An Overview of **MOLDS**: A **M**eta-**O**bject **L**ibrary for **D**istributed **S**ystems

Alexandre Oliva
oliva@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 paper presents a library of meta-objects suitable for developing distributed systems. The reflective architecture of **Guaraná** makes it possible for these meta-objects to be easily combined in order to form complex, dynamically reconfigurable meta-level behavior. We briefly describe the implementation of **Guaraná** on Java[1]. Then, we explain how several meta-level services, such as persistence, distribution, replication and atomicity, can be implemented in a transparent, reconfigurable and flexible way.

1 Introduction

Computational reflection [29, 34] (henceforth just reflection) has proven to be a useful support mechanism for building distributed systems in a transparent way [1, 2, 7, 12, 14, 19, 18, 25, 28, 30, 32, 36, 37, 35, 39, 40]. **Guaraná** [31] is a reflective software architecture that aims at encouraging the reuse of reflective solutions. It provides simple mechanisms for combining multiple meta-objects into the meta-level configuration of a single object. These meta-objects may implement mechanisms such as those required to provide distribution, persistence, replication, atomicity, etc.

This paper is organized as follows. In Section 2, we briefly describe our implementation of the reflective architecture of **Guaraná** atop of the *Kaffe OpenVM*™, freely-available Java Virtual Machine. Then, we introduce **MOLDS**, a library of meta-objects that provide essential features for developing reliable distributed systems. Finally, in Section 4, we summarize the benefits of designing and implementing a library such as **MOLDS** atop of **Guaraná**, and describe the current state of development of both **Guaraná** and **MOLDS**.

2 The Java-based implementation of Guaraná

Java [3, 23] is a simple yet powerful object-oriented language. Java classes are compiled into high-level object-oriented cross-platform byte-codes, that can be executed on Java Virtual Machines (JVM). Since the specification of the JVM [27] is open, anyone can implement it, so Java has become available on several different hardware platforms; freely-modifiable and redistributable source code for some of these implementations is available at no cost. The reflective architecture of **Guaraná** was implemented on

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

top of one of these platforms, namely the *Kaffe OpenVM*.

In **Guaraná**, every Java object may be directly associated with zero or one meta-object, called the object's primary meta-object. An object that is associated with a meta-object will be called a reflective object. Every operation addressed to a reflective object is intercepted, reified (represented) as a meta-level object, and presented to the object's primary meta-object.

By operation, we mean methods and constructors invocations, monitor (`synchronized`) enter and exit operations, field reads and writes. Since Java arrays are objects too, array elements reads and writes, as well as array length reads, are considered operations too.

When a meta-object is presented an operation, it may reply with (i) a result for the operation, (ii) an alternate operation, to be performed instead of the one requested by the base-level, or (iii) a request for the original operation to be performed. Unless it provides a result itself, it may request to be presented the result of the operation after it is performed, or even to be able to modify this result.

Meta-objects, instances of subclasses of the class MetaObject, can be made reflective too, by associating them with other meta-objects, what leads to the so-called potentially infinite tower of meta-objects [29].

Class Composer, one of the standard specializations of MetaObject, is the key concept introduced by the meta-object protocol of **Guaraná**. A composer is a meta-object that delegates operations and results to other meta-objects. Composers may delegate to meta-objects sequentially, or concurrently, or following whatever policy fits the needs of a developer. A sample composer is provided that delegates operations to the elements of an array of meta-objects, and delegates results to the same meta-objects in the reverse order.

Some of the component meta-objects of a composer can be composers themselves, what leads to a hierarchical organization of the meta-objects directly or indirectly associated with a base-level object. This organization, called the object's meta-configuration, is orthogonal to the tower of meta-objects; each meta-object may have its own independent meta-configuration. Furthermore, a meta-object may belong to the meta-configuration of more than one object.

Associating an object with its primary meta-object is an operation provided by the kernel of **Guaraná**. In fact, this operation is so general that it allows any meta-object in a meta-configuration to be replaced with another. Any reconfiguration must be approved by the previous meta-configuration; if the base-level object was not reflective, its class is informed about the reconfiguration request, and may prevent it.

Objects created by reflective objects have their meta-configurations determined by their creator's meta-configuration. Furthermore, the meta-configuration of the class of the new object is notified before the object's constructor is invoked, so that it may try to modify the meta-configuration of its new instance. The kernel of **Guaraná** makes it possible to create pseudo-objects, that are uninitialized instances of a given class. These objects may be turned into real objects by invoking a constructor or initializing its fields, but it may remain a pseudo-object and be used, for example, as a proxy of an object in a separate address spaces. The meta-configuration of the class the pseudo-object belongs to is also notified, so it may modify the pseudo-object's meta-configuration, or even prevent the creation of the pseudo-object.

This notification is done by using another operation provided by the kernel of **Guaraná**: any instance of a class that implements the interface Message can be broadcast to (possibly) all component meta-objects of a meta-configuration of an object. Such an operation is necessary because, for security reasons, we made it impossible to obtain a reference to the primary meta-object of an object. Furthermore, we believe this helps maintaining a clear separation of concerns between the base and the meta level, just like encapsulation encourages good object-oriented design.

Guaraná provides an interface that allows arbitrary Operation objects to be created in the meta-level; even operations that would violate encapsulation can be created and performed by using this interface. However, for the sake of security, such operations must be created using OperationFactory objects, that are given to meta-objects whenever they are associated with an object. This ensures that only component meta-objects of an object's meta-configuration, and meta-level objects trusted by them, can obtain privileged access to this object. Composers may distribute restricted operation factories to meta-objects they delegate to.

3 Reusable Meta-Objects for Distributed Systems

The meta-level protocol of **Guaraná** was designed in a way that makes it possible to create meta-objects that implement specific meta-level behaviors, and to easily compose them into complex meta-configurations. In this section, we delineate how some meta-level services for distributed computing can be implemented in **Guaraná**.

3.1 Persistence

A persistent object [4, 11] is one whose lifetime spans the application that created it. The state of persistent objects can be stored in files, databases or long-running processes. An object can be made persistent by simply adding a persistence meta-object to its meta-configuration.

A persistence meta-object may be implemented using two different approaches: i) it may intercept all field update operations, and update the persistent storage accordingly, possibly in background; or ii) it may update the persistent storage only when the persistent object is no longer used by the running application.

Whatever the choice, every object must be given a unique identifier, that can be used for maintaining references from one persistent object to another, as well as for reincarnating an object from persistent storage into a running application. This unique identifier might be maintained by the persistence meta-object itself, however, a unique identifier may be useful for other purposes, so we recommend the creation of a separate identification meta-object.

Whenever an object-type field of a persistent object is assigned to, the referred-to object must also be made persistent, otherwise it will not be possible to recreate the complete state of the referring object afterwards. This can be accomplished by probing the meta-configuration of this object with a broadcast message. If no persistence nor identification meta-object exists in the object's meta-configuration, the object must be reconfigured so as to become persistent, or the field assignment must be denied by throwing an exception.

In order to reincarnate a persistent object, there are two possible approaches: (i) the persistence meta-library may provide a method, that can be called from the base level, that reincarnates an object, given its unique identification; or (ii) a base-level reflective container, that represents the persistent storage, may be used to reincarnate persistent objects transparently.

An object is reincarnated by creating a pseudo-object, whose fields are filled in from the persistent storage. Reincarnation of referred objects can be done on demand, as they are accessed from the base level. Even fields might be reincarnated individually. The implementation of such meta-object would be much more complicated, but it may pay off if the object's state is large enough.

3.2 Replication

Object replication [33] may be used in order to increase availability and fault-tolerance of an object. If one replica fails, others may keep the object running.

There is a very simple way of implementing replication with **Guaraná**. Every replica executes methods and reads fields without exchanging information with other replicas. Field modifications, however, are broadcast to all replicas in a totally-ordered [8] way, so that all replicas perform field writes in the same sequence. Synchronization operations must be subject to the same total order.

Other replication mechanisms may broadcast method invocations and even field read operations to multiple replicas, then run an election algorithm to select a result. However, this introduces some problems that are hard to solve. For example, when one replicated object interacts with another, the interaction must occur as if the objects were not replicated at all. So, when one replicated object invokes another, all the individual invocations must be identified as replicas of a single invocation, and the operation must be performed only once on each replica of the invoked object.

3.3 Distribution

Implementing transparent interaction between objects located in separate virtual machines was made easy by the introduction of pseudo-objects. An approach similar to that taken for persistence may be used to locate remote objects. There are differences: (i) instead of locating objects in a database, they will be searched for in a distributed name server (what might be viewed as a database, after all); and (ii) instead of reincarnating the object, a proxy of the remote object is created, as a pseudo-object.

Whenever an operation is requested to the proxy, its distribution meta-object marshals the operation and sends a message through a network channel to a meta-object located in the actual target object's address space. This meta-object just creates an operation equivalent to the requested one and delivers it for meta-level interception. As soon as a result for the operation is available, it is marshaled and returned to the proxy's meta-object, that unmarshals the result and returns it as the result of the operation.

This facility may be used as a basis for having distributed replicas. Instead of implementing inter-meta-object communication in the replication meta-objects themselves, now we just have to keep proxies to remote replica's meta-objects in every address space so that they can communicate. This is not an overkill, since group communication protocols usually require every member of the group to know every other member.

3.4 Caching

Having to send every single operation addressed to a remote object through the network may cause serious negative impact on the performance of an application. On the other hand, replicating an object may introduce too much overhead for an object that is frequently updated.

An intermediate solution may be achieved by caching the contents of fields of an object in proxy objects. These fields could be updated periodically, or when synchronization operations take place. Update operations in the proxy object might not need to be immediately forwarded to the actual object (or replicas [26]). This is somewhat dependent on the requirements of the application, but it may prove to be very useful in certain situations.

A caching meta-object can be easily implemented as a composer that selectively delegates operations to a distribution meta-object.

3.5 Migration

Objects such as mobile agents [5] may have to move from one address space to another. This may be achieved by creating a replica of the moving object in the target address space, then removing the replica from the source address space.

However, this may be too costly a way to migrate an object. Another, potentially faster, approach is to have a meta-object that stops delivering operations to the object as soon as it decides the object must migrate. Then it marshals the complete internal state of the object and sends it to a remote meta-object that is going to become a member of the migrated object's meta-configuration. It creates a pseudo-object and fills in its fields with the marshaled image of the object. At this point, the original object will have become a proxy object, that simply forwards operations to the migrated object, until the proxy is garbage collected.

If an object migrates many times, an operation may have to flow through several proxies before it reaches the actual object. In this case, it may be useful to have an algorithm that notices whether an object migrated any further, and sets up a short-cut to the most recently known location of the object from then on [17].

We should note that there is some overlap of the migration and the persistence functionalities. After all, a persistent object may be implemented by migrating it to and from a long-running server process. On the other hand, migration could be easily implemented by storing the mobile object in persistent storage, then reincarnating it in another address space.

Instead of implementing one mechanism on top of another, we believe the correct approach is to factor out the common functionality required by both mechanisms, and implement the differences as specializations. Meta-objects that implement caching may also share functionality with these two mechanisms.

3.6 Accounting

Meta-objects for accounting can be easily associated with arbitrary objects. One may count the invocations of a particular method, or updates of a field, or even complex multi-object patterns of interaction. Classes can be configured so that all instances are given appropriate accounting meta-objects.

3.7 Monitoring

In addition to the ability of maintaining information about base-level objects, it may be useful for meta-objects to interact with base-level objects from the meta level.

It is possible to interconnect otherwise independent objects through meta-objects. This can be used to implement the Model-View-Controller (MVC) [22] pattern, connecting a model object with its views transparently: the control can be totally implemented in the meta-level [15].

We might also use meta-objects that implement Statecharts [24] to model and control the behavior of base-level objects. Transitions in the Statechart could be triggered by the interception of operations or results; there may be additional conditions for the transition to take place, involving the state of the base-level object as well as internal meta-object state [10].

Monitoring multiple distributed objects may require the construction of consistent global snapshots [13, 16, 20, 21]. Algorithms for obtaining consistent global snapshots can be implemented completely in the meta-level.

3.8 Atomicity

Atomic actions [6] involve three properties: (i) serializability, that ensures that the execution of concurrent atomic actions is equivalent to at least one serial execution; (ii) atomicity, that is, either all its effects become visible, or none do; and (iii) permanence of effect.

The last property requires objects involved in an atomic action to be kept in stable storage, so that, even if one of the hosts running a distributed atomic action fails, its effects are permanent.

The atomicity property requires a global coordination of all objects involved in an atomic action. If the atomic action is committed, all objects involved must have its states made persistent; if it aborts, all objects must be reverted to the states previous to the beginning of the atomic action.

The serializability property requires some kind of concurrency control on operations. There are optimistic and pessimistic policies. Pessimistic algorithms rely on locking for ensuring serializable executions; optimistic ones let separate atomic actions operate on separate copies of objects, and check for serializability at commit time.

Atomic actions may be totally controlled at the base level, for example, by providing a class AtomicAction that takes an instance of the Java standard class Runnable as its constructor argument. The method run of this argument is then executed inside the atomic action. If it terminates successfully, the transaction is committed; if it throws an exception, the atomic action may abort.

Concurrency control may take place transparently, at the meta level, using whatever selected policy. However, if it is a pessimistic one, it should be possible to pre-declare locks, for example, from both the base and the meta level.

Instead of explicitly creating and managing atomic actions in the base level, certain objects may be configured as atomic ones [38], so that every operation on that object is performed inside an independent atomic action. It may be useful for meta-level control of atomic actions to be able to specify that a particular operation should be performed inside a given atomic action, as a nested atomic action or sharing data with other threads running the same transaction.

4 Conclusion

The design of **Guaraná** was largely influenced by detected needs of a library like **MOLDS**. In fact, we have only started **Guaraná** because no other reflective platform we knew could provide the modularization, composition, reconfiguration and security features demanded by such a library. The choice of Java as the programming language has just made things easier, because of the existing basic reflection capabilities and of the libraries for developing networked applications.

We believe **MOLDS** will become a very powerful and sound framework for developing distributed applications, but its components still have to be detailed further and implemented.

This library is a basic part of a larger project [9]. The only similar project we have known to date is Apertos [39, 40], a reflective operating system. We should note, however, that it is based on a slightly more limited reflective model, specifically targeted at operating system development.

A Obtaining Guaraná and MOLDS

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. **MOLDS** is currently in early design stage, but, when you read this paper, there may be updated information in the home page of **Guaraná**. Both **Guaraná** and **MOLDS** are *Free Software*, released under the GNU General Public License, but their 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*).

References

[1] G. Agha, S. Frølund, R. Panwar, and D. 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, Sept. 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*, Mar. 1996.

[3] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Weslley, 1996.

[4] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, Dec. 1983.

[5] Y. Berbers, B. De Decker, and W. Joosen. Infrastructure for mobile agents. In *Seventh ACM SIGOPS European Workshop: System Support for Worldwide Applications*, pages 173–180, 1996.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[7] S. Bijnens, W. Joosen, and P. Verbaeten. A reflective invocation scheme to realise advanced object management. In *Object-Based Distributed Programming ECOOP '93 Workshop*, July 1993.

[8] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[9] L. Buzato, H. Liesenberg, C. R. R. Anido, and M. 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.

[10] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, Dec. 1994.

[11] L. E. Buzato and A. Calsavara. Stabilis: A Case study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects. In A. Albano and R. Morrison, editors, *Proceedings of the Fifth International Workshop on Persistent Object Systems*, Workshops in Computing, pages 354–375, San Miniato, Italy, Sept. 1992. Springer-Verlag.

[12] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented system in C++. *Commun. ACM*, 36(9):117–126, Sept. 1993.

[13] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, Feb. 1985.

[14] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In N. Nierstrasz, editor, *ECOOP'93*, pages 482–501, 1993.

[15] M. G. Coelho, C. M. F. Rubira, and L. E. Buzato. Uma abordagem reflexiva para a construção de frameworks para interfaces homem-computador. In *XI Simpósio Brasileiro de Engenharia de Software (SBES'97)*, pages 115–130, Fortaleza, CE, Oct. 1997.

[16] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. *SIGPLAN Notices*, 26(12):167–174, Dec. 1991.

[17] M. S. et al. SSP chains. In *Symposium on Principles of Distributed Computing.* acm, 1992.

[18] J.-C. Fabre, V. Nicomette, T. Pérennou, and Z. 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.

[19] 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, Feb. 1995.

[20] I. C. Garcia and L. E. Buzato. Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, EUA, May 1998. IEEE. Available as Technical Report IC–98–16.

[21] I. C. Garcia and L. E. Buzato. Cortes consistentes em aplicações distribuídas. Technical Report IC–98–17, Instituto de Computação, Universidade Estadual de Campinas, Apr. 1998.

[22] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, first edition, 1983.

[23] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Java Series. Addison–Wesley, Sept. 1996. Version 1.0.

[24] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, Aug. 1987.

[25] J. Kleinöder and M. Golm. Transparent and adaptable object replication using a reflective Java. Technical Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996.

[26] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 43–57, Quebec City, Quebec, Canada, Aug 1990.

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

[28] P. W. Madany, N. Islam, P. Kougiouris, and R. H. Campbell. Reification and reflection in C++: An operating system perspective. Technical report, University of Illinois at Urbana-Champaign, Mar. 1992.

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

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

[31] 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.

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

[33] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

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

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

[36] 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.

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

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

[39] 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.

[40] Y. Yokote, F. Teraoka, and M. Tokoro. A reflective architecture for an object-oriented distributed operating system. In *ECOOP '89*, 1989.