Platform-independent dynamic reconfiguration of distributed applications

João Paulo A. Almeida[a], Marten van Sinderen[a], Luís Ferreira Pires[a] and Maarten Wegdam[a, b]
[a]*Centre for Telematics and Information Technology, University of Twente*
PO Box 217, 7500 AE, Enschede, The Netherlands
[b]*Lucent Technologies, Bell Labs Advanced Technologies EMEA Twente*
Capitool 5, 7521 PL, Enschede, The Netherlands
{alme,sinderen,pires}@cs.utwente.nl, wegdam@lucent.com

Abstract

The aim of dynamic reconfiguration is to allow a system to evolve incrementally from one configuration to another at run-time, without restarting it or taking it off-line. In recent years, support for transparent dynamic reconfiguration has been added to middleware platforms, shifting the complexity required to enable dynamic reconfiguration to the supporting infrastructure. These approaches to dynamic reconfiguration are mostly platform-specific and depend on particular implementation approaches suitable for particular platforms. In this paper, we propose an approach to dynamic reconfiguration of distributed applications that is suitable for application implemented on top of different platforms. This approach supports a platform-independent view of an application that profits from reconfiguration transparency. In this view, requirements on the ability to reconfigure components are expressed in an abstract manner. These requirements are then satisfied by platform-specific realizations.

1. Introduction

The reliance on distributed systems constrains the possibility of restarting them or taking them off-line. It is usually not acceptable, e.g., for economical or safety reasons, to cause major disruptions in the service provided by these systems [9]. The aim of dynamic reconfiguration [5, 6, 9, 17] is to allow a system to evolve incrementally from one configuration to another at run-time. Reconfiguration can be needed, e.g., because the resources the system is using will no longer be available, or the behaviour of the system needs to be adapted by replacing some of its components.

Developing systems that can be dynamically reconfigured is a complex task, since a developer must ensure that dynamic reconfiguration results in a correct and useful system. In recent years, support for different QoS (quality-of-service) mechanisms, including dynamic reconfiguration, load-balancing and replication mechanisms, has been added to middleware infrastructures [4, 16, 17]. This results in a shift in the complexity required to satisfy QoS constraints from the application to the supporting infrastructure. QoS mechanisms implemented in middleware are application-independent (i.e., generic to different applications) and to a large extent transparent to application developers (i.e., they hide from application developers the complexity required to achieve dynamic reconfiguration).

Ideally, it should be possible to leverage the benefits of transparent dynamic reconfiguration (and other transparent QoS mechanisms) to distributed applications regardless of the particular middleware platform on top of which these applications are implemented. However, most approaches to dynamic reconfiguration are platform-specific, in that they depend on mechanisms available on a particular middleware platform, or even on details of a specific implementation of a platform.

In this paper, we propose an approach to dynamic reconfiguration that enables the reuse of generic dynamic reconfiguration functionality in different middleware platforms, while maintaining the separation of application logic and dynamic reconfiguration concerns. When applicable, our approach profits from the availability of middleware extension mechanisms, but it does not depend on these mechanisms.

Our approach is based on the Model-Driven Architecture (MDA) [10, 11]. In MDA development, particular attention is paid to separately modelling and explicitly relating platform-independent and platform-specific aspects of a distributed application. A common pattern in MDA development is to define a platform-independent model (PIM) of a distributed application, and to apply (parameterised) transformations to this PIM to obtain one or more platform-specific models (PSMs). The main benefit of this approach stems from the possibility to derive different alternative PSMs from the same PIM depending on the target platform, and to partially automate the model transformation process and the realization of the distributed application on specific target platforms. Models can be described in languages such as UML or specializations of UML [14] or other suitable design languages.

In our approach, we prescribe the use of platform-independent models when developing distributed applications. In these models, requirements on the ability to reconfigure components are expressed in an abstract manner. These requirements are then satisfied by platform-specific realizations, in platforms that offer different levels of support to dynamic reconfiguration and different opportunities for extension. We also provide some criteria for choosing between different realization strategies.

This paper is further structured as follows: section 2 provides some background on platform-independence; section 3 presents how dynamic reconfiguration is supported in platform-independent modelling; and, section 4 discusses realizations of platform-independent models in different platforms. Finally, section 5 presents some conclusions and outlines some future work.

2. Platform-independent design

Platform-independence is a quality of a model that relates to the extent to which the model abstracts from the characteristics of particular technology platforms. In order to refer to platform-independent or platform-specific models, one must define what a platform is. For the purpose of this paper, we assume that distributed applications are ultimately realized in some specific object- or component-middleware technology that supports operation invocation and asynchronous message exchange, such as CORBA [12], .NET [8], and Web Services [18, 19]. Hence, a platform corresponds ultimately to some specific middleware technology. The goal of platform-independence is to facilitate the realization of a distributed application on top of different middleware platforms.

During platform-independent modelling, the application developer identifies some concerns that are postponed to platform-specific realization. These concerns determine the characteristics of what we call an abstract platform (as we have proposed in [2]). Capabilities of a concrete platform are then used during platform-specific realization to support the characteristics of the abstract platform. For example, if a platform-independent design contains application parts that interact through operation invocations, then support for operation invocation is a characteristic of the abstract platform. If CORBA is selected as a target platform, this characteristic can be mapped to CORBA operation invocations.

Characteristics of an abstract platform may be implied by the set of design concepts used for describing the platform-independent model of a distributed application. These concepts are often inherited from the adopted modelling language. For example, the exchange of "signals" between "agents" in SDL [7] may be considered to define an abstract platform that supports reliable asynchronous message exchange. These concepts may also be specializations of concepts from the adopted modelling language. For example, in UML 2.0 [14], the reliability characteristics of "signals" exchanged between "objects" is a semantic variation point. A UML Profile may specialize UML 2.0 and state that "signals" are exchanged reliably, thereby defining an abstract platform that supports reliable asynchronous message exchange.

Instead of implying an abstract platform definition from the adopted set of design concepts for platform-independent modelling, it may be useful or even necessary to define the characteristics of an abstract platform explicitly, resulting in one or more separate and reusable design artefacts. During platform-independent modelling, parts of a pre-defined abstract platform model may be composed with the model of the distributed application. For example, while UML 2.0 does not support group communication as a primitive design concept, it is possible to specify the behaviour of a group communication sub-system in UML. This sub-system is then re-used in the design of the distributed application. The abstract platform we present in section 3 is another example of this approach.

The different approaches to define an abstract platform are depicted schematically in Figure 1.

Explicitly identifying an abstract platform brings attention to *balancing* between two conflicting goals: (i) platform-independent modelling, and (ii) platform-specific realization. On the one hand, an abstract platform indicates directly the support available for designers during platform-independent modelling, and therefore, reflects the needs of application designers, including portability requirements. On the other hand, an abstract platform is established by considering the set of potential target platforms and their (common and diverging) characteristics; this bottom-up knowledge is useful to reduce the design space to be explored for platform-specific realization.

Our problem at hand is then reformulated into: (i) defining an appropriate abstract platform that supports dynamic reconfiguration transparently, and, (ii) defining transformations from a PIM of a distributed application that relies on this abstract platform to different target middleware platforms.

3. Support for dynamic reconfiguration in an abstract platform

Reconfiguration is specified in terms of entities and operations on these entities. The definition of entity depends on the level of granularity of reconfiguration. Examples of entities are objects, groups of objects, components, groups of components, sub-systems,

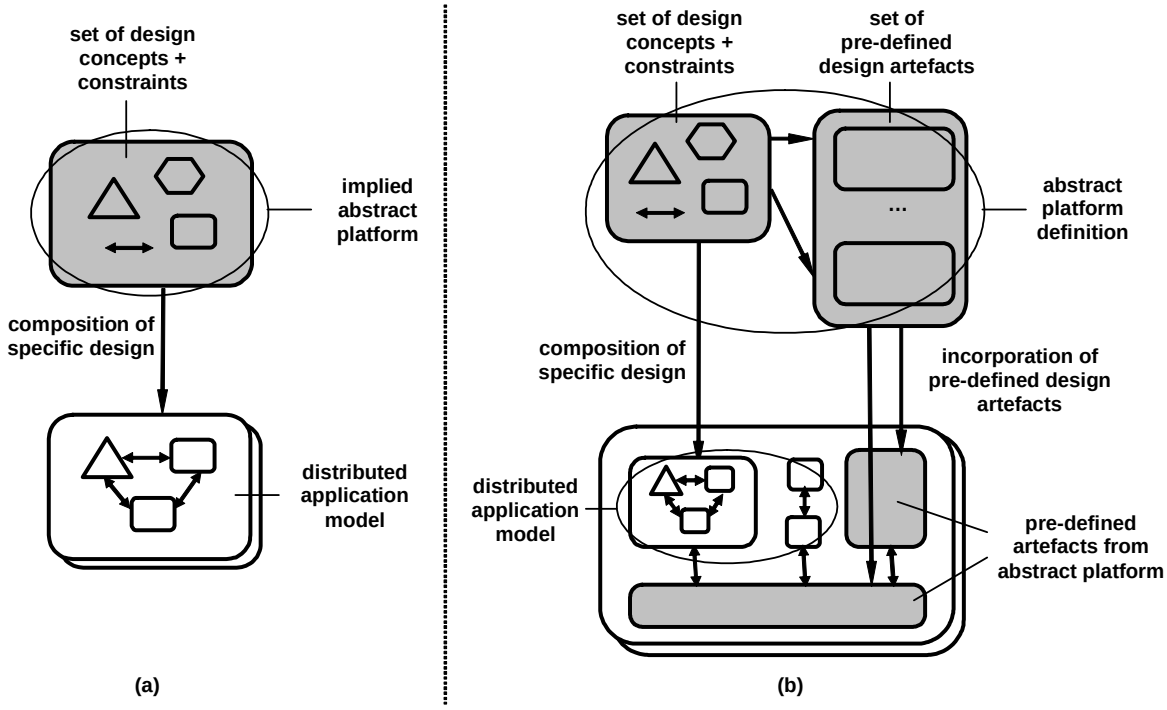

Figure 1. Abstract platforms defined by (a) choice of design concepts and (b) pre-defined design artifacts

modules, bindings and groups of bindings. Typical operations on entities are replacement, migration, creation and removal. In this paper, we focus our attention on component replacement and migration:

1. *Component replacement* allows one version of a component to be replaced by another version, while preserving component identity. We use the term version of a component to denote a set of implementation constructs that realizes the component. The new version of a component may have functional and QoS properties that differ from the old version. Nevertheless, the new version of the component should satisfy both the functional and QoS requirements of the environment in which the component is inserted; and,
2. *Component migration* means that a component is moved from its current node to a destination node. Component migration can be necessary, e.g., when a certain node has to be taken offline.

A system evolves incrementally from its current configuration to a resulting configuration in a *reconfiguration step*. A reconfiguration step is perceived as an atomic action from the perspective of the application. We distinguish between simple and composite reconfiguration steps. A *simple reconfiguration step* consists of the execution of a reconfiguration operation that involves a single *affected* component. A *composite reconfiguration step* consists of the execution of reconfiguration operations involving several affected components. Composite steps are often required for reconfiguration of sets of related components. In a set of related components, a change to a component *A* may require changes to other components that depend on *A*'s characteristics.

We introduce dynamic reconfiguration concepts in a platform-independent design by specializing the notion of a component, distinguishing between *reconfigurable* and *non-reconfigurable* components. Reconfigurable components can be *migrateable*, *replaceable* or both *migrateable* and *replaceable*. This allows a designer to establish these distinctions at a platform-independent level, specifying which components may be manipulated by reconfiguration operations in reconfiguration steps. We represent these specializations of the component concept in UML 2.0 [14] by introducing the stereotype «reconfigurablecomponent», which can be applied to a UML component. This stereotype has tagged values isReplaceable and isMigrateable. UML statecharts can be used to specify the behaviour of (reconfigurable) components.

A (composite) reconfiguration step is specified by a set of simple reconfiguration steps. The definition of a replacement reconfiguration step identifies a component to be replaced and establishes its new version. The definition of a migration reconfiguration step identifies the component to be migrated and establishes its new location. The location should be specified in terms of

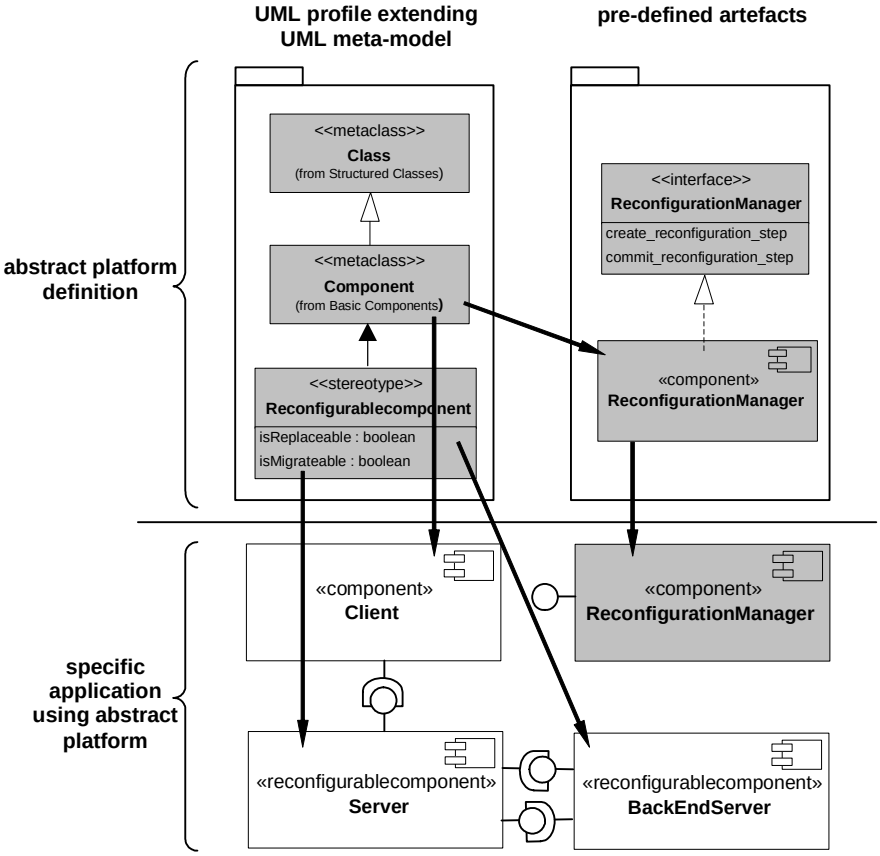

Figure 2. Support for dynamic reconfiguration in an abstract platform

abstract (QoS) properties of the new location. A reconfiguration manager component represents the capabilities of the abstract platform of handing reconfiguration steps. Reconfiguration steps are committed to and handled by the reconfiguration manager. The interface of the reconfiguration manager is an abstraction of the IDL interfaces presented in [17].

Figure 2 depicts the definition of our abstract platform in terms of a UML profile and the reconfiguration manager component.

4. Platform-specific realization

Platform-specific realization may be straightforward when the capabilities of the selected concrete platform correspond (directly) to the characteristics of the abstract platform. When this is not the case, we distinguish two contrasting extreme approaches to proceeding with platform-specific realization:

1. *Adjust the concrete platform*, so that it corresponds to the abstract platform. In this approach, the boundary between abstract platform and platform-independent distributed application model is preserved during platform-specific realization. This implies the introduction of some platform-specific *abstract platform logic* to be composed with the concrete target platform, and;
2. *Adjust the platform-specific model of the application*, while preserving the requirements specified at platform-independent level, so that the application model can be composed with the target platform model. This may imply the introduction of (e.g., QoS) mechanisms in the platform-specific design of the application.

In this paper, we focus on approach 1 to realization, since it enables a clear separation of application and infrastructure functionality, as defined by the abstract platform.

Approach 1 implies the introduction of some platform-specific *abstract platform logic* to be composed with the concrete target platform. The nature of this composition depends on the particular requirements for the abstract platform. It may be possible to implement the abstract platform logic on top of the concrete platform (as depicted in Figure 3 alternative 1a). Nevertheless, this composition may also imply the introduction of platform-specific (QoS) mechanisms in the middleware layer (as depicted in Figure 3, alternatives 1b and 1c). In this case, implementation restrictions imposed by the concrete platform play an important role.

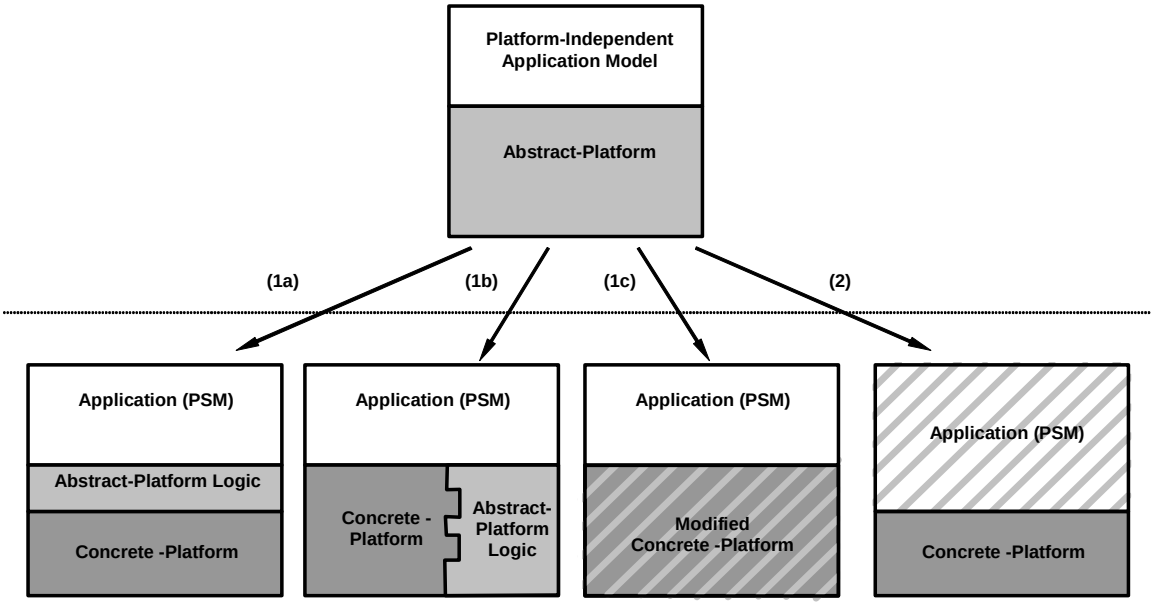

Figure 3. Alternative approaches to platform-specific realization

Figure 3 illustrates possible implementations of the different approaches to platform-specific realization.

Different middleware platforms offer different possibilities for the embedding of QoS mechanisms in the platform. In some platforms, modification or extension of internal components of the platform may be required (Figure 3, 1c). This may be undesirable or impossible for proprietary platforms (for which there is often no access to the platform's source code), or it may require agreement through long standardization cycles for platforms based on open standards. In addition, internal components of a platform are typically vendor-specific.

Extension of the concrete platform in a non-intrusive manner is often the preferable way to adjust the concrete platform (Figure 3, 1b). Techniques that can be used for non-intrusive extension include interceptors with message reflection [12], aspect-oriented programming and composition filters [3]. Using these extension mechanisms, it may be possible to separate dynamic reconfiguration extensions from core standardized middleware functionality. This approach, however, depends on the availability and capabilities of standardized extension mechanisms in middleware platforms.

We have built a Dynamic Reconfiguration Service (DRS) for CORBA that follows approach 1b. This service provides reconfiguration transparency for CORBA application objects, supporting both simple and composite reconfiguration steps. The DRS has been implemented by extending CORBA implementations through the use of portable interceptors, which are standardized extension mechanisms for CORBA ORB implementations [12]. For details on the dynamic reconfiguration algorithm and the DRS implementation please refer to [1, 17]. The DRS freezes on-demand interactions with objects that are being reconfigured, driving the application to what is called a reconfiguration safe state. In this state, the DRS applies the reconfiguration steps, and, after that, unfreezes the interactions. Reconfigurable objects should be classified into active and non-active objects, which should be done by developers during PIM marking (parameterisation of transformation). The service requires that state-access operations be included for reconfigurable objects. Placeholder for these operations should be included in the PIM-PSM transformation. Depending on the availability of behavioural models in the PIM, state derivation and active/non-active classification could be automated during transformation.

In the absence of possibilities for platform extension, approach 1a may still prove to be useful. This is the case for the realization on Web Services hosting platforms. Web Services hosting platforms entail a number of platforms that support the hosting of endpoints described in WSDL [18] and that interact through SOAP [19]. Examples of these platforms are J2EE [15] and .NET [8]. Since Web Services do not imply a particular hosting infrastructure, these platforms provide their own containers and (server-side and client-side) stubs. The suitability of approaches 1b and 1c depends on the level of extension or adjustment that is possible with these containers and stubs. Since we would like to consider an approach for Web Services that does not depend on the hosting platform choice, approach 1a is preferred. The transformation from PIM to PSM can introduce proxy web services that realize the same functionality as portable interceptors in the CORBA DRS.

5. Concluding remarks

By separating infrastructure and application concerns the development of distributed applications can be facilitated. We have shown an approach to the separation of dynamic reconfiguration and application functionality that is suitable for applications being realized on top of different middleware platforms. In this approach, the application developer does not have to be concerned with mechanisms for dynamic reconfiguration. Support for dynamic reconfiguration is provided as extensions to middleware platforms or as reusable components that are composed (or "woven") with the application during platform-specific realization.

Platform-independent models are decoupled from their corresponding platform-specific counterparts by transformations, thereby adding a new dimension to the discussion on the separation of application and distribution infrastructure functionality. There is some degree of freedom between capabilities offered by an abstract platform and capabilities offered by concrete platforms. Identifying an abstract platform brings attention to *balancing* between two conflicting goals: (i) platform-independent modelling, and (ii) platform-specific realization. It makes no sense specifying platform-independent models of applications that cannot be realized in available target platforms. Bottom-up knowledge of the available platforms and their extension/adaptability capabilities is therefore fundamental to define appropriate abstract platforms.

We expect that other QoS mechanisms can be supported with the approach we have described, including load balancing, caching and replication, and other mechanisms that profit from distribution to satisfy QoS constraints. Ideally, it should be possible to select and combine different mechanisms when designing a distributed application. We intend to investigate modularisation criteria for abstract platform definitions to enable this combination. A developer should then be able to compose an abstract platform from abstract platform definition "modules". This modularisation would ideally be reflected in transformation specifications and ultimately at platform-specific level.

Acknowledgements

This work is partly supported by the European Commission, in context of the IST project MODA-TEL (http://www.modatel.org), and by the Dutch Ministry of Economic Affairs, in the context of the Equanet project (http://equanet.cs.utwente.nl).

References

[1] J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, *Proc. 3rd Intl. Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, Sept. 2001, pp. 197-207.

[2] J. P. A. Almeida, M. van Sinderen, L. Ferreira Pires, D. Quartel. A systematic approach to platform-independent design based on the service concept, *Proc. 7th Intl. Conf. on Enterprise Distributed Object Computing (EDOC 2003)*, Brisbane, Australia, Sept. 2003, pp. 112-123.

[3] L. Bergmans and M. Aksit, Composing crosscutting concerns using composition filters, *Communications of the ACM*, Vol. 44, No.10, Oct. 2001, 51-57.

[4] C. Bidan, V. Issarny, T. Saridakis, A. Zarras. A dynamic reconfiguration service for CORBA, *Proc. IEEE Intl. Conf. on Configurable Distributed Systems*, May 1998.

[5] J. Kramer, J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. on Software Engineering* 11(4), April 1985, pp. 424-436.

[6] J. Kramer, J. Magee. The evolving philosophers' problem: dynamic change management. *IEEE Trans. on Software Engineering* 16(11), Nov. 1990, pp. 1293-1306.

[7] ITU-T, Recommendation Z.100 – CCITT Specification and Description Language, International Telecommunications Union (ITU), 2002.

[8] Microsoft Corporation, *Microsoft .NET remoting: A technical overview*, July 2001, available at http://msdn.microsoft.com/library/

[9] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. Ph.D. thesis, Imperial College, London, UK, March 1999.

[10] Object Management Group, *Model driven architecture (MDA)*, OMG document ormsc/01-07-01, July 2001.

[11] Object Management Group, *MDA-Guide*, V1.0.1, omg/03-06-01, June 2003.

[12] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Version 3.0, OMG document formal/02-12-06, Dec. 2002.

[13] Object Management Group, *Online Upgrades Draft Adopted Specification*, OMG document ptc/02-07-01, July 2002.

[14] Object Management Group, *UML 2.0 Superstructure*, ptc/03-08-02, Aug. 2003.

[15] Sun Microsystems, *Java Web Services Developer Pack*, available at http://java.sun.com/webservices/downloads/

[16] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, Coordinating the simultaneous upgrade of multiple CORBA objects, *Proc. 3rd Intl. Symp. on Distributed Objects and Applications (DOA 2001)*, Rome, Italy, Sept., 2001.

[17] M. Wegdam, *Dynamic reconfiguration and load distribution in component middleware*, Ph.D. Thesis, University of Twente, the Netherlands, 2003.

[18] World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001, available at http://www.w3.org/TR/wsdl

[19] World Wide Web Consortium, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Proposed Recommendation, May 2003, available at http://www.w3.org/TR/soap12-part1/