Guaraná: Uma Arquitetura de *Software* para Reflexão Computacional Implementada em Java$^{\mathrm{TM}}$

Alexandre Oliva

Dissertação de Mestrado

Guaraná: Uma Arquitetura de *Software* para Reflexão Computacional Implementada em Java^{TM}

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alexandre Oliva e aprovada pela Banca Examinadora.

Campinas, 17 de agosto de 1998.

Prof. Dr. Luiz Eduardo Buzato
Universidade Estadual de Campinas
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Guaraná: Uma Arquitetura de *Software* para Reflexão Computacional Implementada em Java^{TM}

Alexandre Oliva[1]

30 de julho de 1998

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato
  Universidade Estadual de Campinas (Orientador)
- Prof. Dr. Markus Endler
  Universidade de São Paulo
- Prof. Dr. Jorge Stolfi
  Universidade Estadual de Campinas
- Profª Drª Cecília Mary Fischer Rubira (Suplente)
  Universidade Estadual de Campinas

[1]**Auxílios concedidos pela FAPESP, processo 95/2091-3 para Alexandre Oliva e 96/1532-9 para o Laboratório de Sistemas Distribuídos do IC-UNICAMP.**

Agradecimentos

À FAPESP e ao assessor anônimo, pela confiança e pelo apoio financeiro, mesmo acreditando que a mistura de café com guaraná é esquisita.

Aos funcionários e colegas do Instituto de Computação, e do antigo Departamento de Ciência da Computação, pelo ambiente de trabalho agradável.

Aos professores Paulo Lício de Geus e Rogério Drummond, meus orientadores durante a graduação, por me ensinarem a fazer pesquisa científica e por me incentivarem a permanecer no meio acadêmico, e a todos os demais professores que contribuíram para minha formação.

To Richard Stallman, for creating the GNU project and the Free Software Foundation, and for teaching me the meaning of Free Software.

To Tim Wilkinson, for releasing Kaffe as Free Software and making the development of ***Guaraná*** *possible.*

Aos companheiros Nádia, Delano, Luciane e Alexandre, pelo auxílio na caça aos *bugs* do **Guaraná**.

A meus pais, e a toda a minha família, por terem sempre me incentivado a querer saber mais.

Ao meu orientador e amigo, Luiz Eduardo Buzato, por me dar liberdade para criar, ainda que à custa de *pequenos* atrasos no cronograma, e pela excelente orientação, apesar de toda a sobrecarga.

À minha noiva, Islene, por todo o amor, carinho, amizade, incentivo e apoio, e por não me deixar satisfazer com pouco.

A Deus e ao meu anjo da guarda, pela luz e pela proteção. Ou, como sempre diz meu pai, por eu ter "mais sorte que juízo".

Abstract

This dissertation is a collection of papers written in English, with an introduction and a conclusion in Portuguese.

The first paper describes **Guaraná**, a language-independent reflexive architecture, whose run-time meta-level protocol permits a high degree of reuse of meta-level code. The protocol was designed so as to provide, in a secure manner, flexibility and reconfigurability of meta-level behavior of objects.

The second paper describes our implementation of this architecture through the modification of a free implementation of the Java^{TM} Virtual Machine (JVM) Specification, but keeping the Java Programming Language intact. With our approach, existing Java applications can be made reflexive, even if their source code is not available. We describe the modifications we have introduced in the JVM, as well as the Java classes that complete the implementation, and measure the impact of the modifications on the performance of applications and the JVM.

The third paper is a tutorial directed to Java^{TM} programmers who are willing to know and use the features of **Guaraná**. It covers the workings of **Guaraná**, from basic interception mechanisms to advanced topics, exposing some of the internal details of the implementation of **Guaraná**.

The fourth and last paper introduces **MOLDS**, a library of meta-level components suitable for building distributed applications, that we intend to implement on top of **Guaraná**. This library will explore **Guaraná**'s features to combine independent meta-objects that implement mechanisms such as replication, persistence, etc, in order to form complex meta-level behavior, in a transparent way, from the point of view of the application programmer.

Resumo

Esta dissertação é uma coleção de artigos escritos em inglês, com uma introdução e uma conclusão em português.

O primeiro artigo descreve **Guaraná**, uma arquitetura reflexiva independente de linguagem, cujo protocolo de meta-nível, em tempo de execução, permite um alto grau de reutilização de código de meta-nível. O protocolo foi projetado de forma a prover, de forma segura, flexibilidade e reconfigurabilidade do comportamento de meta-nível de objetos.

O segundo artigo descreve nossa implementação dessa arquitetura, através da modificação de uma implementação aberta da Máquina Virtual de Java$^{\text{TM}}$, que mantém a linguagem de programação Java$^{\text{TM}}$ inalterada. Com nossa abordagem, aplicações Java$^{\text{TM}}$ pré-existentes podem ser tornadas reflexivas, mesmo quando seu código fonte não está disponível. O artigo descreve as alterações que fizemos à máquina virtual, bem como as classes que completam a implementação. Além disso, ele apresenta medidas de degradação de desempenho causadas por nossas alterações.

O terceiro artigo é um tutorial dirigido a programadores Java$^{\text{TM}}$ que pretendam conhecer e utilizar os recursos do **Guaraná**. Ele cobre desde mecanismos básicos de interceptação até tópicos avançados, expondo alguns detalhes internos da implementação do **Guaraná**.

O quarto e último artigo apresenta **MOLDS**, uma biblioteca de componentes de meta-nível adequados para a construção de aplicações distribuídas, que pretendemos implementar sobre o **Guaraná**. Esta biblioteca explorará a capacidade do **Guaraná** de combinar meta-objetos independentes de modo a definir comportamentos de meta-nível complexos, de maneira transparente, do ponto de vista da aplicação do nível base.

Sumário

Agradecimentos **v**

Abstract **vi**

Resumo **vii**

1 Introdução **1**

2 Arquitetura de *Software* **3**
2.1 Introduction 4
2.2 Computational Reflection 6
2.2.1 Reification 8
2.2.2 Reflective Architectures 8
2.2.3 Reflective Languages 10
2.2.4 Transparency 10
2.3 The Meta-level Protocol of **Guaraná** 11
2.3.1 The kernel of **Guaraná** 11
2.3.2 Meta-Objects 12
2.3.3 Composers 13
2.3.4 Meta-configuration management 13
2.3.5 Security 17
2.3.6 Libraries of Meta-Objects 19
2.4 Conclusions 19

3 Implementação **21**
3.1 Introduction 22
3.2 Reflective Architecture 23
3.3 Implementation 24
3.3.1 Classes and Interfaces 25
3.3.2 Changes to the Java Virtual Machine 29

3.4 Performance 30
3.4.1 Empty loop 32
3.4.2 Empty `synchronized` block 32
3.4.3 Invoking a `static` method 33
3.4.4 Invoking a `private` method 34
3.4.5 Invoking a non-`final` method 35
3.4.6 Invoking an `interface` method 36
3.4.7 Loading a `static` field 37
3.4.8 Writing to a `static` field 38
3.4.9 Loading a non-`static` field 38
3.4.10 Writing to a non-`static` field 40
3.4.11 Loading the `length` of an array 40
3.4.12 Loading an element of an array 41
3.4.13 Writing to an element of an array 42
3.4.14 Creating objects 43
3.4.15 Creating arrays 43
3.4.16 Creating arrays of arrays 44
3.4.17 Printing a `String` 44
3.4.18 Compiling a program 44
3.4.19 Overall analysis 46
3.5 Future optimizations 48
3.6 Conclusions 49

4 Tutorial **50**
4.1 Introduction 51
4.2 Basics 52
4.2.1 Starting up 52
4.2.2 Intercepting array operations 54
4.2.3 Intercepting class operations 55
4.2.4 Meta-configuration propagation 57
4.3 Intermediate 59
4.3.1 Dynamic reconfiguration 59
4.3.2 Composing meta-objects 62
4.3.3 Modifying results 64
4.3.4 Modifying operations 67
4.3.5 Using messages 71
4.3.6 Creating Proxies 73
4.4 Advanced 76
4.4.1 Meta-objects with restricted access 76

4.4.2 Multi-object meta-objects 77
4.4.3 Coping with replaced operations 79
4.4.4 Reconfiguration details 81
4.5 Conclusion 82

5 Aplicações **83**
5.1 Introduction 85
5.2 The Java-based implementation of **Guaraná** 85
5.3 Reusable Meta-Objects for Distributed Systems 87
5.3.1 Persistence 87
5.3.2 Replication 88
5.3.3 Distribution 88
5.3.4 Caching 89
5.3.5 Migration 89
5.3.6 Accounting 90
5.3.7 Monitoring 90
5.3.8 Atomicity 90
5.4 Conclusion 91

6 Conclusão **93**

A Obtendo Guaraná e MOLDS **95**

Referências Bibliográficas **96**

Lista de Tabelas

3.1 Empty loop . . . 32
3.2 Empty `synchronized` block . . . 33
3.3 Invoking a `static` method . . . 33
3.4 Invoking a `private` method . . . 35
3.5 Invoking a non-`final` method . . . 36
3.6 Invoking an `interface` method . . . 36
3.7 Loading a `static` field . . . 37
3.8 Writing to a `static` field . . . 39
3.9 Loading a non-`static` field . . . 39
3.10 Writing to a non-`static` field . . . 40
3.11 Loading the `length` of an array . . . 41
3.12 Loading an element of an array . . . 41
3.13 Writing to an element of an array . . . 42
3.14 Printing a `String` . . . 44
3.15 Compiling a program: total execution time . . . 45
3.16 Compiling a program: JIT compilation time . . . 45
3.17 Compiling a program: disregarding JIT-compilation time . . . 45
3.18 Overall analysis (interpreter engine) . . . 46
3.19 Overall analysis (JIT-compiler engine) . . . 47

Lista de Figuras

2.1 UML Sequence Diagrams 7
2.2 Reifying an operation 9
2.3 Sequential composition of meta-objects 14
2.4 Reconfiguring an object's meta-configuration 15

Capítulo 1

Introdução

Nosso objetivo original era criar um ambiente de desenvolvimento de sistemas distribuídos, valendo-nos de técnicas de reflexão computacional [38, 51] para implementar de maneira transparente os diversos mecanismos úteis a essa categoria de aplicações.

Infelizmente (ou felizmente), dentre todas as plataformas reflexivas estudadas, citadas no Capítulo 2, nenhuma delas satisfazia simultaneamente os requisitos que julgávamos essenciais para o desenvolvimento do ambiente almejado, tais como:

Ortogonalidade: arquiteturas reflexivas que associam aspectos de gerência à classe a que um objeto pertence limitam a transparência do sistema, por exigirem alteração no código do nível base (criação de subclasses) para possibilitar diferenciação do comportamento reflexivo para instâncias de uma mesma classe.

Reconfigurabilidade: a alteração dinâmica de elementos de meta-nível é bastante desejável na construção de sistemas que ofereçam suporte a tolerância a falhas, em função da possibilidade de reconfiguração.

Não-intrusão: a necessidade de alterar código do nível base para adicionar comportamento reflexivo a objetos é extremamente indesejável, por reduzir o grau de transparência obtido; consideramos prejudicial até mesmo a possibilidade de o nível base interagir com o meta-nível, por quebrar a separação entre os níveis.

Reuso: a possibilidade de criar componentes de meta-nível reutilizáveis em diferentes contextos é um requisito indispensável para a construção do ambiente desejado, especialmente se esses componentes puderem ser combinados a fim de determinar comportamentos reflexivos complexos.

Aceitação da linguagem: linguagens de propósito geral, com vasta utilização, são certamente preferíveis em comparação com linguagens direcionadas a nichos específicos; a ausência de alteração à linguagem de programação é também bastante desejável, a fim de que aplicações não reflexivas possam ser beneficiadas pela introdução de mecanismos reflexivos sem alteração de seu código.

Portabilidade: a implementação da plataforma reflexiva deve ser portável e inter-operável, a fim de possibilitar a implementação de sistemas distribuídos heterogêneos.

Segurança: a capacidade de reconfigurar dinamicamente o comportamento reflexivo de um objeto abre inúmeras possibilidades, algumas delas possivelmente perigosas; para o desenvolvimento de aplicações confiáveis, julgamos conveniente que se possam estabelecer políticas de segurança, a fim de evitar ou ao menos limitar os danos que reconfigurações indesejadas possam causar.

Na falta de uma plataforma reflexiva que atendesse a esses requisitos, decidimos adiar a criação do ambiente de desenvolvimento de sistemas distribuídos, em favor da definição de uma plataforma que favorecesse sua futura implementação.

Adotamos Java[1] como linguagem de programação alvo, mas tomamos como objetivo a especificação de uma arquitetura reflexiva que pudesse ser implementada, com variados graus de dificuldade, sobre plataformas reflexivas já existentes. Assim surgiu **Guaraná**.

Esta dissertação está organizada na forma de uma coletânea de textos em formato de artigos, alguns já publicados como relatórios técnicos, um deles aceito em *workshop* nacional. Para evitar repetições desnecessárias, foram omitidos dos artigos os apêndices contendo intruções sobre como obter o *software* e sua documentação, apresentado no Apêndice A, e os agradecimentos, transferidos para o corpo da dissertação.

No Capítulo 2, apresenta-se o artigo que descreve a arquitetura reflexiva proposta para atender aos diversos requisitos expostos anteriormente. No Capítulo 3, descrevemos a implementação dessa arquitetura, que envolveu a modificação de uma máquina virtual Java que permitia a livre distribuição de seu código fonte. O Capítulo 4 consiste de um tutorial contendo inúmeros trechos de código que exemplificam a utilização dos principais mecanismos da arquitetura do **Guaraná**. Como retomada do objetivo original, o Capítulo 5 demonstra como a arquitetura do **Guaraná** permite uma implementação simples de diversos serviços de meta-nível reutilizáveis e combináveis, adequados para a implementação de aplicações distribuídas confiáveis. Encerra-se a dissertação com a Conclusão, objeto do Capítulo 6.

[1]Java é uma marca registrada da Sun Microsystems, Inc.

Capítulo 2

Arquitetura de *Software*

Prólogo

Neste primeiro artigo, apresentamos conceitos básicos de reflexão computacional, tais como interceptação e materialização (*reification*) e apontamos limitações das diversas plataformas reflexivas analisadas.

Em seguida, apresentamos o protocolo de meta-objetos do **Guaraná**, especificando o comportamento esperado de meta-objetos e *composers*. *Composers* são meta-objetos que delegam operações para outros meta-objetos, a fim de combinar suas funcionalidades. Descrevemos também o mecanismo de propagação de meta-configurações para objetos criados dinamicamente, que torna possível que o meta-nível seja totalmente transparente para o nível base. Definimos ainda as estratégias para a manutenção de consistência das meta-configurações, através de políticas de segurança no meta-nível.

Uma versão anterior deste artigo está disponível como relatório técnico IC-98-14.

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.

2.1 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) [9, 29, 47, 48]. 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

*Islene Calciolari Garcia *recebeu auxílio da FAPESP através da bolsa 95/1983-8.*
[1]Java is a trademark of Sun Microsystems, Inc.

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 [30, 31] 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 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 [38, 51] (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 [38, 58].

Several offer class-wide reflective facilities: each class is associated with a *single* meta-class [15, 16, 22, 32]. Some allow groups of objects to be attached to groups of meta-objects [27, 40, 41, 42, 60], by giving every meta-object the responsibility for handling a specific aspect of the system–object interaction. Also, hybrid models [39] 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.

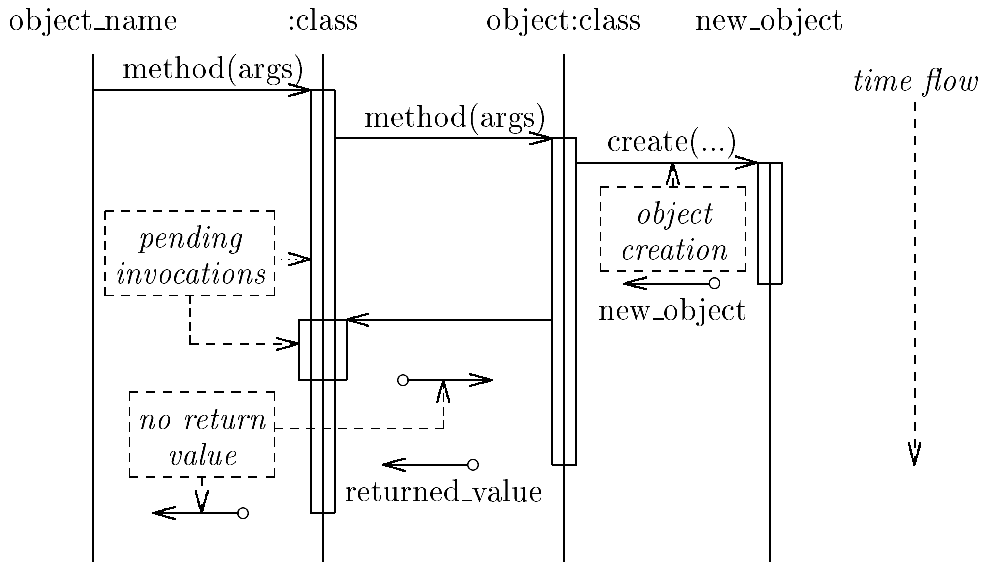
This paper describes the reflective architecture of **Guaraná**, a flexible language-independent meta-protocol that encourages the creation of highly coherent, loosely and transparently coupled meta-level objects. An application developed within **Guaraná**'s framework is easily adapted to conform to new requirements, and the implementation of meta-level requirements can be easily reused in other applications. Section 2.2 contains an introduction to computational reflection, as used to support open architectures. Section 2.3 describes the software architecture and meta-level protocol of **Guaraná**. Finally, in Section 2.4, 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 [47]. Figure 2.1 specifies the semantics of such diagrams.

2.2 Computational Reflection

Computational Reflection [38, 51] 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 2.2.1). Meta-level architectures are discussed in Section 2.2.2 and reflective languages in Section 2.2.3. Finally, Section 2.2.4 shows the use of computational reflection in the structuring and implementation of system-oriented mechanisms.



The meta-diagram above shows the subset of the UML [47] 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 2.1: UML Sequence Diagrams

2.2.1 Reification

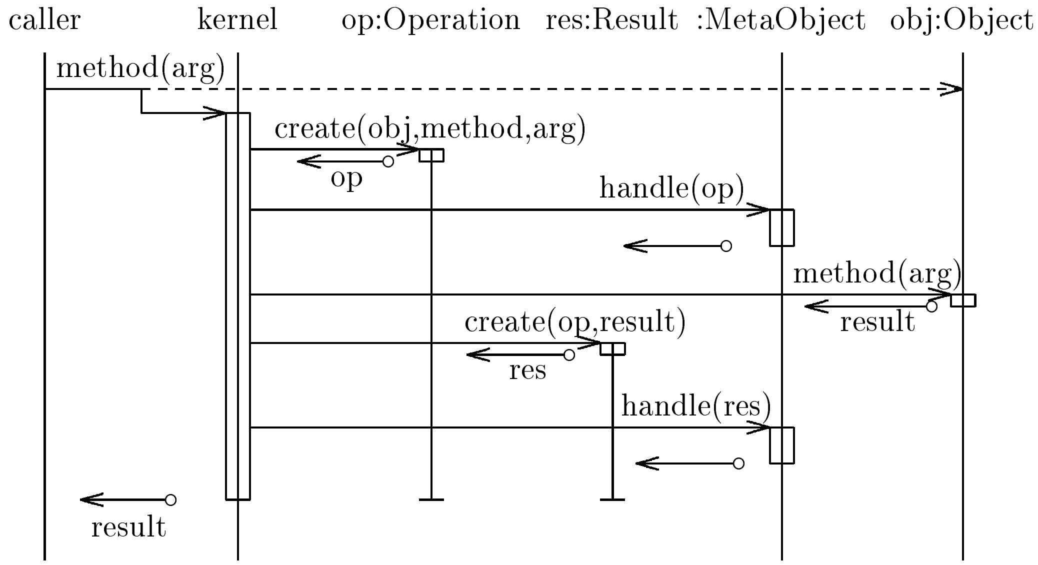
For the meta level to be able to reflect on several objects, specially if they are instances of different classes, it must be given information regarding the internal structure of objects. This meta-level object must be able to find out what are the methods implemented by an object, as well as the fields (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 2.2. 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.

2.2.2 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++ [15, 16] and CLOS [32] 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 [33] (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 [59], MMRF [41, 42], CodA [40] and Iguana [27] 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.



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 2.2: Reifying an operation

2.2.3 Reflective Languages

Several object-oriented languages have already been designed or extended in order to support reflection. Languages such as KRS (3-KRS [38]), LISP (3-LISP [50], CLOS [32]), ABCL (ABCL/R [58] and ACBL/R2 [39]), AL-1/D (MMRF [41, 42]), C++ (Open C++ [15, 16] and Iguana [27]) and Java (MetaXa [33]) 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 [38]. 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 [38, 50, 58]. 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 preprocessing 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 [38] to only two levels; the work by Bijnens et al [7] 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.

2.2.4 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 [45, 53], location transparency [43], replication [16, 34], fault tolerance [1, 2, 21, 20] and atomic actions [54, 55]. Reflection has also been shown to be useful in the development of distributed systems [13, 37, 52, 59, 60] and for simplifying library protocols [57].

Persistence [45, 53], 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 [43] 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.

2.3 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.

2.3.1 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.

2.3.2 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.

2.3.3 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 2.3 illustrates the behavior of a sequential composer.

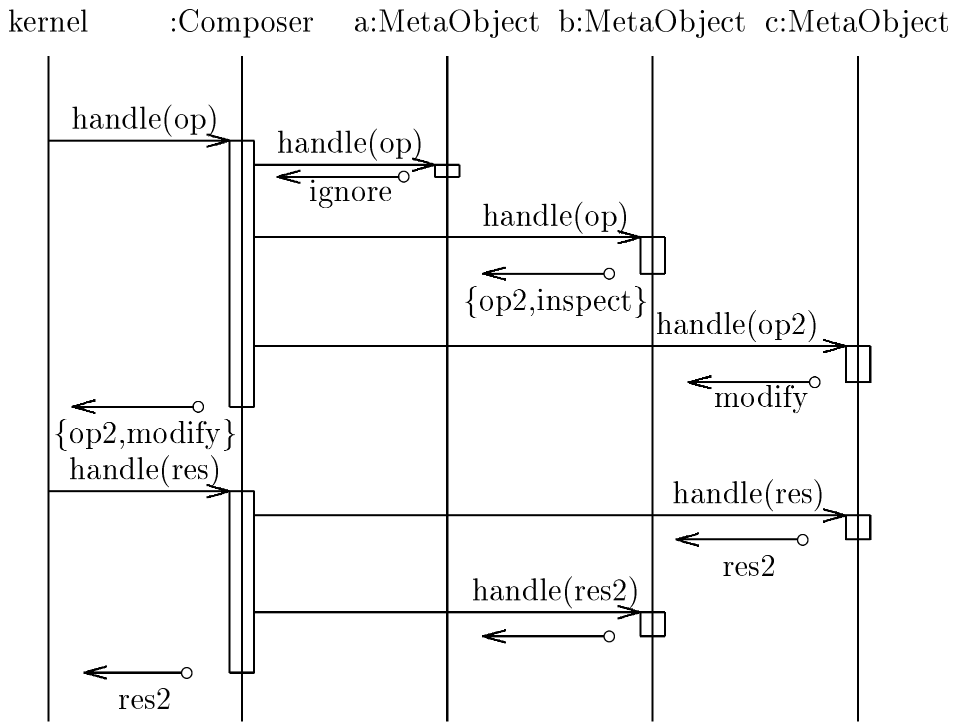
A concurrent composer might have been implemented too: it would present an operation to all meta-objects concurrently. Some may generate results or actions that conflict with those generated by others. In order to cope with this situation a default adjudicator could be provided: it would raise an exception 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 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.

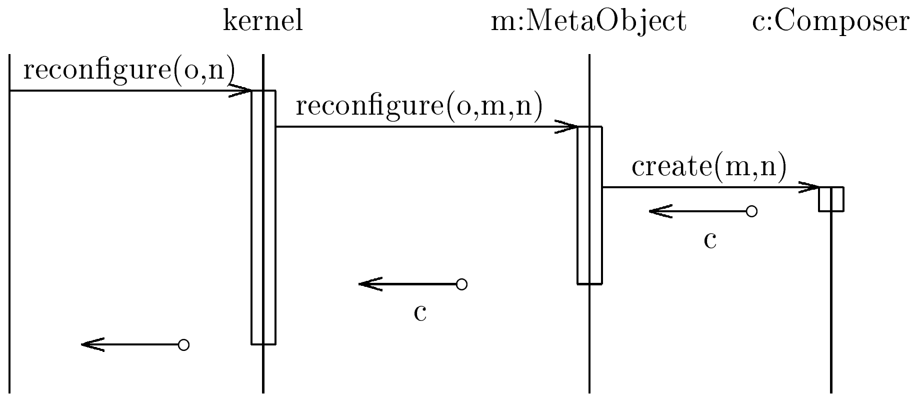
2.3.4 Meta-configuration management

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,



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 2.3: Sequential composition of meta-objects



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 2.4: Reconfiguring an object's meta-configuration

then associate 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 2.4. 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 2.4. 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.

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.

2.3.5 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.

2.3.6 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**, 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 [10].

2.4 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 [46]. 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 [36] 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.

Capítulo 3

Implementação

Prólogo

No artigo a seguir, descrevemos concisamente a arquitetura de *software* do **Guaraná** e as classes utilizadas em sua implementação na linguagem Java; mais detalhes podem ser inferidos a partir do tutorial do Capítulo 4 ou obtidos explicitamente na API disponível na *home-page* do **Guaraná**, no endereço `http://www.dcc.unicamp.br/~oliva/guarana`.

Em maior grau de profundidade, relatamos as alterações efetuadas sobre o *Kaffe*, uma máquina virtual Java de distribuição livre, para que ele oferecesse suporte para interceptação, materialização e criação de operações, conforme a especificação do **Guaraná**, descrita no Capítulo 2. Apresentamos também algumas medidas comparativas de desempenho, tomadas a fim de avaliar o impacto do código de interceptação sobre o tempo de execução das operações.

Uma nova versão deste artigo, com dados mais atualizados, será publicada como relatório técnico IC-98-32.

The Implementation of **Guaraná** on Java^{TM}

Alexandre Oliva Luiz Eduardo Buzato

oliva@dcc.unicamp.br buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

September 1998

Abstract

Guaraná is a reflective architecture that aims at simplicity, flexibility, security and reuse of meta-level code. It is implemented as an extension of *Kaffe OpenVM*^{TM}, a free implementation of the Java^{TM} Virtual Machine.

We describe the Java classes that implement the meta-object protocol of **Guaraná**, and the modifications introduced in the virtual machine to intercept and reify of operations.

Finally, we evaluate the performance impact of our modifications, and suggest some optimizations that may be implemented in the future.

3.1 Introduction

Guaraná is a software architecture for computational reflection [38, 51] that introduces a runtime meta-object protocol that allows for dynamic composition of meta-objects in order to build up potentially complex meta-level behavior. This leads to the development of simple, coherent meta-objects. The possibility for their composition improves code reuse. The fact that this composition can be established and modified at run time, on a per-object basis, makes **Guaraná** flexible. Finally, its meta-object protocol provides controlled access to the meta-level of an object, helping separate meta-level from base-level concerns, just like encapsulation helps with the separation of interface from implementation in the object-oriented paradigm.

*Islene Calciolari Garcia *recebeu auxílio da FAPESP através da bolsa 95/1983-8.*

In the next section, we briefly describe the reflective architecture of **Guaraná**. Section 3.3 contains a description of our implementation of this architecture, extending a freely-available Java[1] Virtual Machine. In Section 3.4, we present some figures about the impact of **Guaraná** on the performance of applications, then we list some possible future optimizations in Section 3.5. Finally, in Section 3.6, we summarize the main points of the text.

3.2 Reflective Architecture

This section assumes familiarity with concepts pertinent to computational reflection. Although not strictly required, some familiarity with **Guaraná** is desirable, as this section serves only to introduce **Guaraná** in a very concise manner. For a detailed explanation, please refer toChapter 2.

Any object may be directly associated with at most one meta-object, called its primary meta-object. If such an association exists, the kernel of **Guaraná** will intercept and reify all method invocations and field accesses made to the base-level object and present them to the associated primary meta-object. This meta-object may reply with a result for the operation, a replacement operation—to be performed instead of the requested one—, or a request to observe or modify the result of the operation (or its replacement), after it is performed.

One of the actions that a meta-object may take when dealing with an intercepted operation or result is to delegate it to other meta-objects. When a meta-object plays the role of delegator, it is called a composer. Composers have a central role in the reflective architecture of **Guaraná**: they are fundamental to allow the combination of several autonomous meta-objects into a meta-configuration. As composers are themselves meta-objects, they may be composed further, in a potentially infinite hierarchy. Furthermore, every meta-object, just like any other object, can have its own meta-configuration, in another potentially infinite orthogonal hierarchy.

There is no way for an object to find out what meta-object is its primary meta-object. This impossibility is intentional: we believe that an object should not interact with its meta-objects; it should not even be aware of their existence, otherwise the separation of concerns between the levels would be broken. The meta level, on the other hand, is responsible for controlling the base level, so a meta-object does know what objects are associated with it.

Meta-objects have privileged access to objects whose meta-configurations they belong to. They can create operations that violate standard access control by using operation factories, distributed whenever a meta-object is associated with the base-level object, and invalidated as soon as the primary meta-object is replaced.

The composition structure determines a hierarchy of authority and control over the base-level object. The primary meta-object is the highest authority, so it is able to restrict the

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

ability of their components to create operations or set their results. The identity of the primary meta-object of an object is protected, so as to prevent this hierarchy from being subverted. Thus, a meta-object is usually unable to tell whether it is the primary meta-object of an object, since it is possible for a composer to hide from its components, by interacting with other meta-objects just like the kernel of **Guaraná** would.

Reconfiguration requests are also subject to the approval of the existing meta-configuration. It is possible to request for the replacement of subsets of the meta-configuration of an object, and it is up to the existing meta-configuration to determine the new meta-configuration, that may be any combination of the requested meta-configuration with the existing one.

The meta-configuration of a class can affect the meta-configuration of its instances in two situations: (i) when a reconfiguration request is issued to an object that lacks a meta-configuration, the meta-configuration of its class and of its base classes will be given the opportunity to inspect and modify the requested meta-configuration, and (ii) when an object is instantiated, the meta-configuration of the creator is requested to provide a meta-configuration for the new object, then the meta-configuration of the class of the new object is given the opportunity to try to reconfigure it.

These two kinds of interactions with the meta-configuration of an object use a general mechanism: broadcasting a message to all components of the meta-configuration of an object. This allows for communication between meta-objects without requiring explicit mutual references, and allows the inter-meta-object protocol to be extended without modifications to the interface of class MetaObject.

3.3 Implementation

Most of **Guaraná** was coded in Java, but some modifications in the Java Virtual Machine were needed, in order to provide for interception of operations such as invocation of methods, read/write from/to fields and array elements, creation of objects and arrays, entries and exits from monitors.

In the next section, we describe the classes that implement the reflective architecture of **Guaraná**. Then, we list the modifications we have introduced in *Kaffe OpenVM*, a free[2] implementation of the JVM developed at Transvirtual Technologies, so that it supports our reflective architecture.

[2]The *Kaffe OpenVM* is distributed under the terms of the GNU General Public License.

3.3.1 Classes and Interfaces

All the classes used in the implementation of **Guaraná** are defined in the package `BR.unicamp.Guarana`.

Guarana

Class Guarana represents the kernel of **Guaraná**. It provides methods for modifying meta-configurations of objects (reconfigure), broadcasting Messages to components of a meta-configuration (broadcast), creating proxy objects (makeProxy) and performing operations (perform). This last method is invoked as part of the interception mechanism: it runs the operation handling protocol, i.e., it presents the reified operation to the primary meta-object of the target object, delivers the operation to the target object, then, if requested, presents the result to the primary meta-object.

Internally, some `private native` methods are used for obtaining and modifying the primary meta-object of an object, and for delivering an operation to its target object.

Some standard Object methods would be useful for gathering information about base-level objects from the meta-level. However, if meta-objects would invoke them directly, these invocations would be intercepted, but then the meta-object might decide to invoke the method again, and infinite recursion would result. Thus, the kernel of **Guaraná** offers alternate implementations of some of these methods, such as toString, hashCode and getClass. They take a reference to a base-level object and produce the result that would be produced by the invocation of the standard implementations of these methods, but they do not interact with the base-level object, removing the risk of infinite recursion. A method getClassName, that obtains the name of a given class, is also provided.

Operation

Every base-level operation addressed to an object or class whose meta-configuration is non-null is reified as an Operation object. This class is mostly implemented in native code, for performance reasons: it would be too expensive to wrap every non-object argument type into an array of arguments.

This class provides methods for querying the operation about its nature (i.e., method or constructor invocation, field read or write, monitor enter or exit, array length read or array element read or write) and arguments (i.e., the target object, the Thread that created the Operation, the method or constructor to be invoked, or the field or array index to be read or written, and the invocation arguments or the value to be written).

There are methods for validating and performing operations. The former checks whether an operation is consistent and can be performed in the base level (because it is possible to create inconsistent operation objects); the latter dispatches the operation for interception, validation and execution.

This class also provides operations for checking whether an operation is a replacement operation and what operations are directly or indirectly replaced by it.

An implementation of method toString, that fully describes the operation, is also provided.

Result

A Result object is implicitly created by the kernel of **Guaraná** after an Operation is executed. It provides methods for querying what Operation it refers to, as well as what is the actual value of the result.

However, results can also be created by MetaObjects to modify the result of an operation, by invoking the `static` methods return**T** (where **T** is Object or a primitive type name) or throwObject. If the kernel of **Guaraná** receives a Result of the throwObject variant as the result of an Operation, it throws or rethrows the contained Throwable, instead of returning control to the caller.

Result objects serve yet another purpose: they can be used by MetaObjects to replace operations, and to request permission to observe or modify the result of an operation after it has been performed. This kind of Result will usually be created and returned by a MetaObject as it handles an Operation.

This class is also implemented mostly in native code, in order to avoid as much as possible wrapping primitive types.

MetaException

Whenever the interception of a base-level operation terminates by throwing an exception, the kernel of **Guaraná** creates a MetaException to encapsulate it. Since MetaException is a subclass of RunTimeException, it can propagate through methods that have not declared it.

MetaObject

The class MetaObject is the root of the class hierarchy for every possible implementation of meta-object. Methods for initialization (initialize) and termination (release), interception of Operations, Results and Messages (handle), reconfiguration (reconfigure) and configuration of new objects (configure) can be overridden by its subclasses.

Although MetaObject is an abstract class, because its instances would not do anything useful, there are no abstract methods. All methods implement reasonable defaults, so that subclasses can focus on relevant methods only.

Composer

A Composer is a meta-object that delegates operations, results and messages to other meta-objects. Thus, it `extends` class MetaObject by adding two other methods, used to query the Composer about which MetaObjects it delegates to. Subclasses may specialize method getMetaObjectsArray so that it returns an array containing all MetaObjects it may delegate to.

A very inefficient implementation of the previous method is provided, based on the Java Enumeration returned by the `abstract` method getMetaObjects. Subclasses *must* implement this method, so that the returned Enumeration iterates on all MetaObjects it may delegate to. However, since an Enumeration may contain arbitrary Objects, some of them may be used to provide additional control information about how and when the Composer delegates to particular MetaObjects. However, there is no standardized form of control information.

SequentialComposer

This is a simple specialization of Composer that maintains an array of MetaObjects, and delegates Operations and Messages (see below) to them sequentially. Results are also delegated sequentially, but in the reverse order.

Besides implementing the standard MetaObject and Composer methods, SequentialComposer provides `static` methods for delegating operations and results to subsets of a meta-object array, as well as for configuring new objects.

Message

This is a Java `interface` that provides no methods at all. Only instances of classes that implement this `interface` can be used as arguments to method broadcast of the kernel of **Guaraná**, to exchange information with components of the meta-configuration of an Object.

This restriction was imposed to prevent arbitrary Objects from being broadcast to meta-configurations. By requiring classes to be defined for new types of message, we avoid possible ambiguities that might have arisen if implementors of different MetaObject had adopted different meanings for predefined classes.

NewObject

This is an implementation of Message that stores a reference to an Object. An instance of this class is created by the kernel of **Guaraná** as part of the Object creation process, if the instantiated class has a `null` meta-configuration.

After an Object is allocated and the meta-configuration of its creator determines its meta-configuration, but before the constructor of the object is invoked, a NewObject Message is broadcast to the meta-configuration of the class of the new Object, so that the meta-configuration of the class can try to affect the meta-configuration of its instances.

The class NewObject implements a single method, that returns a reference to the newly-created Object.

NewProxy

Just after creating a pseudo object, the method makeProxy of class Guarana creates a NewProxy Message and broadcasts it to the meta-configuration of the instantiated class. NewProxy is a subclass of NewObject, whose broadcast allows meta-configurations of classes to reject the creation of proxy Objects by throwing RunTimeExceptions.

If the broadcast terminates successfully, method makeProxy will issue a reconfiguration request to install the meta-object suggested by its caller as the primary meta-object of the created proxy.

InstanceReconfigure

The kernel of **Guaraná** broadcasts a Message of this class to the class of an object, as well as to its superclasses, when it is asked to replace the primary meta-object of an object, but the meta-configuration of the object is `null`.

Class InstanceReconfigure provides methods for obtaining references to the object and to the suggested primary meta-object, as well as for modifying the suggested primary meta-object.

After the Message is broadcast to the meta-configuration of all classes up to root of the class hierarchy, the meta-object in the InstanceReconfigure Message is installed as the primary meta-object, unless it has become `null`.

OperationFactory

An OperationFactory is associated with a single base-level Object, and it can be used by a MetaObject to create Operations addressed to that Object. OperationFactories provide methods for obtaining a reference to the base-level Object and for creating method and constructor invocations, monitor enter and exit, field read and write, array length read, array element read and write, and a special do-nothing operation placeholder (nop).

Operations can be created as replacement or stand-alone ones. In the former case, the Operation to be replaced must be provided as an additional argument, and some validation is performed. In the latter case, an Operation that can be performed independently of any other is created.

OperationFactoryFilter

This class specializes OperationFactory, so that it delegates operation creation requests to another OperationFactory. Its methods can be overridden so as to restrict the set of Operations that can be created.

HashWrapper

Hashtables are a powerful feature of Java, but using them for mapping base-level objects to meta-level data such as operation factories, pending operations, etc, requires some care, because the implementation of Hashtable invokes key object methods such as hashCode and equals.

In order to avoid unintended interactions with base-level objects, a meta-level object should wrap them with HashWrappers, that implement methods hashCode and equals without invoking methods on the base-level object. Note that, if the base-level object specialized any of these methods, the specializations would be disregarded, because HashWrapper simulates the standard implementations of hashCode and equals. A standard implementation of toString is offered too.

3.3.2 Changes to the Java Virtual Machine

In this section, we describe the modifications we have introduced in *Kaffe OpenVM*, for it to support **Guaraná**.

First of all, every Object had to contain a reference to its MetaObject. For the sake of a simpler implementation, we have decided to add a field to the native description of class Object, instead of trying to encode this reference in modified class descriptors. The main drawback of this approach is that every object is augmented by one word; the main advantage is that checking whether an object is reflective or not (i.e., is associated with a non-null meta-configuration or not) is fast.

Classes are also Objects, thus they may also be associated with MetaObjects. The meta-object associated with a class will receive operations addressed to the class object itself, as well as operations involving `static` methods or fields of the class it represents. The reason for this unification is that `synchronized static` methods acquire locks on the class object. Therefore, if the meta-configuration of a class object could be different from the meta-configuration that handles `static` operations of the corresponding class, the semantics of class monitors in Java would not have been properly modeled in the meta level.

The meta-object reference is hidden from Java programs; it is only accessible in the implementation of the kernel of **Guaraná** and in native code.

The bytecodes that originally just invoked non-`static` methods now check whether the target Object is associated with a MetaObject. If it is, after performing dynamic binding, a method invocation Operation is constructed and performed, that is, operation handling, actual execution and result handling take place. The yielded Result is then un-reified. The bytecode used to invoke `static` methods was also modified, so that the Operation is intercepted only if the class is associated with a MetaObject.

Bytecodes that used to read from and write to fields were changed so that, if the Object (for non-`static` fields) or the class (for `static` fields) is associated with a MetaObject, the Operation is intercepted.

Bytecodes used for indexing arrays, both for reading and for writing, as well as the bytecode for obtaining the length of an array, have been extended so that the array MetaObject can intercept such Operations.

The bytecodes that allocate memory for Objects and arrays were changed so as to support interception of object creation. After they allocate memory for the new Object, they request the primary MetaObject of the creator of the new Object to configure it, a mechanism we have named *meta-configuration propagation*.

The *creator* of an Object is determined based on the method in which the object allocation bytecode appears. If it is a `static` method, the creator is the Class in which the method is declared. Otherwise, the creator is the Object referred to by the keyword `this`, within that method invocation.

After the new Object is configured, a NewObject Message is broadcast to the meta-configuration of the Class of the new Object.

Monitor-related bytecodes, as well as implicit entries and exits of object or class monitors in `synchronized` methods, have been extended so that these Operations can be intercepted.

In addition to bytecodes, the `native` implementation of the Java Core Reflection API and of the Java Native Interface had to be modified to support interception.

3.4 Performance

We have run some performance tests to try to evaluate the impact of introducing reflective capabilities into a Java interpreter. Our tests have been performed on four different platforms: a single-processor 167 MHz SPARC Ultra 1 running Solaris 2.6, a dual-processor 200 MHz SPARC Ultra Enterprise 2 running Solaris 2.5, a 100 MHz Pentium running RedHat Linux 5.0, and a 233 MHz Pentium Pro running RedHat Linux 5.0.

On each host, we have run the same Java program, under different interpreters and configurations. A description of each configuration follows:

`KJ-` Kaffe just-in-time (JIT) compiler without **Guaraná**.

`KJG` Kaffe JIT compiler with **Guaraná**.

`JDK` Sun Java Development Kit (JDK).

`KI-` Kaffe interpreter without **Guaraná**.

`KIG` Kaffe interpreter with **Guaraná**.

`KJN` Kaffe JIT compiler with **Guaraná**, intercepting all operations with a do-nothing meta-object.

`KIN` Kaffe interpreter with **Guaraná**, intercepting all operations with a do-nothing meta-object.

`KJM` Kaffe JIT compiler with **Guaraná**, intercepting and logging all operations, results, messages, initializations and configurations with a MetaLogger.

`KIM` Kaffe interpreter with **Guaraná**, intercepting and logging all operations, results, messages, initializations and configurations with a MetaLogger.

MetaLogger is an example of MetaObject, distributed with **Guaraná**, that logs a message to the standard output every time one of its methods is invoked. It is useful for observing the behavior of base-level objects associated with it.

We have used Kaffe 0.10.1, **Guaraná** 1.3,[3] and Sun JDK 1.1.6 (the official Solaris version and the Linux port). Kaffe and **Guaraná** were compiled with EGCS 1.0.3a, with default optimization levels.

For each configuration, we have timed several different operations. Each operation was timed by running it repeatedly inside a loop, with an iteration count large enough for elapsed time in the loop to be greater than 1 second. Each test was run 50 times on each configuration and platform, and the presented values are the average of the runs. All figures are given in seconds. A change in the order of magnitude of the averages obtained is indicated by shifting the figures to the left. As an example, observe Table 3.1. The program used to perform the tests is a slightly modified version of the one distributed with **Guaraná** 1.3.

We have also measured and averaged the compilation time of the test program itself, in the configurations that do not involve meta-objects, so as to estimate the overall performance impact on a real application caused by introducing the *ability* to intercept operations, without actually intercepting them.

Section 3.4.19, page 46, contains a summarizing analysis of the data obtained during the tests. In that section, Table 3.18 and Table 3.19 are used to present a less fragmented view of the test data obtained so far.

[3]Actually, both Kaffe and **Guaraná** were updated to the snapshot of Kaffe released on June 27, 1998, because this fixed a bug that would cause the Kaffe interpreter (without **Guaraná**) to crash on x86, while running the compilation test. This update did not introduce any other significant change.

3.4.1 Empty loop

This test consists exclusively of a loop that decrements a variable until it becomes zero. It does not trigger any reflective mechanism, as it only deals with a variable local to a method. We have included the average times for the empty loop (Table 3.1) because they can be used as a benchmark for the other tests, as they are also based on measures of the time consumed by a loop that executes the block of instructions pertinent to this test. The times shown in Table 3.1 correspond to the time for a single iteration in the loop.

Table 3.1: Empty loop

Conf	`i586`	`i686`	`sparc ultra1`	`sparc ultra2`
`KJ-`	3.1 e−8	2.2 e−8	6.0 e−8	5.0 e−8
`KJG`	3.1 e−8	2.3 e−8	6.0 e−8	5.0 e−8
`JDK`	2.2 e−7	2.0 e−7	3.0 e−7	2.5 e−7
`KI-`	9.2 e−7	5.5 e−7	6.2 e−7	5.2 e−7
`KIG`	1.0 e−6	6.0 e−7	6.3 e−7	5.2 e−7
`KJN`	3.1 e−8	2.8 e−8	6.0 e−8	5.0 e−8
`KIN`	1.0 e−6	5.9 e−7	6.3 e−7	5.2 e−7
`KJM`	3.1 e−8	2.7 e−8	6.0 e−8	5.0 e−8
`KIM`	1.0 e−6	5.9 e−7	6.3 e−7	5.2 e−7

3.4.2 Empty `synchronized` block

The figures presented in Table 3.2 correspond to the time needed to enter and exit an object's monitor in a loop iteration. In the tests that involve meta-objects, both operations are intercepted.

Introducing interception ability in these two operations did not require modification to the definition of bytecodes, as Kaffe implemented `monitorenter` and `monitorexit` by calling C functions through preprocessor macros lockMutex and unlockMutex. These macros were also called just before entering and exiting a `synchronized` method.

We just had to redefine these macros so that alternate functions were called. These functions test whether the object passed as argument is associated with a meta-object or not. If it is, a monitor enter or exit operation object will be created, and perform of the kernel of **Guaraná** will be invoked to intercept it. If the meta-object reference in the object is `null`, the original lock or unlock function will be called.

Table 3.2: Empty `synchronized` block

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	2.5 e−6	1.2 e−6	3.7 e−6	3.1 e−6
KJG	2.8 e−6	1.3 e−6	4.1 e−6	3.4 e−6
JDK	2.9 e−6	1.4 e−6	1.9 e−6	1.5 e−6
KI-	5.7 e−6	3.0 e−6	5.9 e−6	4.9 e−6
KIG	5.8 e−6	3.3 e−6	6.7 e−6	7.1 e−6
KJN	2.8 e−4	1.3 e−4	2.5 e−4	2.0 e−4
KIN	8.5 e−4	2.8 e−4	4.1 e−4	3.9 e−4
KJM	4.3 e−3	1.6 e−3	2.9 e−3	2.0 e−3
KIM	2.4 e−2	8.9 e−3	1.1 e−2	1.0 e−2

3.4.3 Invoking a `static` method

The numbers presented in Table 3.3 represent the amount of time spent on an invocation of a `static` method that takes no arguments, and returns `void`. In the last four cases, the class that declares the method is reflective, so the invocation is intercepted. In order to be able to intercept this kind of invocation, we had to modify the definition of bytecode `invokestatic`.

Table 3.3: Invoking a `static` method

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.4 e−7	1.0 e−7	1.9 e−7	1.5 e−7
KJG	2.2 e−7	1.2 e−7	2.5 e−7	2.1 e−7
JDK	3.1 e−7	1.8 e−7	3.8 e−7	3.2 e−7
KI-	8.9 e−6	3.8 e−6	6.3 e−6	4.6 e−6
KIG	7.8 e−6	4.0 e−6	5.8 e−6	5.0 e−6
KJN	1.8 e−4	7.4 e−5	1.4 e−4	9.1 e−5
KIN	4.1 e−4	1.5 e−4	2.3 e−4	2.1 e−4
KJM	2.6 e−3	1.1 e−3	2.0 e−3	1.1 e−3
KIM	1.2 e−2	4.1 e−3	5.3 e−3	4.9 e−3

In the interpreted case, before invoking recursively the interpreter function to run the static method, the bytecode would check whether the class whose method is to be invoked is reflective, i.e., if it is associated with a meta-object. If not, normal execution proceeds, otherwise a generic method invocation reification function is invoked. This function takes a pointer to the target object (in this case, since the method is `static`, this pointer is `null`), a pointer to the structure that describes the method to be invoked, a pointer to the top of the stack, onto which the arguments have been pushed, and a pointer to the stack slot where the return value should be stored.

This function tests again whether a meta-object is associated with the target class (or object); in general, a meta-object will be found, and an operation object representing the method invocation will be created and performed. In order to create the operation object, the argument list must be copied, so we must first parse the signature of the method in order to find out how many stack slots the argument list takes, so that we do not copy too much or too little. After the execution of the operation, the result is stored in the provided stack slot.

In the case of the just-in-time compiler, native code is generated so that, before pushing the method arguments onto the stack, the target class is tested for the existence of a meta-object. If the method does not need to be intercepted, normal method invocation takes place, otherwise, arguments are pushed onto the stack with an offset of two words, so that a reference to the method to be called and a reference to the target object (`null`, in this case) can be passed as the first two arguments. Then, an interception function is selected, based on the return type of the called method.

These interception functions all call a generic method invocation function provided by Kaffe. This function tests if the target class (or object) is reflective, and generates a direct invocation of the target method or an invocation of yet another interception function, but now a generic one. This function takes a pointer to the target object's meta-object, a pointer to the target object, a pointer to the method to be invoked, a `va_list`[4] containing the arguments to be passed to the method, and a pointer to a `union` where the result of the method should be stored. This function assumes the meta-object pointer is non-`null`, so it always reifies the invocation. First, it parses the method signature to find out how many stack slots it takes, then it allocates a memory area of appropriate size, then it copies the method arguments into this area, parsing the signature again to use the appropriate types to read the argument values from the `va_list`.

After the operation is performed, the result is stored in the provided `union`. The generic method invocation function returns this value as a `union`, and the type-specific interception function extracts the correct return value from this `union` and returns it.

3.4.4 Invoking a `private` method

Table 3.4 shows the cost of invoking a non-`static` `private` method of a potentially reflective object. The `invokespecial` bytecode, used in this kind of invocation, is also used for invoking constructors and, in some cases, `final` methods. It is the fastest non-`static` invocation because it is statically bound. The invoked method, in our test case, has no arguments besides the implicit `this`, and returns `void`.

[4] A `va_list` is a standard C structure that allows a function to accept a variable number of arguments.

Table 3.4: Invoking a `private` method

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.6 e−7	1.1 e−7	2.0 e−7	1.7 e−7
KJG	2.8 e−7	1.4 e−7	2.5 e−7	2.1 e−7
JDK	3.7 e−7	2.1 e−7	4.7 e−7	3.9 e−7
KI-	9.4 e−6	3.9 e−6	5.2 e−6	4.4 e−6
KIG	7.2 e−6	4.2 e−6	5.4 e−6	4.6 e−6
KJN	2.8 e−4	7.0 e−5	1.0 e−4	9.8 e−5
KIN	4.6 e−4	1.5 e−4	2.4 e−4	2.1 e−4
KJM	6.4 e−3	1.0 e−3	2.3 e−3	1.4 e−3
KIM	1.8 e−2	5.1 e−3	7.0 e−3	6.2 e−3

The implementation of this bytecode was modified almost exactly as the previous one, as the address of the method to be called is also known at JIT-compilation time. The only difference is that, instead of passing `null` as the place-holder for the pointer to the target object, the actual target object is passed.

3.4.5 Invoking a non-`final` method

Non-`private` non-`static` methods declared in classes (i.e., not in `interfaces`) are dynamically bound on a per-object basis. Kaffe implements dynamic binding using a per-class dispatch table, so that the element of the dispatch table corresponding to an overridden method points to the most derived overrider. In Table 3.5, we present the amount of time spent on invoking, with the `invokevirtual` bytecode, a do-nothing method that takes only the implicit `this` argument and returns void.

The greatest difficulty for intercepting this bytecode was that, in JIT compiler mode, the dispatch table only contained pointers to the native code generated for each method, but **Guaraná** needed pointers to the structures that describe methods. So, we modified the format of the dispatch table, so as to accommodate our needs: it has become twice as large, in JIT mode, because it contains pointers both to the native code and to the method structure. If the target object is not reflective, the pointer to native code is loaded from the dispatch table, otherwise, the pointer to the method structure is used to intercept the method invocation, just like in the other invocation bytecodes.

Table 3.5: Invoking a non-`final` method

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.2 e−6	9.3 e−7	5.9 e−7	5.0 e−7
KJG	1.1 e−6	4.0 e−7	6.3 e−7	5.3 e−7
JDK	7.9 e−7	5.9 e−7	7.9 e−7	6.5 e−7
KI-	1.0 e−5	4.2 e−6	5.9 e−6	4.9 e−6
KIG	1.2 e−5	4.6 e−6	6.2 e−6	5.4 e−6
KJN	1.8 e−4	7.2 e−5	1.2 e−4	1.2 e−4
KIN	4.4 e−4	1.5 e−4	2.3 e−4	2.0 e−4
KJM	3.0 e−3	9.6 e−4	2.1 e−3	1.4 e−3
KIM	1.5 e−2	5.1 e−3	7.4 e−3	6.3 e−3

3.4.6 Invoking an `interface` method

When the static type of an object (i.e., the type known at compile-time) is an `interface` one, the bytecode used for a method invocation is `invokeinterface`. Dynamic binding is much more expensive than in the `invokevirtual` case, because interface methods cannot share a common index in a dispatch table. Hence, for every invocation, the requested method name and signature must be looked up in the object's class, as well as in its superclasses. Although, in our example, dynamic binding ends up selecting the same method of the object used in the previous test, the dynamic binding takes much longer, as we can observe in Table 3.6.

Table 3.6: Invoking an `interface` method

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	2.6 e−6	1.1 e−6	1.9 e−6	1.6 e−6
KJG	2.7 e−6	1.2 e−6	2.0 e−6	1.7 e−6
JDK	1.4 e−6	1.1 e−6	9.9 e−7	8.2 e−7
KI-	1.4 e−5	5.2 e−6	8.6 e−6	6.4 e−6
KIG	1.2 e−5	5.5 e−6	8.0 e−6	7.0 e−6
KJN	2.2 e−4	7.3 e−5	1.5 e−4	1.1 e−4
KIN	4.6 e−4	1.6 e−4	2.3 e−4	2.1 e−4
KJM	3.0 e−3	9.6 e−4	1.8 e−3	1.5 e−3
KIM	1.6 e−2	5.2 e−3	7.6 e−3	6.3 e−3

Once again, the data provided by the JIT compiler runtime was not enough for intercepting method invocations correctly: the function that would look up the `interface` method and signature in the object's class and its superclasses would return a pointer to the native code of the selected method, but we needed a method structure. Thus, we have modified the look up function, so that it would return the method structure and, if the method did not have to be intercepted, we would load the address of the native code from the method structure, just as the look up function would have done.

3.4.7 Loading a `static` field

In Table 3.7, we show how long it takes to load the value of a `static int` variable from of a potentially reflective class into a local variable.

Table 3.7: Loading a `static` field

Conf	`i586`	`i686`	`sparc ultra1`	`sparc ultra2`
`KJ-`	1.1 e−7	3.2 e−8	6.6 e−8	5.5 e−8
`KJG`	2.0 e−7	6.3 e−8	1.9 e−7	1.6 e−7
`JDK`	3.3 e−7	2.8 e−7	4.6 e−7	3.8 e−7
`KI-`	3.1 e−6	1.6 e−6	2.1 e−6	1.7 e−6
`KIG`	3.2 e−6	1.8 e−6	2.1 e−6	2.3 e−6
`KJN`	1.7 e−4	6.5 e−5	1.2 e−4	9.5 e−5
`KIN`	4.0 e−4	1.4 e−4	2.1 e−4	1.9 e−4
`KJM`	2.3 e−3	6.9 e−4	1.6 e−3	1.1 e−3
`KIM`	1.2 e−2	3.7 e−3	5.7 e−3	4.8 e−3

The Kaffe interpreter implements the `getstatic` bytecode by looking up the address of the field in the field description structure and pushing its value onto the stack. A `switch` statement selects the appropriate number of stack slots to allocate and the size of the data to be copied. Moving the loaded value from the stack to the local variable takes another `bytecode`, that is not modified at all.

For the `getstatic` bytecode to support interception, we have inserted the meta-object test just before the `switch` statement; if no meta-object is available, the original code is executed, otherwise, we run another `switch` statement that just allocates stack space for the field to be loaded, and call a generic field load interception function, passing to it a pointer to the class' meta-object, a pointer to the class structure, a pointer to the class where the field is declared (that is the same as the previous one, but is needed for non-`static` fields), a pointer to the field structure, and a pointer to the stack slot where the value of the field should be stored. This function assumes the meta-object is non-`null`, so it just creates an operation object and performs it, storing the result of the operation into the provided stack slot.

The JIT compiler is much faster, as it encodes the address of the field in the generated code, and it selects the appropriate load and store instructions at compile-time. Furthermore, the value loaded into a register needs not be immediately stored in the stack frame, if it is going to be used in some other computation. Due to limitations in the JIT compiler, though, the register that represents the Java stack slot cannot be the same that represents the local variable: they represent different native stack slots. Furthermore, sooner or later, the registers have to be spilled onto their native stack slots, even if they are not going to be used in the future: Kaffe does not currently perform any kind of global analysis.

Introducing interception abilities in this bytecode has a rather high cost, because the test for meta-object introduces new basic blocks in the program. Since Kaffe does not perform global register allocation, it resets the state of all registers at the beginning of every basic block and spills them all at block's end. A simple optimization has allowed us not to reset the register states before the field load operation, despite the branch just before it, but, nevertheless, all registers are spilled after the field load, and, when the two branches merge back, all registers are reset, so that the field value must be loaded back from the stack if it is going to be used in the next few instructions.

In order to intercept field load operations, different interception functions are used for different field types. They take all arguments the generic field load interception function take, except the pointer to the result stack slot: the result is returned by the type-specific functions. They just call the generic field load interception function, passing them a pointer to a local stack slot, then return the appropriate value extracted from this slot.

3.4.8 Writing to a `static` field

Table 3.8 gives the time needed to write the value of a zero-valued local variable in a `static int` field of a potentially reflective class.

The modified bytecode, in this case, is `putstatic`. In the Kaffe interpreter implementation, it would just perform a `switch` statement, write the value on top of the stack onto the field address, and pop it from the stack. Our modified implementation includes a meta-object existence test. If a non-`null` meta-object is associated with the class that declares the `static` field, a generic field write interception function is called. This function takes the same arguments that the generic field load interception function expects, but, in this case, the pointer to the stack slot contains the value to be written.

In the JIT compiler case, all the optimizations and overheads presented in the previous section apply.

3.4.9 Loading a non-`static` field

The times needed to obtain the value of a field of type `int` from an object, that may be reflective, and store it in a local variable, are displayed in Table 3.9.

Table 3.8: Writing to a `static` field

Conf	`i586`	`i686`	`sparc ultra1`	`sparc ultra2`
`KJ-`	1.3 e−7	3.5 e−8	6.6 e−8	5.5 e−8
`KJG`	3.9 e−7	8.9 e−8	1.4 e−7	1.2 e−7
`JDK`	3.4 e−7	3.1 e−7	4.8 e−7	3.9 e−7
`KI-`	3.8 e−6	1.7 e−6	2.1 e−6	1.7 e−6
`KIG`	3.6 e−6	1.7 e−6	2.1 e−6	2.3 e−6
`KJN`	1.9 e−4	6.4 e−5	1.1 e−4	9.3 e−5
`KIN`	4.0 e−4	1.4 e−4	2.1 e−4	1.9 e−4
`KJM`	3.9 e−3	8.2 e−4	1.6 e−3	1.2 e−3
`KIM`	1.2 e−2	3.9 e−3	5.5 e−3	5.3 e−3

Table 3.9: Loading a non-`static` field

Conf	`i586`	`i686`	`sparc ultra1`	`sparc ultra2`
`KJ-`	6.9 e−8	4.2 e−8	6.0 e−8	5.0 e−8
`KJG`	1.2 e−7	9.7 e−8	1.6 e−7	1.4 e−7
`JDK`	3.3 e−7	2.6 e−7	5.5 e−7	4.5 e−7
`KI-`	4.4 e−6	1.9 e−6	2.7 e−6	1.9 e−6
`KIG`	3.7 e−6	1.9 e−6	2.4 e−6	2.6 e−6
`KJN`	2.2 e−4	6.3 e−5	1.1 e−4	8.7 e−5
`KIN`	4.0 e−4	1.4 e−4	2.1 e−4	1.9 e−4
`KJM`	2.7 e−3	9.4 e−4	1.9 e−3	1.4 e−3
`KIM`	1.5 e−2	4.9 e−3	6.5 e−3	6.7 e−3

One of the two differences between the `getfield` bytecode, used in this case, and `getstatic`, already described, has to do with the arguments passed to the interception functions: in this case, the second argument is the object whose field is going to be loaded, instead of a duplicated pointer to the class object.

The other difference is that, in the `static` case, the addresses of the class and of the field are known at compile-time, so they are treated as constants in the compiled code; in the non-`static` case, the address of the class is still used, as the third argument to the interception function, but the address of the object is only known at execution time, and, instead of the absolute address of the field, the field offset is encoded in the compiled code.

3.4.10 Writing to a non-`static` field

Table 3.10 lists the duration of an operation that stores the value of a local variable, initialized to zero, into an `int`eger field of an object whose meta-object may be non-`null`.

Table 3.10: Writing to a non-`static` field

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.3 e−7	3.2 e−8	5.4 e−8	4.5 e−8
KJG	5.3 e−7	9.1 e−8	1.0 e−7	8.5 e−8
JDK	3.6 e−7	2.9 e−7	5.5 e−7	4.6 e−7
KI-	3.4 e−6	1.8 e−6	2.3 e−6	1.9 e−6
KIG	3.6 e−6	2.0 e−6	4.7 e−6	2.6 e−6
KJN	1.7 e−4	6.8 e−5	1.1 e−4	9.1 e−5
KIN	4.0 e−4	1.4 e−4	2.1 e−4	1.9 e−4
KJM	2.7 e−3	8.6 e−4	1.9 e−3	1.3 e−3
KIM	1.7 e−2	5.0 e−3	7.0 e−3	6.7 e−3

One important difference between the `putfield` bytecode and the other field-related bytecodes is that the stack position of the object whose field is to be written to depends on the type of the field. For this reason, what used to be a single `switch` statement in the implementation of Kaffe has been split into two separate ones: first, we find out where in the stack the target object of the operation is located, so we can check whether it is reflective or not. If it is not, the original `switch` statement stores the top of the stack onto the object's field. Otherwise, a field write interception function is called, just like the `putstatic` does, except that the second argument is a pointer to the object, not to the class. The additional `switch` statement affects only the interpreter, because, in the JIT compiler, it is only evaluated at compile-time. Nevertheless, the discussions in the previous sections apply to this bytecode too.

3.4.11 Loading the `length` of an array

The time needed to load the `length` of a possibly reflective array of `int`, of `length` 1, into a local variable, is presented in Table 3.11.

Although the `length` of an array is not properly a field of the array object, the code produced by the original JIT compiler for the `arraylength` and the `getfield` bytecodes is identical, as the length of an array is stored in a fixed offset of the object that represents the array. On the interpreter, however, `arraylength` operation is much faster `getfield`, because the offset of the `length` is known at interpreter compile-time, it does not have to be looked up in a field structure.

Table 3.11: Loading the `length` of an array

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	7.0 e−8	4.3 e−8	6.0 e−8	5.0 e−8
KJG	1.3 e−7	9.6 e−8	1.6 e−7	1.3 e−7
JDK	3.7 e−7	2.7 e−7	5.4 e−7	4.5 e−7
KI-	1.7 e−6	1.1 e−6	1.2 e−6	9.7 e−7
KIG	1.9 e−6	1.2 e−6	1.2 e−6	1.1 e−6
KJN	1.8 e−4	6.6 e−5	1.1 e−4	8.9 e−5
KIN	3.9 e−4	1.4 e−4	2.1 e−4	1.8 e−4
KJM	3.1 e−3	7.8 e−4	1.5 e−3	9.6 e−4
KIM	1.2 e−2	3.8 e−3	5.4 e−3	4.8 e−3

As usual, we have added the meta-object existence test before the execution of the regular array length operation. If the array is found to be reflective, both engines just call a function that takes a pointer to the array object and returns the array length. This function tests again whether the array is reflective. If its meta-object has become `null`, the length of the array is returned immediately. Otherwise, an operation object is created and performed, then its result is returned. The JIT performance incurs in the register spilling and reloading overhead in this case too.

3.4.12 Loading an element of an array

Storing in a local variable the first element of the array used in the previous section takes the amount of time displayed in Table 3.12.

Table 3.12: Loading an element of an array

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.1 e−7	5.2 e−8	1.0 e−7	9.0 e−8
KJG	2.3 e−7	1.2 e−7	1.9 e−7	1.6 e−7
JDK	3.7 e−7	2.3 e−7	6.4 e−7	5.3 e−7
KI-	2.0 e−6	1.3 e−6	1.4 e−6	1.2 e−6
KIG	2.2 e−6	1.3 e−6	1.6 e−6	1.3 e−6
KJN	2.0 e−4	6.7 e−5	1.1 e−4	8.9 e−5
KIN	4.0 e−4	1.4 e−4	2.1 e−4	1.8 e−4
KJM	3.0 e−3	8.5 e−4	1.3 e−3	1.3 e−3
KIM	1.2 e−2	3.7 e−3	5.5 e−3	5.0 e−3

There are different array element load operations for each primitive type, and yet another for object types. However, their implementations are identical, except for the calculation of the offset from the beginning of the array and the actual element load instruction. Therefore, there is no need to time all possible array operations. We have probably selected an `int` array, accessed through the `iaload` bytecode, so as to save typing, since `int` is the shortest type name in Java. The fact that it fits exactly in the registers of the tested platforms has just made the results look worse, because there is no need for any conversion that might have reduced the relative overhead introduced by **Guaraná**.

We have introduced a meta-object existence test in all array load bytecodes *before* the array bound check, so that the meta-level can make arrays seem larger than they actually are. If a meta-object is associated with the array, interception occurs. In the interpreter case, a generic array load interception function is called. It takes, as arguments, a pointer to the array meta-object, a pointer to the array itself, the index of the array element to be loaded, and the stack slot where it should be stored. The JIT, on the other hand, uses specialized functions for each different type, that return the loaded values. Because of the additional basic blocks, register spilling and reloading becomes necessary.

3.4.13 Writing to an element of an array

The figures in Table 3.13 represent the time spent by an operation that stores the value of a zero-initialized local variable into the first element of the aforementioned array.

Table 3.13: Writing to an element of an array

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.0 e−7	5.8 e−8	9.3 e−8	7.7 e−8
KJG	6.0 e−7	1.2 e−7	1.5 e−7	1.3 e−7
JDK	4.2 e−7	3.1 e−7	6.5 e−7	5.3 e−7
KI-	2.0 e−6	1.2 e−6	1.4 e−6	1.1 e−6
KIG	2.2 e−6	1.3 e−6	1.4 e−6	1.2 e−6
KJN	1.7 e−4	6.5 e−5	1.1 e−4	9.7 e−5
KIN	3.9 e−4	1.4 e−4	2.1 e−4	1.8 e−4
KJM	2.4 e−3	8.7 e−4	1.3 e−3	1.3 e−3
KIM	1.4 e−2	4.0 e−3	5.8 e−3	5.3 e−3

The only difference between the array load and the array store bytecodes is the direction of the array element data. In this case, the interpreter calls a generic array store interception function that takes a pointer to the stack slot that contains the value to be stored, and the JIT compiler calls type-specific interception functions that take the value to be stored as an additional argument. Register management overhead due to additional basic blocks applies too.

3.4.14 Creating objects

Creating an object involves allocating memory for an object and invoking its constructor, that are two separate operations at bytecode level. Furthermore, if we were to run a test program to time this operation, it would inevitably be influenced by the cost of garbage collection. Since the garbage collector cannot be controlled reliably, because of different object sizes and increased size of JIT-generated code, we have decided not to run this test.

The first bytecode involved in the creation of an object is `new`, that allocates the amount of memory necessary for an object of a specified class and initializes all its fields with zeroes and `null`s, except its dispatch table, that is initialized to point to the dispatch table of its class. This bytecode is implemented as an invocation of a function that takes a pointer to a class and returns a pointer to the newly created object. We have modified this bytecode so that it calls an alternate function that takes an additional argument: a pointer to the creator of the new object. After this function calls the original object creation function, it calls a generic meta-configuration propagation one.

This propagation function checks whether the creator has a meta-object. If it does, it invokes method configure of that meta-object, then sets the new object's meta-configuration to the meta-object returned by this method. Afterwards, if the object's class has a non-`null` meta-configuration, a NewObject message is created and broadcast to the class' meta-configuration, by invoking method broadcast of the kernel of **Guaraná**.

If neither the creator nor the class are reflective, the reflection overhead in bytecode `new` is minimal, and it is the same for both the interpreter and the JIT compiler.

The second bytecode involved in object creation is `invokespecial`, used to invoke the new object's constructor. We have already presented the overhead introduced in this operation in Table 3.4.

3.4.15 Creating arrays

There are two different bytecodes for creating arrays: one that creates arrays of primitive types, and another that creates arrays of class types. The implementation of these bytecodes is very similar, and both were modified exactly like bytecode `new`: the array creation function calls gained an additional argument, a pointer to the array creator, and were changed so as to call alternate functions that supported this additional argument. After invoking the original array creation functions, they would call the generic meta-configuration propagation function. Since arrays do not have constructors, the additional overhead due to constructor invocation does not exist.

3.4.16 Creating arrays of arrays

The `multianewarray` bytecode creates multi-dimensional arrays as a single operation. This bytecode was modified in a slightly different manner: instead of creating an alternate function that would call the original multi-array creation one, we have modified the function itself, so that it would support an additional argument, a pointer to the array creator.

The rationale for this difference is that, just after creating the top-level array, the creator's meta-configuration must be propagated into this array, so that, when the sub-arrays are created, the meta-configuration of the container array is already set up to propagate into them.

3.4.17 Printing a `String`

As a first attempt to measure the overall impact of the introduction of reflection, we have measured how long it takes for the System.err object to print the `String` `''Hello world!''` and skip to the next line. The obtained times are listed in Table 3.14.

Table 3.14: Printing a `String`

Conf	`i586`	`i686`	`sparc ultra1`	`sparc ultra2`
`KJ-`	3.5 e−4	1.1 e−4	1.9 e−4	1.5 e−4
`KJG`	6.2 e−4	1.6 e−4	2.6 e−4	2.1 e−4
`JDK`	3.9 e−4	2.0 e−4	2.3 e−4	1.9 e−4
`KI-`	2.9 e−3	1.0 e−3	1.4 e−3	1.1 e−3
`KIG`	3.1 e−3	1.0 e−3	1.4 e−3	1.3 e−3
`KJN`	6.0 e−4	1.6 e−4	2.7 e−4	2.1 e−4
`KIN`	3.2 e−3	1.0 e−3	1.4 e−3	1.4 e−3
`KJM`	6.0 e−4	1.7 e−4	2.9 e−4	2.2 e−4
`KIM`	3.3 e−3	1.0 e−3	1.4 e−3	1.4 e−3

No objects are created in this operation, so no garbage collection takes place. Furthermore, since neither class System nor object System.err are reflective, no interception takes place.

3.4.18 Compiling a program

Timing the compilation of the test program with the various available interpreters has produced the figures in Table 3.15. We have not timed the executions with meta-objects, because, at this point, we are only interested in measuring the overall performance penalty introduced by the potential of intercepting operations.

Table 3.15: Compiling a program: total execution time

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	1.3 e+1	4.3 e+0	6.2 e+0	5.2 e+0
KJG	2.4 e+1	8.6 e+0	1.0 e+1	8.4 e+0
JDK	8.6 e+0	3.9 e+0	3.5 e+0	2.8 e+0
KI-	3.2 e+1	1.0 e+1	1.6 e+1	1.3 e+1
KIG	3.3 e+1	1.1 e+1	1.6 e+1	1.4 e+1

On short-running applications like this, most of the time is spent on virtual machine initialization and JIT compilation, not on running the application itself. The virtual machine start-up, for example, involves executing very large array initialization methods, whose JIT-compilation wastes a lot of memory and CPU cycles, because these methods are executed only once.

Although a complex program, involving several similar classes, is being compiled, Table 3.16 shows that more than 50% of the total time was spent on JIT-compiling Java Core classes and the Java compiler itself. Table 3.17 presents the differences between the total time and the JIT-compilation time, that represents the time spent on running the actual application. Hence, long running applications, that repeatedly run the same methods, should present a reflection overhead similar to the relative overhead of this table.

Table 3.16: Compiling a program: JIT compilation time

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	8.0 e+0	2.8 e+0	3.5 e+0	2.9 e+0
KJG	1.8 e+1	7.2 e+0	6.8 e+0	5.7 e+0

Table 3.17: Compiling a program: disregarding JIT-compilation time

Conf	i586	i686	sparc ultra1	sparc ultra2
KJ-	5.1 e+0	1.5 e+0	2.7 e+0	2.3 e+0
KJG	6.4 e+0	1.4 e+0	3.4 e+0	2.7 e+0

3.4.19 Overall analysis

In Table 3.18, we present the relative slow down caused by adding interception code to the Kaffe interpreter all tested platforms. Table 3.19 contains the corresponding data for the Kaffe JIT compiler. The listed figures are calculated from numbers with a higher precision than the ones presented in the previous tables, so that rounding of those values does not affect the figures in this table.

Table 3.18: Overall analysis (interpreter engine)

Table Number and Mnemonic	i586	i686	sparc ultra1	sparc ultra2
3.1: emptyloop	+9%	+8%	+1%	+1%
3.2: monitorenter/exit	+2%	+7%	+13%	+45%
3.3: invokestatic	−12%	+3%	−8%	+9%
3.4: invokespecial	−23%	+8%	+3%	+4%
3.5: invokevirtual	+18%	+7%	+6%	+9%
3.6: invokeinterface	−14%	+6%	−7%	+8%
3.7: getstatic	+2%	+8%	+1%	+31%
3.8: putstatic	−7%	+3%	+0%	+29%
3.9: getfield	−14%	+4%	−11%	+35%
3.10: putfield	+6%	+6%	+104%	+38%
3.11: arraylength	+13%	+9%	+5%	+11%
3.12: iaload	+11%	+7%	+15%	+15%
3.13: iastore	+12%	+8%	+4%	+4%
3.14: println	+7%	+0%	+1%	+17%
3.15: compile	+3%	+2%	+2%	+13%

The table only compares executions that do not involve meta-objects, because, when a meta-object intercepts an operation, the cost of the operation grows by some orders of magnitude. In fact, intercepting a simple operation involves dozens of method invocations, some of them implemented in native code that calls Java code. In addition to the fact that Kaffe interface for calling Java from native code is very slow, we should also consider that every intercepted operation causes the creation of an operation object and a result object, that must be garbage collected, and garbage collection is slow and unpredictable.

In certain combinations of platform and engine, an operation executes faster on **Guaraná** than on the corresponding combination without it. This is quite hard to explain, since **Guaraná** always executes at least as much code as Kaffe does. The tests have been verified so as to ensure that the results are correct, and the generation of the tables from the test runs is totally automated, so there is no place for human error. The better performance can be attributed to factors such as improved fast-RAM cache hit rate or code alignment issues (on x86).

Table 3.19: Overall analysis (JIT-compiler engine)

Table Number and Mnemonic	i586	i686	sparc ultra1	sparc ultra2
3.1: emptyloop	−0%	+3%	+0%	+0%
3.2: monitorenter/exit	+8%	+9%	+9%	+10%
3.3: invokestatic	+54%	+21%	+35%	+36%
3.4: invokespecial	+73%	+26%	+24%	+25%
3.5: invokevirtual	−7%	−56%	+6%	+6%
3.6: invokeinterface	+1%	+2%	+7%	+8%
3.7: getstatic	+86%	+98%	+181%	+182%
3.8: putstatic	+204%	+154%	+109%	+109%
3.9: getfield	+77%	+130%	+169%	+171%
3.10: putfield	+319%	+184%	+88%	+88%
3.11: arraylength	+86%	+125%	+159%	+160%
3.12: iaload	+101%	+131%	+86%	+71%
3.13: iastore	+501%	+107%	+62%	+62%
3.14: println	+75%	+54%	+40%	+42%
3.15: compile	+83%	+98%	+64%	+62%
3.16: compile-JIT	+121%	+154%	+96%	+94%
3.17: compile-diff	+24%	−5%	+24%	+21%

The overhead introduced by interception on the interpreter engine is mostly small, because the interpreter is usually orders of magnitude slower than the test for existence of a meta-object. The JIT, however, is severely affected by increased register pressure and additional register spilling and reloading. JIT-compilation costs have increased too, as our tests have shown, but they have only affected the last two tests, because we ensure that a method is JIT-compiled before we start timing its execution.

Although the interception code has introduced large penalties for invoking `static` and `private` methods, the most common kind of invocation (non-`final`) causes a very small overhead, and `interface` invocations are almost not affected at all. The bad results for all load and store operations on the JIT engines are expected, for the reasons already presented.

Fortunately, in object-oriented applications, field and array operations are usually intertwined with method invocations and object creations. Since the latter operations have a much smaller penalty, and they are one order of magnitude slower than the former ones, the net performance penalty may be acceptable, as the introduction of reflective capabilities may pay off.

3.5 Future optimizations

The reflection overhead on the interpreter is almost negligible, so there is very little need to worry about optimizing it any further. For the JIT code, there is little hope for similarly small overheads, though. One possible approach would be to implement all operations, even field and array ones, as invocations of dynamically generated JIT-compiled code. Then, instead of having to load the meta-object reference before performing an operation, an extended dispatch table would contain pointers to these JIT-generated functions, on non-reflective objects, and to interceptor functions, in the case of reflective objects.

Unfortunately, we do not think this solution would do very well: first, because we would have to look up the dispatch table before executing every single operation, and the virtual method invocation cost is currently much higher than non-virtual method invocation, so we would end up increasing the cost of most operations, instead of reducing it.

Furthermore, invoking a function requires saving most registers on some ABIs, but this is not required when contents of memory addresses are loaded directly, as field load operations are currently implemented. In fact, the way **Guaraná** is currently implemented means that, whenever a field or array operation is performed, registers must be saved because it *might* be necessary to invoke an interceptor function. A promising optimization involves not saving registers at all in case no interception is necessary, and modifying the interception code so that it leaves registers just like the original code would. This would decrease the cost of both branches, because they currently save all registers and mark them all as unused before they join to proceed to the next instruction. Furthermore, if the JIT compiler ever gets smarter about global register allocation, the additional branches introduced by **Guaraná** will not get it confused.

One of the reasons why actually intercepting an operation is slow is that it always involves creating two objects: the reified operation and the reified result. We might think of optimizing away the instantiation of these objects, by defining specialized interceptor (handle) methods for different kinds of operations and results. However, this would complicate the meta-object protocol without solving the problem, because then, instead of instantiating operations and results, it would be necessary to dynamically create instances of Method and Field.

However, it might be possible to optimize away the construction of a result object, if it is not requested. In the case of method invocations, we could also try to allocate a single chunk of memory, to contain both the operation object and the argument list, instead of allocating two chunks.

The most important optimization is in the native method invocation interface. Whenever a method is invoked from native code, its signature is parsed several times, so as to calculate its size, build an argument list, fill it in, then actually invoke the method. A lot of this work could be done in advance, and, in the JIT engine, it would be possible to generate specialized dispatchers, fragments of code that would take an argument list in a standard format and call the desired method with the arguments properly converted to the calling convention. In case of intercepting a method invocation, we could also have a pre-compiled interceptor, that would reify the invocation much faster than the current code.

3.6 Conclusions

The implementation of the reflective architecture of **Guaraná** required some modifications in a Java interpreter, but not in the Java programming language. Thus, 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 extend them.

Unfortunately, our implementation depends on a particular interpreter, but we can prove it is *impossible* to implement our meta-object protocol transparently in 100% Pure Java, due to limitations related with `native` methods and bytecode verification.

Our modifications have reduced the speed of the interpreter, but we believe the flexibility introduced by the reflective capabilities outweighs this inconvenience. Furthermore, the performance impact analysis has revealed the current hot spots in the interception mechanisms. We expect to reduce this impact by implementing the suggested optimizations.

Capítulo 4

Tutorial

Prólogo

O texto deste capítulo pretende ser uma introdução ao **Guaraná** para programadores Java que desejem utilizá-lo. Apresentamos exemplos básicos, que demonstram que tipos de interações entre objetos podem ser processadas pelo protocolo de meta-objetos do **Guaraná**. Em seguida, apresentamos algumas implementações de meta-objetos, demonstrando de que forma se podem utilizar operações, resultados e mensagens resultantes de interceptações ou de interações de meta-nível. Finalmente, abordamos tópicos mais avançados, tais como mecanismos de segurança e reconfiguração dinâmica. Oferecemos dicas para a implementação correta de alguns padrões de interação entre meta-objetos e seus respectivos objetos de nível base.

A ser publicado como relatório técnico IC-98-31.

Guaraná: A Tutorial

Alexandre Oliva   Luiz Eduardo Buzato
oliva@dcc.unicamp.br   buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

September 1998

Abstract

This text is a tutorial for people interested in using our Java^{TM}-based implementation of **Guaraná**, a reflective architecture that aims at flexibility, security and reuse of meta-level code. It shows what kind of operations can be intercepted with **Guaraná** and how meta-objects can monitor and modify base-level behavior. It also introduces composition of meta-objects, and discusses dynamic reconfiguration and management of meta-configurations. Several tricks and internal details of the implementation are exposed, through the use of numerous examples and detailed explanations.

4.1 Introduction

Guaraná is a reflective architecture whose meta-object protocol allows reuse of meta-level code through composition of meta-objects, in a simple, flexible and secure manner. It has been implemented by modifying an open-source Java[1] Virtual Machine [36], but **Guaraná** does not require any change to the Java^{TM} Programming Language [3, 26].

This text assumes some familiarity with the Java^{TM} Programming Language and the reflective architecture of **Guaraná**, as it contains several examples coded in Java that demonstrate how to use the Application Programming Interface (API) of **Guaraná**.

In Section 4.2, we describe **what kind** of base-level interactions can be intercepted by **Guaraná**. Section 4.3 considers the implementation of meta-objects, showing **how** to

*Islene Calciolari Garcia *recebeu auxílio da FAPESP através da bolsa 95/1983-8.*

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

enable and use these intercepted interactions. In Section 4.4, we discuss some details of the implementation and present some advanced programming tips. Finally, in Section 4.5, we summarize the topics presented.

4.2 Basics

Just like several other reflective architectures, **Guaraná**'s reflective capabilities are based on meta-objects intercepting interactions between objects. However, unlike other architectures, we have not modified the original programming language. Thus, there is no compile-time association of classes with meta-classes: the meta-level behavior of an object is orthogonal to its class. Meta-objects can be dynamically associated with objects.

Instead of introducing the reflective capabilities of **Guaraná** through the examination of a complex application, we have decided to use a very simple meta-object, called MetaLogger, to explain the use of the reflective mechanisms implemented by **Guaraná**. MetaLogger writes onto the console a descriptive log every time it is activated, due to interception or to other kinds of meta-level interaction. The format of these logs is going to be explained as they appear along the tutorial.

We provide several example programs, that are included in the latest releases of **Guaraná**. Line-numbered listings of the source files and outputs of their executions are provided. In the text, we are going to refer to source lines with numbers between curly braces, such as {1} or {3–5}, and to output lines with numbers between angle brackets, such as <4> or <7–10>.

4.2.1 Starting up

Let us take a look at a very basic example, that demonstrates how to associate an object with a meta-object. Method main of Program 1, that is invoked when the program is started as a Java application, creates an instance of class ConfBasic {7} and a MetaObject of class MetaLogger {8}. Then, it requests the kernel of **Guaraná**, represented by the class Guarana in package `BR.unicamp.Guarana`, to replace the `null` meta-object with the newly created one, in the meta-configuration of object o {9}. Note, in the output of this program, Output 1, that the reconfiguration has succeeded, since the MetaLogger was initialized <1>: method initialize is invoked just before a meta-object is associated with a base-level object.

The ugly string printed after the colon is the Java-standard String representation of the base-level object the meta-object was just associated with. The name of the class of the object is separated with an "`@`" sign from the memory address of the object, printed in hexadecimal notation. We are going to refer to this String with the term *object-id*.

After the reconfiguration, method meth is invoked {10}. This invocation is intercepted

Program 1 `ConfBasic.java`

```
1  public class ConfBasic {
2      int i = 3;
3      synchronized int meth(int j) {
4          return i + j;
5      }
6      public static void main(String[ ] argv) {
7          ConfBasic o = new ConfBasic();
8          BR.unicamp.Guarana.MetaObject mo = new MetaLogger();
9          BR.unicamp.Guarana.Guarana.reconfigure(o, null, mo);
10         o.i = o.meth(1);
11     }
12 }
```

```
1  Initialize: ConfBasic@10de80
2  Operation: ConfBasic@10de80.ConfBasic.meth(int 1)
3  Operation: ConfBasic@10de80.<monitor enter>
4  Result: return null
5  Operation: ConfBasic@10de80.ConfBasic.i
6  Result: return 3
7  Operation: ConfBasic@10de80.<monitor exit>
8  Result: return null
9  Result: return 4
10 Operation: ConfBasic@10de80.ConfBasic.i=4
11 Result: return null
```

Output 1: `guarana ConfBasic`

by the kernel of **Guaraná** and presented to the MetaLogger, that prints the object-id, the method-id of the method to be invoked and the argument list <2>. A *method-id* is defined as the name of the class where the method is defined, followed by a dot and the method name. The name of the class is included to avoid ambiguities that might arise if a superclass defined a method with the same name. The type of each argument is also printed, to remove potential ambiguities due to overloaded methods.

Since this method is `synchronized` {3}, just before it starts running, a <monitor enter> operation is intercepted <3> and performed. A MetaLogger is programmed to request the result of any operation it receives, and, even though monitor-related operations produce no useful result, a `null` result is presented anyway <4>, as a notice that the monitor operation was successful.

Since method meth adds the contents of field i with the value of the argument j {4}, it must read the value of field i. The MetaLogger receives this operation too, and prints the object-id and the field-id <5>. Analogously to the method-id, a field-id is defined as the name of the class where the field is declared, followed by a dot and the field name. The MetaLogger also prints the result of the operation <6>: the value with which field i was initialized {2}.

When the execution of method meth terminates, the lock associated with object o is implicitly released, so a <monitor exit> operation is performed <7>, its result (`null`) is printed <8>, and so is the result of the execution of method meth <9>.

Finally, this result of the method is assigned to field i of object o {10}. The MetaLogger represents the non-`static` field assignment operation as the object-id, the field-id, an assignment operator and the assigned value <10>. Since assignments produce no useful result, the result of any such operation is `null` <11>.

4.2.2 Intercepting array operations

In addition to method invocations, field accesses and monitor operations, **Guaraná** can intercept operations on arrays, such as `length`, element read and element write. Program 2 presents a simple example of such interceptions; the example is accompanied by Output 2.

Program 2 `ArrayBasic.java`

```
 1  public class ArrayBasic {
 2      public static void main(String[ ] argv) {
 3          Object[ ] array = new Object[3];
 4          BR.unicamp.Guarana.MetaObject mo = new MetaLogger();
 5          BR.unicamp.Guarana.Guarana.reconfigure(array, null, mo);
 6          array[0] = new Integer(array.length);
 7          array[1] = array[0].toString();
 8          array[2] = array.getClass().getClass();
 9      }
10  }
```

First, method main creates an array of 3 references to Objects {3}, then a MetaLogger {4}. Since any Java array is also an Object, it can be given a meta-configuration. The program instructs the kernel of **Guaraná** to associate the MetaLogger with the array of Objects {5}. The association is successful <1>. Note that the Java-standard String representation of an array of a class type is composed by a left square bracket, a capital "L", the name of the class and a semicolon.

```
 1  Initialize: [Ljava.lang.Object;@115ec8
 2  Operation: [Ljava.lang.Object;@115ec8.length
 3  Result: return 3
 4  Operation: [Ljava.lang.Object;@115ec8[0]=java.lang.Integer@1b9b68
 5  Result: return null
 6  Operation: [Ljava.lang.Object;@115ec8[0]
 7  Result: return java.lang.Integer@1b9b68
 8  Operation: [Ljava.lang.Object;@115ec8[1]=java.lang.String@1ec450
 9  Result: return null
10  Operation: [Ljava.lang.Object;@115ec8.java.lang.Object.getClass()
11  Result: return java.lang.Class@1ae258
12  Operation: [Ljava.lang.Object;@115ec8[2]=java.lang.Class@31a58
13  Result: return null
```

Output 2: `guarana ArrayBasic`

Next, the program obtains the `length` of the array, to compute the first argument to be passed to the constructor of the `new` Integer {6}; this operation <2> and its result <3> are intercepted (by the kernel of **Guaraná**) and printed (by the MetaLogger). The Integer object is assigned to element 0 of the array {6}. This assignment is intercepted and printed <4>, as is its `null` result <5>. The representation of an array element-id is intuitive, albeit visually unpleasant: the array object-id is followed by the index of the array element between square brackets. For an array assignment operation, the MetaLogger will print, after the element-id, an assignment operator and the assigned value; in this case, an object-id.

Afterwards, method main reads the value just stored in array[0] <6–7>, invokes method toString of the returned Integer {7} (not intercepted, because the Integer was not associated with a meta-object), and assigns the returned String to element 1 of the array <8–9>.

Finally, the program invokes method getClass of the array object <10–11>, then invokes method getClass of the returned Class object {8}. Note that the second invocation of getClass is not intercepted, because the Class object that represents the class array of Objects has a `null` meta-configuration. The result of this second invocation is stored in element 2 of the array <12–13>. Observe that the Class reference returned as the result of the first invocation of getClass <11> is different from the one stored in array[2] <12>; while the former is the Object that reprents the class Object[ ], the latter is the Object that reprents the class Class itself.

4.2.3 Intercepting class operations

Class (`static`) operations will be intercepted by the meta-configuration of the Class object that represents the class, as in Program 3. First, the program instantiates ClassBasic {5}, and this instance will remain with a `null` meta-configuration. Then, it creates a MetaLogger {6} and obtains a reference to the Class object that represents the class ClassBasic {7}, using the class pseudo-field notation introduced in Java 1.1. In line {8}, it installs the MetaLogger as the primary meta-object of class ClassBasic, i.e., of the Class object that represents it. Line <1> of Output 3 shows the association was successful.

Program 3 `ClassBasic.java`

```
 1  public class ClassBasic {
 2      static void doNothing() {}
 3      static synchronized void doNothing(boolean b) {}
 4      public static void main(String[ ] argv) {
 5          ClassBasic o = new ClassBasic();
 6          MetaLogger mo = new MetaLogger();
 7          Class c = ClassBasic.class;
 8          BR.unicamp.Guarana.Guarana.reconfigure(c, null, mo);
 9          c.hashCode();
10          o.hashCode();
11          doNothing();
12          o.doNothing(true);      // calls static method
13      }
14  }
```

```
 1  Initialize: java.lang.Class@dda58
 2  Operation: java.lang.Class@dda58.java.lang.Object.hashCode()
 3  Result: return 907864
 4  Operation: ClassBasic.doNothing()
 5  Result: return null
 6  Operation: ClassBasic.doNothing(boolean true)
 7  Operation: java.lang.Class@dda58.<monitor enter>
 8  Result: return null
 9  Operation: java.lang.Class@dda58.<monitor exit>
10  Result: return null
11  Result: return null
```

Output 3: `guarana ClassBasic`

Invocations of methods of the Class object {9} are intercepted and presented to the MetaLogger <2–3>. Note, however, that the meta-configuration of the class does not affect the interception of operations on its instances: their meta-configurations are unrelated.

Therefore, a method invocation of an instance of a reflective class (that is, a class whose meta-configuration is not empty) would only be intercepted if the instance itself were reflective. In the example, Object o is not reflective, so the second invocation of hashCode is not intercepted {10}.

Invocations of `static` methods, operations on `static` fields, and synchronization operations on a class (for instance, invoking a `static synchronized` method) are also presented to the meta-object associated with the class represented by the Class object. For example, the invocation of a `static` method {11} is intercepted and presented to the class meta-configuration <4–5>. Even if the invocation expression appears to refer to a non-`static` method, as in line {12}, if compile-time overload resolution selects a `static` method, the operation is intercepted and presented to the meta-object of the class <6>, as is its result <11>. In this case, because the selected method is `synchronized`, monitor enter and exit operations are also intercepted <7–10>.

4.2.4 Meta-configuration propagation

In the previous examples, meta- and base-level code are intertwined, that is, code that modifies meta-configurations and base-level code appear together, in the same class. This clearly goes against the separation of concerns that can be attained through reflection. Fortunately, **Guaraná** provides mechanisms that allow a complete separation of the base and the meta application.

The meta application starts first, and sets up meta-configurations for classes and objects of the base application it is programmed to control. Then, it starts the base application. From then on, the meta application will only regain control by intercepting operations addressed to base-level classes or objects associated with meta-objects it has installed.

The MetaLogger class, for example, can be used as a meta application, because it provides a method main that creates a MetaLogger and associates it with a class specified as its first command-line argument. Then, it invokes method main of that class, passing to it the remaining command-line arguments.

Program 4, for example, is a simple non-reflective application, that can be made reflective by starting it as specified in the caption of Output 4. Method main of class MetaLogger, the meta application, associates a MetaLogger with class PropagBasic <1>, then invokes method main of that class <2>, just like the Java interpreter would do if it had been started as "`java PropagBasic`".

The meta application can configure the base application so as to determine the meta-configurations of dynamically created objects, due to the ability to intercept object creation provided by **Guaraná**. Whenever a class is instantiated, the primary meta-object of the creator is requested to provide a primary meta-object for the new object, before the object is constructed. If the object is created in a static method, its creator is defined as the class that contains the static method; otherwise, the creator is the object whose non-`static` method

Program 4 `PropagBasic.java`

```
1 public class PropagBasic {
2     public static void main(String[ ] argv) {
3         new PropagBasic(true);
4     }
5     PropagBasic(boolean another) {
6         if (another)
7             new PropagBasic(false);
8     }
9 }
```

```
 1 Initialize: java.lang.Class@1b46d8
 2 Operation: PropagBasic.main([Ljava.lang.String; [Ljava.lang.String;@1b9fa0)
 3 Configure PropagBasic@134c28 based on java.lang.Class@1b46d8: propagated
 4 Initialize: PropagBasic@134c28
 5 Message: BR.unicamp.Guarana.NewObject@1f8410 for java.lang.Class@1b46d8
 6 Operation: PropagBasic@134c28.PropagBasic(boolean true)
 7 Operation: PropagBasic@134c28.java.lang.Object()
 8 Result: return null
 9 Configure PropagBasic@134c78 based on PropagBasic@134c28: propagated
10 Initialize: PropagBasic@134c78
11 Message: BR.unicamp.Guarana.NewObject@1f8590 for java.lang.Class@1b46d8
12 Operation: PropagBasic@134c78.PropagBasic(boolean false)
13 Operation: PropagBasic@134c78.java.lang.Object()
14 Result: return null
15 Result: return null
16 Result: return null
17 Result: return null
```

Output 4: `guarana MetaLogger PropagBasic`

was being executed.

In line {3}, a `static` method of class PropagBasic creates a new object, so the meta-object of this class is requested to configure it <3>. Since MetaLoggers are programmed to *propagate* into meta-configurations of new objects, the new object is associated with a MetaLogger too <4>. Although it cannot be deduced from the output, no additional MetaLogger has to be created in this case. Since a single instance of MetaLogger can handle multiple base-level objects, the MetaLogger just installs itself in the meta-configuration of the new object.

After the meta-configuration of a new object is established, the meta-configuration of the class of the new object is informed of the instantiation, so that it can try to affect the meta-configuration of the object. The mechanism used to inform the meta-configuration of the class is one example of use of a general inter-meta-object communication facility that is part of the meta-object protocol of **Guaraná**: instances of classes that implement `interface` Message can be broadcast by the components of the meta-configuration of any given object. An instance of the class NewObject is thus broadcast to the meta-configuration of the class PropagBasic, to notify the creation of an instance thereof <5>. When a MetaLogger is presented a Message, it will always print the the String representation of the Message (by default, its object-id), the word "`for`" and the base-level object to whose meta-configuration the Message was broadcast.

Only after the broadcast terminates, the constructor of the new object is invoked. It is intercepted by the object's meta-configuration <6>, as is the implicit invocation of the constructor of the base class <7> that, like any other constructor invocation, returns `null` <8>.

Since the constructor is invoked with a `true` argument {3}, the test in line {6} results `true`, so a new PropagBasic object will be created {7}. Once again, the meta-object of the creator (the first PropagBasic object) is implicitly requested to provide a meta-configuration for the new object <9>, so the MetaLogger propagates itself <10>. Then, a NewObject message is broadcast to the meta-configuration of the class of the new object <11>, and its constructors are invoked <12–13>.

Finally, the outstanding invocations return: first, the two constructors of the second object <14–15>, then the pending constructor of the first object <16>, called in <6>, and finally the method main <17>, called in <2>.

4.3 Intermediate

Up to this point, we have covered almost every kind of interaction between the kernel of **Guaraná** and meta-objects associated with base objects. From now on, we are going to cover more complex interaction patterns, such as dynamic reconfiguration and meta-object composition. Then, we are going to present real implementations of meta-objects.

4.3.1 Dynamic reconfiguration

We have already introduced the method reconfigure, provided by the kernel of **Guaraná**. However, there are some details that have yet to be presented. Program 5 and Output 5 uncover such details.

In this example, three MetaLoggers are created {6–8}. In order to identify the output produced by each MetaLogger, each one is given a prefix, that will be inserted in the beginning of every line it produces. The first of these MetaLoggers, named cl, is associated with class

Program 5 `Reconfigure.java`

```java
import BR.unicamp.Guarana.Guarana;
class ReconfigureBase {}
public class Reconfigure extends ReconfigureBase {
    public static void main(String[ ] argv) {
        final MetaLogger
            cl = new MetaLogger().setPrefix("cl:  "),
            a = new MetaLogger().setPrefix("a:  "),
            b = new MetaLogger().setPrefix("b:  ");
        Guarana.reconfigure(ReconfigureBase.class, null, cl);
        final Reconfigure o = new Reconfigure();
        Guarana.reconfigure(o, null, a);     // cl lets a become primary
        Guarana.reconfigure(o, null, b);     // a replaced with b
        Guarana.reconfigure(o, b, a);     // b replaced with a
        Guarana.reconfigure(o, b, a);     // ignored by a
        Guarana.reconfigure(o, null, null);     // b replaced with null
        Guarana.reconfigure(o, a, b);     // ignored by the kernel
    }
}
```

```
cl: Initialize: java.lang.Class@ddab8
cl: Message: BR.unicamp.Guarana.InstanceReconfigure@1b9cb8 for java.lang.Cla
ss@ddab8
cl: Operation: java.lang.Class@ddab8.java.lang.Class.getSuperclass()
cl: Result: return java.lang.Class@30438
a: Initialize: Reconfigure@134c08
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> MetaLogger@1b8938
b: Initialize: Reconfigure@134c08
a: Release: Reconfigure@134c08
b: Reconfigure Reconfigure@134c08: MetaLogger@1b8938 -> MetaLogger@1b88f0
a: Initialize: Reconfigure@134c08
b: Release: Reconfigure@134c08
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> MetaLogger@1b88f0
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> null
a: Release: Reconfigure@134c08
```

Output 5: `guarana Reconfigure`

ReconfigureBase {9}<1>, a base class of Reconfigure {3}.

When the program instantiates class Reconfigure {10}, unlike in the previous example, neither a configure request is issued nor a NewObject message is broadcast. These meta-level operations do not take place because class Reconfigure, that is both the creator of the new object and its class, has an empty meta-configuration, in spite of its superclass having a non-empty one.

When the program reconfigures object o so that its meta-object becomes a {11}, a new kind of message is broadcast <2>. Whenever an object whose meta-configuration is `null` is reconfigured, an InstanceReconfigure message is broadcast to the meta-configuration of the class of the object, then to the meta-configuration of its base class, and so on, up to the root of the inheritance hierarchy. Thus, meta-objects that belong to meta-configurations of classes may modify reconfiguration requests issued to its non-reflective instances. Note that method getSuperclass is invoked on class ReconfigureBase <3–4>: this is the kernel of **Guaraná** moving up in the inheritance hierarchy.

Since, in this example, no meta-object modified the reconfiguration request, meta-object a becomes the primary meta-object of o <5>. But another reconfiguration is soon requested {12}. Since the base object is reflective already, the meta-configurations of its classes do not participate in the reconfiguration process, only its primary meta-object does.

The kernel of **Guaraná** replaces the `null` argument in the reconfiguration request with a reference to the current primary meta-object before delegating the request to it, so the MetaLogger a prints a reference to itself on the left of the arrow in <6>, and a reference to b on the right of the arrow. Although it might have ignored the request or modified it, we can see it has accepted to be replaced, because b is initialized <7>, then a is requested to release the object <8>. A release invocation takes place just after a meta-object is removed from the meta-configuration of an object.

Instead of specifying `null`, we may name any particular meta-object as the second argument of a reconfigure request. In line {13}, a meta-object known to be the primary one is specified <9>. However, in **Guaraná**, a meta-object (called *composer*) may delegate operations to others, so the second argument in the reconfiguration request can be used to specify other meta-objects in the composition hierarchy.

When presented a reconfiguration request, a meta-object is supposed to return a meta-object to replace it. In this example, b finds itself in the second argument <9>, and accepts to be replaced with a, the third argument <10–11>. If it did not intend to be replaced, it should have returned a reference to itself, even if the second argument of the reconfiguration request were not a reference to itself—the kernel of **Guaraná** does not care whether the references compare equal, it will just replace a meta-object with the value it returns. Furthermore, although the third argument is supposed to be the meta-object that intends to replace the one passed as the second argument, it may be completely disregarded by the existing meta-objects, or used just as a hint to create the components of the new meta-configuration.

The program continues in line {14}, by repeating the reconfiguration request, but now it is a that listens to it <12>. Since it does not match the second argument, it ignores the reconfiguration request, returning itself. No initialization nor release takes place.

In line {15}, once again, the second argument to reconfigure is `null`, so the kernel of **Guaraná** passes the primary meta-object as the second argument. Thus, MetaLogger a matches the request <13>, and accepts to be replaced with a `null` meta-object. No initialization takes place—it is pointless to `initialize` a `null` meta-object—, but the previous meta-object is released <14>, and object o has become non-reflective again.

The final invocation of reconfigure {16} is ignored by the kernel of **Guaraná**, because a `null` meta-configuration would only match a request that had `null` as the second argument.

4.3.2 Composing meta-objects

The meta-object protocol of **Guaraná** was designed so as to make it possible to create a meta-object that interacts with other meta-objects just like the kernel of **Guaraná** would, providing them with operations, results, messages, initialization and release requests in a way that these meta-objects may believe they are directly called by the kernel of **Guaraná**.

This special kind of meta-object is called a composer. The implementation of **Guaraná** includes a useful implementation of composer: the SequentialComposer. Essentially, it delegates operations to an array of meta-objects, and delegates results to these meta-objects in reverse order.

Program 6 creates an instance of class Composer {4}, two MetaLoggers {6–7}, an array containing these MetaLoggers {8} and a SequentialComposer that delegates to them {9}. As you may notice in Output 6, one of the MetaLoggers prefixes all lines it outputs with "`a: `", and the other, with "`b: `". Lines <1–2>, for example, are printed when the composer is associated with the Compose object {10}. When method initialize of the composer is invoked, it delegates the initialization request to the meta-objects contained in the array passed to its constructor.

When method another is invoked on the Compose object {11}, it is intercepted and presented to the composer, that delegates first to meta-object a <3>, then to meta-object b <4>, and finally tells the kernel of **Guaraná** to perform the Operation.

Method another just creates a new Compose object {14}, but this involves propagation of meta-configuration. The SequentialComposer requests its meta-objects to configure the new object <5–6>, then it creates a new SequentialComposer that delegates to the meta-objects selected for the new object <7–8>.

Note that the invocations of the constructors of the new object are intercepted and delegated to both meta-objects <9–12>, then the results of the constructor invocations

Program 6 `Compose.java`

```
import BR.unicamp.Guarana.*;
public class Compose {
    public static void main(String[ ] argv) {
        Compose o = new Compose();
        MetaObject
            a = new MetaLogger().setPrefix("a:  "),
            b = new MetaLogger().setPrefix("b:  "),
            mos[ ] = { a, b },
            mo = new SequentialComposer(mos);
        Guarana.reconfigure(o, null, mo);
        o.another();
    }
    Compose another() {
        return new Compose();
    }
}
```

```
a: Initialize: Compose@134608
b: Initialize: Compose@134608
a: Operation: Compose@134608.Compose.another()
b: Operation: Compose@134608.Compose.another()
a: Configure Compose@134cc8 based on Compose@134608: propagated
b: Configure Compose@134cc8 based on Compose@134608: propagated
a: Initialize: Compose@134cc8
b: Initialize: Compose@134cc8
a: Operation: Compose@134cc8.Compose()
b: Operation: Compose@134cc8.Compose()
a: Operation: Compose@134cc8.java.lang.Object()
b: Operation: Compose@134cc8.java.lang.Object()
b: Result: return null
a: Result: return null
b: Result: return null
a: Result: return null
b: Result: return Compose@134cc8
a: Result: return Compose@134cc8
```

Output 6: `guarana Compose`

(null) are presented to them, but first to b <13,15>, then to a <14,16>. This inversion is a particular characteristic of SequentialComposer; other composers might do it differently.

Finally, the result of the invocation of method another is presented to the original composer, that delegates it to the MetaLoggers <17–18>.

4.3.3 Modifying results

In the next example, we are going to show an actual implementation of MetaObject. Program 7 shows how a meta-object is supposed to handle intercepted operations and results, and how it can modify the results of intercepted operations, after its execution, or even prevent their execution.

First, let us take an overview of the presented code. Two static variables are defined, namely, hashCode and toString {3}. Variable hashCode is initialized {5–7} by searching class Object for a method named "hashCode", that does not take any argument—so the array of classes that specify its argument list has length zero. Method getDeclaredMethod of class Class returns an instance of class Method that represents this searched method. Variable toString is initialized similarly {8–10}. These variables will ease the verification of whether operations correspond to invocations of the methods they represent.

The first method handle {12–19} is invoked by the kernel of **Guaraná** (or by a composer) before an operation (the first argument) is delivered to its target object (the second one). Our implementation will check whether the operation is an invocation of any of the two selected methods. Although it would be syntactically correct to compare the methods using operator ==, the program would be semantically wrong, because there may be multiple Method objects associated with a single method, and this operator only compares *references*. Thus, the comparison must be performed using method equals, that compares the actual *values* of the objects, that is, it yields true if, and only if, two Method objects correspond to the same method of the same class. Note that our implementation does not test whether the operation is a method invocation. Nevertheless, it is correct, because method getMethod of class Operation returns null if the operation is not a method invocation, causing the comparison performed by method equals to yield false. However, if the program invoked method equals of object meth, passing hashCode or toString as arguments, it would be incorrect: a NullPointerException would be raised for operations other than method invocations, because meth would be null.

If the intercepted operation is an invocation of method hashCode {14}, the MetaObject will produce a result for the operation {15}, so that the operation will never be delivered to the target object for execution. Otherwise, if it is an invocation of method toString {16}, the MetaObject will request to be presented, and to possibly modify, the result of the operation {17}, after it is executed. For any other operation, the meta-object indicates it is not interested in the result {18}.

Program 7 `ModifyResult.java`

```
import BR.unicamp.Guarana.*;
public class ModifyResult extends MetaObject {
    final static java.lang.reflect.Method hashCode, toString;
    static {
        try { hashCode =
                Object.class.getDeclaredMethod("hashCode", new Class[0]);
        } catch (NoSuchMethodException e) { hashCode = null; }
        try { toString =
                Object.class.getDeclaredMethod("toString", new Class[0]);
        } catch (NoSuchMethodException e) { toString = null; }
    }
    public Result handle(final Operation op, final Object ob) {
        final java.lang.reflect.Method meth = op.getMethod();
        if (hashCode.equals(meth))
            return Result.returnInt(0, op);     // make it return 0
        if (toString.equals(meth))
            return Result.modifyResult;     // asks for permission to modify it
        return Result.noResult;     // not interested in the result
    }
    public Result handle(final Result res, final Object ob) {
        Operation op = res.getOperation();
        if (toString.equals(op.getMethod()))
            return Result.returnObject("modified "     // replace result
                                       + res.getObjectValue(), op);
        return null;
    }
    public static void main(String[ ] argv) {
        Object o = new Object();
        MetaObject[ ] mos = {
            new MetaLogger().setPrefix("a:  "),
            new ModifyResult(),
            new MetaLogger().setPrefix("b:  ") };
        Guarana.reconfigure(o, null, new SequentialComposer(mos));
        System.out.println(o);
    }
}
```

The second method handle {20–26} is invoked after a result is produced for an operation, either by its execution or by another meta-object. Our implementation just checks whether the Operation the Result refers to {21} is an invocation of method toString {22}, in which case it will return a new Result object, containing a modified String {23–24}. Otherwise, it returns `null`, what is equivalent to returning the Result it was presented.

The execution of method main {27–35} produces Output 7. First, it creates an Object {28} and associate it with a SequentialComposer {33} that delegates to a MetaLogger {30}, an instance of our implementation of meta-object {31} and another MetaLogger {32}. We may notice the configuration was successful from the initialization messages printed by the MetaLoggers <1–2>.

```
 1  a: Initialize: java.lang.Object@134898
 2  b: Initialize: java.lang.Object@134898
 3  a: Operation: java.lang.Object@134898.java.lang.Object.toString()
 4  b: Operation: java.lang.Object@134898.java.lang.Object.toString()
 5  a: Configure java.lang.StringBuffer@1f0f30 based on java.lang.Object@134898:
      not propagated
 6  b: Configure java.lang.StringBuffer@1f0f30 based on java.lang.Object@134898:
      not propagated
 7  a: Operation: java.lang.Object@134898.java.lang.Object.getClass()
 8  b: Operation: java.lang.Object@134898.java.lang.Object.getClass()
 9  b: Result: return java.lang.Class@30438
10  a: Result: return java.lang.Class@30438
11  a: Operation: java.lang.Object@134898.java.lang.Object.hashCode()
12  a: Result: return 0
13  b: Result: return java.lang.String@1f1d10
14  a: Result: return java.lang.String@1f1f50
15  modified java.lang.Object@0
```

Output 7: `guarana ModifyResult`

Last thing method main does is to print the Object it has created to the standard output. This requires representing the Object as a String, so method println invokes toString, as logged by the MetaLoggers <3–4>. Between <3> and <4>, the ModifyResult meta-object was prrequested to handle the operation, but it produces no visible output.

The standard implementation of method toString concatenates, in a StringBuffer, the name of the class of the object with an "@" sign and an hexadecimal representation of the hash code of the object. MetaLoggers do not propagate into meta-configurations of StringBuffers to avoid excessive noise <5–6>, and the standard implementation of method configure in class MetaObject does not propagate: it just returns `null`. The SequentialComposer notices that none of its meta-objects propagated into the meta-configuration of the new object, and neither does it, so the StringBuffer gains an empty meta-configuration.

The method toString invokes getClass to find out what class the object belongs to <7–10>, then invokes getName on this class, but does not produce any output, because the class of the object, java.lang.Object, has an empty meta-configuration.

Then, the method toString obtains the hash code of the target object, by invoking method hashCode <11>. When the ModifyResult meta-object is presented this operation {14}, it provides a zero result for it {15}. Thus, meta-object b never sees the operation; the result is presented to a <12> and returned as if the operation had yielded it.

Finally, method toString returns the String representation of the StringBuffer that was used to concatenate the class name, the "@" sign and the hash code of the object. MetaLogger b is presented this String as the result of the operation first <13>, then the ModifyResult meta-object is. Instead of letting the result reach a unmodified, the ModifyResult meta-object notices the original operation was an invocation of method toString, and concatenates the string "`modified `" to the original result. Note that the String object presented as result for a <14> is not the same that was presented to b <13>.

Since both a and the SequentialComposer return the modified result, the modified String is printed by method println: it contains the "`modified `" prefix, and the hash code after the "@" sign is zero <15>.

4.3.4 Modifying operations

In addition to modifying results of operations, meta-objects can also create operations from the meta level, as does Program 8. The subclass of MetaObject implemented in this example assumes its instances will be associated with instances of the class defined in Program 9.

Like in the previous example, a few `static` variables are defined in ModifyOperation: toString {3} denotes method toString of class Object {6–8}, whereas callSuper {4} denotes the field callSuper of class ModifyOperationBase {9–11}.

This example introduces the class OperationFactory: it is used to create Operations addressed to base-level objects from the meta level. If a meta-object intends to create arbitrary operations to objects it reflects upon, it must save the OperationFactory it is initialized with, as does method initialize {14–15}.

Operation factories allow meta-objects to violate access control, giving them privileged access even to private fields and methods of base-level objects. Furthermore, using operation factories, it is possible to create and perform synchronization operations such as entering and leaving an object's monitor. Another ability provided by operation factories, explored in our example, is to modify overload resolution and dynamic dispatching mechanisms.

Method handle {16–28} prints a description of every operation it is requested to handle, just like the corresponding method of class MetaLogger does. Note that method toString

Program 8 `ModifyOperation.java`

```
import BR.unicamp.Guarana.*;
public class ModifyOperation extends MetaObject {
    static final java.lang.reflect.Method toString;
    static final java.lang.reflect.Field callSuper;
    static {
        try { toString =
                Object.class.getDeclaredMethod("toString", new Class[0]);
        } catch (NoSuchMethodException e) { toString = null; }
        try { callSuper =
                ModifyOperationBase.class.getDeclaredField("callSuper");
        } catch (NoSuchFieldException e) { callSuper = null; }
    }
    OperationFactory opf = null;
    public void initialize(final OperationFactory opf, final Object o)
    { this.opf = opf; }
    public Result handle(final Operation op, final Object ob) {
        System.out.println("Operation:  " + op);
        if (op.isMethodInvocation()) try {
            final java.lang.reflect.Method m = op.getMethod();
            if (!toString.equals(m) && m.getName().equals("toString")
                && opf.read(callSuper).perform().getBooleanValue()) {
                Operation newOp = opf.invoke(toString, new Object[0], op);
                System.out.println("Replaced with:  " + newOp);
                return Result.operation(newOp, Result.inspectResultMode);
            }
        } catch (IllegalAccessException e) { }
        return Result.inspectResult;
    }
    public Result handle(final Result res, final Object ob)
    { System.out.println("Result:  " + res); return null; }
    public static void main(String[ ] argv) {
        final Object oFalse = new ModifyOperationBase(false);
        Guarana.reconfigure(oFalse, null, new ModifyOperation());
        System.out.println("oFalse:  " + oFalse);
        final Object oTrue = new ModifyOperationBase(true);
        Guarana.reconfigure(oTrue, null, new ModifyOperation());
        System.out.println("oTrue:  " + oTrue);
    }
}
```

Program 9 `ModifyOperationBase.java`

```
import BR.unicamp.Guarana.*;
class ModifyOperationBase {
    final private boolean callSuper;
    public ModifyOperationBase(final boolean callSuper) {
        this.callSuper = callSuper;
    }
    public String toString() { return "derived method was called"; }
}
```

of class Operation does not invoke any method of the base-level object. If it did, these interactions would have to be intercepted, possibly leading to infinite recursion. Instead of calling method toString of the base-level Object, it calls an alternate method of the kernel of **Guaraná**, that emulates the execution of the former, without any interceptable interaction with the Object.

After printing a description of the operation, method handle checks whether the operation is a method invocation {18} of a method {19} named toString, but not the one implemented in class Object {20}, and finally tests whether field callSuper of the base-level object is `true` or `false` {21}.

This last condition illustrates one of the possible uses of operation factories: to create operations from the meta level and perform them. In this case, the meta-object uses its OperationFactory to create an Operation that reads the value of field callSuper, then performs it. Note that, although field callSuper is `private`, the operation can be created and performed, due to the fact that an operation factory gives privileged access to the object it refers to. Also note that the target object of the operation is not specified: an operation factory is associated with a base-level object when it is instantiated, and it can only create operations addressed to that object.

After the field read operation is performed, its result is returned in a Result object, so method getBooleanValue must be used to obtain its value. If it yields `false`, so does the whole condition, and method handle terminates returning Result.inspectResult {27}. Otherwise, we may observe the other use for an operation factory: to replace an operation that is being handled.

In line {22}, the operation factory is requested to create an operation that will invoke method toString of class Object, without any arguments—this is why the array of Objects is created with length zero. But note that a third argument is passed to invoke: the operation currently being handled.

When an operation is passed as the last argument to a method of an operation factory, the new operation will be a replacement operation, that is, it may be used to replace the original operation. But it will only effectively replace the current operation if the method handle returns a Result object containing the new Operation {24}. Such a Result object may also contain a request to inspect (or modify) the result of the operation, after it is performed.

The other method handle {29–30} just prints every result it is presented, almost exactly like the corresponding method of MetaLogger does.

```
 1  Operation: ModifyOperationBase@1b9328.ModifyOperationBase.toString()
 2  Operation: ModifyOperationBase@1b9328.ModifyOperationBase.callSuper
 3  Result: return false
 4  Result: return java.lang.String@1be830
 5  oFalse: derived method was called
 6  Operation: ModifyOperationBase@1e0350.ModifyOperationBase.toString()
 7  Operation: ModifyOperationBase@1e0350.ModifyOperationBase.callSuper
 8  Result: return true
 9  Replaced with: ModifyOperationBase@1e0350.java.lang.Object.toString()
10  Operation: ModifyOperationBase@1e0350.java.lang.Object.getClass()
11  Result: return java.lang.Class@1b2498
12  Operation: ModifyOperationBase@1e0350.java.lang.Object.hashCode()
13  Result: return 1966928
14  Result: return java.lang.String@1eb510
15  oTrue: ModifyOperationBase@1e0350
```

Output 8: `guarana ModifyOperation`

Output 8 presents the output produced by running method main {31–38}. First, it creates an instance of ModifyOperationBase {32} with a `false` callSuper {3} (in Program 9), associates it with an instance of a ModifyOperation meta-object {33}, then prints a String representation of it preceded by "`oFalse: `" {34}.

In order to obtain the String representation of the Object, method toString is invoked, and dynamic dispatching selects the implementation {7} in class ModifyOperationBase <1>. The meta-object notices that a method named toString was invoked {20}, and creates an operation to read field callSuper {21}. This operation is intercepted <2>, despite the fact that it was created in the meta level. The result is presented to the meta-object too <3>. Since callSuper is `false`, the test performed by the meta-object {20–21} fails, so it lets the operation pass unchanged {27}, and it returns the string hard-coded in method toString {7} of class ModifyOperationBase <4>, as shows the line printed {34} by method main <5>.

When an instance of ModifyOperationBase is created with a `true` callSuper {35}, however, the meta-object behaves differently. Dynamic dispatching selects the derived implementation <6>, but now, the meta-object finds callSuper to be true <7–8>, so it replaces the invocation of method toString of class ModifyOperationBase with an invocation of method toString of class Object {22–24}<9>. No further dynamic dispatching takes place, so the standard implementation of method toString is really executed, as we can notice by the methods this execution invokes <10–13>. Finally, method toString returns <14>, and method main prints {37} the standard String representation of object oTrue <15>.

As a final note, we should observe that the meta-object of this example should never be associated with more than one base-level object, because it can only store one operation factory {13}. If it were ever associated with two or more objects, it would always read field callSuper from the most recently initialized object, that might not be the target of the operation being handled. If it tried to create a replacement operation for a different target object, an exception would be thrown, but creating non-replacement operations has no such restriction, so mistakes might have gone unnoticed.

4.3.5 Using messages

Guaraná provides a mechanism that allows any object whose class `implements interface` Message to be broadcast to the components of the meta-configuration of an object. This allows communication between meta-level components without requiring them to be explicitly named or even known in advance. In Program 10, for example, defines a Message that can be used to look for a meta-object in the meta-configuration of an object. It could be easily modified to carry any kind of information to be given to a selected meta-object, or to look for meta-objects that present a certain property, such as being an instance of a selected class, instead of just comparing references.

When an instance of class AreYouThere is created, a meta-object to be looked for {4} must be specified {5–7}, and wasThere is initialized to `false` {3}. A meta-object that understands this kind of Message should invoke method IamHere {8–11} to flag its existence; if the given reference is the desired one {9}, wasThere is set to `true` {10}. One could check whether the meta-object was found by invoking method wasThere {12}, but method lookFor {13–17} provides a more convenient interface for using this mechanism. It creates the AreYouThere message {14}, broadcasts it to meta-configuration of the specified object {15}, then tests whether the specified meta-object was found {16}.

Method main {18–31} instantiates a meta-object that can handle AreYouThere messages {19–24}, using the anonymous inner class notation introduced in Java 1.1. Method handle {20–23} is invoked by broadcast of the kernel of **Guaraná**; if the broadcast Message is an instance of class AreYouThere {21}, it invokes method IamHere {22}.

Then, the program creates an object {25} and looks for the created meta-object in its meta-configuration {26}. As expected, Output 9 shows it is not there <1>. Then, method main reconfigures the object so that mo becomes its primary meta-object {27}. After that, the program looks for it again {28}, and now it is found <2>. Finally, the object

Program 10 `AreYouThere.java`

```
import BR.unicamp.Guarana.*;
public class AreYouThere implements Message {
    private boolean wasThere = false;
    private final MetaObject mo;
    public AreYouThere(final MetaObject mo) {
        this.mo = mo;
    }
    public void IamHere(final MetaObject mo) {
        if (this.mo == mo)
            wasThere = true;
    }
    public boolean wasThere() { return wasThere; }
    public static boolean lookFor(final MetaObject mo, final Object o) {
        AreYouThere m = new AreYouThere(mo);
        Guarana.broadcast(m, o);
        return m.wasThere();
    }
    public static void main(String[ ] argv) {
        final MetaObject mo = new MetaObject() {
            public void handle(final Message m, final Object o) {
                if (m instanceof AreYouThere)
                    ((AreYouThere)m).IamHere(this);
            }
        };
        Object o = new Object();
        System.out.println(lookFor(mo, o));
        Guarana.reconfigure(o, null, mo);
        System.out.println(lookFor(mo, o));
        Guarana.reconfigure(o, mo, null);
        System.out.println(lookFor(mo, o));
    }
}
```

```
false
true
false
```

Output 9: `guarana AreYouThere`

is reconfigured again, so that its primary meta-object becomes `null` {29}, and, once again, the meta-object cannot be found any more {30}<3>.

4.3.6 Creating Proxies

Proxies are useful for at least two purposes: they can be used to represent objects from different address spaces and as object shells in which real objects are going to be created or reincarnated from persistent storage. In this example, we show how to turn a proxy, created using a meta-level interface, into a real object.

Program 11 defines class MakeProxy, a specialization of MetaObject, and two inner classes, namely, Base {3} and Init {4}. Base is just an arbitrary base-level class that will be instantiated from the meta-level, and Init is an implementation of Message we are going to use to communicate with the MakeProxy meta-object.

Method main associates a MetaLogger with prefix "`c: `" with class Base {6–7}, as you may see in Output 10 <1>, then it creates an instance of an anonymous subclass of SequentialComposer {8–16} that delegates to a MetaLogger with prefix "`o: `" and a MakeProxy meta-object {9}. The sequential composer is specialized so that its method reconfigure {11–15} accepts to be replaced {13} (the default implementation {14} will only accept reconfiguration requests for meta-objects it delegates to).

A proxy of class Base is created by invoking method makeProxy of class Guarana {17}. Whenever a class is instantiated, the meta-configuration of the class is notified with a NewObject message. However, when a proxy instance of a class is created, the broadcast message is an instance of class NewProxy <2>, a subclass of NewObject, so that meta-objects of the class can behave differently for proxy objects.

The makeProxy request also includes a MetaObject specification; after the NewProxy message is broadcast, a reconfigure request is issued to install the specified MetaObject as the primary meta-object of the proxy. If the meta-configuration of the class has already configured the proxy with a meta-object, the reconfiguration request would be presented to this meta-object. However, in our example, the proxy was not made reflective yet, so the meta-configurations of the class of the proxy and of its superclasses will be given the opportunity to modify the reconfigure request.

First, method reconfigure synchronizes on the Class object that represents the class of the proxy <3–4>, to ensure that the reconfiguration is atomic (if the object were reflective already, the synchronization would be performed on its primary meta-object). Then, an InstanceReconfigure message is broadcast to class of the proxy <5>, then to its superclass, and so on, as implied in <6,7>. Since no class meta-configuration modifies the reconfigure request, the SequentialComposer is associated with the proxy and initialized. It propagates the initialization to its component meta-objects <8>. Method initialize of our meta-object saves the operation factory it is presented in opf {23–25}. When reconfiguration terminates,

Program 11 `MakeProxy.java`

```
import BR.unicamp.Guarana.*;
public class MakeProxy extends MetaObject {
    public static class Base {}
    public static class Init implements Message {}
    public static void main(String [ ] argv) {
        Guarana.reconfigure(Base.class, null,
                                new MetaLogger().setPrefix("c:  "));
        MetaObject mo = new SequentialComposer(new MetaObject [ ] {
            new MetaLogger().setPrefix("o:  "), new MakeProxy()
        }) {
            public MetaObject reconfigure
            (final Object o, final MetaObject pre, final MetaObject pos) {
                if (pre == this) return pos;
                else return super.reconfigure(o, pre, pos);
            }
        };
        Base o = (Base)Guarana.makeProxy(Base.class, mo);
        Guarana.reconfigure(Base.class, null, null);
        Guarana.broadcast(new Init(), o);
        Guarana.reconfigure(o, null, null);
        System.out.println(o);
    }
    private OperationFactory opf;
    public void initialize(final OperationFactory opf, final Object ob)
    { this.opf = opf; }
    public void handle(final Message m, final Object ob) {
        if (m instanceof Init) try {
            final java.lang.reflect.Constructor
                c = Base.class.getDeclaredConstructor(new Class[0]);
            opf.construct(c, new Object[0]).perform();
        } catch (NoSuchMethodException e) {
        } catch (IllegalAccessException e) {}
    }
}
```

```
 1  c: Initialize: java.lang.Class@1b04f8
 2  c: Message: BR.unicamp.Guarana.NewProxy@1e4728 for java.lang.Class@1b04f8
 3  c: Operation: java.lang.Class@1b04f8.<monitor enter>
 4  c: Result: return null
 5  c: Message: BR.unicamp.Guarana.InstanceReconfigure@1e4770 for java.lang.Clas
    s@1b04f8
 6  c: Operation: java.lang.Class@1b04f8.java.lang.Class.getSuperclass()
 7  c: Result: return java.lang.Class@30438
 8  o: Initialize: MakeProxy$Base@134ce8
 9  c: Operation: java.lang.Class@1b04f8.<monitor exit>
10  c: Result: return null
11  c: Reconfigure java.lang.Class@1b04f8: MetaLogger@1b9298 -> null
12  c: Release: java.lang.Class@1b04f8
13  o: Message: MakeProxy$Init@134d78 for MakeProxy$Base@134ce8
14  o: Operation: MakeProxy$Base@134ce8.MakeProxy$Base()
15  o: Operation: MakeProxy$Base@134ce8.java.lang.Object()
16  o: Result: return null
17  o: Result: return null
18  o: Release: MakeProxy$Base@134ce8
19  MakeProxy$Base@134ce8
```

Output 10: `guarana MakeProxy`

so does the synchronization on the class of the proxy <9–10>.

In order to avoid excessive noise in the output, the MetaLogger is removed from the meta-configuration of class Base {18}<11–12>, then we broadcast an Init message to the meta-configuration of the proxy {19}<13>. When method handle {26–33} is invoked, it notices the Message is an Init message {27}. So it looks up a constructor taking no arguments in class Base {28–29}, creates an operation that invokes that constructor and performs it {30}. This invocation is intercepted <14>, and so is the implicit invocation of the constructor of the superclass {15}, as well as their `null` results <16–17>.

Finally, method main removes the composer from the meta-configuration of the object {20}<18>, and the object becomes a regular non-reflective object, indistinguishable from any object created with a `new` expression. Just to probe the behavior of the object, we print its String representation {21}<19>.

It is worth noting that invoking a constructor on a proxy is not strictly necessary. When the object is going to be used only as a proxy, no operation would ever reach it, so there is no reason to construct a real object. But even if this is not the case, and the object will actually execute operations, constructor invocation may be skipped, and fields may be initialized, if necessary, by creating and performing operations from the meta level.

4.4 Advanced

In this section, we are no longer going to present full executable examples: they would be too long and complex. Instead, we are going to discuss some advanced issues regarding meta-configuration management and security, using small code snippets for illustrative purposes only.

4.4.1 Meta-objects with restricted access

When a meta-object is initialized, it is given an operation factory that may allow it to create operations for the base level object from the meta-level. However, composers may initialize meta-objects they delegate to with operation factories that restrict the kind of operations they can create. The primary meta-object is always initialized with an operation factory that provides full access to the base-level object (unless the meta-object is reflective itself, and its own meta-object modifies the invocation of initialize).

In Program 12, for example, we present an implementation of OperationFactory that throws IllegalAccessExceptions {8,15} when requested the creation of any operation that accesses `private` fields {7,14}. The constructor of this class takes another OperationFactory as argument {4}, and the default implementation of methods from class OperationFactoryFilter delegate requests to the operation factory given as the constructor argument {9,16}.

This allows the creation of a hierarchy of operation factories that resembles the hierarchy of composers and meta-objects, except that composers refer to their children, whereas operation factories refer to their parents. Thus, the identity of meta-objects higher in the composition hierarchy is protected from lower ones.

However, OperationFactoryFilters carry a potential security hole: if a child meta-object is able to associate a meta-object with an OperationFactoryFilter, it may be able to obtain a reference to the OperationFactory the filter delegates to, as this reference is stored in a private field of it. Thus, we have associated a meta-object with class OpFactNoPrivate {19–27} that prevents its non-reflective instances from being reconfigured {20–23} and rejects any reconfiguration that might remove itself from the meta-configuration of its class {24–26}. This is accomplished by handling messages of type InstanceReconfigure {21} and removing any meta-object they might carry into the meta-configuration of its instances {22}, and by always returning `this` for any reconfiguration request {26}.

This meta-object does not take care of meta-configurations introduced by meta-configuration propagation, though: if the meta-object that creates this operation factory is reflective, its meta-configuration may freely propagate into the meta-configuration of the operation factory, spoiling the offerred protection mechanism.

Program 12 `OpFactNoPrivate.java`

```
import java.lang.reflect.*;
import BR.unicamp.Guarana.*;
public class OpFactNoPrivate extends OperationFactoryFilter {
    public OpFactNoPrivate(final OperationFactory opf) { super(opf); }
    public Operation read(final Field field, final Operation op)
    throws IllegalAccessException, IllegalArgumentException {
        if (Modifier.isPrivate(field.getModifiers()))
            throw new IllegalAccessException("access denied");
        else return super.read(field, op);
    }
    public Operation write(final Field field, final Object value,
                           final Operation op)
    throws IllegalAccessException, IllegalArgumentException {
        if (Modifier.isPrivate(field.getModifiers()))
            throw new IllegalAccessException("access denied");
        else return super.write(field, value, op);
    }
    static {
        Guarana.reconfigure(OpFactNoPrivate.class, null, new MetaObject() {
            public void handle(final Message m, final Object o) {
                if (m instanceof InstanceReconfigure)
                    ((InstanceReconfigure)m).setMetaObject(null);
            }
            public MetaObject reconfigure
                (final Object o, final MetaObject m, final MetaObject n)
                { return this; }
        });
    }
}
```

4.4.2 Multi-object meta-objects

Meta-objects sometimes have to interact with multiple base-level objects. Some are stateless, in the sense that they do not need to store any information about base-level objects they interact with, so they may safely disregard methods initialize and release.

Other meta-objects, like the one presented in Program 13, must maintain information regarding several objects, for example, in order to create operations for all those objects. This could be redesigned so as to have single-object meta-objects only, but this stricter requirement may lead to complicated solutions, so we have decided to support stateful multi-object meta-objects.

Program 13 `MultiMeta.java`

```
1  import BR.unicamp.Guarana.*;
2  import java.util.Hashtable;
3  public abstract class MultiMeta extends MetaObject {
4      protected Hashtable objDict = new Hashtable();
5      protected static class ObjData {
6          public OperationFactory opf;
7          public int initcount = 1;
8          public ObjData(OperationFactory opf) { this.opf = opf; }
9          public synchronized void initialize(OperationFactory opf)
10         { this.opf = opf; ++initcount; }
11         public synchronized boolean release() { return --initcount == 0; }
12     }
13     public synchronized void
14         initialize(final OperationFactory opf, final Object ob) {
15         final HashWrapper obw = new HashWrapper(ob);
16         final ObjData od = (ObjData)objDict.get(obw);
17         if (od ≠ null) od.initialize(opf);
18         else objDict.put(obw, new ObjData(opf));
19     }
20     public synchronized void release(final Object ob) {
21         final HashWrapper obw = new HashWrapper(ob);
22         final ObjData od = (ObjData)objDict.get(obw);
23         if (od ≠ null && od.release())
24             objDict.remove(ob);
25     }
26 }
```

The easiest way to maintain information about multiple objects is to create a Hashtable that maps an object to the information stored about the object {4}. However, the implementation of hash table compares objects by invoking method equals, and obtains their hash codes by invoking method hashCode. When these methods are invoked, they are likely to be intercepted, and this may lead to infinite recursion.

In order to avoid this kind of problem, **Guaraná** provides the class HashWrapper, that should be used to wrap references to base-level objects used as keys in hash tables. HashWrappers only compare references to objects, instead of invoking method equals, and use the hash code returned by method hashCode defined in class Guarana, that emulates the behavior of method hashCode of class Object without interacting with the object.

Another problem that multi-object meta-objects must be aware of is garbage collection: storing references to base-level objects in meta-objects may prevent objects from being garbage collected. Assume, for example, that meta-object m belongs to the meta-configurations of objects o1 and o2, so m stores explicit references to them. Assume o1 is referred by other active objects, but o2 is not. Under normal conditions, o2 would be candidate for garbage collection. However, since o1 holds an implicit (possibly indirect) reference to m, and m holds a reference to o2, o2 will not be considered unreachable.

The only way to solve this problem would be to use weak references, but this concept is not supported as of release 1.1 of the Java API [56]. Since it will be available in the next release of the Java API, in the future, we may decide to modify HashWrapper so that it only stores weak references to objects.

The implementation in Program 13 handles multiple invocations of initialize and release, that may take place at reconfiguration time. For each base-level object, the most recently provided operation factory is maintained {6}, as well as a count indicating the difference between the number of invocations of initialize and the number of invocations of release {7}. When the meta-object is initialized {13–19}, the base-level object is wrapped in a HashWrapper {15} and looked up in the objDict Hashtable {16}. If the object is listed in the objDict already, method initialize of the object data {9} is executed, so as to save the new operation factory and increment the initialization count {10}. If the object was not listed in the objDict yet, a new ObjData object is created and registered in the objDict {18}.

When method release {20–25} is invoked, the object is also wrapped {21} and looked up in the objDict {22}. If it is not found {22}, something must be wrong, but the error condition is ignored. If it is found, method release of the ObjData {11} is performed to decrement the initcount. When this counter reaches zero, the meta-object was told to release the object as many times as it was initialized, so it can remove the object from the objDict {24}.

4.4.3 Coping with replaced operations

There are two issues to care about, regarding replaced operations. First, there is the composer issue: a composer must only accept replacement operations that actually replace the operation it requested a component meta-object to handle. Similarly, replacement results should refer to the same operation the replaced result referred to.

The other issue has to do with stateful meta-objects. A meta-object must be aware that it may be presented results of operations it was never requested to handle or whose results it was not interested in. This may be caused by reconfiguration, composer laziness (not remembering whether the meta-object requested for a result or not) or operation replacement.

The reconfiguration problem might be avoided if composers refused or delayed reconfigurations while there were pending operations, but this deadlocks if the thread that is handling an operation requests a reconfiguration. Another solution, that fixes the second problem too, is to implement composers that remember what meta-objects should be presented the results for each operation.

But this leads to the third problem: the result handed to the composer may refer to an operation that replaced the operation the composer was originally requested to handle. Therefore, composers, and meta-objects in general, that intend to match results with operations, should store a reference to the original operation, instead of any replacement thereof. For this reason, every Operation provides methods to find out whether it is a replacement (isReplacement), the replaced operation (getReplaced) and the original operation (getOriginal), that iterates through the replaced operations until it reaches the original operation.

A replacement operation must have result types compatible with operations it replaces, i.e., the result type of the replacement must be implicitly convertible to the result type of the replaced operation, and the exception types that may propagate out of the replacement operation must be convertible to exception types that might propagate out of the replaced operation.

However, there is one exception to this rule: it is possible to create untyped placeholder operations using method nop of class OperationFactory. These operations are intended to be replaced with actual operations by some meta-object, usually the one that created it. This is useful to simulate operations intended to be intercepted only by meta-objects located after its creator in the operation handling sequence. When the meta-object receives the placeholder, it returns the actual intended operation as a replacement of the placeholder. With regard to the return value, it may either let it remain unchanged or replace it with, say, an exceptional value.

It should also be noted that it is possible to create other invalid operations from the meta-level. For instance, it is possible to create a method invocation operation with arguments that cannot be converted to the expected parameter types, or an assignment operation to a field that does not exist in the target object. However, before any operation is delivered to the base object, it is validated, and an exceptional result is produced if validation fails.

One possible use of this feature is to create *pseudo* methods and fields, that are introduced by meta-objects, and do not exist in the base level. Pseudo-objects of type Method and Field can be created (makeProxy) and used to identify such inexistent operations.

A more interesting application of this feature is to extend arrays. For example, meta-objects can arrange for arrays to seem to contain more (or less) elements, or even to grow or shrink on demand. Elements of a persistent array, for example, can be reincarnated on demand, instead of all at once.

4.4.4 Reconfiguration details

Guaraná goes to a great length to ensure that meta-level reconfigurations are atomic, and that operations are only delivered to the base-level object after its current primary meta-object has been requested to handle the operation. Two methods of class Guarana participate in this effort: reconfigure and perform.

Method reconfigure tries to ensure that reconfigurations are atomic. This is attained by synchronizing the reconfiguration operation either on the object's primary meta-object, if it is not `null`, or on the class the object belongs to. In either case, after entering the `synchronized` block, if the primary meta-object has changed already, the block is left and entered again, until it holds a lock on the current primary meta-object or class object.

The reconfiguration will take place while the lock is held, so that no other thread can start a reconfiguration on the same object. First, if the primary meta-object is currently null, the class meta-configuration and its superclasses are presented an InstanceReconfigure message; otherwise, the current meta-object is presented a reconfigure request.

Whatever the method, it will end up returning the candidate primary meta-object. Before establishing this candidate as the new primary meta-object, **Guaraná** will test whether the primary meta-object changed. This is possible because the InstanceReconfigure message or the reconfigure request may have caused some meta-object to decide to reconfigure the same object. If the meta-object is found to have changed, the candidate meta-object is simply discarded. Otherwise, it will be initialized with a still invalid operation factory.

If, during the initialization, the primary meta-object reference changes, the candidate meta-object is just requested to release the object and the reconfiguration terminates. Otherwise, the candidate meta-object is finally installed as the new primary meta-object, and the previous meta-object is told to release the object.

At the moment the primary meta-object is modified, the operation factory presented to the previous meta-object and any other operation factories based on it are immediately invalidated. This ensures that meta-objects removed from a meta-configuration cannot create new operations.

However, an evil meta-object might have already created an operation for later (ab)use before it was removed from a meta-configuration. It will be able to request **Guaraná** to perform it. However, even operations created from the meta-level are subject to interception, so the new primary meta-object will be requested to handle the operation, and it may refuse to let the operation reach the object.

But the previous meta-object might have foreseen the new meta-object's refusal, and it could have already requested **Guaraná** to perform the operation, and it could be delaying the execution of the operation by not returning from method handle.

In order to prevent this kind of misbehavior, before delivering any operation to a base-level object, method perform will check whether the primary meta-object is still the same that was requested to handle the operation. If it has changed, and the original meta-object requested the result of the operation, it is presented a `null thrown` result. After that, the new primary meta-object is requested to handle the operation. This process repeats until the meta-configuration becomes stable, that is, the primary meta-object remains unchanged while it handles the operation.

After the operation is delivered to the base-level object, the result is presented (if requested) to the meta-object that was requested to handle it, even if the primary meta-object has already changed, and the old one will not be able to do much with the result.

4.5 Conclusion

By studying this tutorial, you should have learned how to code meta-objects so as to monitor and extend the behavior of base-level objects. You should have understood the protocol for handling operations and results, the protocol for dynamic reconfiguration. You may also have seen some utility for the message broadcasting mechanism, as well as the meta-configuration propagation protocol.

You may have gained some insight on how composers are supposed to behave, and how they can prevent untrustworthy meta-objects from entering the meta-configuration of objects they control, or how to let them in, albeit preventing them from creating unwanted operations or ignoring their results.

Despite its length, this tutorial does not present a single whole picture; instead, it shows several small examples that we hope are enough for one to be able to start using **Guaraná**, gaining experience with it and possibly mastering it with the advanced discussions.

Capítulo 5

Aplicações

Prólogo

Após uma descrição dos poderes reflexivos oferecidos pela arquitetura do **Guaraná**, o artigo deste capítulo descreve maneiras de implementar, de forma elegante e transparente, diversos serviços de meta-nível relevantes para o desenvolvimento de aplicações distribuídas confiáveis.

As descrições de serviços são o início dos trabalhos de desenvolvimento da biblioteca de meta-objetos que intitula o artigo, e vem sendo objeto de intensos estudos e trabalhos de implementação, com participação de alguns alunos de graduação de pós-graduação do Laboratório de Sistemas Distribuídos do Instituto de Computação da UNICAMP. A biblioteca pretende explorar ao máximo as possibilidades de (de)composição de comportamento de meta-nível e reconfiguração dinâmica oferecidas pelo **Guaraná**.

Uma versão anterior deste artigo foi publicada nos Anais do II Workshop em Sistemas Distribuídos, realizado em Curitiba, PR, de 17 a 19 de junho de 1998, após sua primeira edição como relatório técnico IC-98-15.

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.

Resumo

Este artigo apresenta uma biblioteca de meta-objetos adequada para o desenvolvimento de sistemas distribuídos. A arquitetura reflexiva **Guaraná** torna possível que esses meta-objetos sejam facilmente combinados, a fim de desempenhar comportamento de meta-nível complexo e reconfigurável dinamicamente. Descreve-se sucintamente a implementação de **Guaraná** em Java™. Em seguida, explica-se como vários serviços de meta-nível, como persistência, distribuição, replicação e atomicidade, podem ser implementados de forma transparente e flexível.

---

*Islene Calciolari Garcia *recebeu auxílio da FAPESP através da bolsa 95/1983-8.*

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

Keywords: Reflection, Distributed Objects, Persistence, Replication, Atomic Actions.

5.1 Introduction

Computational reflection [38, 51] (henceforth just reflection) has proven to be a useful support mechanism for building distributed systems in a transparent way [1, 2, 7, 13, 16, 21, 20, 34, 37, 43, 45, 53, 54, 52, 59, 60]. **Guaraná** 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 5.2, 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 5.4, 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**.

5.2 The Java-based implementation of Guaraná

Java [3, 26] is a simple yet powerful object-oriented language. Java classes are compiled into high-level object-oriented cross-platform bytecodes, that can be executed on Java Virtual Machines (JVM). Since the specification of the JVM [36] 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 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 [38].

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.

5.3 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**.

5.3.1 Persistence

A persistent object [4, 12] 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.

5.3.2 Replication

Object replication [49] 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.

5.3.3 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.

5.3.4 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 [35]). 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.

5.3.5 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 [19].

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.

5.3.6 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.

5.3.7 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) [25] pattern, connecting a model object with its views transparently: the control can be totally implemented in the meta-level [17].

We might also use meta-objects that implement Statecharts [28] 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 [11].

Monitoring multiple distributed objects may require the construction of consistent global snapshots [14, 18, 23, 24]. Algorithms for obtaining consistent global snapshots can be implemented completely in the meta-level.

5.3.8 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 [55], 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.

5.4 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 [10]. The only similar project we have known to date is Apertos [59, 60], 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.

Capítulo 6

Conclusão

Após um longo tempo de maturação, em que sofreu inúmeras reformulações, a arquitetura reflexiva do **Guaraná** finalmente atingiu um alto grau de estabilidade e coerência. Com isso, foi possível concluir uma primeira implementação já em fevereiro deste ano. A partir de então, novas versões foram criadas, à média de uma por mês, corrigindo os inúmeros erros das primeiras versões, introduzindo pequenas alterações, com objetivo de facilitar a utilização da plataforma, melhorar seu desempenho e atualizá-la em relação a avanços introduzidos nas versões mais recentes do *Kaffe OpenVM*.

O fato de o **Guaraná** ser distribuído livremente, sob a GPL (*General Public License*) do projeto GNU, além do grande volume de documentação disponível, toda escrita em inglês, tem favorecido a aceitação desta implementação do **Guaraná** na comunidade nacional e internacional. Além dos diversos alunos do Instituto de Computação da UNICAMP que têm baseado seus trabalhos nesta plataforma, temos notícia de pesquisadores de algumas universidades brasileiras, como Universidade de São Paulo e Universidade Federal do Pará, e até mesmo de uma empresa multinacional (*ParaGraph Inc.*, empresa associada à *Silicon Graphics Inc.*), que têm manifestado interesse em utilizar **Guaraná** como ambiente de apoio para a construção de aplicações reflexivas.

Atribuímos o crescente uso do **Guaraná** à linguagem escolhida para a implementação, cuja utilização encontra-se em franca expansão, e ao fato de possibilitar a implementação de serviços de gerência de forma transparente, modular, reutilizável e segura, e com um desempenho razoável.

Queremos crer que, à medida em que o trabalho for divulgado, através de anúncios ainda não realizados em listas de *e-mail* e *newsgroups*, além de publicações em periódicos de boa visibilidade, esta procura só venha a crescer. Vale mencionar que, três dias após a defesa desta dissertação, recebemos comunicação sobre a aceitação de um artigo sobre o modelo de composição de meta-objetos do **Guaraná** [44] num *workshop* da OOPSLA'98, uma das maiores conferências do mundo na área de computação.

Como trabalhos futuros, podemos citar as otimizações propostas no Capítulo 3, isto é, (i) preservar o estado dos registradores em caso de não interceptação de operações envolvendo campos ou elementos de *arrays* e (ii) criar interceptadores e acionadores de métodos especializados, que não incorram no custo adicional de interpretar a assinatura dos métodos correspondentes a cada interceptação. Pretendemos implementar essas otimizações em futuro próximo.

Finalmente, o trabalho futuro de maior destaque será a estruturação e a implementação parcial da biblioteca **MOLDS**, que consiste no objeto de estudo do programa de doutoramento recém-iniciado.

Apêndice A

Obtendo Guaraná e MOLDS

Informação adicional a respeito do **Guaraná** pode ser obtida na Home Page do **Guaraná**, na URL `http://www.dcc.unicamp.br/~oliva/guarana/`. O código fonte de sua implementação, baseada no *Kaffe OpenVM*, documentação *on-line* e artigos completos estão disponíveis. **MOLDS** está em estágio inicial de projeto mas, quando você ler esta dissertação, pode haver informação mais atualizada na Home Page do **Guaraná**. Tanto **Guaraná** quanto **MOLDS** são *free software*, distribuídos sob os termos da *GNU General Public License*, mas suas especificações são abertas, portanto implementações independentes podem ser distribuídas com licenças distintas.

*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.*

Referências Bibliográficas

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

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

[29] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley/ACM Press, Reading, Mass., 1992.

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

[31] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, Jan. 1996.

[32] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*, chapter 5,6. MIT Press, 1991.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] S. Matsuoka, T. Watanabe, Y. Ichisugi, and A. Yonezawa. Object-oriented concurrent reflective architectures. In *ECOOP'91*, July 1991.

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

[41] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *IMSA'92 International Workshop on Reflection and Meta-level Architecture*, Nov. 1992.

[42] H. Okamura, Y. Ishikawa, and M. Tokoro. Metalevel decomposition in AL-1/D. In *1st International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, Nov. 1993.

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

[44] A. Oliva and L. E. Buzato. Composition of meta-objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, Oct. 1998.

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

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

[47] Rational Software Corporation. *Unified Modeling Language v1.0.1*, Jan. 1997.

[48] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Desing.* Prentice Hall, 1991.

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

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

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, Dec. 1996. Version 1.1.

[57] K. Ushijima, S. Chiba, and T. Masuda. Meta-level programming for simplifying library protocols. In *ISOTAS'96 (Submitted to)*, 1996.

[58] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA'88*, volume 23, pages 306–315, Sept. 1988.

[59] 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.

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