Composition of Meta-Objects in **Guaraná**

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

Abstract

There are meta-object protocols (MOPs) that do not provide support for meta-object composition. Others require explicit modification of existing meta-level code or provide a limited delegation mechanism in order to support it. There is much room for improvement in this field.

The MOP of **Guaraná** favors the development of meta-objects that can be easily composed. *Composers* are meta-objects that define arbitrary policies of delegation to other meta-objects, separating the *implementation* of meta-level functionality from its *organization*. Composers can also implement meta-level *security policies*, limiting the abilities of its component meta-objects. Composers can be further composed, forming a potentially infinite *reconfigurable* hierarchy.

Our MOP is currently implemented in Java^{TM}. Nevertheless, most design decisions presented in this paper can be transported to other programming languages and MOPs, improving their flexibility, reconfigurability, security and code reuse.

1 Introduction

The development of generic mechanisms for the composition of meta-objects is still in its initial stages. OpenC++ [1] does not provide direct support for composition. MOOSTRAP [6] and MetaXa [2] (formerly known as MetaJava) support sequential composition of *similar* meta-objects. We say that meta-objects are similar if they implement the same interface.

Apertos [11] and CodA [5] admit composition of dissimilar meta-objects, through an extensible set of meta-level aspects. In order to introduce new aspects, pre-existing ones must be modified, and these modifications may clash.

Guaraná [9] allows similar meta-objects to be combined through the use of *composer* meta-objects. Composers can be used to define arbitrary policies for delegating control to other meta-objects, including other composers. We argue that this feature of **Guaraná** is fundamental to the separation of the *structure* of the meta level from the *implementation* of individual management aspects. It represents a step forward towards MOPs that not only allow but encourage and support the implementation of diverse meta-object composition strategies.

In Section 2, we present a description of the reflective architecture of **Guaraná**. In Section 3, we shortly describe its implementation. Finally, in Section 4, we summarize the main points of the paper.

2 The Reflective Architecture of Guaraná

Several run-time MOPs have been designed so that, when a meta-object is requested to *handle* a reified operation (for example, a method invocation), it is *obliged*, by design of the MOP, to return a valid result for the operation. The meta-level computation that yields the result can include or not the delivery of the operation to the base-level object.

This design implies that the only way to combine the behavior of meta-objects is by arranging for one meta-object, say M, to forward operation handling requests to another, say N, delegating to N the responsibility for computing the result of the operation. Only after N returns a result will M be able to observe and/or to modify it.

Given such a protocol, meta-objects are likely to be organized in a chain, so that each meta-object delegates operation handling requests to its successor. The last element of the chain is either the base-level object [2] or a special meta-object that delivers operations to it [6]. We argue that this design presents some serious drawbacks:

- it is intrusive upon the meta-object implementation, in the sense that a meta-object must *explicitly* forward operations to its successor;
- it forbids multiple meta-objects from concurrently handling the same operation, because, at a given moment, at most one meta-object can be responsible for producing a result or delivering the operation to the base level;
- it forces meta-objects to receive the results of operations they handled, even if they are not interested in them;
- the order of presentation of results is necessarily the reverse order of the reception of operations, even though different (possibly concurrent) orderings might be more appropriate or efficient, according to the semantics of the application.

2.1 The MOP of Guaraná

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

1. If the target object of the operation is associated with a meta-object, the kernel of **Guaraná** intercepts and reifies the operation and requests the meta-object to handle it; otherwise, no meta-level computation occurs, reducing the overhead for non-reflective objects.
2. A meta-object may produce a result for an 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, the meta-object is not required to reply with a result, nor can it deliver the operation to the base-level object. Instead, it can reply with an operation to be delivered to the base level —usually the operation it was requested to handle— and with an indication of whether it is interested in observing 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 meta-object, depending on its previous reply. If it had requested for permission to modify the result, it may now reply with a different result for the operation.

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

We have been able to define *composers* by separating operation handling from result handling, implemented in two distict methods, namely, handle operation and handle result. A composer is a meta-object that delegates operations and results to multiple meta-objects, then composes their replies in its own replies. For example, a composer can implement the chain of meta-objects presented before, but in a way that one meta-object does not have to keep track of its successor. Another implementation of composer may delegate operations and/or results concurrently to multiple meta-objects, or refrain from delegating an operation to some meta-objects if it is aware they are not interested in that operation.

In **Guaraná**, at any given moment, each object can be associated with at most one meta-object, called its *primary meta-object*. If there is no such association, operations addressed to that object are not intercepted, and we say that the object is not reflective at that moment.

The fact that **Guaraná** associates a single (primary) meta-object with an object keeps the design of the interception mechanism simple. Since the primary meta-object can be a composer, as can any meta-object it delegates to, multiple meta-objects can reflect upon an object. These meta-objects form what we call the *meta-configuration* of that object, a potentially infinite hierarchy of composition that is orthogonal to the well-known infinite tower of meta-objects [4].

2.2 Meta-configuration management

Guaraná presents two additional features that enforce the separation of concerns between the base level and the meta level: (i) the meta configuration of an object is completely hidden from the base level and even from the meta level itself; and (ii) the initial meta-configuration of an object is determined by the meta-configurations of its creator and of its class, a mechanism we call *meta-configuration propagation*.

The first design decision implies that there is no way to find out what is the primary meta-object associated with an object. It is possible, however, to send arbitrary *messages* and *reconfiguration requests* to the components of the meta-configuration of an object, through the kernel of **Guaraná**.

Messages can be used to extend the MOP of **Guaraná**, as they allow meta-objects to exchange information even if they do not hold references to each other. Meta-objects that do not understand a message are supposed to ignore it, and composers are expected to forward messages to their components. The kernel operation that implements this mechanism is called broadcast.

A reconfiguration request carries a pair of meta-objects, suggesting that the first meta-object should be replaced with the second. A special value can be used to refer to the primary meta-object. It is up to the existing meta-configuration to decide whether the request is acceptable or not. However, if the base-level object is not reflective, an InstanceReconfigure message is broadcast to the meta-configurations of its class and of its superclasses. Their components can modify the suggested meta-configuration, for example, forcing it to remain empty.

In most object-oriented programming languages, creating an object consists of two steps: (i) allocating storage for the object, possibly initialized with default values, then (ii) invoking its constructor. We say that these steps are performed by the *creator* of the object.

In **Guaraná**, between these two steps, meta-configuration propagation takes place. The primary meta-object of the creator is provided with a meta-object for the new object. It may return `null`, a different meta-object or even itself, as a meta-object can belong to the meta-configurations of multiple objects. A composer is expected to forward this request to its components and to create a composer that delegates to the meta-objects returned by them.

After meta-configuration propagation, the kernel of **Guaraná** broadcasts a NewObject message to the meta-configuration of the class of the new object, so that its meta-objects can try to reconfigure it. Finally, the object is constructed, but the constructor invocation will be intercepted if the new object has become reflective.

2.3 Support for proxy objects

Guaraná provides a mechanism that allows *proxy* objects to be created from the meta level, *without* invoking their constructors. In addition to the traditional use of a proxy, namely, for representing an object from another address space, a proxy can be used to reincarnate an object from persistent storage, to migrate an object, etc. When a proxy is created, the kernel of **Guaraná** broadcasts to the meta-configuration of its class a NewProxy message, a subclass of NewObject. A proxy will usually be given a meta-configuration that prevents operations from reaching it, but it may be transformed in a real object by its meta-configuration, through constructor invocation or direct initialization.

3 Implementation

We had originally intended to implement **Guaraná** in 100% Pure Java, by writing an extended Java interpreter in Java or by introducing interception mechanisms through a bytecode preprocessor. The first alternative was immediately discarded because it would imply poor performance and difficulties in handling native methods [10].

A bytecode preprocessor implementation was not possible either, due to restrictions imposed by the Java bytecode verifier [3] and the impossibility to rename native methods, needed in order to ensure their interception. We are currently working on a paper describing thoroughly the limitations we have encountered that have prevented us from pursuing further these implementation strategies.

Therefore, we have decided to implement **Guaraná** by modifying the Kaffe OpenVM^{TM}, an open-source Java Virtual Machine. The performance impact due to the introduction of interception capabilities was quite small [7].

The Java Programming Language, however, has not been modified: any Java program, compiled with any Java compiler, will run on our implementation, within the limitations of the Kaffe OpenVM, and it will be possible to use reflective mechanisms in order to adapt and/or extend it.

The MOP of **Guaraná** can also be implemented in other object-oriented programming languages, or even upon existing reflective platforms, as an extension to their built-in MOPs. However, some particular features of **Guaraná** may be difficult to duplicate, if some design decisions of the target language or MOP conflict with those of **Guaraná**.

4 Conclusions

Guaraná provides a powerful and secure mechanism to combine meta-objects into dynamically modifiable, elaborate meta-configurations.

In addition to enforcing a clear separation between the reflective levels of an application, the MOP of **Guaraná** improves reuse of meta-level code by defining a meta-object interface that eases flexible composition. Furthermore, it suggests a separation between meta-objects that implement meta-level behavior from ones that define policies of composition and organization.

Security is another advantage of the MOP of **Guaraná**. The hierarchy of composition can be used to limit the ability of a meta-object to affect a base-level object. For example, a composer may decide not to present an operation to a meta-object, or to ignore results or replacement operations it produces. The composer can withhold a message to a component, reject a meta-object produced by a component at a reconfiguration or propagation request, or provide restrictive operation factories to its components, thus limiting their ability to create operations. Furthermore, since the identity of the primary meta-object of an object is not exposed, the hierarchy cannot be subverted.

MOLDS [8], a library of reusable and combinable meta-level components useful for distributed applications, is currently being developed atop of our Java-based implementation of **Guaraná**. Meta-level functionalities, such as persistence, distribution, replication and atomicity, will be provided transparently, from the point of view of the base-level application programmer.

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 and 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*). Additional support is provided by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*).

Special thanks go to Islene Calciolari Garcia for many useful insights and suggestions.

References

[1] S. Chiba. A metaobject protocol for C++. In *OOPSLA'95*, volume 30, pages 285–299, Oct. 1995.

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

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Java Series. Addison–Wesley, Jan. 1997.

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

[5] J. McAffer. Meta-level programming with CodA. In *ECOOP'95*, pages 190–214, Aug. 1995.

[6] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA'95*, volume 30 of *ACM SIGPLAN Notices*, pages 316–330, Austin, TX, Oct. 1995.

[7] A. Oliva and L. E. Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, Instituto de Computação, Universidade Estadual de Campinas, Sept. 1998.

[8] A. Oliva and L. E. Buzato. An overview of MOLDS: A Meta-Object Library for Distributed Systems. Technical Report IC-98-15, Instituto de Computação, Universidade Estadual de Campinas, Apr. 1998.

[9] A. Oliva, I. C. Garcia, and L. E. Buzato. The reflexive architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, Apr. 1998.

[10] A. Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report SMLI TR-98-64, Sun Microsystems Laboratories, Mar. 1998.

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