Composing non-orthogonal meta-programs

Bart Vanhaute Eddy Truyen Wouter Joosen
Pierre Verbaeten
Distrinet
Computer Science Department
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven BELGIUM
{bartvh,eddy,wouter,pv}@cs.kuleuven.ac.be

September 20, 1999

Abstract

A key feature of meta-level architectures is the strict separation between the application and the non-functional requirements. In a base-level, the application behaviour is modelled. Other requirements are handled in a meta-program. This makes both application and meta-program reusable. We introduce a mechanism based on high-level declarative policies to overcome the difficulty of tailoring a meta-program to application demands, without sacrificing its genericity. These policies can also be used in the composition of multiple meta-programs in a meta-tower. Often, the requirements are not completely orthogonal, and dependencies between the meta-programs in the tower are generated. Through policy specifications, these dependencies can be described separately from the implementation. This results in reusable and composable meta-programs in a more general case.

1 Introduction

Meta-level architectures have become an important research topic to master the complexity of developing robust distributed applications. Through the use of a meta-object protocol (MOP hereafter), an object-oriented program can manipulate the state of its own execution [6] and implement non-functional requirements such as fault-tolerance and distributed execution [2, 4, 10]. The major benefit of this approach is the separation of concerns between the application behaviour and the non-functional requirements. In addition, the FRIENDS system [4] demonstrated how multiple orthogonal requirements, in this case distribution, security, and reliability, can be applied at the same time by carefully stacking the meta-programs on top of each other.

However, one particularly relevant problem is the complexity of the meta-programs that reflect upon the base-level behavior. This complexity results from both the inherent complexity of reflective systems and the non-trivial protocols and algorithms that are used to meet the non-functional requirements. As a

result, it is very hard for application programmers, who are typically not experts in these domains, to specialize such meta-programs to their needs.

In our work on Correlate[3], a concurrent object-oriented language and runtime with a meta-level architecture, we have developed a new approach which tries to bridge the gap between application programmers and meta-level programmers. Meta-level programmers use a specialized language to define a template that expresses how the meta-program can be configured. Application programmers instantiate these templates to define application specific policies. Here, we can combine high-level declarative policies with reusable and flexible meta-programs. An important property of our approach is that it is not specific to a particular non-functional property, making it applicable to any meta-program.

It is the position of this paper that this mechanism can be used not only to bind an application and its meta-level program, but also to describe the relation between meta-levels that are combined into a meta-tower configuration.

The rest of this paper is structured as follows. In the following two sections, the general problem and our approach to solving it is explained. In the next section the application of this approach in the case of a meta-tower is evaluated.

2 Problem Description

Run-time reflection is a powerful technique that can be used to control an application's behaviour. Existing research has shown how this technique is used successfully to implement non-functional requirements such as reliability, security and physical distribution [10]. In this approach, the code that realizes the non-functional requirements is expressed as a meta-program. This meta-program consists of a collection of meta-level objects that use the MOP to control the application. This MOP is a set of meta-objects that define an abstract, application independent view on the base-level objects. An excellent separation of concerns is achieved as the base-level code is completely free of any non-functional concerns and the meta-program does not contain direct references to the base application.

An important issue with this approach is the specification of the binding between base and meta-level program. Non-functional requirements are largely independent of application behaviour, but not completely. For example, to achieve optimal performance, application specific requirements have to be taken into account. This means that one can not simply select the correct meta-program that implements the requirements, as this would treat every object in the application the same. Indeed, some application objects must be treated differently at the meta-level. In a dynamic environment, it might even be the case that this depends on the current state of the application objects. In such cases, a much more expressive specification of the binding is required. We believe that most of the current work on run-time reflection does not address this problem in a satisfactory way.

In some work [2, 5], the application's source code is annotated with some special constructs. Through these annotations, an application programmer can indicate the binding between the base-level and meta-level classes. In our opinion, these approaches are not satisfactory because the separation of concerns is violated. Indeed, the application's source code is tangled with references to the

meta-program. Each time the binding between application and meta-program changes, the application's source code needs to be changed. In addition, the expressive power of these mechanisms is limited because the annotations only allow a static binding defined at compile-time.

A more modular solution is provided by Dalang [11]. In this system, a separate configuration file is used to specify the binding. This file also specifies which methods and constructors are to be reflected upon. This yields a much better separation of concerns. A change in the configuration does not require a recompilation of the application classes. However, the expressive power and flexibility is very limited in Dalang. It is not possible to decide on configuration parameters at run-time[1] or to provide detailed application specific information.

In the following section we will briefly present our approach that provides high-level support for the integration of meta-program and application. A more elaborate description of this approach, together with its usage and an implementation for Correlate can be found in [9].

3 Application Policies

In our approach, application specific characteristics can be defined in policies. A policy specifies how the mechanisms that are provided by the meta-program should be applied for a specific application. Policies are strictly separated both from the application and the meta-program. This strict separation enables the reuse of meta-programs for multiple applications.

The policies are declared by an application programmer at a high level of abstraction. At run-time, policies are interpreted by the meta-program. This allows the meta-program to take into account the application specific preferences concerning the implementation of the non-functional requirements. This also means that the semantics of the policies is ultimately defined by the meta-program. The interpretation of the policies is done in terms of a general template, defined by the meta-programmer. This keeps the meta-programs independent of specific applications.

This idea is explained in further detail in the following two sections. In section 3.1, the definition of policies is discussed in detail. Section 3.2 explains the interpretation.

3.1 Defining Policies

We make a difference between *templates* and *policies*. A template defines a declarative language that indicates the various possible customisations an application might require. A policy is an instantiation of such a template. Instantiating a template consists of making a selection between a number of possible customisations and completing certain missing information. The scope of the resulting policy is a single application class.

A template is expressed as a set of *properties*. In the simplest case, a property is an enumeration of a set of atomic values. In more complex cases, a property is declared as a function of the internal state of the object and the current state of the environment, expressed as a set additional parameters that contain

[1] This is in the assumption that the configuration files are created before the application starts.

```
distributor {
  constructorproperty
    creation = BALANCED|LOCAL|CUSTOM;
  constructorproperty
    Host allocate(Host[] h) {
     return h[0];
  }
  objectproperty
    migration = NONE | BALANCED;
  objectproperty
    double getLoad() {
      return 1.0;
    }
}
```

```
distributor WorkUnit {
  WorkUnit(Point pos, Dimension size) {
    creation = CUSTOM;
    Host allocate(Host[] h) {
      return host[pos.x % h.length];
    }
  }
  migration = BALANCED;
  double getLoad() {
    return mySize.x * mySize.y;
  }
}
```

Figure 1: A sample template and a policy specification for distribution.

essential information for that specific property. It is the responsibility of the meta-program to provide these parameters. Orthogonal to this, the declaration scope of a property can be an entire application class or a single method or constructor.

A policy is the instantiation of a template. It is always related to a single application class. A policy can instantiate any number of properties of its template. Properties that are not instantiated have the default value as declared in the template. Instantiating an enumeration consists of selecting a single value from the set. To instantiate a function, an alternative implementation must be given. This implementation can make use of the instance variables of the application class (as free variables).

An example of a template definition for the distribution requirement, and a instantiation for a application object that performs some mathematical computation upon a square area can be seen in figure 1.

3.2 Interpreting Policies

Each template is transformed into an abstract class. This class serves as interface for the meta-program to query the policy an application wants. Each property is represented as a public operation on this class. Their parameters depend on the declaration scope of property and also contain the set of additional parameters as defined in the template. Finally, a static operation is defined that can be used to retrieve the policy object for a certain application class.

Policies for application classes are transformed into specialisations of the abstract template classes. These specialisations implement of course the property operations of the template class, as defined in the policy.

4 Composition of meta-programs

Many applicatons will have multiple non-functional needs. In this case meta-programs will have to be composed. There are currently several approaches to this composition. In a first approach, composition is explicitly supported by the MOP [8, 1]. Here, every aspect of the overall meta-program functionality is handled in a specific meta-object. For instance, to implement object migration

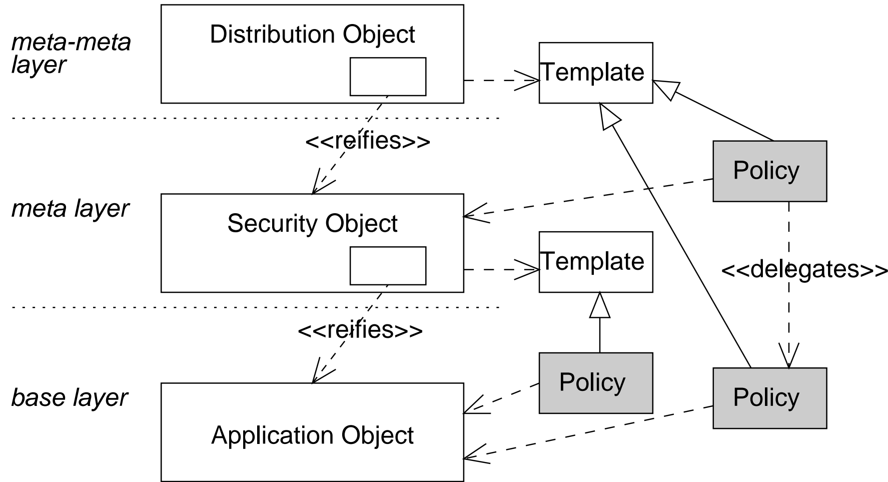

Figure 2: An example mapping for security and distribution.

one only needs to change the distributed environment model. However, this kind of composition breaks if the non-functional requirement cuts across several aspects of the MOP.

The second, and most obvious approach to meta-program composition is through explicit composition, using object-oriented techniques and design patterns. The result is usually a very complex meta-program, which is hard to maintain and evolve. This therefore violates the principle of separation of concerns.

A third alternative uses multiple meta-levels to create a meta-tower. The reasoning behind this is that a MOP gives the programmer much more control over the base application, such that composition with a meta-tower should be easier. This seems very promising, but has severe consequences on the level of control a meta-program at a higher level will have over the application. Indeed, as each meta-level only sees the level directly beneath it, the higher meta-levels can make no decisions about the state of the base application. We believe that the policy specifications as discussed earlier can help overcome this problem in many practical cases.

The general idea is as follows. Each level defines the policies for all of the levels above it. The definition of these policies can be based on the corresponding definitions for the level beneath it, but accomodated to its own needs. Let us give an example (see figure 2). Suppose a security meta-program is to be composed with a distribution meta-program. The distribution policy of a security meta-level object then depends on the distribution policy of the base-level object it is controling. Consider for instance the `allocation` property of figure 1. A particular security policy could be to only allow creation of some particular objects on trusted hosts. The implementation of an allocaton property in a policy for security meta-level object would first query the policy of its base object, and only if the latter selects a trusted host, it would forward this. Otherwise a different but trusted host is selected.

This delegation from a policy at one level to a policy from a lower level

has two dimensions of complexity. First, when the implemented non-functional requirements are not completely orthogonal a strict delegation will not be sufficient. The selection when and how to delegate are dictated by the semantics of the meta-program at the considered level. This is clearly a complex task. Here, the declarative nature of the specifications is helpful. The composition of behavioural descriptions would be much harder, as can be illustrated by AspectJ[7]. Here composition of aspects means deciding when and in what order the `before`, `after`, and other *advices* are executed. This will require careful study of the behaviour of these advices.

First, it is already clear that the mapping of policies of one level to the correct policies on the lower level is a complex task. It depends on the orthogonality of the meta-programs, and the non-functional requirements they implement.

Second, there is the question of what policy specification to delegate to. The answer to this is dependent on the relation between the entities (objects, methods, ...) at the one level and their reified versions at the next. This relation is specified in the MOP. An object in the base-level is reified into an abstract representation. This representation is then controlled by one (or more) objects at the meta-level. Therefore, a property in the context for the meta-level object would query the corresponding property in the policy for the base-level object this object controls. The same reasoning can be applied to method invocation and object construction. A method invocation is reified as a message. At some point, this will cause an invocation in the meta-level, to send the message to the meta-level object of receiver. So, a property definition for the base-level method would be used in a definition for the method used for receiving messages from meta-level objects.

5 Conclusion

This paper explains how our approach to enable application-specific policies for non-functional requirements to be expressed at a high level of abstraction can also be used to compose several meta-programs in a meta-tower arrangement. Through delegation of policy specifications of one meta-level to a lower (meta-) level, this composition becomes feasible in the case the non-functional requirements and implementations are not entirely orthogonal. The complexity of this delegation arises from the questions of how and to what to delegate.

Through further experimentation, we hope to have a better view on its applicability in complex, real-world applications.

References

[1] Lodewijk Bergmans. Composing Concurrent Objects - Applying Composition Filters for Development and Reuse of Concurrent Object-Oriented Programs. PhD Thesis, Universiteit Twente, 1994.

[2] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In Proceedings of ECOOP '93, pages 483-502, Kaiserslautern, Springer-Verlag, July 1993.

[3] The Correlate home page. http://www.cs.kuleuven.ac.be/~xenoops/CORRELATE/.

[4] Jean-Charles Fabre and Tanguy Prennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. In IEEE Transactions on Computers, 47(1), January 1998.

[5] Brendan Gowing, Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. In Proceedings of Reflection'96, San Francisco, 1996.

[6] G. Kiczales, J. des Rivieres and D. Bobrow. The Art of the Meta-Object Protocol, MIT Press, 1991.

[7] Cristina Lopes and Gregor Kiczales. Recent Developments in AspectJ. In ECOOP'98 Workshop Reader, Springer-Verlag, 1998.

[8] Hideaki Okamura and Yataku Ishikawa. Object Location Control using Metalevel Programming. In Proceedings of ECOOP '94, 299-319, Bologna, July 1994.

[9] Bert Robben, Bart Vanhaute, Wouter Joosen and Pierre Verbaeten. Non-Functional Policies. In Proceedings of the Second International Conference on Metalevel Architectures and Reflection. Saint-Malo, France, Springer-Verlag, July 1999.

[10] Robert J. Stroud and Zhixue Wue. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmermann, editor, Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996.

[11] Ian Welch and Robert Stroud. Dalang - A Reflective Java Extension, OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, October 1998. To be published as part of the OOPSLA'98 Workshop Reader.