA1" Exclusiveness="Singleton">
    <ConcernAreaElement>Structural Information</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="MClass" ReificationCategory="Class"/>
  <MetaObject ClassName="MAttribute" ReificationCategory="Attribute"/>
  <MetaObject ClassName="MMethod" ReificationCategory="Method"/>
</MOP>
```

A.2 MOP descriptor for the Default MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Default C++ object behaviour -->
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Default" Composability="Auto" Distribution="Local" Locality="NotApplicable">
  <SuperMOPs>
    <MOPRef Name="TypeInfo"/>
  </SuperMOPs>
  <ConcernArea Id="CA2" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Default</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="DefaultCreation" ReificationCategory="Creation"/>
  <MetaObject ClassName="DefaultDeletion" ReificationCategory="Deletion"/>
  <MetaObject ClassName="DefaultInvocation" ReificationCategory="Invocation"/>
  <MetaObject ClassName="DefaultStateRead" ReificationCategory="StateRead"/>
  <MetaObject ClassName="DefaultStateWrite" ReificationCategory="StateWrite"/>
  <MetaObject ClassName="DefaultSend" ReificationCategory="Send"/>
```

```
</MOP>
```

A.3 MOP descriptor for the Persistent MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Persistent" Composability="Auto" Distribution="Local" Locality="NotApplicable">
  <SuperMOPs>
    <MOPRef Name="Default"/>
  </SuperMOPs>
  <ConcernArea Id="CA3" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Persistence</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="PersistentClass" ReificationCategory="Class">
    <MiddlewareComponentRef Name="ClassRegister"/>
  </MetaObject>
  <MetaObject ClassName="PersistentObjectReference" ReificationCategory="ObjectReference"/>
  <MetaObject ClassName="PersistentCreation" ReificationCategory="Creation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ClassRegister"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentDeletion" ReificationCategory="Deletion">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentInvocation" ReificationCategory="Invocation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentStateRead" ReificationCategory="StateRead">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentStateWrite" ReificationCategory="StateWrite">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <ExtensionProtocol ClassName="Common"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="NameService"/>
    <MiddlewareComponent ClassName="ReferenceManager"/>
    <MiddlewareComponent ClassName="ClassRegister"/>
    <MiddlewareComponent ClassName="ObjectManager"/>
    <MiddlewareComponent ClassName="StorageManager"/>
  </MiddlewareLinks>
</MOP>
```

A.4 MOP descriptor for the Persistent2 MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Persistent2" Composability="Auto" Distribution="Local" Locality="NotApplicable">
  <SuperMOPs>
    <MOPRef Name="Default"/>
  </SuperMOPs>
  <ConcernArea Id="CA3" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Persistence</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="PersistentObjectReference" ReificationCategory="ObjectReference"/>
  <MetaObject ClassName="PersistentCreation" ReificationCategory="Creation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <MiddlewareComponentRef Name="ClassRegister"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentDeletion" ReificationCategory="Deletion">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <ExtensionProtocol ClassName="Common"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="NameService"/>
    <MiddlewareComponent ClassName="ReferenceManager"/>
    <MiddlewareComponent ClassName="ClassRegister"/>
    <MiddlewareComponent ClassName="ObjectManager"/>
    <MiddlewareComponent ClassName="StorageManager"/>
  </MiddlewareLinks>
</MOP>
```

A.5 MOP descriptor for the Persistent2Absent MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Persistent2Absent" Composability="Auto" Distribution="Local" Locality="NotApplicable">
  <SuperMOPs>
    <MOPRef Name="Persistent2"/>
  </SuperMOPs>
  <ConcernArea Id="CA3" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Persistence</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="PersistentInvocation" ReificationCategory="Invocation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentStateRead" ReificationCategory="StateRead">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <MetaObject ClassName="PersistentStateWrite" ReificationCategory="StateWrite">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
    <MiddlewareComponentRef Name="StorageManager"/>
  </MetaObject>
  <ExtensionProtocol ClassName="Common"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="NameService"/>
    <MiddlewareComponent ClassName="ReferenceManager"/>
    <MiddlewareComponent ClassName="ClassRegister"/>
    <MiddlewareComponent ClassName="ObjectManager"/>
    <MiddlewareComponent ClassName="StorageManager"/>
  </MiddlewareLinks>
</MOP>
```

A.6 MOP descriptor for the Logging MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Logging" Composability="Auto" Distribution="Common" Locality="Common">
  <SuperMOPs>
    <MOPRef Name="Default"/>
  </SuperMOPs>
  <ConcernArea Id="CA5" Exclusiveness="Multi">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Logging</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="LoggingCreation" ReificationCategory="Creation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="LogManager"/>
  </MetaObject>
  <MetaObject ClassName="LoggingDeletion" ReificationCategory="Deletion">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="LogManager"/>
  </MetaObject>
  <MetaObject ClassName="LoggingInvocation" ReificationCategory="Invocation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="LogManager"/>
  </MetaObject>
  <MetaObject ClassName="LoggingStateRead" ReificationCategory="StateRead">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="LogManager"/>
  </MetaObject>
  <MetaObject ClassName="LoggingStateWrite" ReificationCategory="StateWrite">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="LogManager"/>
  </MetaObject>
  <ExtensionProtocol ClassName="LEP"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="LogManager"/>
  </MiddlewareLinks>
</MOP>
```

A.7 MOP descriptor for the Remote MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Remote" Composability="Auto" Distribution="Distributed" Locality="Server">
  <SuperMOPs>
    <MOPRef Name="Default"/>
  </SuperMOPs>
  <ConcernArea Id="CA4" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Remote</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="RemoteObjectReference" ReificationCategory="ObjectReference"/>
  <MetaObject ClassName="RemoteCreation" ReificationCategory="Creation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ReferenceManager"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="RemoteDeletion" ReificationCategory="Deletion">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ReferenceManager"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="RemoteInvocation" ReificationCategory="Invocation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <MetaObject ClassName="RemoteStateRead" ReificationCategory="StateRead">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <MetaObject ClassName="RemoteStateWrite" ReificationCategory="StateWrite">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <ExtensionProtocol ClassName="Common"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="ReferenceManager"/>
    <MiddlewareComponent ClassName="ObjectManager"/>
    <MiddlewareComponent ClassName="NameService"/>
    <MiddlewareComponent ClassName="CommunicationService"/>
  </MiddlewareLinks>
</MOP>
```

A.8 MOP descriptor for the RemoteProxy MOP

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOP SYSTEM "composition.dtd">
<MOP Name="Remote" Composability="Auto" Distribution="Distributed" Locality="Proxy">
  <SuperMOPs>
    <MOPRef Name="Default"/>
  </SuperMOPs>
  <ConcernArea Id="CA4" Exclusiveness="Singleton">
    <ConcernAreaElement>Behaviours</ConcernAreaElement>
    <ConcernAreaElement>Remote</ConcernAreaElement>
  </ConcernArea>
  <MetaObject ClassName="RemoteObjectReference" ReificationCategory="ObjectReference"/>
  <MetaObject ClassName="RemoteCreation" ReificationCategory="Creation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ReferenceManager"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="RemoteDeletion" ReificationCategory="Deletion">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <AfterBehaviour CallNext="UnconditionalNext" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="ReferenceManager"/>
    <MiddlewareComponentRef Name="ObjectManager"/>
  </MetaObject>
  <MetaObject ClassName="RemoteInvocation" ReificationCategory="Invocation">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <MetaObject ClassName="RemoteStateRead" ReificationCategory="StateRead">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <MetaObject ClassName="RemoteStateWrite" ReificationCategory="StateWrite">
    <BeforeBehaviour CallNext="UnconditionalNext" CalledBy="Middleware"/>
    <AfterBehaviour CallNext="ActivatesMiddleware" CalledBy="Metalevel"/>
    <MiddlewareComponentRef Name="CommunicationService"/>
  </MetaObject>
  <ExtensionProtocol ClassName="Common"/>
  <MiddlewareLinks>
    <MiddlewareComponent ClassName="ReferenceManager"/>
    <MiddlewareComponent ClassName="ObjectManager"/>
    <MiddlewareComponent ClassName="NameService"/>
    <MiddlewareComponent ClassName="CommunicationService"/>
  </MiddlewareLinks>
</MOP>
```