Abstract

Middleware platforms have filled up an important gap in distributed application development, as they enable a high level of abstraction, masking the complexities of distributed systems programming. Recently, though, technology developments in areas such as multimedia systems, networking and mobile computing have made feasible new categories of applications that are not properly supported by conventional middleware. In common, such applications pose important requirements regarding platform configurability and adaptability, for which the black box nature of conventional middleware is not suitable. In this context, reflection and meta-level architectures represent a concrete solution, offering a principled way to open up the design and implementation of a platform, enabling its internal structure and behaviour to be dynamically inspected and adapted. In this way, new application requirements and environmental changes can be accommodated by platform evolution. This paper presents an approach for reflective middleware which combines meta-information management with meta-object protocols, offering a common basis for platform configurability and adaptability. The paper also describes a concrete implementation of the approach.

Keywords: Middleware, Reflection, Reflective Middleware, Distributed Multimedia Systems.

1 Introduction

The advent and popularisation of middleware platforms have greatly contributed to the widespread development and use of distributed systems and applications. Technologies such as CORBA [1], Java RMI [2], DCOM [3] and, more recently, Enterprise Java Beans [4] and .NET [5] have gained considerable space as part of the software development scenario. Such technologies offer a software layer that sits above operating systems and provide standard interfaces and services on top of which applications are developed. This enables a distributed environment where the problems originated from distribution are hidden from the programmer. Typical non-functional concerns, such as location of services and resources, language and operating system interoperability, and replication, are made transparent, leaving the programmer free to deal with the functional concerns of the applications.

Middleware technologies have also enabled the notion of open systems to be extended to the realm of distributed systems. Applications built using middleware can typically rely on the platform’s standard interfaces to provide the common distributed systems services in a way that is both interoperable and portable across different operating systems and languages. Nevertheless, this conventional notion of openness has begun to be challenged. Current practice in middleware standards requires only interfaces to be open, while the details of platform implementation are left for the vendors to decide. Usually, this leads to rigid, monolithic platforms, which cannot be tailored to the particular needs of different applications. Firstly, not all applications use all services built into a platform, causing memory footprints larger than needed. In addition, some applications may need specialised services
which are not provided by mainstream middleware (such as in the case of multimedia applications). Secondly, application requirements may evolve during runtime, demanding adaptations on the platform implementation, so that the set of services provided remain appropriate after, for instance, changes in the execution environment. This leads us to conclude that middleware platforms should be built around more open architectures, following an open implementation approach [6]. In this way, programmers can configure customised versions of a platform, as well as change their internals at runtime as need arises.

Reflection and meta-level architectures represent a principled way to open up the internal implementation of a platform for dynamic inspection and modification. As a complement to this, meta-information management offers a basis for middleware configuration, allowing one to specify and build customised instances of a platform. This paper presents an architecture for reflective middleware, called Meta-ORB, which exploits the combined use of such techniques in order to provide a flexible middleware architecture to support distributed multimedia applications. The paper is structured as follows. Section 2 reviews the basic concepts of reflection and meta-level architectures, while section 3 discusses the principles of meta-information management. Section 4 considers the application of reflection in the area of middleware, with an emphasis on requirements for reflective middleware. Sections 5 then presents the specific approach adopted in the research and describes the Meta-ORB reflective middleware architecture, followed by section 6, which discusses the implementation of the platform. Finally, section 7 reviews important related work, while section 8 presents relevant conclusions and further considerations.

2 Reflection and meta-level architectures

2.1 Basic concepts

The fundamentals of reflective computing systems were introduced by B. C. Smith and can be summarised by his reflection hypothesis [7], which argues that a system can be made to manipulate representations of itself in the same way as it manipulates representations of its application domain. Such a system is said to have a self-representation, which can encompass both its state and behaviour. In addition, if there is a relationship of causal connection [8] between the self-representation and the actual state and behaviour of the system, meaning that changes in one have corresponding effects in the other, the system is said to be reflective. The self-representation can thus be used for inspection and adaptation of the system’s internals.

The architecture of a reflective system, also known as a meta-level architecture, is usually structured in levels, where the bottom level, known as base-level, deals with computation about the domain of application, whereas the levels above it, known as meta-levels, perform computation about the system itself. More precisely, each meta-level is concerned with the representation and manipulation of the level below it (which is its relative base-level), giving rise to the notion of a reflective tower of meta-levels, as illustrated in figure 1. In principle, as with recursive procedures, this tower can have infinite levels. In practice, however, the use of techniques such as the lazy creation of meta-levels means that only a few levels are actually present.

As shown in figure 1, the act of a meta-level exposing the internals of its (relative) base-level is known as reification. This corresponds to the establishment of an explicit representation of the base-level system and its internal implementation, in terms of programming entities that can then be manipulated at runtime. Modifications to this self-
representation result in corresponding changes to the reified elements of the base-level, a process known as *reflection* or *absorption*. Given a particular base-level entity, the set of meta-level entities reifying it is know as the entity’s *meta-space*.

![Figure 1 - Reification, reflection in a meta-level architecture.](image)

### 2.2 Behavioural and structural reflection

The design of reflective systems usually follow a distinction between *structural reflection* and *behavioural reflection*, initially conceived in the context of programming languages [9]. Structural reflection is defined as the ability of a language to provide a complete reification of the program currently executing, together with the abstract data types that are part of the program. On the other hand, behavioural reflection (also referred to as computational reflection [8]) is the ability of a language to provide a complete representation of its own semantics, in terms of the internal aspects of its runtime environment. Hence, while structural reflection usually deals with the functional properties of a system, behavioural reflection is typically concerned with the non-functional properties. Importantly, these two styles of reflection are complementary to each other, with many reflective architectures providing both.

### 2.3 Object-oriented reflection

A well-defined meta-level structure is an important ingredient to facilitate the use of a reflective architecture, due to the multitude of aspects that may need to be handled. What is needed is a meta-level that allows each of the concepts of the system to be easily identified, in terms of discrete elements that can be handled separately from each other.

The *object-oriented* paradigm provides a clean way to structure the meta-level. In general, object-orientation allows the distribution of the reflection mechanisms and interfaces among multiple, distinct meta-level entities [10]. Regarding terminology, in object-oriented reflection, the entities that populate the meta-level are called *meta-objects*, while those entities at the base-level are known as *base-level objects*. Thus, while the interfaces of base-level objects provide an object protocol for access to application functionality, the interfaces of meta-objects provide a *meta-object protocol* (MOP) [11], which allows reflective access to the implementation of the system. Importantly, the same object model should be employed at both base- and meta-level, meaning that reflection can be re-applied at the meta-level itself.
3 Meta-information management

3.1 Basic concepts

Reflective techniques inherently deal with meta-information in order to build the self-representation of base-level entities. Meta-information is kept about the reified aspects of a system, in explicit or implicit form, as part of the state of the meta-objects. Reflection, however, does not imply a consistent framework for modelling and maintaining this meta-information, especially considering issues of sharing and distribution. The provision of such a framework is precisely the goal of meta-information management, and its presence is an important, often overlooked requirement for reflective middleware.

For the purposes of this paper meta-information can be defined as information about a system itself, instead of about the application domain of the system. The structured use of meta-information depends on the concepts of modelling and meta-modelling. The former refers to the description of structured meta-information representing the runtime entities that compose a given target application or system, as well as the relationships between such entities. The collection of such meta-information comprises a model of the application or system, and may provide sufficient detail to enable instantiation and introspection. On the other hand, meta-modelling refers to the ability to represent and manipulate meta-models (models that represent other models), thus defining the constructs used when modelling a system or application [12]. As an example, consider the object-oriented paradigm, in which a meta-model may define constructs such as classes and methods, as well as relationships among them, such as inheritance. The role of the meta-model is thus to define the features that characterise the modelling constructs. This enables the meta-model to provide the necessary meta-information for model instantiation and introspection, in the same way as models are used to introspect on application entities.

3.2 Functionality of meta-information management

Such structured use of meta-information, based on models and meta-models, is the basic prerequisite for meta-information management. However, in order to have an effective architecture for the management of meta-information, it is also necessary to provide facilities which assist with [13]:

- **meta-information definition**, such as with a language with well-defined syntax and semantics (conforming to the meta-model), as well as tools, such as compilers to validate and translate textual meta-information into a machine-readable form; alternatively, interactive tools (such as with a GUI) can be used for this purpose;
- **meta-information maintenance**, with a distributed and persistent repository with features for creating, deleting, managing and manipulating meta-information;
- **definition, storage and evaluation of relationships**, such as compatibility and substitutability, between different entities of meta-information; and
- **meta-information interchange**, based on mappings and tools to transfer meta-information between different repositories, possibly using different meta-models.

A well-known example of a general-purpose meta-information management architecture is the OMG Meta-Object Facility [14], which provides a framework for defining and managing models and meta-models, along with the meta-information they comprise. Another example, although restricted to the CORBA meta-model, is the Interface Repository defined as part of the CORBA specification [1].
Importantly, the use of a unified architecture for meta-information management enables a consistent view of meta-information in a distributed environment. This is crucial when meta-information must be shared by different users in a heterogeneous setting, as conformance to a common meta-model enables its unambiguous interpretation. Furthermore, typical meta-information management architectures impose the requirement that meta-information, once defined and published, must be immutable, so that replicated meta-information elements are always kept consistent [15]. This is crucial in open distributed systems, as meta-information must provide a reliable description of runtime objects (e.g., to enable dynamic type checking).

3.3 Meta-information management for middleware

The demand for meta-information management in middleware platforms comes from two basic needs. Firstly, middleware platforms need to support an open services environment, where service users dynamically bind to service providers. In such an environment, new services can be dynamically introduced and existing services can evolve through versioning. In this context, the availability of runtime meta-information describing the types of servers and clients is vital for the dynamic discovery of services, as well as for type checking and bridging of service types before binding [16]. Meta-information management techniques thus play an important role in open distributed systems, as a means to organise and maintain such runtime meta-information and to support service discovery and interoperability.

Another area where the management of meta-information can prove useful is in the configuration of open distributed systems [17]. In this context, the configuration activities include object creation and allocation (to physical locations in the system), and binding of objects through their interfaces. Explicit meta-information can be used to describe the internal configuration of the components of a system, in terms of templates with enough detail to allow instantiation. In addition, such specifications serve as runtime documentation of the configuration of a system and its components, thus providing a basis for reconfiguration. Using meta-information management techniques, templates can be defined and managed in terms of a meta-model, using repository tools. This also enables the association between templates and typing meta-information, which in turn permits the use of type relationships to search and compare configurations, as well as to validate interconnections between the elements of a configuration. Finally, version management can be used to control and track the evolution of configurations, by enabling multiple template versions to co-exist.

It is therefore important to recognise the role of meta-information management as a principled basis for the definition, instantiation and management of customised middleware platforms. A promising scenario for the future will be the widespread existence of libraries of template and type meta-information describing alternative implementations for the several aspects of middleware, which can be selected and combined (or even extended) in order to produce platforms that are tailored to particular applications.

4 The reflective middleware approach

4.1 Motivation

The overall rationale for reflective middleware comes from the need to open up the platform implementation, in order to allow the customisation and runtime adaptation required by dynamic applications. Existing middleware technologies have recognised the need to
address this problem, although tackling it with a different approach, by adding flexible features on top of their core architectures (an example is the portable interceptor feature recently added to CORBA [18]). Despite the usefulness of such features, the degree of support for customisation and dynamic adaptation is only partial, not covering all aspects of the design and the different phases of a platform’s lifecycle. This is mostly due to the inherent “black-box” nature of these technologies, which limits the extent to which elements of the design can be opened and exposed to the programmer. Reflection, on the other hand, offers a truly generic solution to the above problem, as it enables a principled approach to the design of middleware in a way that naturally renders itself to openness [19]. In addition, the use of reflection enables the different aspects of a platform to be manipulated and adapted in ways that were not anticipated during its design.

In general terms, reflective middleware refers to the use of a causally connected self-representation to support the inspection and adaptation of the middleware system [20]. Thus, the same reflection techniques used in traditional areas (such as programming languages) apply to middleware as well [21]. Besides providing the usual middleware services through standard interfaces (e.g., as defined in the CORBA specification), a reflective middleware platform also provides meta-interfaces that allow the programmer to inspect and manipulate the internals of the platform.

4.2 Overall design issues for reflective middleware

The design of reflective middleware must usually observe the same requirements that are typical of other kinds of reflective systems (such as safety of reflection, the performance impact, and a uniform reflection model). Some requirements, however, are of particular importance in the context of middleware, and form the basis for the approach presented in this paper (see [22] for a more complete list of requirements):

1. **Modular platform infrastructure.** This is essential in order to facilitate a clear identification, at runtime, of the services and components of a platform. It also enables each of these elements to be manipulated independently from the others.

2. **Language and system independence.** In order to enable portability of applications that use reflection functionality, it is important that a MOP for reflective middleware is defined at a language-neutral level. Thus, the use of reflection capabilities of specific languages or operating systems should be discouraged.

3. **Pervasiveness of reflection.** This issue refers to the range of aspects of a middleware platform (e.g. marshalling, synchronisation, communications protocols, and the several distributed systems services) that can be reified with the reflection mechanisms. Ideally, all aspects of interest in a given context should be amenable to manipulation via reflection.

4. **Unified approach to configurability / re-configurability.** Essentially, facilities for static configuration (which enable the initial selection of the middleware components required in a particular scenario) should be integrated with the mechanisms used for dynamic reconfiguration. In particular, both should be based on a unified terminology, also using a consistent set of meta-level constructs.

5. **Management of meta-level complexity.** Meta-levels for middleware platforms tend to be highly complex, due to the large number of aspects that need to be represented and the interactions among them. In addition, the need to handle such aspects in a distributed environment only adds to the complexity. This requires an appropriate meta-level structure in order to make reflective programming a more manageable task.
5 The Meta-ORB architecture

5.1 Overview

The architecture of the Meta-ORB platform is based on the combination of meta-information management and reflection techniques. While the former is the basis for platform configurability, the latter provides the infrastructure and interfaces for dynamic adaptability. In this section, each of these aspects of the architecture will be examined in separate, describing the specific approaches adopted in the work. This will be followed by considerations about the integrated view of meta-information.

5.2 Platform configurability

5.2.1 The Meta-ORB meta-model

The design of Meta-ORB is centred around an explicit meta-model, described using the Meta-Object Facility [14]. This meta-model defines the kinds of entities used in the construction of instances of the platform, as well as the meta-information structures used to describe such entities. The use of such meta-information to specify and instantiate customised platform configurations qualifies the Meta-ORB as a highly configurable framework for middleware.

The elements of the meta-model correspond to the constructs of the Meta-ORB programming model, which has been influenced by CORBA and also by the RM-ODP computational language (in particular, by the interpretation of the standard as proposed in [23]). The first-class constructs in the meta-model are the following:

- Interfaces, which represent access points to the services provided by a component. Three styles of interfaces are provided: operational (for client/server-like interaction), stream (for continuous media interaction) and signal (for more primitive one-way interaction).
- Component objects (or, simply, components), which represent the units of functionality in the platform. Components can have multiple interfaces and can be defined in a hierarchical way by means of configurations of finer-grained components connected by their respective interfaces.
- Binding objects, which are the equivalent of distributed components, whose internal components can be dispersed across the network. Binding objects embed middleware services and communication capabilities, which enable structured and flexible support to the interaction among remotely located application (and middleware) components.

The corresponding meta-model elements (meta-types) represent both the type and template aspects of such constructs, meaning that the meta-model provides a basis for the functions of type and configuration management. (In this paper, instances of such meta-types will simply be referred to as types, although their definitions also include template meta-information).

The meta-model also includes elements to define auxiliary types, which do not correspond to first-class entities in the platform, but are essential to their description. Examples include: media types, constructed types and primitive types. In addition, the meta-model includes non-type-related meta-model elements. These elements correspond to the scope-defining constructs of the type system (e.g., module) and to auxiliary constructs, used in the definition of the first-class meta-types (e.g. operation, flow, signal and QoS annotation). A complete definition of the Meta-ORB meta-model can be found in [22].
5.2.2 Defining platform configurations

A complete definition of the meta-model, describing all the features available for platform configuration, is out of scope in this paper. However, this section presents a few representative examples that should provide an idea of how the basic meta-model constructs can be used to build customised platform instances. A textual notation is used, based on the Meta-ORB ODL (Object Definition Language) [22]. Typically, the platform designer provides a set of specifications conforming to this ODL, which are interpreted and stored as meta-information objects in a repository. Such meta-information objects can later be used to instantiate the whole or parts of a particular middleware configuration. Additionally, meta-information stored in the repository can be re-used as part of newly defined configurations.

In the first example, as shown in figure 2 below, ODL definitions for a composite component are presented. Note that auxiliary definitions have been omitted for lack of space (notably those for interfaces, which are based on a multimedia extension to OMG IDL). The last definition specifies a component for audio/video processing (AVDeviceComp), which is composed of three primitive components, also defined in the example. The configuration of the composite component is specified in terms of its set of internal components, the object graph representing the way such internal components are connected (adjacent components are linked by means of their interfaces), and the interfaces that the overall component presents to its users. This example illustrates how arbitrarily complex units of functionality can be modelled and configured in terms of structured component composition, using primitive components (which encapsulate binary implementations) and composite components.

```
module Example {
  primitive component AudioDeviceComp {
    implementation: AudioDeviceImpl;
    interfaces: AudioDevice audio_interf;
  };
  primitive component VideoDeviceComp {
    implementation: VideoDeviceImpl;
    interfaces: VideoDevice video_interf;
  };
  interface <stream> AVDevice : AudioDevice, VideoDevice {};
  primitive component MixerComp {
    implementation: MixerCompImpl;
    interfaces: AudioDevice audio_interf;
    VideoDevice video_interf;
    AVDevice av_interf;
  };
  component AVDeviceComp {
    internal components: AudioDeviceComp audio_comp;
    VideoDeviceComp video_comp;
    MixerComp mixer_comp;
    object graph: { audio_comp, audio_interf}:(mixer_comp, audio_interf);
    (video_comp, video_interf): (mixer_comp, video_interf);
    interfaces: AVDevice av is (mixer_comp, av_interf);
  };
}
```

*Figure 2 - An example component configuration specification.*

The next example similarly shows how distributed configurations can be specified using the constructs of the Meta-ORB meta-model. Figure 3 shows the specification of a complex binding object, aimed at connecting the interfaces of audio/video components of the kind defined above. The binding is built out of components and other binding objects (their definitions were omitted for brevity) that implement the different elements of middleware functionality, such as stubs, protocol filters and transport protocols. The binding definition is given in terms of the type of binding control interface (which exposes functionality to control
the operation of the binding, such as to pause and resume its operation), the type of the
internal binding objects used in the configuration, and the roles implemented at each of the
binding endpoints. In this particular case, a single role is defined as the binding is symmetrical
(i.e., both its endpoints are meant to connect interfaces of the same type). The definition of the
binding role is similar to a composite component definition, except for the cardinality part,
which specifies the maximum number of endpoints conforming to the role that can be created
in a given binding instance (this means that multipoint bindings are supported). In addition,
the definition of the configuration of a binding role (i.e., its object graph) must also specify
the connection points with other binding roles, in order to connect the whole binding.

```plaintext
module Example {
    binding AVBinding {
        control interfaces: CtrlInterf ctrl is (CtrlComp, ctrl_interf);
        internal bindings: AudioBinding audio_binding;
                        VideoBinding video_binding;
        role AVBindingPartic {
            components: AVStubComp stub;
                        AudioFilterComp audio_filter;
                        VideoFilterComp video_filter;
            target interface: AVDevice is (stub, av_interf);
            cardinality: 2;
            configuration:
                (stub, audio_interf):(audio_filter, audio_interf);
                (stub, video_interf):(audio_filter, video_interf);
                (audio_filter, forward_interf):(audio_binding, audio_role);
                (video_filter, forward_interf):(video_binding, video_role);
        }
    }
};
```

Figure 3 - An example binding configuration specification.

The way such definitions are provided, stored and managed is an implementation-specific
issue (see section 6). The essential requirement is that such definitions can be made public in
some form (typically using a repository), so that they can be accessed by platform
configuration tools, by designers looking for pre-defined configurations (for re-use as part of
more complex configurations), as well as by the reflection mechanisms, as seen in below.

5.3 Dynamic adaptation

5.3.1 Principles and structure of the meta-level

Customised platform configurations can be defined in terms of meta-information elements
that represent the entities comprising the platform. Dynamic reconfiguration, on the other
hand, requires some means to manipulate such meta-information at runtime, in a way that is
causally connected with the specific instances of platform configuration. This is the role of the
reflective meta-level, which completes the architecture of the Meta-ORB and contributes
towards fulfilling the requirements identified in section 4.

Reflection in the Meta-ORB is used for dynamic inspection and adaptation in the context
of both platform and application elements. To this end, the overall architecture is conceptually
divided into base-level, where the actual functionality of the platform and applications is
defined, and meta-level, where the reflection capabilities are defined. The design of the meta-
level follows the principles of the Open-ORB reflective middleware architecture [24], as discussed below.

A fundamental principle of the meta-level is the use of object-oriented reflection, which means that the entities that populate the meta-level are uniquely identifiable objects. The definition of object, in the Meta-ORB, refers to the structural constructs defined in the meta-model, namely components, binding objects and interfaces, although, typically, only component objects (and their interfaces) are used in the constitution of the meta-level. Hence, the meta-object protocol (MOP) is realised in terms of the interfaces of components that play the role of meta-objects. In addition, object-oriented reflection also assumes that the base-level is similarly structured in terms of objects, meaning that meta-objects are used to reify components, binding objects and interfaces. Importantly, in the Meta-ORB approach the state of meta-objects must always have a direct correspondence with the meta-information elements that describe their respective base-level objects. In practice, such meta-information is used, during the reification process, as the basis for initialising the state of meta-objects.

The other important principle of used in the design of the meta-level is known as multi-model reflection framework, first introduced by [25] in the AL-1/D language. This approach consists in applying separation of concerns to the design of the meta-level itself, so that the meta-space of an object is partitioned into a number of distinct aspects, each one realised by a different meta-object. This is important to reduce the complexity of the meta-level, especially considering the multitude of features that must be managed in a meta-level for middleware.

Each separate aspect of the meta-level is defined in terms of a meta-space model, which represents the structure and functionality for the reification of a base-level object according to that aspect. Figure 4 below illustrates the concept of using distinct meta-objects (each one corresponding to a different meta-space model) to reify a given base-level object.

![Figure 4 - Reifying a base-level object according to multiple meta-space models.](image)

Currently, five meta-space models are specified, with well-defined abstract design and semantics. The meta-space models are categorised according to the usual distinction between behavioural and structural reflection, discussed in section 2. The behavioural part of the meta-space consists of two meta-space models: Resources and Interception. The former is responsible for reifying and managing the resources (such as storage and processing) used by base-level objects, while the latter deals with the manipulation of implicit behaviour associated with the interfaces of components (e.g., pre- and post-processing that affect the
non-functional properties of the interactions on a given interface). Typical operations provided by the behavioural meta-object protocols include the addition or removal of interceptors at an interface, and the re-definition of the properties of the resources allocated to an object (such as the amount of memory and scheduling parameters).

Structural reflection, on the other hand, is represented by three distinct meta-space models: 

*Interface Discovery* (which reifies the set of interfaces supported by a component or binding object), *Interface* (which reifies the constitution of a particular interface, in terms of the operations, flows or signals it provides), and *Architecture* (which reifies the internal configuration of a component or binding object, in terms an object graph representing its internal components and the way they are connected).

### 5.3.2 Dynamic adaptation of platform structure

The current version of Meta-ORB is focused on structural reflection, based on the *Interface Discovery, Interface, and Architecture* meta-space models. However, only the latter is meant for adaptation, whereas the former two are meant for inspection only (i.e., to discover the services provided by a component, in terms of interfaces and their operations).

Adaptation according to the *Architecture* meta-space model is mainly done through the manipulation of the object graph that represents the configuration of a given platform element. The meta-object protocol associated with this meta-space model offers operations for inspecting the structure of a configuration, as well for changing it, by adding, removing or replacing components. For instance, in a binding configuration, such as the one described in figure 3, if the available bandwidth of the underlying network suffers a drop, it may become impossible to sustain the previously agreed quality of service. Under the circumstances of rigid middleware infrastructures, such as with conventional middleware, this would typically cause the binding to be torn down. On the other hand, in the Meta-ORB reflective middleware, the use of the *Architecture* meta-object may help overcome the problem in a more satisfactory way. The solution could involve the selection of an alternative video encoding method with lower bandwidth requirements, as well as a component type (defined in the meta-information repository) that implements it. The *Architecture* meta-object can then be used to replace the current video filter components (at each of the binding endpoints) with components of the selected type, without disrupting the overall service (although the user might experience some downgrading of the video output quality).

Other examples of use of dynamic adaptation can be found in the so called 24x7 applications (i.e., those applications that need continous, non-stop availability). In such cases, the use of the *Architecture* meta-object protocol represents a natural way to adapt the application or the underlying platform (e.g., when user requirements or business rules change) without having to stop, reconfigure and then re-start it.

The bottom line for using reflection in such a way is therefore the convenience of making runtime structural changes to an application or to the underlying platform. In addition to smoothing the change process (by preserving continuous availability of the adapted service), this approach also enables to simplify the process of system evolution, as changes can be made in a localised way, without affecting the whole system. There are, however, problems that need to be solved in order to make the approach completely feasible. One problem is the handling of state in the components involved in an adaptation, which requires some mechanism to transfer the state between old and new components (in case of component replacement). The current design of Meta-ORB requires such mechanisms to be defined at the
level of the individual component types, as state transfer is usually an application-specific issue. Another related problem occurs when the adaptation of a binding object involves the manipulation of components that lie on its data path. In such cases, performing the adaptation will cause the flow of data through the binding to be shortly interrupted, which may cause the loss or the inconsistency of information transferred. In the current design, the approach to solve this problem is twofold. First, a mechanism is provided that allows the meta-level programmer to pause the operation of the binding before starting the adaptations and to resume it once adaptations are completed. Second, the design encourages efficient implementations of the MOP, so that, for instance, adaptations can be made to fit within inter-packet delay periods. Nevertheless, more sophisticated solutions to the problem of smooth adaptation are still required, and are the subject of future work.

5.4 Exploiting the unified view of meta-information

As seen above, both the meta-information management facilities and the MOP deal with the same kind of meta-information. While the former uses structural meta-information for static configuration, the latter uses it for dynamic inspection and adaptation. This suggests a strong potential for the close integration of the two techniques.

One approach that has been investigated in this research is represented by type evolution [26], a technique to enable new types of components and bindings to be derived as a result of reflective adaptations. This is especially useful when a series of changes are made to an object, resulting in a different configuration that may be further useful in other scenarios. In this case, the Meta-ORB allows new type meta-information to be generated, based on running objects, and then published in the meta-information repository. Such new types are handled as versions of the original types and can be used in a normal way by configuration tools in order to produce configurations that contain, from scratch, the results of previous adaptation efforts.

6 Implementation

A prototype implementation of the Meta-ORB architecture has been developed with the goal of demonstrating its feasibility and applicability [27]. The focus of this work was on the functionality and the qualities of the architecture, rather than its performance. This is reflected on the chosen implementation environment, which is based on the Python programming language [28], which favours rapid prototyping instead of performance. Despite this, experiments have shown that the performance of the prototype is still appropriate for simple multimedia applications [22]. In addition, by implementing the prototype purely in Python, portability to a variety of operating systems is guaranteed, which was also a factor when choosing the language.

The implementation is structured in three main modules, according to the abstract design discussed in the previous two sections:

- **The Platform Core** – consists of the features that are necessary to support the Meta-ORB programming model. Specifically, it contains the basic distribution infrastructure, with naming and capsule management services, as well as the core constructs to support the meta-model, such as interface references and local bindings (which are links between the interfaces of locally connected components). In addition, this module defines the runtime representation for the first-class constructs of the programming model (interfaces, components and binding objects). Finally, higher-level services are also defined in this module, notably default implementations of component and binding
factories, which are the entities responsible for the instantiation of components and binding objects based on specified type meta-information.

- **The Type Repository** – this module represents the meta-information management framework of the Meta-ORB, providing support for both the platform core and its meta-level. Its structure is based on an extension of the CORBA Interface Repository, in order to define the set of meta-types that constitutes the Meta-ORB meta-model. Importantly, this module also introduces tools and mechanisms to facilitate the definition and manipulation of meta-information, such as a graphic browser, used to specify, edit, publish and search for type definitions.

- **The Meta-level** – this module corresponds to the mechanisms and facilities for structural reflection provided by the platform. It follows the framework defined in section 5.3, with the design defined in terms of the constructs of the programming model (i.e., meta-objects are themselves components). The approach is to provide a default design and implementation, with meta-object types that offer a representative meta-object protocol. This can later be extended with new meta-object types, either through static type definition, or through reflection (i.e., using meta-meta-objects) and type evolution. The precise meta-object protocols currently implemented are described in the Appendix.

Further details about the prototype can be found in [22].

### 7 Related work

This section examines representative related work on the use of reflection as a design principle for distributed systems middleware. The common theme is the use of reflection at a more fundamental level, in order to eliminate some of the crucial limitations of more conventional technologies, such as those highlighted in the introduction. The systems examined here were chosen to illustrate and compare the main reflection techniques that have been used in the context of middleware, in particular: interception, meta-classes and component-based reflection.

FlexiNet is a configurable ORB architecture developed by APM, aiming at mobile object environments [29]. FlexiNet proposes an open binding framework that defines the connection between clients and servers in terms of protocol stacks, which can be dynamically configured by means of method interceptors, as defined by APM’s Reflective Java extension [30]. This feature is mainly used for the adaptation of the non-functional properties of a binding, such as access control, replication and transactional properties. While this approach resembles the use of architectural reflection in our work, it is restricted to linear protocol stacks, while in the Meta-ORB architecture, arbitrary configurations can be handled.

OpenCORBA [31] is a reflective implementation of CORBA based on the meta-class approach and on the idea of modifying the behaviour of a middleware service by replacing the meta-class of the class defining that service. This is mainly used to dynamically adapt the behaviour of remote invocations, by applying the above idea to the classes of stubs and skeletons. The use of meta-classes, however, has the consequence of making such adaptations reflect on all instances of a class. In contrast, in the Meta-ORB reflection is based on per-object meta-objects, enabling to isolate the effects of reflection (so that other objects are not affected when reflection is used to alter a particular object). In reflective middleware, this is a desirable property as the components of a middleware system tend to be fairly independent of each other (even though they might have the same class).

DynamicTAO [32] is another representative reflective middleware architecture. It is based
on an extension of the TAO ORB [33] with the concept of architectural awareness, making explicit the architectural structure of a system in a causally connected way. Middleware configurations are defined in terms of prerequisite specifications, which represent the components of the platform and the dependencies among them. These specifications are used by an automatic configuration service to instantiate the platform components and the components on which they depend. At runtime, such prerequisites are managed by component configurators, which are in charge of keeping the consistency of dependencies as new components are added or removed from the system. This approach is similar to the use of architectural reflection in the Meta-ORB, with the added value of dependency management. However, dynamicTAO restricts the use of reflection to coarse-grained components, limiting its applicability to control more detailed structures of the platform.

Regarding the management of meta-information, although all reflective middleware architectures (such as the ones discussed above) deal with meta-information in one way or another, the treatment is typically ad hoc. On the other hand, the isolated use of meta-information management in middleware, notably for type management purposes has been proposed in the literature (such as in [34]). To our knowledge, however, Meta-ORB is the first middleware architecture to integrate a comprehensive and pervasive framework for meta-information management with a principled reflective meta-level. This has the benefit of unifying the use of meta-information in the system (e.g., preventing that different meta-object implementations use different meta-level representations), as well as providing a basis to closely integrate the configuration and adaptation features of the platform.

8 Conclusion and further considerations

This paper has presented Meta-ORB, a reflective middleware framework based on a combination of a meta-level architecture with a meta-information management framework. The overall aim of the research has been to propose an approach that permits the integration of configuration and reconfiguration facilities in a highly flexible middleware architecture. The foundation concepts used in the research have been surveyed, and motivation for their use in middleware has been considered. The paper also discussed the use and implementation of the features proposed in the architecture.

The design of Meta-ORB was guided by the requirements presented in section 4, which are used here as a framework against which to evaluate the proposed approach. Firstly, the use of components as the basic building blocks of middleware emphasises modular platform architectures, which naturally eases the identification of the entities that need to be adapted through reflection. Second, the definition of the architecture has been presented in an abstract way, not tied to the specific facilities of any given programming language. This enables its implementation (and consequent interoperability) in different programming environments. Third, the fact that the reflection mechanisms are built to operate on the basic building blocks of the architecture means that any aspects of a middleware platform can be subject to reflection. Fourth, the integration of meta-information management and reflection provides a natural basis to unify the facilities used for configuration and dynamic reconfiguration, contributing to preserve the consistency between the mechanisms used during design and run time. Last, but not least, the use of a multi-model reflection framework as the principle to structure the meta-level is an important contribution to lower the complexity of meta-level programming. In this way, reflection can be performed according to a particular meta-space model, largely ignoring the details of aspects managed by other meta-space models. Overall,
the approach has been proven feasible for the support of distributed multimedia applications [22]. In particular, the dynamic adaptation features enable to the underlying platform structure to continuously match the requirements of the application as they evolve over time.

Nevertheless, enhancements to the Meta-ORB architecture are still possible, in order to fully explore the possibilities of the proposed approach. In particular, the author is currently investigating the feasibility of the approach in mobile and wireless environments. In this context, the configuration and adaptation features should decisively contribute to the definition of custom middleware platforms that suit, both statically and dynamically, the limited and variable resource availability that is typical of such environments. To make this feasible, however, future work is required in order to define more lightweight implementations of the reflection and meta-information mechanisms of the platform. Another area for future work consists in the implementation of the behavioural reflection part of the multi-model reflection framework. In particular, the Resources meta-space model is required in order to take full advantage of adaptability as a means to control quality of service. This, however, requires further investigation to extend the current approach in order to meaningfully represent and manage behavioural meta-information within the repository framework. Finally, there is scope for research to further exploit the use of the meta-information framework. In particular, the use of (meta-)meta-information at the meta-model level (the current design only deals with meta-information at the level of models) offers a promising new approach to platform extensibility and interoperability. More specifically, the use of meta-meta-information to interpret the structure and semantics of model elements can help to bridge platforms with different meta-models. In addition, by enabling the explicit manipulation of meta-models, it is possible to dynamically extend a platform with new kinds of constructs (such as to introduce new styles of communication). These enhancements, among others, are envisaged for future versions of the Meta-ORB platform.

Acknowledgements

This research was developed at Lancaster University, as part of of the author’s Ph.D. Thesis, under the sponsorship of CNPq and UFG. The author would like to thank the members of the OpenORB team at Lancaster for their useful contributions, in particular: Gordon Blair, Geoff Coulson, Katia Saikoski, Rui Moreira, Nikos Parlavantzas, Hector Duran and Tom Fitzpatrick.

References

Appendix: Meta-object protocols of Meta-ORB

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_interf_names</td>
<td>returns a list with the unique names of all interfaces of the base-level object (which can be either a component or a binding object)</td>
</tr>
<tr>
<td>get_endpoints</td>
<td>for binding objects only; returns a list with information about all the endpoints (i.e. role instances) of the base-level binding object; endpoint information includes the respective role names and capsule identifiers</td>
</tr>
<tr>
<td>get_target_interf_type_id</td>
<td>for bindings only; returns the repository identifier of the type of target interfaces expected by a binding role</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the type of the base-level component or binding</td>
</tr>
<tr>
<td>replaceable_by</td>
<td>compares the type of the base-level component (binding) with another type, in order to check if a component (binding) of the other type can be a correct substitute for it; substitutability is currently determined by structural compatibility, i.e., if the new component (binding) supports all the interfaces (roles) supported by the original one; if yes, the return is a list with a mapping between the interfaces (roles) of the base-level component (binding) and the names of the interfaces (roles) declared in the substitute type; if not, an empty list is returned. (Note, however, that in future versions of the prototype, such functionality should be migrated to the repository objects representing component and binding types.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_interf_name</td>
<td>returns the unique name of the interface</td>
</tr>
<tr>
<td>get_interf_style</td>
<td>returns a string with the interface style (operational, stream or signal)</td>
</tr>
<tr>
<td>get_interf_descr</td>
<td>returns a complete description of the interface</td>
</tr>
<tr>
<td>get_attr_list</td>
<td>returns a list of strings with the names of all attributes of the interface</td>
</tr>
<tr>
<td>get_attr_type</td>
<td>returns the typecode that represents the type of a given attribute</td>
</tr>
<tr>
<td>get_interaction_list</td>
<td>returns a list of strings with the names of all interactions (operations, flows or signals) defined for the interface</td>
</tr>
<tr>
<td>get_prov_interactions</td>
<td>these two operations return a list of strings with the names of interactions offered by the interface, according to their causality; the results must be interpreted according to the interface style (operations, flows or signals)</td>
</tr>
<tr>
<td>get_req_interactions</td>
<td></td>
</tr>
<tr>
<td>get_interaction_descr</td>
<td>returns a structure containing the complete description of a given interaction; the result must be interpreted according to the interface style</td>
</tr>
<tr>
<td>get_attr_value</td>
<td>returns the current value of an attribute</td>
</tr>
<tr>
<td>set_attr_value</td>
<td>modifies the value of an attribute</td>
</tr>
<tr>
<td>invoke</td>
<td>dynamically invokes an operation</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the interface’s type</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>get_obj_graph</td>
<td>returns a data structure representing the internal configuration (i.e. the corresponding object graph) of the base-level object</td>
</tr>
<tr>
<td>get_internal_comps</td>
<td>returns a list with the unique names of the internal components</td>
</tr>
<tr>
<td>get_bound_comps</td>
<td>given a component, returns information about the other components bound to it (including the unique names of the components and their respective interfaces)</td>
</tr>
<tr>
<td>get_role_config</td>
<td>for bindings only; returns a data structure representing the abstract configuration associated with the given role, as in the binding type</td>
</tr>
<tr>
<td>get_endp_config</td>
<td>for bindings only; returns a data structure representing the concrete configuration of the given binding endpoint</td>
</tr>
<tr>
<td>get_internal_bindings</td>
<td>for bindings only; returns the unique names of the nested bindings (but only at the current level of composition)</td>
</tr>
</tbody>
</table>

**Operations for configuration inspection**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_bind</td>
<td>performs a local binding between the interfaces of two components (or a component and a binding endpoint) that are part of the current configuration</td>
</tr>
<tr>
<td>break_local_binding</td>
<td>breaks an existing local binding within the current configuration</td>
</tr>
<tr>
<td>insert_component</td>
<td>inserts a component of a given type and name at a given location in the configuration (the location is expressed as a pair of interfaces between which the component is to be inserted)</td>
</tr>
<tr>
<td>remove_component</td>
<td>removes and deletes a component from the configuration, re-binding the adjacent interfaces when possible (i.e., when the removed component was bound to two other components, the operation tries to bind their interfaces, so that the configuration is not left broken)</td>
</tr>
<tr>
<td>replace_component</td>
<td>replaces a given component in the configuration with a new one; the operation only succeeds if the new component can correctly replace the old one in all local bindings it took part (this is verified by calling operation replaceable_by of the Interface Discovery meta-object); the old component is deleted</td>
</tr>
<tr>
<td>replace_binding</td>
<td>for bindings only; replaces a nested binding, with the requirement that the new binding must support all the roles of the replaced one; the configuration of the old binding is deleted</td>
</tr>
<tr>
<td>add_component</td>
<td>for components only; adds a new, non-connected component to the configuration of the base-level component</td>
</tr>
<tr>
<td>add_binding_component</td>
<td>for bindings only; adds a new, non-connected component to a particular endpoint of the base-level composite binding object</td>
</tr>
<tr>
<td>role_add_component</td>
<td>for bindings only; these operations act like their counterparts defined above (operations with the same name, except for the “role_” prefix), but applied to the abstract configuration of a given binding role, and with their effects reproduced on all endpoints that are instances of this role in the base-level binding object</td>
</tr>
<tr>
<td>role_local_bind</td>
<td></td>
</tr>
<tr>
<td>role_break_local_binding</td>
<td></td>
</tr>
<tr>
<td>role_insert_component</td>
<td></td>
</tr>
<tr>
<td>role_remove_component</td>
<td></td>
</tr>
<tr>
<td>role_replace_component</td>
<td></td>
</tr>
</tbody>
</table>
Table C (cont.) – Architecture MOP (interface type: ArchMeta_interf)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_interf_exposer</td>
<td>returns the unique name of the internal component exposing a particular interface of the base-level component (or binding object); useful, e.g., when replacing (part of) the implementation behind a given interface</td>
</tr>
<tr>
<td>expose_interf</td>
<td>maps a particular interface of a given internal component as an external interface of the overall composite</td>
</tr>
<tr>
<td>get_type_id</td>
<td>returns the repository identifier of the type of the base-level object</td>
</tr>
<tr>
<td>get_type_descr</td>
<td>returns an up-to-date description of the type (either as a ComponentDescription or as a BindingDescription structure)</td>
</tr>
<tr>
<td>commit_type</td>
<td>used for type evolution; when the copy of the type held by the meta-object has been changed, this operation can be called in order to publish the changed type as a new type version in the Type Repository</td>
</tr>
</tbody>
</table>

Miscellaneous operations