Designing a Secure and Reconfigurable Meta-Object Protocol

Alexandre Oliva
Luiz Eduardo Buzato

Relatório Técnico IC–99-08

Fevereiro de 1999

Designing a Secure and Reconfigurable Meta-Object Protocol

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

February 1999

Abstract

Running code downloaded from the network raises several security issues. Unlike the Java™ programming language, most existing reflective architectures have failed to address these issues. There is a clear need for mechanisms to impose some discipline on the interactions between objects and meta-objects, so as to retain or improve the security mechanisms offered by programming languages.

While the ability to dynamically associate objects with meta-objects is essential for developing flexible and adaptable reflective applications, reliability depends on mechanisms to regulate reconfigurations.

In the design of **Guaraná**, a language-independent meta-object protocol, we have attempted to address these issues. This papers describes and justifies some of the decisions we have made in order to allow developers of reflective applications to balance flexibility and security.

1 Introduction

One of the main benefits of computational reflection is to allow a clear and disciplined separation of concerns between base- and meta-level functionality [2]. However, widely used reflective software architectures do not enforce this separation, allowing undisciplined cross-level interaction. This brings several drawbacks to the development of reflective applications.

If it is possible for a base-level object to directly use features of meta-objects, the developers of the base-level object may be tempted to do so, and this will harden the evolution of the application, because meta-objects will not be easily replaceable.

Allowing direct access to meta-objects, from the base level or from the meta level itself, can also make it impossible to use the meta level to implement robust security mechanisms.

We have considered these issues during the design of the Meta-Object Protocol (MOP) of **Guaraná** [4, 6]. The present paper relates each design decision with the reasoning and the requirements associated with them, such as security, reliability, adaptability and ease of maintenance.

Even though **Guaraná** is currently implemented in Java™, the design of its MOP, and, particularly, the arguments presented in this paper, are not limited to this programming language. The main contribution of this paper is to provide guidelines for the design of other MOPs, so that they can benefit from the security issues learnt from our experiences during the design and implementation of **Guaraná**.

In Section 2, we shortly describe a simplified meta-object protocol and show that it satisfies the requirements presented. Section 3 introduces reconfiguration mechanisms, designed in a way that does not violate the presented security concerns. Section 4 concludes the paper.

2 A Secure Meta-Object Protocol

In this section, we assume that each object may be associated with at most one meta-object. We say the object is reflective if it is associated with a meta-object. It does not matter, at this point, how the association is specified. It may be determined at compile time, with special declarations, as in OpenC++ [1], or at run-time, as in MetaXa [3].

Interactions from the base level to the meta level can be *implicit* or *explicit*. Interception of operations is an example of implicit interaction, since the base level may be unaware of its occurrence. Explicit interactions may be carried out by direct invocations of methods of meta-object.

2.1 Explicit interactions

There are times in which interactions with meta-objects are desirable, for example, when a meta-object of one object must provide some information to the meta-object of another object, or obtain some information from it. Even in this case, direct access should be avoided, because it strengthens the coupling between the levels, reducing the adaptability of the application.

One possibility is to offer a mechanism that allows one meta-object to find out which is the meta-object of another object, and then interact with it through this indirect reference, thus eliminating explicit references.

We have decided to discard this solution for two reasons. First, it would be hard to prevent base-level objects from using this mechanism, and this use would break the inter-level separation. Even if only meta-objects could have access to this mechanism (say, a `protected` method in a base class for all meta-objects), a base-level object might be able to create a meta-object under its control to gain access to the mechanism.

Second, exposing references to meta-objects, even if indirect, would make it possible to present an arbitrary operation for the meta-object to handle, and the meta-object might be unable to tell whether the operation was intercepted from the base level or faked.

Moreover, even if references to meta-objects could only be obtained indirectly, explicit dependencies on their types could still be introduced (say, by a type cast followed by direct invocations). This defeats the whole purpose of the indirection, which is to ease adaptation, i.e., substitution of a meta-object for another of a different type. Therefore, the mechanism for exchanging information with meta-objects should be based upon weak typing and dynamic dispatching, even if the language that the MOP extends is strongly typed.

We propose a mechanism that is based on *message* objects: given a base-level object and a message object, the mechanism will present the message to the object's meta-object, if the object is reflective, or discard it, otherwise. Meta-objects should be able to tell, based on type and content of a message, whether it is prepared to handle it. If it is, it can extract information from it, or respond to it, by invoking its methods.

This mechanism does not allow the sender of the message to gain any information about the meta-object, or even about its existence, unless the meta-object is willing to disclose it. A meta-object has absolute control over message handling, so its behavior may range from completely ignoring messages that do not satisfy some acceptance criteria to providing references to itself, in order to ease further communication with authorized partners.

We should note that, although this messaging mechanism is intended to be used for communication between meta-objects, base-level objects may be able to use it, and thus to break inter-level separation. Restricting access to this mechanism only to meta-objects would not help, unless creation of meta-objects could also be restricted. In any case, the main argument for preventing such interactions, which is that of reducing coupling between levels, is sustained.

2.2 Implicit interactions

Whenever an operation is requested to a reflective object, an implicit interaction with its meta-object takes place. An interception mechanism reifies the operation and invokes a method that the target object's meta-object must implement to handle intercepted operations.

This method is supplied with the reified operation, i.e., an object that allows the meta-object to find out which base-level object the operation refers to (because a meta-object might be associated with several objects), what kind of operation it is (method invocation, or attribute read or write), which method or attribute it operates upon, the argument list of a method invocation, or the value to be stored in an attribute.

Several reflective architectures assume that, when a meta-object is requested to handle an operation, it becomes responsible for providing a result for it, by producing the result itself or by requesting the base-level object to perform the intercepted operation.

Unfortunately, allowing a meta-object to deliver an operation to an object introduces two potential security holes. Whatever mechanism a meta-object would use to deliver the operation to its target object might be used by other objects to access an object, possibly bypassing interception. However, even if access to this mechanism was restricted so that only the meta-object associated with an object could deliver operations to it, the second hole remains: if an intruder manages to obtain a reference to a meta-object, it may pretend to be the interception mechanism and ask the meta-object to handle arbitrary operations, and the meta-object may end up delivering them to the base-level object.

In order to avoid these security holes, we propose that the *only* entity able to deliver operations to objects must be the interception mechanism, and the only way a meta-object can request it to do so is by arranging that the operation handling method *returns* a delivery request. Hence, a meta-object cannot be led into delivering an operation, since it is impossible to bypass the interception mechanism.

Thus, when a meta-object is requested to handle an operation, it may return a result for the presented operation or a request for an operation to be delivered. In the latter case, it may indicate that it wishes to view or modify the result of the operation, after it is performed. If it does not indicate it, the result can be returned directly to the requester of the operation, without reification.

Otherwise, the interception mechanism will invoke a method that the meta-object must implement to handle results. This method may return a different result for the operation.

It is desirable for meta-objects to be able to modify operations they are requested to handle, as well as to create stand-alone operations and submit them for execution by the objects they control. Furthermore, allowing a meta-object to judiciously break encapsulation of a base-level object may ease the implementation of some typical meta-level features such as persistence and remote invocation, but this ability should be restricted to the meta-object associated with the object.

Unlike the case of operation delivery, this mechanism cannot be based upon returning from operation handling requests, because it should be possible to create operations autonomously. A publicly accessible method that would allow for creation of operations after verifying that its caller is the meta-object of the target object would probably work. However, it would limit the use of other meta-level objects as part of the implementation of meta-objects, because these other meta-level objects would be unable to create operations.

The mechanism we have designed is based on operation factories, meta-level objects that function as keys that allow their possessors to open the implementation of an object. For each reflective object an operation factory is created for its meta-object. The operation factory will only create operations targeted at that object. It is up to the meta-object to keep the operation factory secret. One interesting feature of this mechanism is that meta-objects can disclose operations factories to other meta-objects they collaborate with.

Operation factories provide methods to create *stand-alone* and *replacement* operations. A stand-alone operation can be submitted for execution by the invocation of a method it offers, but it must undergo interception, because only the interception mechanism can deliver an operation to its target object.

A replacement operation, on the other hand, can be returned by the operation handling method to the interception mechanism, that will deliver it instead of the operation it replaces. For this reason, the operation factory methods used to create replacement operations must ensure that a replacement operation is compatible, in terms of result type, with the replaced operation, and it carries a reference to the replaced operation, to prove that it has not been created to replace another operation.

The existence of a mechanism to create replacement operations obviates the need for mechanisms to modify existing operation objects, so we have simplified the interface of operation objects by making them immutable.

3 Reconfiguration

In the previous section, we assumed objects were permanently associated with meta-objects. In this section, we are going to present a way that allows this association to be modified at run-time, while preserving security properties.

The basic principle we have adopted is that the meta-object associated with an object has total control over the object, so it can only be replaced by another meta-object if it agrees to leave.

We suggest a mechanism that allows a meta-object to be proposed as a replacement to the meta-object of an object. This mechanism will implicitly ask the existing meta-object whether it accepts to be replaced with the proposed one, and will abide by its decision. The identity of the existing meta-object is protected, and it is impossible to inconditionally replace it.

There are times in which a meta-object might wish to accept the suggested reconfiguration only in part. For example, being aware of its own properties, and being able to find out the properties of the suggested meta-object, it may name an existing meta-object, or create a new one, that somehow combines some selected properties of them. Therefore, the reconfiguration mechanism should allow a meta-object to reply more than just "yes" or "no": it may return (a reference to) the meta-object that will replace it.

Sometimes, the requester of the reconfiguration may wish to provide some hint for an existing meta-object about what it expects to take place during the reconfiguration. For example, it may indicate that it wishes meta-object N to become the meta-object of object O only if meta-object M is the current meta-object of O. Since the requester cannot find out which is the current meta-object of O, this hint must be part of the reconfiguration request, and the current meta-object can base its decision on it.

The semantics of not providing a hint meta-object M (for example, making it `null`) is a way to request the current meta-object, whatever it is, to be replaced with N. In any case, the current meta-object can overrule this request.

However, if object O is not reflective, there is no "current meta-object" to protect it from arbitrary reconfigurations. It would be nice to be able to specify global or per-class reconfiguration policies, to handle this case without having to make every single object reflective.

The mechanism we propose allows for such per-class policies. Whenever a meta-level reconfiguration request is issued on a non-reflective object, a *message* is created and presented to the meta-object associated with (the object that represents) the class of the object, if there is such a meta-object. This message, whose type is "instance reconfiguration", contains a reference to the object being reconfigured and a modifiable reference to the proposed meta-object, so that the class meta-object can change it. The message is also presented to meta-objects of base classes, and the reference that remains at the end of this process determines the meta-object to be installed as the object's meta-object. If the reference is nullified, the object remains non-reflective.

It is worth noting that, even if an object is reflective, its meta-object may hide and pretend it is not, by creating itself an "instance reconfiguration" message and using the messaging mechanism to send the message to the meta-objects of the classes of the object.

An important issue to be considered, when reconfiguration is possible, is how to ensure that meta-objects that have given up control over an object do not get privileged access to it any more. For example, the operation factory held by such a meta-object must be invalidated when it is replaced.

Note that a meta-object might create a stand-alone operation while its operation factory is still valid, and request it to be performed, i.e., submit it for interception, only after a reconfiguration. The operation may still be considered valid and delivered to the object, but it is up to the new meta-object to decide it.

Furthermore, if a meta-object is replaced while it handles an operation, its reply should be ignored and the operation should be handled by the new meta-object. So, even if a meta-object creates an operation, submits it for interception and holds its processing until after a reconfiguration, the operation will not be delivered at its choice: the new meta-object will be able handle the operation and approve it or reject it. In any case, the replaced meta-object should have an opportunity to reset any internal state, with a result that indicates the cancellation of the operation it had handled.

However, if an operation has already been delivered to the base-level object when a reconfiguration takes place, there is no way to cancel it without possibly creating inconsistencies in the base level, so the result should be handled by the same meta-object that handled the operation.

There is a possibility that, upon a reconfiguration request, a meta-object issues a new reconfiguration request. In this case, if the nested reconfiguration is successful, the pending reconfiguration should be ignored, otherwise a meta-object would be able to inconditionally re-take control on the object, by returning itself from the pending reconfiguration request. Concurrent reconfigurations, in multi-threaded systems, had better be serialized.

3.1 Object creation

An object created dynamically could be created non-reflective. However, we do not think this is a good policy because, in some languages, it would be impossible to intercept operations performed during object construction. Moreover, it would probably lead to inter-level interactions, since the obvious place in which to specify a meta-object for the new object would be just after creating it.

Leaving an object non-reflective after its creation may also have security consequences, because a base-level object might use the knowledge that just-created objects are non-reflective in order to configure them with arbitrary meta-objects, unless prevented by class meta-objects.

However, leaving it up only to class meta-objects to decide when to disallow certain reconfigurations may be insufficient, or just too cumbersome. We advocate that the context in which an object is created, i.e., the meta-object of the creator, should be requested to provide a meta-object for the new object, so as to *propagate* reflective and security properties.

There are three reasons for the meta-object to return the meta-object it proposes for the new object, instead of requesting a reconfiguration: (i) returning is simpler; (ii) a reconfiguration request would allow the meta-object of the class of the object to overrule the proposal of the execution context; and (iii) allowing reconfigurations, or any other operations, before the new object is configured, could allow other meta-objects to take over the new object.

In order to prevent access to a new object before it is configured, a meta-object that rejects all requests controls the new object until the meta-object of the creator returns another meta-object.

At last, the messaging mechanism is implicitly used to notify the meta-object of a class about the creation of a new instance. This meta-object can keep track of the instances of the class, as well as affect their meta-configurations, by issuing reconfiguration requests.

4 Conclusion

Security has become a major concern as the execution of code downloaded from the Internet gained wide acceptance. Designers of programming languages such as Java$^{\mathrm{TM}}$ have addressed this issue by introducing built-in security mechanisms. Most reflective software architectures, however, have failed to do so.

We have argued that the lack of security mechanisms in reflective architectures, i.e., the lack of discipline in inter-level and inter-meta-object communication, has negative implications for the reliability and adaptability of meta-level components.

We advocate that MOPs should be designed so as to avoid direct explicit interactions between levels and to favor implicit interactions. Meta-objects should be hidden from objects and other meta-objects, so that they can retain full control over objects under their responsibility and prevent unauthorized access to them.

The MOP of **Guaraná** is a simple extension of the secure MOP and the reconfiguration mechanisms described in this paper, in order to support composition of meta-objects [5]. This extension does not violate the discussed security properties; on the contrary: it enriches the MOP by allowing code reuse and hierarchical control of meta-objects.

Although our current implementation of **Guaraná** is an extension of the Java platform, the security mechanisms we have proposed are not limited to the Java programming language, and can be adopted by MOPs based upon other programming languages. Even for languages not designed with security in mind, a secure MOP can help developing reliable, adaptable and secure reflective applications.

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*), and CNPQ (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), for the PRONEX programme and for a PhD scholarship for Alexandre Oliva.

Islene Calciolari Garcia has helped us very much in reviewing and improving this paper. She also made important suggestions to the architecture of **Guaraná**.

References

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

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

[3] 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.

[4] Alexandre Oliva. Guaraná: Uma arquitetura de *software* para reflexão computacional implementada em Java. Master's

thesis, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brazil, September 1998. Mostly in English.

[5] Alexandre Oliva and Luiz Eduardo Buzato. Composition of meta-objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 86–90, Vancouver, BC, Canada, October 1998.

[6] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of Guaraná. In *5th Conference on Object-Oriented Technologies and Systems*, San Diego, CA, USA, May 1999. USENIX. to appear.