The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms

Gordon S. Blair[1], Lynne Blair[1], Valérie Issarny[2], Petr Tuma[3], Apostolos Zarras[2]

[1]Distributed Multimedia Research Group, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.
{gordon,lb}@comp.lancs.ac.uk

[2]Solidor Research Group, INRIA-Rocquencourt, Domaine de Voluceau, Rocquencourt - BP 105, 78153 Le Chesnay Cédex, France.
{Valerie.Issarny,Apostolos.Zarras}@inria.fr

[3]Distributed Systems Research Group, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.
tuma@nenya.ms.mff.cuni.cz

Abstract. Future middleware platforms will need to be more *configurable* in order to meet the demands of a wide variety of application domains. Furthermore, we believe that such platforms will also need to be *re-configurable*, for example to enable systems to adapt to changes in the underlying systems infrastructure. A number of technologies are emerging to support this level of configurability and re-configurability, most notably middleware platforms based on the concepts of open implementation and reflection. One problem with this general approach is that widespread changes can often be made to the middleware platform, potentially jeopardizing the integrity of the overall system. This paper discusses the role of *software architecture* in maintaining the overall integrity of the system in such an environment. More specifically, the paper discusses extensions to the Aster framework to support the re-configuration of a reflective (component-based) middleware platform in a constrained manner. The approach is based on i) the formal specification of a range of possible component configurations, ii) the systematic selection of configurations based on a given set of non-functional properties, and iii) the orderly re-configuration between configurations, again based on formally specified rules.

1 Introduction

Middleware technologies such as CORBA and DCOM are now established as central elements in modern computer systems, offering portability and interoperability through their platform independent programming models. However, with the rapid deployment of such platforms, there is an increasing demand to introduce more configurability and re-configurability into middleware. For example, configurability is important to meet the often conflicting requirements in areas such as embedded systems, real-time systems, telecommunications, digital libraries, etc. Similarly, re-configurability is increasingly important to cope with rapidly fluctuating environmental conditions as found, for example, in mobile computing. In addition, re-

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 164-184, 2000.

configurability can help enormously in supporting systems evolution, as functional or non-functional requirements change over time (e.g. the scaling up of a system to deal with an order of magnitude increase in activity).

A number of researchers have investigated issues of configurability and re-configurability in middleware platforms. Such solutions often involve the selective replacement or tailoring of key components of the system, e.g. parts of the protocol stack. However, a more general solution is offered by emerging reflective middleware platforms, exploiting the concepts of open implementation and reflection to offer a more principled access to the underlying virtual machine. A number of such platforms have been developed, including Open-ORB [Blair98], Dynamic TAO [Román99], Open CORBA [Ledoux97], and Flexinet [Hayton97]. Such platforms offer great flexibility in terms of meeting the needs of a wide range of application domains. However, they all suffer from the problem of offering too much flexibility, i.e. it is possible to compromise the overall integrity of the system by allowing unbridled access to details of implementation. This paper examines this issue in depth. In particular, the paper considers the role of software architecture in constraining adaptation in reflective middleware platforms. More specifically, the paper examines the results of the Aster Project [Issarny96] and considers the potential of Aster tools and techniques to ensure robust adaptation in the context of Open-ORB (although the results are equally applicable to a range of other reflective middleware platforms). Some important extensions are suggested for Aster to ensure more seamless support for adaptation in such environments.

The paper is structured as follows. Section 2 presents some background information on reflective middleware in general, and the Open-ORB Project in particular. Emphasis is given to the component-based design of this platform. Section 3 then examines the key results of the Aster Project, illustrating how Aster can be used to support the automatic synthesis of middleware platforms from statements of the required non-functional properties (cf configurability). Following this, section 4 provides a more in-depth examination of the adaptation process, highlighting 4 key phases that must be supported. Our solution is then discussed in section 5. This section also includes an explicit statement of the required extensions to Aster, including exploitation of the specifics of reflective middleware. Finally, related work and concluding remarks are given in sections 6 and 7 respectively.

2 Background on Open-ORB

2.1 Why Reflective Middleware?

The concept of reflection was first introduced by Smith in 1982 [Smith82]. In this work, he introduced the reflection hypothesis which states:

> "In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures".

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a *meta-object protocol (MOP)* which defines the services available at the *meta-level* [Kiczales91]. Access to the meta-level is provided through a process of *reification*. Reification effectively makes some aspect of the internal representation explicit and hence accessible from the program.

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open engineering. In contrast, the present-day approach to developing middleware platforms is generally to adopt a *black box* philosophy, whereby implementation details are hidden from the platform user (cf. distribution transparency). There is increasing evidence though that the black box philosophy is becoming untenable. For example, the OMG have recently added internal interfaces to CORBA to support services such as transactions and security. The recently defined Portable Object Adapter is another attempt to introduce more openness in their design. Nevertheless, their overall approach can be criticised for being rather ad hoc. Similarly, a number of ORB vendors have felt obliged to expose selected aspects of the underlying system (e.g. filters in Orbix or interceptors in COOL). These are however non-standard and hence compromise the portability of CORBA applications and services. The authors believe that the solution is to provide flexible middleware platforms through application of the principle of reflection.

2.2 The Open-ORB Architecture

In the Open-ORB architecture, we adopt a *component-based* model of computation [Szyperski98]. A middleware platform is then viewed as a particular configuration of components, which can be selected at build-time and re-configured at run-time. We therefore provide an open and extensible library of components, and component factories, supporting the construction of such platforms, e.g. protocol components, schedulers, etc. The use of components is important given the trend towards the application of this technology in open distributed processing, e.g. CORBA v3 [OMG99] and DCOM. Note however that these technologies exploit component technology at the application level; we extend this approach to the structuring of the middleware platform itself. Our particular component model includes features to support multimedia applications, and is derived from previous work on the Computational Language from RM-ODP [Blair97]. One notable feature of this component model is that communications channels are explicitly represented by components. Such components are referred to as bindings and have the important role in supporting inspection and adaptation of communications related aspects. A fuller description of the component model can be found in [Blair97].

Each component (or more strictly, each interface) has an associated meta-space, offering access to meta-level for the component, i.e. the underlying virtual machine supporting the execution of this component. Because of the complexity of distributed systems, this meta-space is partitioned into a number of orthogonal *meta-space models*. This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [Okamura92]. The benefit of this approach is to simplify the interface offered by the meta-space, by maintaining a

separation of concerns between different system aspects. Currently, four meta-space models are supported in Open-ORB as discussed below.

1. *The Encapsulation Meta-model.* This meta-model provides access to the representation of a particular interface in terms of its set of methods and associated attributes, together with key properties of the interface. This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation.

2. *The Compositional Meta-model.* In reality, many components will in fact be *composite*, using a number of other components in their construction. In recognition of this fact, we also provide a compositional meta-model offering access to such constituent components. In the meta-model, the composition of a component is represented as a *component graph*, in which the constituent components are connected together by *local bindings*[1]. The interface offered by the meta-model then supports operations to inspect and adapt the graph structure, i.e. to view the structure of the graph, to access individual components in the graph, and to adapt the graph structure and content. This meta-model is particularly useful when dealing with binding components [Blair98]. In this context, the composition meta-model reifies the internal structure of the binding in terms of the components used to realise the end-to-end communication path. For example the component graph could feature an MPEG compressor and decompressor and an RTP binding component. The structure can also be exposed recursively; for example, the composition meta-model of the RTP binding might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol. It is argued in [Fitzpatrick98] that open bindings alone provide strong support for adaptation.

3. *The Environmental Meta-model.* The environmental meta-model supports the reification of activity related to a particular interface of a component. In terms of middleware, this equates to functions such as message arrival, enqueueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [Watanabe88, McAffer96]. At present, our architecture offers a simple environmental meta-model enabling the insertion of pre- and post- methods. Such a mechanism can then be used to introduce, for example, additional levels of distribution transparency (such as concurrency control) or to insert functions such as security managers or compression components.

4. *The Resources Meta-model.* Most reflective languages and systems restrict their scope to the above styles of reflection. In experiments, however, we have identified a significant weakness of this approach, namely that we have no means of accessing the level of resources and resource management in the system. This is a particular problem for mobile, multimedia and real-time systems where it is often important to be aware of the resources currently available at a given node (e.g. if it is intended to introduce a new software compression component). We therefore introduce a fourth meta-model, referred to as the resource meta-model [Blair99a].

[1] The RM-ODP inspired concept of local binding is crucial in our design, providing a language-independent means of implementing the interaction point between interfaces within a given address space.

This meta-model supports the reification of resource creation, scheduling and, more generally, management. The meta-model provides access to a set of components representing resources, together with the associated managers. As with other meta-models, it is then possible to either inspect or adapt activity associated with resources. For example, it is possible to insert monitors to capture statistics on the effectiveness of a thread scheduling policy and then possibly change this policy based on the information collected.

The first two meta-space models support what is often referred to as structural reflection in the literature, whereas the latter two models offer behavioural reflection [Watanabe88].

The complete Open-ORB architecture is summarised in figure 1. This highlights the recursive nature of the architecture in that each meta-space is populated by components that, consequently, have their own meta-space models, etc.

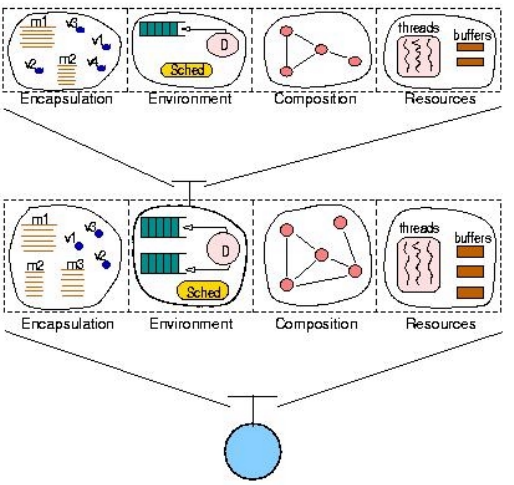

Fig. 1. The Structure of Meta-Space in Open-ORB

An implementation of the Open-ORB architecture has been carried out using Python. In addition, a second implementation is now underway using a lightweight and efficient reflective component model, based on COM. Further information on the reflective middleware architecture and associated implementation work can be found in the literature [Blair98, Blair99a, Costa98].

2.3 Analysis

The Open-ORB architecture described above provides a principled means of supporting both inspection and adaptation of the underlying middleware platform. Through inspection, the programmer can discover information about the structure of a component, together with any sub-components used in its construction. The

programmer can also analyse the behaviour of key parts of the system and resource usage through the insertion of monitoring behaviour. Based on this, the programmer can then make a number of fine or course grained changes to the underlying platform. For example, the programer can locate a key compression component and request a change to the compression strategy (e.g. drop B-frames in MPEG). Alternatively, he/she can change the compression and matching decompression component completely (e.g. to H.263). Other strategies include allocating more resources or changing the resource management policy for a subset of resources, or extending the system with new capabilities (e.g. encryption).

As can be seen, this *procedural* approach provides great flexibility to the programmer. Indeed, as argued in the introduction, there is often too much flexibility. At present, this is constrained only by (run-time) type checking on creating a local binding between components. While offering a degree of safety, this is clearly not sufficient. What we therefore require is a mechanism that i) can delegate responsibility for adaptation to a (trusted) management entity or entities, ii) can offer a more declarative view of adaptation, and iii) can ensure the architectural integrity of the system before, during and after re-configurations. This is precisely the problem that is addressed in this paper, where we investigate the role of software architecture, in general, and the Aster techniques and tools, in particular, in providing this level of robust adaptation management.

3 Software Architecture for Component-Based Middleware

3.1 What is Software Architecture?

Research effort in the software architecture domain aims at reducing costs of developing large, complex software systems. Towards that goal, formal notations are being provided to describe software architectures, replacing the usual informal description of software architectures in terms of box-and-line diagrams [Perry92]. These notations are generically referred to as *Architecture Description Languages* (ADLs). Basically, an ADL allows the developer to describe the gross organisation of the system in terms of coarse-grained architectural elements, thus abstracting away from implementation details. Prominent elements of a software architecture subdivide into the three following categories:

1. *components* that abstractly define computational units written in any programming language,
2. *connectors* that abstractly define types of interactions between components (*e.g.*, pipe or client-server), and
3. the *configuration* that defines an application structure in terms of the interconnection of components through connectors.

The interested reader is referred to [Medvidovic97] for a survey of existing ADLs and associated CASE tools.

3.2 The Aster Approach

The main goal of the Aster Project is to apply ideas from the software architecture community to the field of component-based middleware. More specifically, the aim of the work is to support the systematic synthesis of middleware configurations from architectural descriptions. Through this approach, it is then possible to match the services offered by the middleware platform to the demands of a given application domain. In more detail, the input of the synthesis process is an architectural description of the application that, apart from specifying the overall structure of the application in terms of components and connectors, also includes a specification of the required properties of the middleware connectors that mediate the interaction among the components, i.e. *non-functional properties* required by the application (such as scalability, real-time performance, etc). The output of the process is a middleware configuration (e.g. a component graph as exposed by the compositional meta-model associated to the middleware in terms of Open-ORB) consisting of components implementing services (e.g. OMG's CORBA services, Open-ORB's constituent components) that are interconnected through a specific connector that corresponds to the underlying middleware platform (e.g. OMG's Object Request Broker, Open-ORB's base binding components). Interestingly, the whole configuration can also be viewed as a connector (with added value) as it serves to support interactions between application components (e.g. Open-ORB's binding components defined through composition). Given this, a recursive approach to architectural description is required as already illustrated by the definition of the Open-ORB's compositional meta-model.

In more detail, the Aster environment for the systematic synthesis of middleware relies on the following key elements:

1. *An ADL for the description of both application and middleware software architectures.* Any architecture description may further embed the specification of the non-functional properties that are either provided or required by the constituting software elements, using the framework given below.

2. *A framework for the formal specification of properties offered by middleware configurations.* The framework is based on linear temporal logic and has been used so far for the specification of properties relating to dependability [Saridakis99], security [Bidan98] and transactions [Zarras98]. This framework is central to the Aster environment as it forms the basis for reasoning about the matching of available middleware configurations to the properties required by applications. Basically, a middleware configuration matches application requirements if the properties enforced by the configuration, as specified by the corresponding logic formula, imply the non-functional properties required by the application. In addition, the framework is used for structuring a *repository* of available middleware architectures according to a lattice structure, which encodes the refinement relationship holding over properties provided by available middleware elements. This repository effectively provides a pool of tried and tested software architectures offering a variety of non-functional behaviour.

3. *A tool for the integration of a chosen middleware configuration retrieved from the repository with the application that initiated the search.*

As an example of Aster usage, consider an application that requires transactional support from the underlying middleware. Given this, it is necessary to search the Aster repository for an appropriate match. A possible repository structure is depicted in figure 2. In this repository, all the architectures at the bottom of the figure match the application's requirements, with each corresponding to a different middleware style (i.e. EJB, CORBA, DCOM). In the remainder of this paper, we will be using the CORBA-style architecture of transactional middleware for illustration purpose. This architecture is composed of the CORBA ORB together with OTS (Object Transaction Service) and CCS (Concurrency Control Service) services. Due to the lack of space, we do not provide here the Aster ADL description of the above architecture nor formal specification of related non-functional properties; the interested reader is referred to the aforementioned references for detail. However, let us notice that property specifications within the ADL are given in terms of textual names in order to simplify the developer's task. However, these should have a corresponding formal definition within the Aster environment, which stores each property as a pair giving the name of the property and the corresponding temporal logic formula.

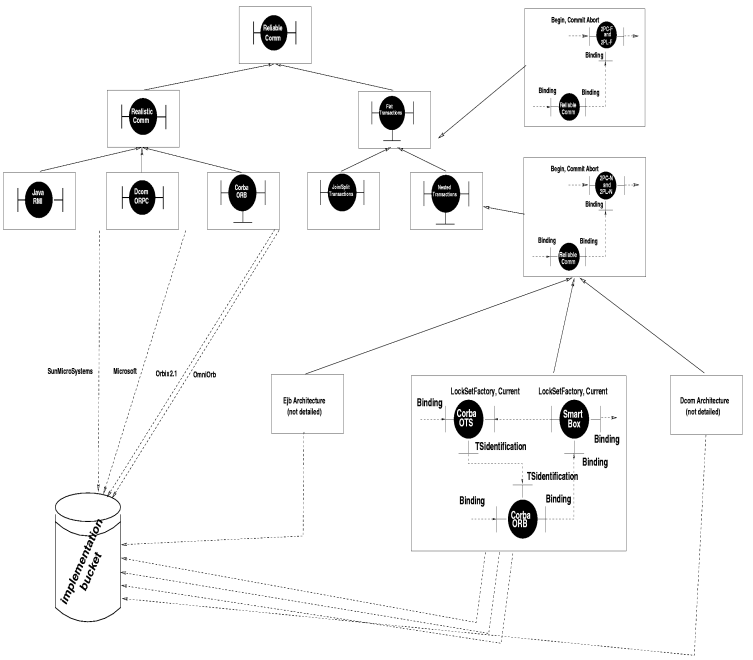

Fig. 2. A Middleware Repository

Each middleware architecture available within the repository may actually represent a number of concrete middleware configurations, depending on available implementations for the constituting components. Such configurations are stored within the *implementation bucket*, as depicted in figure 2. When multiple configurations implement a retrieved middleware architecture, the designer is

requested to select the one that best suits the application regarding environmental parameters such as available resources.

3.3 Analysis

Since the early design of the Aster environment presented in [Issarny96], prototype implementation has been carried out so as to assess the proposed approach. The current Aster prototype relies on the STeP theorem prover [Bjorner98] for the implementation of property matching when either inserting or seeking an element in the middleware repository. Experimentation has generally been carried out within the CORBA framework. Although work is still needed for improving the environment's usability (*e.g.* the use of formal methods by software designers may be seen as a challenge), our experience shows that the approach actually eases the construction of middleware, contributes to software robustness, and fosters software and design reuse.

The Aster approach has focussed on the synthesis of middleware platforms whose properties are defined at design time. However, it does not deal with the adaptation of middleware at run-time, resulting either from evolution of the application design or from changes in the underlying infrastructure. Hence, it does not yet provide the mechanisms we seek for reflective middleware architectures (as identified in section 2.3 above). We believe though that the Aster approach *a priori* provides a way to alleviate the penalty of middleware adaptation. Furthermore, the Aster approach naturally extends to accommodate the seamless nature of the Open-ORB reflective architecture, i.e. the use of components above and within the middleware platform. In particular, we have already seen how the recursive nature of architectural description in Aster can accommodate such structures. Before examining extensions to Aster, it is helpful, however, to examine the issue of adaptation more closely.

4 A Closer Look at Adaptation

In order to manage the adaptation process, it is necessary to provide support for each of the four phases presented below.

1. *Identifying conditions for initiating middleware changes.* We consider two separate triggers for middleware adaptation: i) revision of the requested middleware properties, and ii) modification of the environment. The former type of change typically follows evolution of the application's design. The latter type of change results from the evolution of either the execution or the software environment, requiring the corresponding revision of the middleware implementation.
2. *Computing the middleware architecture complying to a requested change.* Once it has been determined that a change must take place, it is then necessary to compute the corresponding impact on the middleware configuration. Note that such changes may range from trivial component/connector exchanges to major structural changes, including revision of the middleware architecture.
3. *Detecting when it is safe to actually change the current middleware platform.* The adaptation process must ensure that an application remains in a consistent state throughout the adaptation of the underlying middleware configuration. For

example, in the case of a platform offering transactional support, the changes to the configuration should not disrupt the execution of ongoing transactions. As such, it is necessary to detect when it is safe to carry out such changes. In doing so, the primary design concern is to find the right trade-off between minimal application disruption and timely exchange.

4. *Adapting the current middleware.* This step is highly dependent on the support provided by the underlying middleware platform, e.g. in terms of reflective facilities (if available). In terms of Open-ORB, for example, the adaptation process would exploit the different meta-space models, as appropriate. For example, the adaptation may require access to a compositional meta-model to exchange one component for another. Adaptation may also require additional support in terms of, for example, the transfer of state information between configurations. Again, this can often be supported by underlying reflective facilities.

We consider the support provided for each of these phases in some detail below.

5 Extensions to Aster to Support Architectural Adaptation

5.1 Overall Approach

There are a number of ways of approaching the problem of adaptation as defined above. The most constraining solution consists of allowing only planned changes. In such a case, the application designer must provide a specification of the range of middleware configurations, together with the conditions for switching between configurations. In such an approach, the number of possible configurations is clearly finite and pre-determined. In contrast, the less constraining solution consists of tolerating any change at runtime, almost independently of design decisions. We believe that both these solutions are unsatisfactory. The former is impractical, as there is no way to exhaustively anticipate changes to the middleware that may be required. The latter option is more flexible from the standpoint of supported changes. However, this is at the expense of software robustness because there is no way to guarantee that the middleware adaptation still complies to the application design. It seems then necessary to constrain the range of accepted changes at runtime whilst not requiring an exhaustive list of supported changes.

This debate relates very closely to research in the area of dynamic reconfiguration [Edler92]. Briefly stated, work in this area investigates support for expressing and achieving reconfiguration of applications at runtime. Within this field, two general approaches have been proposed: *programmed* and *evolutionary reconfiguration*[2]. Programmed reconfiguration is a form of planned evolution where the set of possible changes and conditions for changing are prescribed at design time (typically by associating state predicates with each system configuration, e.g. see [Barbacci93]). In evolutionary reconfiguration, in contrast, a configuration manager responds to requests for reconfiguration (typically from the user) and ensures that the necessary changes are carried out. Our proposal is for an evolutionary approach whereby a form

[2] A number of hybrid solutions have also been defined.

of configuration manager, which we refer to as an *adaptation manager*, takes responsibility for all changes and also ensures that architectural integrity is maintained. Further details of the role of the adaptation manager will emerge from the discussion below.

5.2 The Four Steps Revisited

5.2.1 Conditions for Initiating Changes

Technical Approach

As stated above, there are two distinct cases that must be considered, i.e. revision of the requested non-functional properties and modification of the environment.

Revision of non-functional properties can only be made known to the adaptation manager through user interaction (typically, designer intervention) and hence this requires a corresponding GUI tool. Such changes are expected to be infrequent as they map on to evolution of the general design. Architectural constraints are maintained by extending applications architectural descriptions with denotations of the *weakest properties* that must be satisfied by the middleware configuration supporting a given application. The adaptation manager can then accept such changes as long as the required weakest properties remain satisfied. Note that, if we set the weakest properties to true, then any changes will be tolerated. However, allowing such flexibility is under the responsibility of the application's designer who decides to not constrain changes and hence to favor flexibility over robustness. In general terms, this proposal favors software robustness without unduly restricting adaptation; essentially, the application designer, based on knowledge of the application domain and environmental conditions, determines the right trade-off between flexibility and robustness.

In contrast, modifications of the environment can be initiated either through user intervention or, more likely, as a result of events generated by underlying monitoring components. Crucially, such monitoring components can be inserted dynamically into the middleware configuration using reflection. For example, a monitor can be inserted in the resource meta-space to report on the current processor load. Similarly, the compositional meta-space can be used to insert a monitor into a binding object to report on inter-arrival times of video frames. The frequency of such changes is expected to be somewhat greater than that for non-functional properties.

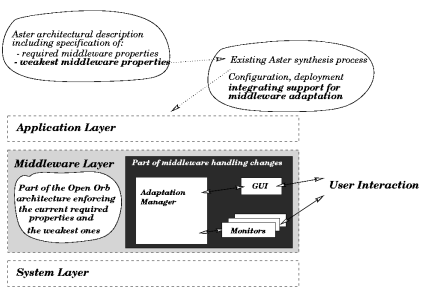

Fig. 3. Support for Initiating Change

Figure 3 summarises the support required for initiating middleware adaptation (with highlighting in boldface).

Required Changes to Aster
The first change is the incorporation of *weakest properties* specifications into the Aster framework. This can easily be achieved. The Aster ADL already supports the specification of non-functional properties required from a middleware configuration (as used by the synthesis process). Given this, it is sufficient to enrich such a description with a clause prescribing the weakest properties that should always be enforced by the middleware platform (written using temporal logic). The middleware properties specified at design time should imply the weakest ones, a condition that can easily be checked using the Aster toolset.

The proposed changes also require the introduction of an *adaptation manager*. While the provision of such a manager relates more to the middleware infrastructure than to the Aster environment, we present it here as an Aster extension as it is central to the adaptation process.

5.2.2 Computing the New Middleware Configuration

Technical Approach.
Once the adaptation manager has checked the required properties, the corresponding middleware architecture must be computed. This is easy to achieve using Aster, as it amounts to exploiting the *synthesis process*. In general, though, this is a difficult problem, potentially requiring complex analysis of the current configuration, the desired non-functional properties, and the set of possible changes that can be made to achieve such properties. Considering the Open-ORB infrastructure, this requires us to identify the necessary updates within the current component graph managed by the compositional meta-model. This is also made more difficult because of potential dependencies between components in the current configuration, e.g. in the example of transactional middleware in section 3.2., the OTS and CCS services are closely inter-related at the implementation level. The Aster approach does not enable us to directly deal with such dependencies due to the highly abstract description of middleware properties. In addition, the synthesis process would need to be adapted so as to integrate computation of a middleware configuration with respect to both an existing configuration and required properties and not just middleware properties. Investigation of this is an area for future work. Currently, we adopt a more pragmatic approach, in order to minimise Aster extensions whilst supporting robust middleware adaptation.

Dealing with changes in the required non-functional properties is quite straightforward. In this case, the synthesis process is carried out with the new abstract properties as input, the result being the target configuration. In other words, this means that the component graph managed by the compositional meta-model is fully replaced by the graph corresponding to the newly synthesized middleware configuration. Dealing with environmental changes is however more complex and requires further discussion. In this case, it is crucial to be able to distinguish between middleware configurations implementing a given architecture with given non-

functional properties but differing in terms of *environmental parameters*[3]. For example, one architecture might be more appropriate in a resource rich environment with plentiful bandwidth, processing capacity, battery life, etc, whereas other solutions may be better in the cases of drops in bandwidth or problems with battery life. It is important not to place bounds on the criteria taken into account in the selection process of middleware configurations implementing a given architecture. The only prerequisite is to ensure that the various concrete middleware configurations implementing a single middleware architecture can be compared and hence all prescribe the same set of selection criteria (or at least have a set of common selection criteria). In addition, it should be possible to assess the environment features with respect to those common criteria so as actually be able to select the most suitable configuration. In the case where multiple configurations are still eligible, one is chosen randomly. We now consider the impact of this on Aster.

Required Changes to Aster.
The current Aster repository enables us to distinguish middleware architectures with respect to the abstract properties they provide. However, Aster does not enable us to distinguish between valid implementations that differ in terms of environmental parameters. In order to extend this, we introduce the concept of a *sub-repository* (possibly empty) which gives all the eligible configurations for implementing a given architecture, hence making the Aster repository close to a multi-dimensional database. A sub-repository is further defined by a vocabulary of selection criteria and, for each term of the vocabulary, a value needs to be specified for each embedded (concrete) configuration. More formally, let P denote the power set, a sub-repository is denoted by:

```
Concrete: P (CONFIGURATION)
Criteria: P (STRING)
Value: Criteria → TYPE
```

and each element of `Concrete`, which is a concrete configuration, is such that for all the elements of `Criteria`, it defines a function:

```
Valid: Criteria → ConfigElement → TYPE  ∪ ⊥
```

where `ConfigElement` is the set of configuration elements (i.e. component, connector implementations) and `TYPE` should be equal to `Value(c)` with `c` being the criterion under consideration. The `Valid` function then serves to identify for each relevant criterion, the value of the environment parameter that is required for each configuration element, ⊥ meaning that the criterion does not apply for this specific element.

As an example, let us consider the CORBA-based transactional middleware architecture that was discussed in Section 3.2. Although not part of the standard, we

[3] A second important consideration is the cost of adaptation. As stated earlier, this can vary between changing properties of a particular component, through to major reconstruction of the configuration. This should be taken into account in the decision making process, e.g. by associating a cost metric with each possible transition. This has not yet been addressed in our work, although this is a crucial area for future research.

may consider different implementations for the OTS service based on existing results in the area of transaction management: transaction commitment using either a 2-phase or a 3-phase protocol according to the frequency of failure occurrences, or transaction management supporting transaction execution over a disconnected laptop. These implementations relate to the same middleware architecture. However, the corresponding configurations differ depending on the following set of criteria: {*MTBF, ConnectionMode*}, where *MTBF* gives the Mean Time Between Failure of components and takes its values in the set of integers (i.e. `Value(MTBF) = INT`), and *ConnectionMode* gives the connection mode of components and takes its values in the set {*LAN, Wireless*} (i.e. `Value(ConnectionMode) = {LAN, Wireless}`). Hence, the middleware configuration made out of the OTS implementing the 2-phase commit protocol is valid as long as the value of the *MTBF* of each component exceeds a given threshold *t*, and the value of the ORB *ConnectionMode* is equal to *LAN*. In the same way, the 3-phase commit implementation is eligible when at least one of the *MTBF*s is lower than *t*, the value of ORB's *ConnectionMode* being still *LAN*. Finally, the middleware configuration composed of the OTS implementation supporting disconnected transactions is valid when the *ConnectionMode* is equal to Wireless, whatever are the values of the *MTBFs*.

5.2.3 Detecting When it is Safe to Adapt the Configuration

Technical Approach

Adaptation of the middleware configuration should be carried out in a way that maintains the overall system in a consistent state. At the time of middleware change, there might be requests, qualified as *pending*, which are issued through the old middleware and for which the requested properties (i.e. properties enforced by the old middleware) are not yet satisfied. For the middleware exchange to be correct, it must be ensured that, for all the requests pending at the time of the integration, the new middleware satisfies the non-functional requirements for those particular requests. A trivial pre-condition upon the middleware state for achieving exchange is then to have no pending request. Such a state is qualified as *idle*. Requiring an idle state for the middleware prior to carrying out adaptation is at the expense of timeliness. Although the execution environment may enforce reaching an idle state by selectively blocking requests (as long as this does not prevent termination of ongoing requests) [Kramer90], this may take quite a long time. A weaker pre-condition is then to require a *safe state*, which guarantees that for all the pending requests at the time of the exchange, these requests will eventually be terminated and their non-functional properties will be satisfied.

Middleware exchange under safe state detection requires the following capabilities from the middleware, so as to ensure the correct termination of pending requests: i) the non-functional properties provided by the new middleware should imply the properties of the current configuration, and ii) part of the state of the current middleware configuration relating to pending requests should be mapped onto the state of the new middleware. The former condition always holds when the middleware adaptation is due to environmental changes. On the other hand,

adaptation due to new non-functional requirements must be dealt with on a case-by-case basis. The latter condition requires the middleware to be able to import and export its state (e.g. [Hofmeister93]), a function that can be directly supported by reflective middleware (e.g. see [Killijian99]). Note that meeting this condition is specific for each pair of middleware configurations involved in the exchange (regarding implementation of the state mapping function). We cannot afford the definition of an exhaustive mapping function, covering all the possible middleware adaptations, due to either changing non-functional requirements or environmental parameters. However, such a function may be devised for the specific case of environmental changes: the range of middleware adaptations is bounded in this case by the number of configurations stored within the corresponding sub-repository. Consequently, middleware adaptation due to environmental changes is carried out under safe state detection while idle state detection is used in the other cases.

Required Changes to Aster
Support for either safe or idle state detection does not require changes to Aster per se, apart from enriching sub-repository management for storing and retrieving *state mapping functions*. However, this imposes a number of requirements upon the middleware infrastructure, which are easy to handle given a reflective middleware.

State mapping functions should be introduced within sub-repositories for pairs of configurations; these functions are then used whenever the middleware changes relate to the corresponding pairs. A *safe state detector* should also be provided with each mapping function. This safe state detector can then be installed when the change is requested, so as to detect when the safe state is reached. In the absence of this information, idle state detection should be carried out by default.

Unlike safe state, the definition of idle state depends only on the properties of the old middleware configuration. Hence, it is possible to build the required middleware configuration with an *idle state detection* mechanism already in place. A second important property of the idle state definition is that it is based only on predicates that qualify the interaction between the middleware configuration and the application components, which derives from the definition of middleware properties that are always given in terms of component interactions. This makes it possible to implement a module that detects whether a middleware is in an idle state with respect to a specific property only by monitoring the interaction between the middleware and the components. Since such an idle state detector is independent of the specific middleware configuration, it can be stored in the middleware repository together with each architecture, and integrated into the middleware configuration during the middleware synthesis.

We do not further detail the implementation of idle and safe state detectors. These exploit the meta-space models for inspecting the middleware state.

5.2.4 Carrying out the Adaptation

Technical Approach
Once the above adaptation decision has been taken, the task of re-configuration is quite straightforward. In terms of implementation, re-configuration will make

extensive use of the different meta-space models in carrying out the required adaptation. For example, for fine-grained adaptations, the compositional meta-model can be used to identify encapsulated components. Once identified, direct changes of the properties of a given component can then be made. For more course-grained adaptations, changes to the component graph can be requested. Note that support is provided to enable *smooth* transition between different configurations. For example, consider the transaction example given in section 5.2.2. In this example, support is provided to enable the new configuration to be constructed and primed in parallel with the operation of the old configuration. A hand-over can then be made as a single action, thus minimizing the disruption to service. Further information on such facilities can be found in [Fitzpatrick99].

Required Changes to Aster
The main change is to associate an *adaptation script* for pairs of configurations in the sub-repository. This is then invoked when changes are due to environmental conditions. For other changes (i.e. those due to evolving non-functional requirements), it must be assumed that the code for adaptation is provided by the user/ designer. Note that, if an adaptation script is not provided for a given pair, then it is necessary for the Aster environment to replace completely the old configuration with the new one. In this respect, adaptation scripts can be viewed as an important optimisation. For example, in many cases the adaptation could be quite lightweight, e.g. changing the properties of a single component. It would then not make sense to build a complete new configuration. Note also that this adaptation script should invoke the state mapping function (as appropriate) as part of its operation.

5.3 Analysis

This section has considered extensions to Aster to deal with adaptation, both in terms of the overall evolution of the design, e.g. to include new non-functional properties, or in response to changes in the underlying environment. Briefly, the required changes are as follows:

1. the incorporation of weakest properties specifications into architectural descriptions, to place constraints on the range of possible responses to new requirements,
2. the inclusion of environmental parameters in architectural descriptions to support the selection of the most appropriate configuration following changes in environmental conditions, through the introduction of a sub-repository of configurations per middleware architecture.
3. the introduction of an adaptation manager as a key middleware service,
4. the definition of safe state or idle state detectors, again provided through the sub-repositories, to indicate when it is safe to change,
5. the provision of a set of associated state mapping functions in sub-repositories to perform the required state transfers between pairs of configurations, and
6. the provision of adaptation scripts, again in sub-repositories, to carry out the necessary steps to switch between pairs of configurations.

Let us further recall that the proposed extensions have been addressed assuming a reflective middleware platform, which gives the necessary support for actually changing middleware configurations in an efficient way. We believe that this approach provides a good compromise between flexibility and robustness. In particular, it enables the designer to constrain the supported middleware changes with respect to the application's requirements, and it relieves the application programmer from writing code specific to middleware adaptation.

6 Related Work

Among existing ADLs, DARWIN [Magee92], DURRA [Barbacci93], and C2 [Medvidovic96] support dynamic configuration of applications. DARWIN provides language primitives allowing the user to describe the dynamic instantiation, removal and re-binding of ports. A DARWIN specific configuration manager is used to apply changes into a running configuration. The basic language primitives that describe dynamic instantiation and port re-binding can be used by both the application components and the application administrator. In DURRA, runtime configuration changes are modeled in terms of reconfiguration rules. A reconfiguration rule describes a transition from one configuration to another. Possible transitions are described at design time and are applied at runtime by a set of configuration managers, with each manager being responsible for changing a particular cluster of component instances. In C2, configuration changes are modeled as sequences of primitive reconfiguration actions like component creation, removal, welding, etc. The language provides corresponding primitives for the expression of such reconfiguration actions. All the aforementioned ADLs provide support for creating, removing and re-binding components of the application. However, none of them confronts changes in the middleware connector that mediates the interactions among the components.

There are many projects investigating adaptation/ dynamic QoS management in middleware platforms. BBN's QuO (QUality Objects) project [Vanegas98] is perhaps the closest in approach to our own work in that it adopts a similar open implementation philosophy. QuO, however, adopts an approach reminiscent of aspect-oriented programming [Kiczales97] to specify different aspects of QoS support using a range of specialised (high-level) languages. As such, it provides a more declarative, as opposed to procedural, approach to QoS management. Similarly, researchers at the University of Illinois have developed a task control model to support dynamic reconfiguration in middleware platforms [Li99]. This project is complementary to the Dynamic Tao project mentioned earlier [Román99], exploiting the openness of this underlying platform for more fine-grained control. Their approach is based on the use of a fuzzy logic inference engine together with a rule base of possible adaptations. The crucial difference between these projects and our proposals is that our work introduces overall architectural constraints based on explicit modeling of the software architecture. In previous research, the Lancaster authors have also developed a QoS management scheme for adaptation based on the use of timed automata [Blair99b]. While providing some support for adaptation, this work does not have the breadth of coverage of the work described in this paper. Finally, a number of other middleware projects include important elements of

configurability and re-configurability including DIMMA [Donaldson98], Tao [Schmidt97] and Jonathan [Dumant97], again, however, with limited coverage of the range of issues raised by adaptation.

7 Concluding Remarks

This paper has argued that the next generation of middleware platforms will need to be more configurable and re-configurable. Furthermore, it is argued that reflection, together with component technology, provide the right technical solution to meet these requirements. However, such approaches have significant problems in terms of maintaining the integrity of the underlying configuration. We argue that such problems can be overcome by adopting ideas from the software architecture community.

More specifically, we have explored the role of the Aster framework in supporting dynamic adaptation in the context of the Open-ORB middleware platform. Following a detailed examination of adaptation, it was concluded that Aster could usefully be extended to meet our requirements. The key extensions to Aster include the incorporation of weakest properties and environmental parameters in architectural descriptions to accommodate re-configurations due to changing non-functional parameters and environmental conditions respectively.

The most significant contributions of this paper are:

1. the development of techniques to constrain changes in an open environment, and
2. the extension of software architecture techniques to accommodate dynamic change.

The work also has a more general contribution to make to the automatic synthesis of component-based configurations, and their subsequent monitoring and adaptation, based entirely on architectural descriptions.

The Aster toolkit is fully developed, although the changes described above have not yet been incorporated. In addition, as mentioned above, prototype implementations of Open-ORB have been developed. Although the proposed framework still needs to be integrated within Aster for further validating our approach, it is our belief that such an integration should not pose any difficulty, although it would be demanding in terms of implementation effort. However, work is still needed for a thorough assessment of our approach from a practical standpoint. In particular, despite the benefits of our approach with respect to sofware system robustness, it should be demonstrably efficient for it to be used in practice. We are currently working on such an issue by further investigating support for efficient middleware adaptation regarding both the implementation of the Aster extensions and the OpenORB infrastructure. We are also investigating the application of our framework to multimedia systems which are among the most demanding in terms of middleware adaptation due to their highly demanding resource usage and increasing use over heterogeneous platforms ranging from powerful workstations to wireless PDAs.

Acknowledgements

Research in the Open-ORB Project is partly funded by CNET, France Telecom (CNET Grant 96-1B-239) and partly by the EPSRC together with BT Labs (Research Grant GR/K72575). We would like to thank our collaborators for their support. Particular thanks are due to Jean-Bernard Stefani and his group at CNET, and also Ian Fairman, Alan Smith and Steve Rudkin at BT Labs. We would also like to acknowledge the contributions of a number of researchers at Lancaster to the Open-ORB Project, namely Mike Clarke, Fabio Costa, Geoff Coulson, Tom Fitzpatrick, Hector Duran, Nikos Parlavantzas and Katia Saikoski.

Research in the Aster project has been partly funded by CNET, France Telecom, and partly by the Esprit LTR C3DS project. Particular thanks are due to former members of the Aster project, namely C. Bidan and T. Saridakis.

Finally, this joint paper would not have been possible without the support of INRIA, which enabled Gordon and Lynne Blair to visit the Solidor Group during Easter, 1999.

References

[Barbacci93] Barbacci, M., C. Weinstock, D. Doubleday, M. Gardner, R. Lichota, “DURRA: A Structure Description Language for Developing Distributed Applications”, Software Engineering Journal, pp 83-94, March 1993.

[Bidan98] Bidan, C., V. Issarny, “Dealing with Multi-Policy Security in Large Open Distributed Systems”, Proceedings of the 5th European Symposium on Research in Computer Security, pp 51-66, September 1998.

[Bjorner98] Bjorner N., A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. Pichora, H.B. Sipma, T.E. Uribe, “STeP: The Stanford Temporal Prover Educational Release”, Stanford University, 1.4-a edition, July 1998.

[Blair97] Blair, G.S., J.B. Stefani, “Open Distributed Processing and Multimedia”, Addison-Wesley, 1997.

[Blair98] Blair, G.S., G. Coulson, P. Robin, M. Papathomas, “An Architecture for Next Generation Middleware”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), Springer, 1998.

[Blair99a] Blair, G.S., F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, J.B. Stefani, “The Design of a Resource-Aware Reflective Middleware Architecture”, Proc. of the 2nd Int. Conference on Meta-Level Architectures and Reflection (Reflection’99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp 115-134, 1999.

[Blair99b] Blair, G.S., A. Amdersen, L. Blair, G. Coulson, "The Role of Reflection in Supporting Dynamic QoS Management Functions", Proceedings of the IEEE/IFIP International Workshop on Quality of Service (IWQoS'99), London, June 1999.

[Costa98] Costa, F., G.S. Blair, G. Coulson, “Experiments with Reflective Middleware”, Proceedings of the ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP’98 Workshop Reader, Springer-Verlag, 1998.

[Donaldson98] Donaldson, D., M. Faupel, R. Hayton, A. Herbert, N. Howarth, A. Kramer, I. MacMillan, S. Waterhouse, “DIMMA - A Multi-media ORB”, Proc. Middleware ‘98, The Low Wood Hotel, Ambleside, England, September 1998.

[Dumant97] Dumant, B., F. Horn, F. Dang Tran, J.B. Stefani, “Jonathan: An Open Distributed Processing Environment in Java”, Proc. IFIP International Conference on Distributed

Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Edler92] Edler, M., J. Wei, "Programming Generic Dynamic Reconfiguration for Distributed Applications", Proceedings of the International Workshop on Configurable Distributed Systems, pp 68-79, March 1992.

[Fitzpatrick98] Fitzpatrick, T., G.S. Blair, G. Coulson, N. Davies, P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings", Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE, 1998.

[Fitzpatrick99] Fitzpatrick, T., "Open Multimedia Component Middleware for Adaptive Distributed Applications", PhD Thesis, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, UK, September 1999.

[Hayton97] Hayton, R., "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, 1997.

[Hofmeister93] Hofmeister, C., J. Purtilo, "Dynamic Reconfiguration for Distributed Systems, Adapting Software Modules for Replacement", Proceedings of the 13th IEEE International Conference on Distributed Computing Systems (ICDCS'93), May 1993.

[Issarny96] Issarny, V., C. Bidan, "Aster: A Framework for Sound Customization of Distributed Runtime Systems", Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS'96), May 1996.

[Kiczales91] Kiczales, G., J. des Rivières, D.G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.

[Kiczales97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, J. Irwin, "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag, June 1997.

[Killijian99] Killijian, M.O., J.C. Ruiz-Garcia, J.C. Fabre, "Using Compile-Time Reflection for Objects' State Capture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp 150-152, 1999.

[Kramer90] Kramer, J., J. Magee, "The Evolving Philosophers Problem", IEEE Transactions on Software Engineering, Vol. 15, No. 1, pp 1293-1306, November 1990.

[Ledoux97] Ledoux, T., "Implementing Proxy Objects in a Reflective ORB", Proc. ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, 1997.

[Li99] Li, B. and K. Nahrstedt, "Dynamic Reconfiguration for Complex Multimedia Applications, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (IEEE Multimedia Systems '99), Florence, Italy, June 7-11, 1999.

[Magee92] Magee, J., N. Dulay, J. Kramer, "Structuring Parallel and Distributed Systems", Proc. of the International Workshop on Configurable Distributed Systems, March 1992.

[McAffer96] McAffer, J., "Meta-Level Architecture Support for Distributed Objects", In Proceedings of Reflection 96, G. Kiczales (end), pp 39-62, San Francisco; Also available from Department of Information Science, The University of Tokyo, 1996.

[Medvidovic96], "ADLs and Dynamic Architecture Changes", Proceedings of the 2nd ACM SIGSOFT International Software Architecture Workshop (ISAW-2), pp 24-27, October 1996.

[Medvidovic97] N. Medvidovic, R. Taylor, "A framework for classifying and comparing architecture description languages", Proc. of the Joint European Software Engineering - ACM SIGSOFT Symposium on Foundations of Software Engineering, pp 60-76, September 1997.

[Okamura92] Okamura, H., Y. Ishikawa, M. Tokoro, "AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework", Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[OMG99] The Common Object Request Broker: Architecture and Specification versions 3.0., available at http://www.omg.org/.

[Perry92] D. E. Perry, A. L. Wolf, " Foundations for the Study of Software Architecture ", ACM SIGSOFT Software Engineering Notes, 17(4), pp 40-52, October 1992.

[Román99] Román, M., F. Kon, R. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case", Proceedings of the ICDCS'99 Workshop on Middleware, Austin, Texas, May-June 1999.

[Saridakis99] Saridakis, T., V. Issarny, "Developing Dependable Systems Using Software Architecture", Proceedings of the 1st Working IFIP Conference on Software Architecture, pp 80-104, February 1999.

[Schmidt97] Schmidt, D.C., R. Bector, D. Levine, S. Mungee, G. Parulkar, "An ORB End System Architecture for Statically Scheduled Real-Time Applications", Proceedings of the Workshop on Middleware for Real-Time Systems and Services", IEEE, San Francisco, 1997.

[Smith82] Smith, B.C., "Procedural Reflection in Programming Languages", PhD Thesis, MIT, Available as MIT Computer Science Technical Report 272, Cambridge, Mass., 1982.

[Szyperski98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Vanegas98] Vanegas, R., J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Watanabe88] Watanabe, T., A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language", In Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of "Object-Oriented Concurrent Programming", A. Yonezawa, M. Tokoro (eds.), pp 45-70, MIT Press, 1987.

[Zarras98] Zarras, A., V. Issarny, "A Framework for Systematic Synthesis of Transactional Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), pp 257-272, Springer, September 1998.