The Reflective Architecture of **Guaraná**

Alexandre Oliva
oliva@dcc.unicamp.br

Islene Calciolari Garcia
islene@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

September 1998

Abstract

This text describes a reflective software architecture called **Guaraná**. Its run-time meta-level protocol has been designed to achieve a very high degree of flexibility, reconfigurability, security and reuse of meta-level code. *Composers* are *meta-objects* that can be used to combine meta-objects, that may themselves be composers, into dynamically modifiable *meta-configurations*. Instances of a class may have different meta-configurations, either determined explicitly or derived from the context in which every single object was created.

A free Java[1]-based implementation of the language-independent **Guaraná** reflective architecture is currently available.

1 Introduction

As the size and complexity of systems increase, so does the need for mechanisms to deal with such complexity. Object-oriented design is based on abstraction and information hiding (encapsulation) [4, 13, 30, 31]. These concepts have provided an effective framework for the management of complexity of applications. Within this framework, software developers strive to obtain applications that are highly coherent and loosely coupled. High coherence translates into narrow and easy-to-understand interfaces, as highly coherent components tend to do just one thing, leading to functional simplicity and component autonomy. Loosely coupled components are components that are connected by simple communication structures, thus their relationships are easy to understand and less prone to domino effects caused when errors occur at one location and propagate through the application. Unfortunately, object-oriented design alone does not address the development of software that can be easily adapted. *Adaptability* and *transparency of coupling* is playing an increasingly important role in the software development process, that is now carried out in a much more dynamic market where requirement shifts force developers to adapt already existent software to originally unforeseen conditions (requirements). The next two paragraphs review solutions that have been proposed to these two problems, respectively. After presenting these solutions, we argue that our reflective software architecture, **Guaraná**, represents a step forward towards the construction of open, easily adaptable applications.

The concept of open architectures [14, 15] has been proposed as a partial solution to the problem of creating software that is not only modular, well-structured, but also easier to adapt. Open architectures are based on the existence of an additional component (object) interface that allows

[1] Java is a trademark of Sun Microsystems, Inc.

them—acting as servers—to dynamically adapt to new requirements presented by their clients. Open architectures encourage a modular design where there is a clear separation of *policy*, that is, *what* a module has been designed for, from the *mechanisms* that implement a policy, that is, *how* a policy is materialized. Although open architectures might seem to confront the modular design approach by exposing parts of their designs, in fact, the opposite is valid. Open architectures may provide elegant solutions to the design and implementation of highly adaptable software. In particular, the implementation of system-oriented mechanisms such as concurrency control, distribution, persistence and fault-tolerance can benefit from this approach to software construction.

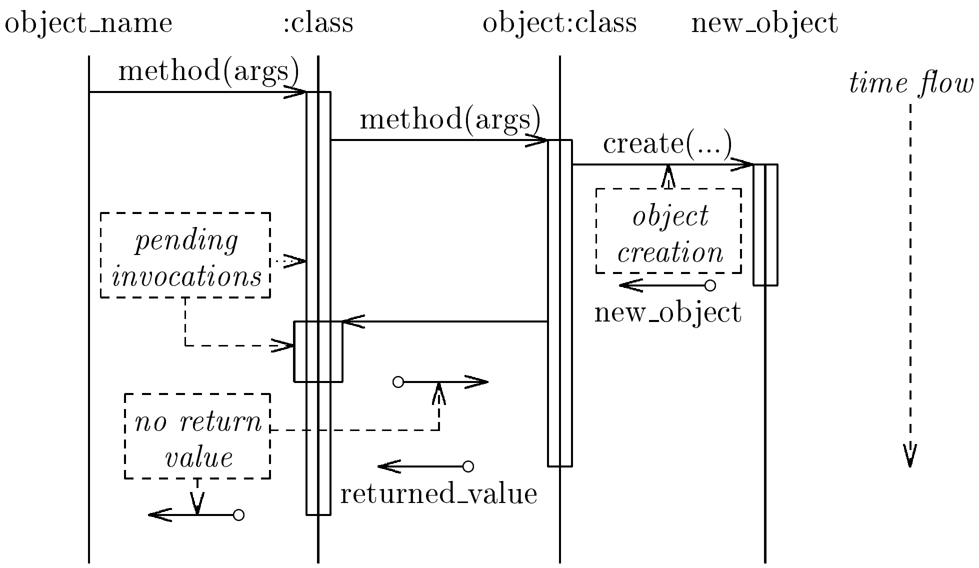
Computational reflection [21, 33] (henceforth just reflection) has been proposed as a solution to the problem of creating applications that are able to maintain, use and change representations of their own designs (structural or behavioral). Reflective systems are able to use self-representations to extend and *adapt* their computation. Due to this property, they are being used to implement open software architectures. Additionally, the mechanisms used to implement reflective systems can also be used to partially solve the problem of coupling software components transparently. In reflective architectures, components that deal with the processing of self-representation and management of an application reside in a software layer called *meta-layer* or *meta-level*. Components that deal with the functionality of the application are assigned to a software layer called *base-layer* or *base-level*. The transparent coupling of the base-level to its meta-level is implemented using *interception* mechanisms. In object-oriented reflective systems, objects that reside in the meta-level and base-level are called *meta-level objects* and *base-level objects*, respectively.

A comparative study of existent object-oriented reflective architectures reveal that some associate every base-level object with a *single meta-level object* called meta-object [21, 39]. Several offer class-wide reflective facilities: each class is associated with a *single* meta-class [7, 8, 11, 16]. Some allow groups of objects to be attached to groups of meta-objects [12, 23, 24, 25, 41], by giving every meta-object the responsibility for handling a specific aspect of the system–object interaction. Also, hybrid models [22] exist.

Due to their inherent structure, the existing reflective architectures may induce developers to create complex meta-objects that, in an all-in-one approach, implement many policies (management aspects) of an application or, alternatively, to construct coherent but tightly coupled meta-objects and objects. Both alternatives harden reuse, maintenance, and adaptation of an application, especially of its meta-level, as it is where most of the adaptations tend to occur in an open architecture. There is certainly room for improvement of the mechanisms that give support to adaptation and transparent loose coupling.

This paper describes the reflective architecture of **Guaraná**, a flexible language-independent meta-protocol that encourages the creation of highly coherent, loosely and transparently coupled meta-level objects. An application developed within **Guaraná**'s framework is easily adapted to conform to new requirements, and the implementation of meta-level requirements can be easily reused in other applications. Section 2 contains an introduction to computational reflection, as used to support open architectures. Section 3 describes the software architecture and meta-level protocol of **Guaraná**. Finally, in Section 4, we present some conclusions and discuss future work.

Along the text, we are going use diagrams to illustrate the main characteristics of **Guaraná**. The graphical conventions used in these diagrams are those of UML (Unified Modeling Language) sequential diagrams [30]. Figure 1 specifies the semantics of such diagrams.



The meta-diagram above shows the subset of the UML [30] sequence diagrams notation (formerly known as interaction diagrams) we are going to use in this paper. The first lines of the diagram define one column for each object. The name of an object and the name of its class (or a superclass thereof, for the sake of generalization) may be specified.

Time flows downwards. Method invocations are shown as arrows from the caller to the callee. Until the callee returns, its time line is adorned with a rectangle. The return is denoted with a short arrow adorned with a circle; the returned value is specified just below the arrow. Nested invocations are represented by wider rectangles.

The time line of an object that is created during the time span covered by the diagram starts at the moment of the invocation of the pseudo-method `create`. Sometimes, this pseudo-method will be explicitly split into invocations of the pseudo-methods `alloc` and `init`: the former just allocates memory for the object; the latter initializes (constructs) it.

Figure 1: UML Sequence Diagrams

2 Computational Reflection

Computational Reflection [21, 33] is a technique that allows a system to maintain information about itself (meta-information) and use this information to change its behavior (adapt).

This is achieved by processing in two well-defined levels: functional level (also known as base level or application level) and management (or meta) level. Aspects of the base level are represented as objects in the meta level, in a process called *reification* (Section 2.1). Meta-level architectures are discussed in Section 2.2 and reflective languages in Section 2.3. Finally, Section 2.4 shows the use of computational reflection in the structuring and implementation of system-oriented mechanisms.

2.1 Reification

For the meta level to be able to reflect on several objects, specially if they are instances of different classes, it must be given information regarding the internal structure of objects. This meta-level object must be able to find out what are the methods implemented by an object, as well as the fields (attributes) defined by this object. Such base-level representation, that is available for the meta level, is called *structural meta-information*. The representation, in form of objects, of abstract language concepts, such as classes and methods, is called *reification*.

Base-level behavior, however, cannot be completely modeled by reifying only structural aspects of objects. Interactions between objects must also be materialized as objects, so that meta-level objects can inspect and possibly alter them. This is achieved by intercepting base-level operations such as method invocations, field value inquiries or assignments, creating operation objects that represent them, and transferring control to the meta level, as shown in Figure 2. In addition to receiving reified base-level operations from the reflective kernel, meta-level objects should also be able to create operation objects, and this should be reflected in the base level as the execution of such operations.

2.2 Reflective Architectures

In this Section we briefly summarize some of the drawbacks of existent reflective architectures, that have motivated the creation of **Guaraná**. Open C++ [7, 8] and CLOS [16] collapse all the meta-level processing in a single monolithic meta-object. However, there are situations where several mechanisms related to independent requirements have to be combined to serve a single object. The systems above encourage the all-in-one approach for the implementation of these mechanisms in the meta-level, creating objects that are complex and hard to adapt. Ideally, we would like to have a reflective architecture that would allow developers to create several smaller and very coherent objects linked by simple couplings. MetaXa [17] (formerly known as MetaJava), for example, allows multiple meta-objects to form of a linked list whose tail is the base-level object, but it lacks a coordination mechanism for allowing them to cooperate.

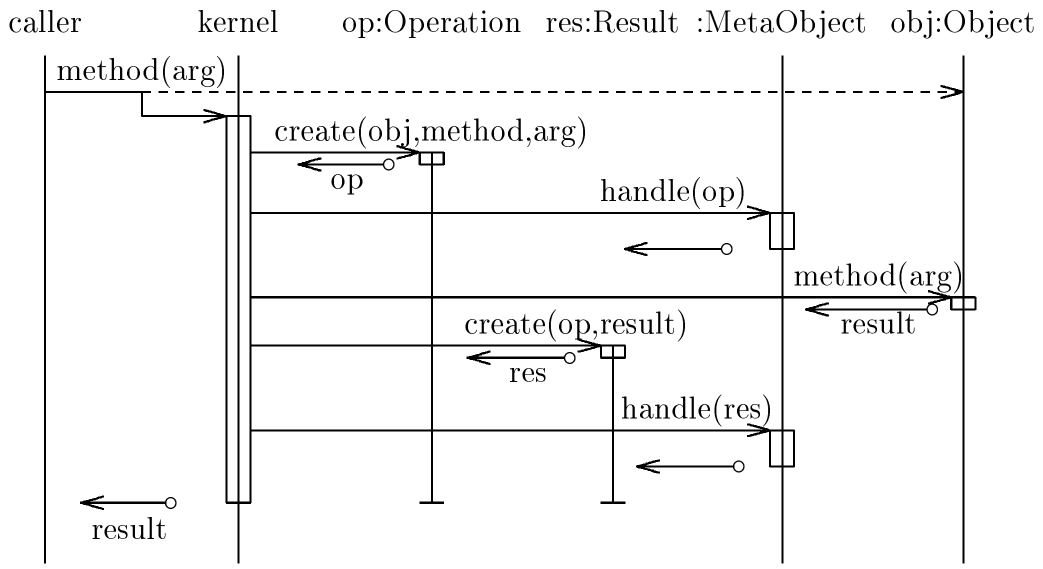
Apertos [40], MMRF [24, 25], CodA [23] and Iguana [12] base their reflective architecture upon fine-grained coordinated meta-level objects. However, their meta-objects present different interfaces and interaction patterns, with tight coupling (interdependency). We believe this architecture complicates usage and composition of meta-objects. The drawback due to multiple interfaces can be weakened through the provision of a single narrow yet powerful meta-object interface. In this paper, we are going to present such a narrow interface that is easy to learn and simple to use, yet it supports mechanisms for easy composition and reconfiguration of meta-objects through the provision of a loosely coupling pattern.

2.3 Reflective Languages

Several object-oriented languages have already been designed or extended in order to support reflection. Languages such as KRS (3-KRS [21]), LISP (3-LISP [32], CLOS [16]), ABCL (ABCL/R [39] and ACBL/R2 [22]), AL-1/D (MMRF [24, 25]), C++ (Open C++ [7, 8] and Iguana [12]) and Java (MetaXa [17]) are examples of languages that provide varied levels of support to reflection.

In a totally reflective system, any kind of meta-information should be modifiable, and any such modification should reflect upon the base-level behavior, in a causally-connected way [21]. Although changing reified operations is possible even in compiled languages, changing structural meta-information is usually possible only in interpreted languages. Some interpreted reflective languages allow replacement interpreters to be written in the language itself [21, 32, 39]. Such interpreters may change the behavior of the built-in interpreter, and may themselves be interpreted by other replacement interpreters. These interpreters are called meta-circular interpreters.

Extending non-reflective compiled languages to support reflection usually involves some kind of source code preprocessing. Such preprocess-



A reflective kernel is responsible for implementing an interception mechanism. The method invocation is reified as an operation object and passed for the callee's meta-object to reflect upon (handle). Eventually, the meta-object requests the kernel to deliver the operation to the callee's application object, by returning control (as in the diagram) or performing some special meta-level action. Finally, the result of the operation is reified and presented to the meta-object.

Figure 2: Reifying an operation

ing adds interception and control mechanisms, so that meta-objects are informed of operations sent to base-level objects and can deliver operations to them. If the original language does not provide structural meta-information, the preprocessor is also responsible for collecting it and arranging that it is available to meta-level objects at run-time.

Some reflection techniques can be used in programming languages that offer none or some very restricted form of the mechanisms used by reflective systems. These shortcomings usually restrict the form of reflection implemented, limiting the tower of meta-objects [21] to only two levels; the work by Bijnens et al [3] is an example of this restricted use of reflection. Ideally, reflective software architectures should allow infinite tower of meta-objects to be built, that is, objects have meta-objects, meta-objects have meta-meta-objects, and so on.

2.4 Transparency

In a reflective application, the base level implements the main functionality of an application, while the meta level is usually reserved for the implementation of management requirements, such as persistence [28, 34], location transparency [26], replication [8, 18], fault tolerance [1, 2, 9, 10] and atomic actions [35, 37]. Reflection has also been shown to be useful in the development of distributed systems [6, 20, 36, 40, 41] and for simplifying library protocols [38].

Persistence [28, 34], for example, can be implemented through the interception of update operations sent to an object. The intercepted operations are sent to the meta-level where the object responsible for the persistence mechanism guarantees that changes made to the object are kept in stable storage. Transparency of locality [26] may be achieved by intercepting operations addressed to proxies of objects in other address spaces: meta-level objects in the caller's address space would forward operations to meta-level objects located in the callee's address space. Finally, the operation is performed and its result is sent back to the caller.

Reflective architectures can be used to create applications that attend to certain requirements and later evolve to comply with new requirements, added to or removed from its specification, depending on how their environment evolves. As an example, consider the case of an application whose objects are persistent but not replicated. Later, due to the requirement of availability, some of its objects have to become replicated. Persistence and replication, in a reflective architecture, can be implemented at the meta level of the architecture. The addition of these mechanisms can be attained with varied degrees of transparency, depending on the coupling and interception mechanisms offered by the architecture.

Being able to dynamically associate non-functional requirements to groups of objects of an application, independently of the types of these objects, favors transparency and adaptability, but may be costly in terms of performance.

3 The Meta-level Protocol of Guaraná

One of the most important features of a reflective architecture is its meta-level protocol. This is also valid for **Guaraná**, its meta-level protocol is greatly responsible for the communication and coupling pattern that induces software developers to create well-structured and adaptable configurations of meta-objects.

This section begins with an analysis of features of programming languages and/or existing reflective kernels that may ease the implementation of **Guaraná** upon them. Next, we present **Guaraná**'s meta-level protocol, namely, *meta-objects* and *composers* and the coupling patterns they induce on meta-objects. Then, we show how these components can be combined to form *meta-configurations*. They are the key to the creation of highly coherent and loosely coupled—adaptable—implementations of well-structured object-oriented designs. Finally, we discuss some security aspects of the **Guaraná** reflective architecture.

3.1 The kernel of Guaraná

The basic architecture of **Guaraná**, its kernel, can be implemented atop of any software platform, with different levels of difficulty, depending on how close the mechanisms implemented by the platform are to the mechanisms necessary to implement **Guaraná**.

The kernel realizes the following basic mechanisms: (i) operation interception and reification, (ii) dynamic binding and invocation for objects of the meta level, and (iii) maintenance of the structural meta-information.

3.2 Meta-Objects

We define a meta-object as a compoundable meta-level object responsible for implementing part of the reflective behavior of an application. Each object may be directly associated with either zero or one meta-object, called the *primary meta-object* of that object. Its role is to observe all operations addressed to its associated object, as well as their results. The observation is guaranteed by the interception and reification mechanisms implemented in the kernel.

A class can also be associated with a primary meta-object, that will observe all class-related operations, and no instance-related ones. Thus, the meta-objects of classes and its instances are independent of each other. Even if a class is associated with a meta-object, if some of its instances are not, operations addressed to these instances will not be intercepted.

Software engineering techniques, inclusive object-oriented, recommend the design and implementation of highly coherent and loosely coupled objects. One of the interesting attributes of **Guaraná** is its support for transparent loose coupling between objects. In **Guaraná**, unlike most of the other existent reflective architectures, base-level objects do not refer to their meta-level counterparts; they are not allowed to obtain references to their meta-objects. Coupling between object and meta-object is supported by the interception and reification of operations and by a dynamic binding mechanism; the kernel method reconfigure is responsible for binding objects to their meta-objects.

A primary meta-object inspects operations and reflects upon their contents, returning to the kernel one of three possible outcomes:

1. a result, that will be regarded by the kernel as if it were produced by the actual execution of the operation;

2. a replacement operation, that the kernel will deliver to the base-level object, discarding the original one; or

3. none of the above, i.e., the kernel will deliver the original operation to the application object.

In the alternatives 2 and 3, where the meta-object does not provide a result, it may signal to the kernel that it intends to inspect or even to modify the result of the operation. In this case, after the operation is performed, the kernel will reify its result and present it to the primary meta-object. At this point, the primary meta-object may perform any appropriate action. For example, it may compute a different result for the operation, and return it. The kernel will only accept this modified result if the meta-object had indicated that it would modify it.

3.3 Composers

Guaraná allows multiple meta-objects to be (indirectly) associated with an application object. This design creates the problem of organizing the flow of intercepted operations through the meta-objects. A specialized form of meta-object called *composer* is responsible for the enforcement of the policies that give structure and order to the flow of operations delegated to meta-objects.

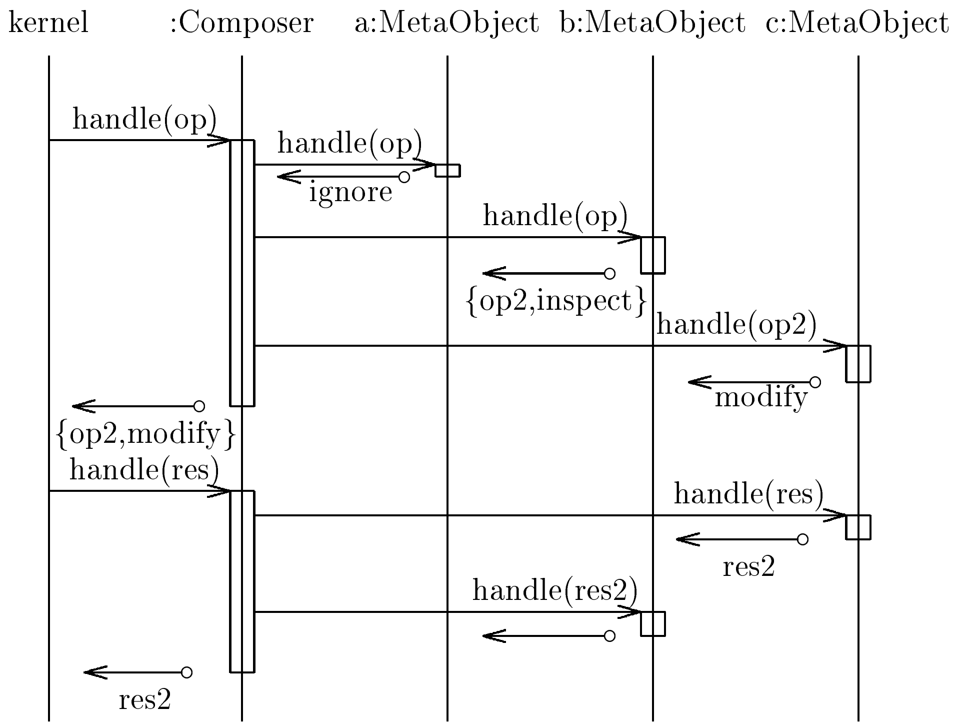
Composers can be used to group meta-objects that are commonly used together, and these groups can be composed further, forming recursive, potentially infinite, hierarchy of meta-objects. These groups can be used as building blocks for setting up complex meta-level configurations.

We have implemented a simple (yet very useful) type of composer: the sequential composer. It organizes meta-objects in sequence, mostly like a stack: operations are fed to meta-objects descending in the stack, whereas results are presented in the reverse order to meta-objects that have requested to inspect them or to change them. Figure 3 illustrates the behavior of a sequential composer.

A concurrent composer might have been implemented too: it would present an operation to all meta-objects concurrently. Some may generate results or actions that conflict with those generated by others. In order to cope with this situation a default adjudicator could be provided: it would raise an exception to signal the conflict. Specializations of this composer may add decision-making mechanisms to the adjudicator, so that it is able to solve some conflicts.

Other more specialized composers can be implemented, as well as other generic implementations that handle conflicts in different ways, or that specify different policies for ordering the flow of operations forwarded to meta-objects. Composers may also be used to filter operations that need not be forwarded to certain meta-objects.

To guarantee adaptability, the design of **Guaraná** precludes non-composer meta-objects



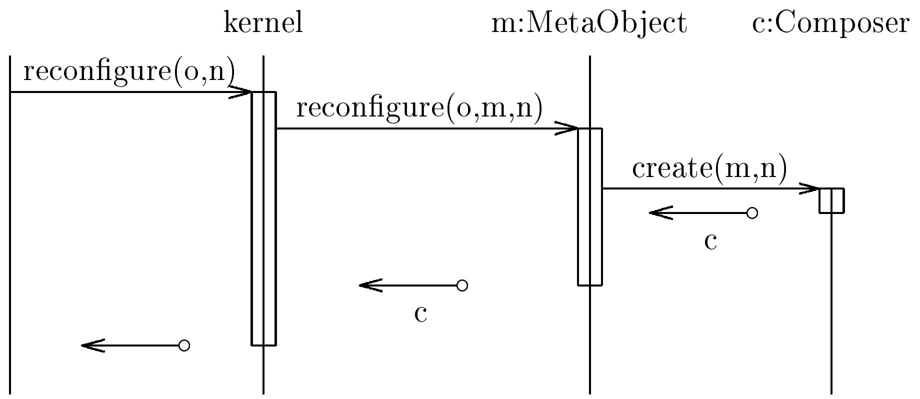
An operation `op` is intercepted by the kernel and passed to a sequential composer, that is the primary meta-object of the target object of the operation. Next, the composer forwards it to meta-object `a`, that indicates it is not interested in the result of the operation. Then, to meta-object `b`, that produces a replacement operation `op2`, suggesting it is interested in inspecting its result. Finally, the replacement operation is presented to meta-object `c`, that returns a request to modify the result. After the operation is delivered, its result `res` is presented to meta-object `c`, that modifies the result to `res2`. Then, meta-object `b` is informed of the replaced result `res2`, but meta-object `a` is not. It is interesting to note that meta-objects `a`, `b` or `c` could be composers themselves, delegating operations to other meta-objects.

Figure 3: Sequential composition of meta-objects

of maintaining direct references to other meta-objects. However, meta-objects may have to interact. To illustrate the need for interaction, consider the case of a persistent aggregate object. Making the whole aggregate persistent requires the application of the persistence mechanism to each of the component objects. In this case, the meta-objects of each component must communicate to ensure that all components have their state saved to stable storage. **Guaraná** implements a `broadcast` operation that can deliver arbitrary messages to all meta-objects associated with an application object.

3.4 Meta-configuration management

When an application starts up, every object has an empty meta-configuration, that is, no object is subject to reflection. The application may create objects and meta-objects, then associate



This figure shows an arbitrary object issuing a request to change the primary meta-object of object o. The requester intends to have meta-object n as the primary meta-object of o. However, meta-object m is the primary meta-object of o, so it can determine what the new meta-configuration is going to be, possibly overruling the requester. Meta-object m decides to create a composer c, that will delegate to both m and n, and returns this new composer, that becomes the primary meta-object of object o.

Figure 4: Reconfiguring an object's meta-configuration

them, by requesting the kernel of **Guaraná** to reconfigure the object's meta-configuration with a given meta-object.

When a reconfiguration request is issued on a reflective object, i.e., one that already has a non-empty meta-configuration, its existing meta-configuration is requested to determine the new meta-configuration, based on itself and on the requested meta-configuration. It may either (i) allow a complete replacement of itself with the suggested meta-configuration, (ii) reject any modifications, (iii) incorporate the suggested meta-configuration, or (iv) create a completely new meta-configuration, as depicted in Figure 4. Note that, when the kernel invokes the method reconfigure of the primary meta-object, it passes the primary meta-object itself as an argument. This signals the meta-object it is the root of the reconfiguration that should take place.

If the object whose meta-configuration was to be reconfigured had an empty meta-configuration, a message with the object and its suggested meta-configuration will be broadcast to the meta-configuration of the object's class, whose meta-objects may modify the meta-configuration to be used. The message will also be broadcast to all superclasses of the object's class, so that any class will be able to reject or modify reconfiguration requests of its previously non-reflective instances.

In addition to a mechanism to replaces the whole meta-configuration of an object, the kernel of **Guaraná** provides an additional method that allows the caller to specify which meta-object should be looked for and replaced. This reconfiguration request traverses the meta-configuration just like a broadcast message. Each meta-object should check whether it is the root of the meta-configuration, and decide what to do with regard to the reconfiguration request accordingly. A meta-object that is not the root of the meta-configuration request will usually ignore it, returning itself. However, one that is the root should behave just like the primary meta-object in Figure 4. Thus, in order to be able to replace a meta-object, it is not necessary to know whether it is the primary meta-object or if some composer refers to it.

3.4.1 Propagation of meta-configurations

The meta-level protocol of **Guaraná** defines the way meta-configurations for newly-created objects are established. Whenever a reflective object c creates another object, say n, just after the system allocates memory for n, but before it is initialized (constructed), the primary meta-object of the creator meta(c) is requested to provide a primary meta-object for o. meta(c) may create a new meta-object, return itself or even an empty meta-configuration. A composer may delegate the configuration request to its component meta-objects, then create and return a new composer for n that composes the meta-objects returned by them.

After the meta-configuration of n is established, a NewObject message is broadcast to the meta-configuration of n's class. With this mechanism, the components of the meta-configuration of a class can affect the meta-configurations of its instances, by issuing reconfiguration requests.

In addition to the language-defined mechanism to create new objects, the kernel of **Guaraná** provides a method that creates a proxy object of a given class and associates it with a specified primary meta-object. This method allocates memory for an object without constructing an actual object there. This is useful for transparent distribution and for re-instantiating objects previously saved in stable storage.

When a proxy is created, a message of type NewProxy (instead of NewObject) is broadcast to the proxy object's class before the suggested meta-object is associated with the proxy object. Although NewProxy is a subclass of NewObject, these messages may be handled differently. For example, meta-configurations of certain classes may prevent the creation of proxy objects by throwing exceptions when presented NewProxy messages.

After the message is broadcast, the kernel of **Guaraná** issues a reconfiguration request, so as to set up the suggested meta-object as the primary meta-object of the proxy. However, during the broadcast, a different meta-configuration may have been established already.

3.5 Security

The need for what we call the security model of **Guaraná**, in a reflective architecture, is the same as for data encapsulation in an object-oriented architecture. By completely hiding from the base level details of the implementation of the meta level, we help ensure a proper separation of concerns between the application level and the management level. Furthermore, by preventing arbitrary interactions of the base level with the meta level, we make it possible to implement meta-objects and verify their correctness, no matter what base-level object they are associated with. For the same reasoning, we forbid meta-objects from accessing objects they are not associated with.

Some design decisions that support these goals have been exposed already:

- it is not possible to obtain a reference to any meta-object associated with an object, unless the meta-object itself is willing to provide such a reference;
- it is possible to request for a modification of the meta-configuration of an object without previous knowledge of any of its component meta-objects, but any modification must be approved by the existing meta-configuration;
- the meta-configuration of an object defines an execution context that can be propagated to whatever additional objects this object creates;
- a class is able to prevent instances that would violate internal security constraints from being created.

The hierarchical organization of meta-objects in a meta-configuration may be seen as directed graph, but it is likely to be an acyclic one, and most likely it will be a tree. The primary meta-object is the root of the tree, and composers are parents of meta-objects they delegate to. This view suggests a natural definition of a hierarchy of control of meta-level processing. Meta-objects closer to the root of the tree are able to filter out messages, operations, results and reconfiguration requests, preventing that they reach untrustworthy components of their subtrees. Furthermore, in the operation and result handling protocol, a meta-object that is higher in the hierarchy may decide not to accept a replacement operation or a result provided by a meta-object lower in the hierarchy.

There is an additional issue regarding meta-level security, that has to do with the ability to create meta-level representations of operations addressed to a base-level object. In principle, only component meta-objects of the meta-configuration of an object should be able to create operations addressed to that object. On one hand, this provides an apparently reasonable security constraint that prevents any object from gaining privileged access to any other object. On the other hand, this model may be excessively restrictive on some situations, because such meta-objects might prefer to delegate their security privileges to other meta-level objects. Furthermore, this model would provide an all-or-nothing authorization control: the meta-objects of an object would be able to create *any* operation addressed to the base-level object; that might be undesirable if we would rather restrict the set of operations available for meta-objects.

Given this analysis, we have come up with a solution based on operation factories. The primary meta-object of an object is given an operation factory, an object that is capable of creating any operation addressed to the base-level object the meta-object is associated with. It may distribute this operation factory to other meta-objects it delegates to, or to other meta-level objects that may need to create operations addressed to the base-level object. However, it may create another operation factory that refuses to create certain kinds of operations, but that delegates valid requests to the operation factory it had access to. Then, it may distribute such restricted operation factories to its sub-meta-objects.

With this model, the mechanism that controls the creation of operations resembles the hierarchy of meta-objects, without requiring lower meta-objects to have references to higher ones. A similar mechanism for result objects is not necessary, since results are always returned by lower meta-objects to higher ones, whereas operations may be created and performed.

An important fact to note is that an operation created in the meta-level is never performed unless the primary meta-object associated with its target object determines so, when requested to handle it. This is important because, even if an operation factory becomes invalid—it does so whenever the primary meta-object changes—, an evil meta-object might have created an operation while the operation factory was still valid. This operation will *not* be delivered to the target object before it is handled by the current object's meta-configuration. If the primary meta-object of a meta-configuration changes while an operation is being handled, an abort result will be presented to the previous primary meta-object, and the operation handling is restarted with the new primary meta-object.

It is possible to create invalid operations—that refer to inexistent methods or fields, or whose types or argument counts do not match the expected ones. Such operations might have been rejected at operation creation time, but we decided to postpone this verification to the moment just before the operation is delivered, or whenever a meta-object requests so. The rationale is that one may create additional pseudo-fields or pseudo-methods in an object, that are only accessible from the meta level. Such invalid operations will never be delivered to the target object: meta-objects should create suitable results or replacement operations but, if they do not, the operation will fail, and its result will be an exception indicating this failure. This feature has proven to be useful for creating pseudo arrays that map into databases, for providing advanced overload resolution mechanisms and for sharing data among meta-objects in a safer way than using broadcast messages—because only the components of the meta-configuration or objects authorized by them may create such operations.

An operation factory can be used to create do-nothing operations, that act as mere placeholders until they reach a particular meta-object that replaces it with another operation. This makes it possible for a meta-object to create an operation that will only be observed by meta-objects that are placed after itself in a sequence of meta-objects.

3.6 Libraries of Meta-Objects

The meta-level protocol of **Guaraná** was designed in a way that makes it possible to create libraries of meta-objects that implement specific meta-level behaviors, and to easily compose them into complex meta-configurations. A good example of this is **MOLDS** [27], a Meta-Object Library for Distributed Systems, that provides meta-objects for persistence, distribution, replication, atomicity and migration. **Guaraná** and **MOLDS** are parts of a larger project [5].

4 Conclusions

As a response to technological changes, such as the massive use of microprocessors, fast networks and bit-mapped monitors, the software engineering process has moved from the traditional sequential software production paradigm to an evolutionary model of software development. The sequential approach will remain applicable to those problems in which requirements are well defined and complexity is relatively low. Problems that do not fit into this category will very likely be subject to an adaptative or evolutionary software engineering process. Object-orientation, openness and computational reflection offer promise of shortening development cycles through reuse/adaptation of software developed and tested in the previous iteration of the evolutionary development process. **Guaraná** has been designed to take the benefits of open and reflective software architectures a step further. We have studied existent open software architectures and determined that they could be improved through the use of composers and meta-objects that allow the implementation of highly coherent and loosely coupled software [29]. Composers and meta-objects form a framework that allows software developers to map a loosely-coupled and highly-coherent object model into an implementation that preserves these properties. The preservation of structural and communication properties in the implementation is essential to facilitate the application of the evolutionary software engineering techniques.

We have implemented this software architecture by modifying the *Kaffe OpenVM*™, an implementation of the Java Virtual Machine Specification [19] whose source code is distributed under the GNU General Public License. The Java programming language, on the other hand, has not been changed at all: any program created and compiled with any Java compiler will run on our implementation, and it will be possible to use reflective mechanisms in order to adapt and/or extend them.

A Obtaining Guaraná

Additional information about **Guaraná** can be obtained in the Home Page of **Guaraná**, at the URL `http://www.dcc.unicamp.br/~oliva/guarana/`. The source code of its implementation atop of the *Kaffe OpenVM*, on-line documentation and full papers are available for download. **Guaraná** is *Free Software*, released under the GNU General Public License, but its specifications are open, so non-free clean-room implementations are possible.

B Acknowledgments

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grants 95/2091-3 for Alexandre Oliva, 95/1983-8 for Islene Calciolari Garcia and 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*).

References

[1] Gul Agha, Svend Frølund, Rajendra Panwar, and Danield Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *DCCA3 — Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 197–206, September 1993.

[2] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis, and M. L. Lisboa. Reflective Architectures for Reusable Fault-Tolerant Software. In *PANEL'95 — XXI Conferência Latino–Americana de Informática*, March 1996.

[3] Stijn Bijnens, Wouter Joosen, and Pierre Verbaeten. A reflective invocation scheme to realise advanced object management. In *Object-Based Distributed Programming ECOOP '93 Workshop*, July 1993.

[4] Grady Booch. *Object Oriented Analysis & Design.* Benjamin Cummings, second edition, 1994.

[5] L.E. Buzato, H.K. Liesenberg, C.M.F. Rubira R. Anido, and M.B.F. de Toledo. Uma arquitetura de software para o desenvolvimento de aplicações distribuídas confiáveis. In *First Workshop on Distributed Systems (WoSid'96)*, Salvador, BA, Brasil, May 1996.

[6] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and Implementing Choices: An Object-Oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[7] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA'95*, volume 30, pages 285–299, October 1995.

[8] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In N.M. Nierstrasz, editor, *ECOOP'93*, pages 482–501, 1993.

[9] J. C. Fabre, T. Perennou, and L. Blain. Meta-object Protocols for Implementing Reliable and Secure Distributed Applications. Technical Report LASS–95037, Centre National de la Recherche Scientifique, February 1995.

[10] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, and Zhixue Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *25th Simposium on Fault-Tolerant Computing Systems*, pages 291–311, Pasadena, CA, June 1995.

[11] J. Ferber. Computation reflection in class-based object-oriented languages. *OOPSLA'89*, 24(10), October 1989.

[12] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. In *Proceedings of Reflection '96*, pages 137–152, San Francisco, USA, April 1996.

[13] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley/ACM Press, Reading, Mass., 1992.

[14] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[15] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[16] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, chapter 5,6. MIT Press, 1991.

[17] Jürgen Kleinöder and Michael Golm. MetaJava: An efficient run-time meta architecture for Java. In *International Workshop on Object Orientation in Operating Systems – IWOOOS'96*, Seattle, Washington, October 1996. IEEE.

[18] Jürgen Kleinöder and Michael Golm. Transparent and adaptable object replication using a reflective Java. Technical Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, September 1996.

[19] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Java Series. Addison–Wesley, January 1997.

[20] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: An operating system perspective. Technical report, University of Illinois at Urbana-Champaign, March 1992.

[21] Pattie Maes. Concepts and experiments in computation reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[22] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. In *ECOOP'91*, July 1991.

[23] Jeff McAffer. Meta-level programming with CodA. In *ECOOP'95*, pages 190–214, August 1995.

[24] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *IMSA'92 International Workshop on Reflection and Meta-level Architecture*, November 1992.

[25] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Metalevel decomposition in AL-1/D. In *1st International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, November 1993.

[26] Kideaki Okamura and Yutaka Ishikawa. Object Location Control Using Meta-level Programming. In *ECOOP'94*, pages 299–319, 1994.

[27] Alexandre Oliva and Luiz Eduardo Buzato. An overview of MOLDS: A Meta-Object Library for Distributed Systems. Technical Report IC-98-15, Instituto de Computação, Universidade Estadual de Campinas, April 1998.

[28] Andreas Paepcke. PCLOS: A flexible implementation of CLOS Persistence. In *ECOOP'88*, pages 3 74–389, August 1988.

[29] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 4th edition, 1997.

[30] Rational Software Corporation. *Unified Modeling Language v1.0.1*, January 1997.

[31] Jim Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Desing*. Prentice Hall, 1991.

[32] Brian C. Smith. Reflection and semantics in lisp. In *ACM POPL '84*, pages 23–35, 1984.

[33] Brian C. Smith. Prologue to "Reflection and Semantics in a Procedural Language". PhD Thesis Prologue, 1985.

[34] R. J. Stroud and Z. Wu. Using meta-objects to adapt a persistent object system to meet applications needs. In *6th SIGOPS European Workshop on Matching Operating Systems to Applications Needs*, 1994.

[35] R. J. Stroud and Z. Wu. Using Metaobject Protocols to Implement Atomic Data Types. In *ECOOP'95 – 9th European Conference*, pages 168–189, August 1995.

[36] Robert Stroud. Transparency and reflection in distributed systems. In *5th European SIGOPS Workshop, on Models and Paradigms for Distributed Systems Structuring*, Mont Saint-Michel, France, September 1992. ACM SIGOPS, IRISA, INRIA-Rennes.

[37] Robert J. Stroud and Zhixue Wu. Using metaobject protocols to satisfy non-functional requirements. In Chris Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, 1996.

[38] Kazutomo Ushijima, Shigeru Chiba, and Takashi Masuda. Meta-level programming for simplifying library protocols. In *ISOTAS'96 (Submitted to)*, 1996.

[39] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA'88*, volume 23, pages 306–315, September 1988.

[40] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 414–434, October 1992.

[41] Yosuhiko Yokote, Fimio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *ECOOP '89*, 1989.