Lecture Notes in Computer Science 1999

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Wolfgang Emmerich Stefan Tai (Eds.)

Engineering Distributed Objects

Second International Workshop, EDO 2000
Davis, CA, USA, November 2-3, 2000
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Wolfgang Emmerich
University College London, Department of Computer Science
Gower Street, London WC1E 6BT, UK
E-mail: W.Emmerich@cs.ucl.ac.uk

Stefan Tai
IBM T.J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA
E-mail: stai@us.ibm.com

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Engineering distributed objects : second international workshop ;
revised papers / EDO 2000, Davis, CA, USA, November 2-3, 2000.
Wolfgang Emmerich ; Stefan Tai (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 2001
(Lecture notes in computer science ; Vol. 1999)
ISBN 3-540-41792-3

CR Subject Classification (1998): D.2, C.2.4, C.2, D.1.3, D.4, D.3

ISSN 0302-9743
ISBN 3-540-41792-3 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH
http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10782206 06/3142 5 4 3 2 1 0

Preface

EDO 2000 was the second international workshop on software engineering for distributed object systems. EDO 2000 was a continuation of EDO'99, the first workshop in this series, which was held in conjunction with ICSE '99.

Distributed object technologies – as exemplified by CORBA and the CORBA Services, COM+, EJB, and the J2EE – are increasingly being adopted by various enterprises as a fundamental technology for their IT infrastructures. As a consequence, extensive industry practice of using the technologies is being gained. At the same time, the technologies continue to advance and new functionality and services continue to be introduced.

In order to use the existing and emerging functions of distributed object technologies effectively, and to better meet today's demanding business and computing requirements, advances in software engineering methods and techniques for distributed objects are strongly needed. Software engineering for distributed objects is the research area that provides solutions of proven practice for issues and problems that are unique to systems employing distributed object technologies. EDO is the premier workshop that brings together the research and practice community of software engineering for distributed objects.

We received about 30 submissions and the international program committee selected 15 papers. We clustered accepted papers into sessions and the authors of these papers championed the sessions and took a lead in the discussions. The workshop organizers selected particular authors to give brief presentations that were aimed to kick off the discussion in each session. The result of the different sessions was summarized at the end of the workshop and these session summaries are included in these proceedings.

Also in tradition with the workshop series, we had an invited industrial presentation. This year Walter Schwarz talked about an enterprise application integration project in the financial domain that deployed a judicious combination of distributed object middleware and markup languages to achieve integration of financial trading systems.

December 2000

Wolfgang Emmerich and Stefan Tai
Program Co-chairs
EDO 2000

Program Committee

Organization

Conference Chair:	Volker Gruhn, University of Dortmund, Germany
Program Co-chairs:	Wolfgang Emmerich, University College London, UK
	Stefan Tai, IBM Watson Research, U.S.A.
Organizing Chair:	Prem Devanbu, University of California, Davis, U.S.A.

Referees

Jean Bezivin, University of Nantes, France
Gordon Blair, Lancaster University, UK
Alfred Bröckers, Adesso GmbH, Germany
Peter Croll, University of Wollongong, Australia
Elisabetta Di Nitto, Politecnico di Milano, Italy
Alfonso Fuggetta, Politecnico di Milano, Italy
Willi Hasselbring, University of Oldenburg, Germany
Jusuke Hashimoto, NEC, Japan
Walter Huersch, Zuehlke, Switzerland
Arno Jacobson, INRIA, France
Mehdi Jazayeri, TU Vienna, Austria
Gerti Kappel, University of Linz, Austria
Wojtek Kozacynski, Rational, USA
Bernd Krämer, FU Hagen, Germany
Jeff Magee, Imperial College, UK
Nenad Medvidovic, University of Southern California, USA
Neil Roodyn, Cognitech, UK
David Rosenblum, University of California Irvine, USA
Isabelle Rouvellou, IBM Watson Research, USA
Walter Schwarz, DG Bank, Germany
Dirk Slama, Shinka Technologies, Germany
Daniel Steinmann, UBS, Switzerland
Alfred Strohmeier, EPFL, Switzerland
Stan Sutton, IBM Watson Research, USA

Sponsoring Institutions

Adesso GmbH, Germany
IBM Watson Research, U.S.A.
Zühlke Engineering GmbH, Germany

Table of Contents

Application Integration with CORBA and XML

Walter Schwarz

OIHE, DG Bank AG, Am Platz der Republik
60265 Frankfurt, Germany
walter_schwarz@dgbank.de

Abstract. We report on experience that we made in the Trading room InteGRation Architecture project (TIGRA). TIGRA developed a distributed system architecture for integrating different financial front-office trading applications with middle- and back-office applications. We discuss the detailed requirements that led us to adopt a judicious combination of object-oriented middleware and markup languages. In this combination an object request broker implements reliable trade data transport. Markup languages, particularly the eXtensible Markup Language (XML), are used to address semantic data integration problems. We show that the strengths of middleware and markup languages are complementary and discuss the synergies yielded by deploying middleware and markup.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, p. 1, 2001.

Middleware Selection

Stanley M. Sutton Jr.

IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532 USA,
suttonsm@us.ibm.com

1 Introduction

An increasing variety of middleware systems is available for use in enterprises today. Two widely used but very different middleware styles are object-oriented and message-oriented. Within each of these styles, there are multiple products to choose from. Moreover, any of these products may be used alone or in combination with other products. Thus the problem of middleware selection is increasingly important in the engineering of enterprise software systems.

Middleware selection, construed broadly, is the determination of middleware to be used in a software development or integration project. The middleware may already exist, in which case selection reflects an intention to acquire it, or the middleware may not yet exist, in which case selection implies an intention to develop it.

Middleware selection is important for a number of reasons. It is an essential part of the way in which distributed systems get built, both by new development and by integration of existing applications and services. Moreover, middleware is a key enabling technology: it provides services, supports application functions and features, separates concerns, and integrates components. In these roles, middleware interacts with and may impact many other kinds of technologies, such as database systems, workflow engines, web servers, and applications. It further affects system architecture and development processes.

Middleware selection is also challenging, for these same reasons. Middleware selection can be an involved software (and systems) engineering process in its own right, with all the technological, organizational, economic, and political aspects that this may imply. Because of the central position and critical function of middleware, if it is selected or applied inappropriately, it can become a key disabling technology.

The two papers in this session provide broad views on middleware selection for enterprise-scale distributed systems. "A Key Technology Evaluation Case Study: Applying a New Middleware Architecture on the Enterprise Scale", by Michael Goedicke and Uwe Zdun of the University of Essen, Germany, describes a method for evaluating and selecting middleware in an enterprise context. It emphasizes the importance of understanding and communication among all stakeholders in the enterprise system, notably management and engineers. It also emphasizes the enterprise-specific nature of the middleware evaluation and selection process.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 2–7, 2001.

"An Architecture Proposal for Enterprise Message Brokers", by Jörn Guy Süß of the Technische Universität and Michael Mewes of the Fraunhofer ISST, both in Berlin, Germany, defines and motivates an enterprise architecture based on message brokers. This work also represents a kind of middleware selection process, since the motivation for the message broker architecture arises from failure of other kinds of considered middleware to effectively address enterprise integration needs.

Both of the papers focus on communication issues as key concerns in middleware selection. Interestingly, each paper arrives at a different primary communication style: synchronous in the former case, asynchronous in the latter. This divergence of results, both of which are well justified, only begins to suggest the diversity of concerns, issues, and approaches that may characterize middleware selection.

To begin to organize these topics, we considered the following general questions:

- What is the role of middleware selection in the engineering of distributed object systems?
- What is the role of middleware selection in the engineering of distributed object systems?

2 What is the Role of Middleware Selection?

When selecting middleware we ultimately want to know what process we should use. However, the most appropriate selection process depends on the role of middleware selection in the overall systems development or integration process.

The role of middleware selection depends on the current and projected shape of the "enterprise system landscape." The main elements shaping this landscape are:

- The proportion of integration of existing applications and systems versus development of new applications and systems
- The level and scope of the organization or enterprise over which middleware is to be applied. This may vary from individual projects and work groups, up through individual departments, to span departments within an enterprise, and even to integrate distinct enterprises
- The time frames for middleware selection, implementation, and use
- The risks, costs, and feasibility of alternative approaches

Integration projects may present more or fewer challenges for middleware selection, depending on the diversity of systems and applications to be integrated. For new development, the constraints on middleware may be initially less restrictive, but middleware selection must be closely integrated with the design of other elements of the system, so selection can still be complex.

Typically, the larger the scale of the effort for which middleware is being selected, the more numerous and complex the issues affecting middleware and

thus the middleware selection process. However, complicating and simplifying factors may occur at any level or scale of project.

The time frames for middleware selection, implementation, and use affect the potential selection process in several ways. If the period for selection is short, then the role of selection may be relatively limited, for example, to identify an approach based on a restricted criterion (such as flexibility, familiarity, or cost) or to consider just a small number of alternatives. If the period for selection is longer, then the role of the selection process may be to compare many technologies, identify and consider a wide range of selection criteria, and so on. If the implementation period is short, then the selection process and its results may be constrained, and an approach oriented toward acquisition may be favored over one oriented toward implementation. If the implementation period is longer, then the selection process can be open to considering and recommending a wider range of technologies and processes. If the period of use (over which the cost of middleware will be amortized) is short, then the selection process should focus on short term concerns and rapid gains and the selection and implementation processes should be constrained accordingly. If the period of use is to be long, then the selection process must focus on long term concerns and benefits, and the selection and implementation processes may be opened up accordingly.

Risks, costs, and feasibility affect all systems development and integration efforts to some degree, but any of these may be especially important in particular cases. Consequently, the role of middleware selection may emphasize risk reduction, cost control, or feasibility assessment.

The above concerns primarily reflect a technical viewpoint. However, organizational factors (including social factors) can be just as important in determining the role of the middleware selection process. Such factors include economics, politics, business processes, business relationships, organizational history, personnel, and so on. A primary function of the middleware selection process may be to identify stakeholders in the middleware selection process and to educate or inform them regarding technical and other concerns relevant to the selection process. Stakeholders may include project managers, development engineers, business partners, customers, third-party systems integrators, consultants, and others, with very different interests, goals, and levels or areas of expertise. According to their stakes, each of these groups may see middleware selection in a different role and have different objectives for it.

Of course, organizational as well as technical concerns can affect practically any aspect of the enterprise system landscape: the degree of integration versus new development, the level and scale of software projects, the time scales on which projects operate, and the perspectives on risk, costs, and feasibility. Standards (or the lack of them) are also a critical consideration at the interface between the technical and non-technical elements of enterprise system engineering. There are choices involving standards that are open or closed, official or de facto. Just as standards help to achieve integration and interoperation on technical plane, they can also help to promote communication and understanding on

the social plane. The role of middleware selection must be defined with respect to the use of standards in these ways.

Finally, the role of middleware selection must be defined with respect to the rest of the software life cycle. Middleware selection should generally be based on system requirements specifications (that account for both technological and organizational factors), but middleware selection may not (and probably should not) be wholly determined by those specifications. Depending on circumstances, middleware selection may occur prior to, along with, or subsequent to system architectural specifications. Consequently, middleware selection may constrain, or be constrained by, architectural design.

Generally, depending on the variety of technological and organizational factors outlined above, middleware selection can be a relatively independent or dependent process. The potential variety of interdependencies between middleware selection and other development or integration activities points out that what is needed is a more general methodology for the software engineering of distributed object systems. Such a methodology should enable all of the activities and technologies relevant to the development and integration of enterprise-scale systems to be defined, organized, and interrelated in a coherent, consistent, and effective way.

3 What is the Process of Middleware Selection?

Due to the variety of roles that middleware selection may play, and to the relative immaturity of software methodologies for distributed object systems, it is evident that there is not, and cannot be, one process for middleware selection. The role that middleware selection plays, and the context in which that role is determined, set important parameters that govern the particular selection process used in (or appropriate for) a particular enterprise or project. These parameters include, for example,

- The parties involved, their roles, the amount of communication needed between them, and the level of discourse required
- The applicability and use of standards
- The breadth of middleware technologies considered
- The criteria and means for technology evaluation

The level and scope of development or integration, and whether it is within a group (team, department, or enterprise) or crosses group boundaries, also has an effect on the complexity of the software process in general and of middleware selection in particular. A particular defined selection process may execute variously, and variously well, depending on the level and scope of the project to which it is applied.

Despite the potential variety of middleware selection processes, a number of common characteristics or recommendations regarding middleware selection processes were identified in this session.

Selection may be made based on a number of approaches to comparing and evaluating middleware technologies. One of the papers in this session describes an analytical approach, while the other is based on an experimental approach. It may also be possible to use simulation for comparison, but this requires the use of simulation technology, which may not be available yet in many cases.

The frameworks, taxonomies, and criteria that are used for comparing and evaluating middleware must reflect enterprise-specific and project-relevant conditions. General frameworks, taxonomies, and criteria may be useful to some extent, but they must be tailored for a particular organizational situation.

Any particular development or integration process is also likely to involve a number of kinds of stakeholders with diverse background and interests. Consequently, an effective middleware selection process will promote communication and understanding among these stakeholders.

Standards can be useful for enhancing communication among stakeholders as well as facilitating technical integration. For example, XML provides a means to define standard document types that facilitate information exchange between heterogeneous systems; the use of XML requires agreement among cooperating organizations on the structure of these documents, and a number of domain-specific XML standards are under development. Where official standards are not published, certain widely use middleware technologies, for better or worse, may implicitly define de facto standards. Even where standards are used, though, non-standard extensions offered by particular technology providers may determine the selection of particular middleware products. These extensions may provide useful features or functionality, but at the cost of interoperability or flexibility with respect to other technologies or technology providers. Whether this trade-off is worthwhile depends on the particular enterprise, project, and technology involved.

To afford an appropriate degree of flexibility in middleware selection, the selection process should allow for the adoption of multiple alternative (or complementary) technologies in the solution to any particular problem. Additionally, the selection process should allow for the refinement of a general development or integration problem into more specific problems, in different areas or on different levels, to which different solution constraints and opportunities may apply.

One of the most critical challenges in middleware selection is accommodating the various time frames over which middleware selection, implementation, and use are to occur. One concern is controlling costs through the stages of the middleware life cycle and assuring the amortization of those costs across the life-cycle period. Another concern is accommodating change during the life of the system, as unexpected changes may incur unexpected costs or prematurely cut short the effective life of the system.

Especially at the enterprise scale, the long-term view is important. Two general approaches to building long-lived systems were advocated in this session:

- Anticipate changes and plan for them (build in points or mechanisms of change)

– Do not try to anticipate changes but use simple, flexible constructs (that can accommodate change as it occurs)

Enterprise system builders find themselves in a somewhat paradoxical situation. The cost of enterprise systems development and integration, and the time required to complete such projects (which may be years), means that careful planning is necessary. However, the evolution of both enterprise needs and software technologies cannot be totally controlled. It is inevitable that plans will go astray. Therefore, the development and integration of enterprise systems must be based on a long-term strategy, but it must also be recognized that the object of that strategy will be a moving target. Indeed, middleware selection itself may be an ongoing activity that outlives individual application and integration projects in a long-term enterprise.

Acknowledgements

This paper is based in part on the comments of the following workshop attendees, who contributed to the discussion during this session: Premkumar Devanbu, Wolfgang Emmerich, Alfonso Fuggetta, Michael Goedicke, Cecilia Mascolo, Walter Schwarz, Antonio Rito Silva, Jörn Guy Süß, Stanley Sutton, and Stefan Tai. Thanks to Stefan Tai, Jörn Guy Süß, and Michael Goedicke for their review of a draft of this manuscript.

A Key Technology Evaluation Case Study: Applying a New Middleware Architecture on the Enterprise Scale

Michael Goedicke and Uwe Zdun

Specification of Software Systems, University of Essen, Germany
{goedicke,uzdun}@informatik.uni-essen.de

Abstract. Decisions for key technologies, like middleware, for large scale projects are hard, because the impact and relevance of key technologies go beyond their core technological field. E.g., object-oriented middleware has its core in realizing distributed object calls. But choosing a technology and product also implies to adopt its services, tools, software architectures, object and component paradigms, etc. Moreover, legacy applications and several other key technologies have to be integrated. And since no middleware product serves all requirements in the enterprise context, various middleware products have to be integrated, too. Another key problem of middleware evaluation is, that often the studies have to be performed very early in a project. In this paper we try to tackle these problems and describe how we can communicate the outcomes – which come from a technical viewpoint – to the management and other non-experts in the technological field.

1 Key Technology Evaluation Case Studies

Early key technology evaluations are a case study type that we have performed several times in different business setups for different companies. They aim at the very early investigation of key technologies, like communication infrastructure, database management systems, or programming languages, for large-scale, business-critical projects from a technical viewpoint. The projects were focussed on re-engineering of large existing systems with numerous applications, though many similar studies are performed for development projects of new software systems. In such projects management normally wants to decide which key technologies the project will use, before the project is actually launched, in order to estimate the savings/costs caused by the technology.

With the term "key technology" we generally refer to business-critical technologies, that drive companies to launch evaluation studies in early phases of (large-scale) projects. With other words: technologies that lead to high costs, if they fail to satisfy the company's expectations. Examples of key technologies could be communication infrastructure, database management, programming language, operating system, etc. technologies. It heavily depends on the nature of the project, whether one of these technologies is a key technology for the project or not.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 8–26, 2001.

In this paper we will concentrate on a study, which aims at the re-organization of the information systems of a large German enterprise with its core business in the field of logistics. The case study is embedded in a process that includes modeling the business processes, designing appropriate conceptual models, strategic decisions for used technologies, and specification of platforms. The presented case study had the task to identify and exemplify suitable middleware solutions for the information system base-line architecture of the enterprise. The great challenge of this study was, that it had to analyze key aspects of technology, before all application areas have named their actual requirements. The enterprise was confronted with the problem that a huge number of applications were developed independently by the various departments. But these applications had to work in concert to a certain degree in order to allow the departments to flexibly interoperate and to exchange their respective information.

Newer key technology trends, which have become mainstream recently, like distributed object systems, normally promise a lot, but also imply a set of risks. Since the company had not experts in (all) the new key technology areas, it was hard for the management to estimate which new technologies are valuable/necessary for the company. Therefore, a middleware evaluation study was launched during early requirements analysis. We believe that this situation is recurring for different key technologies in many companies of various size and in various business fields.

1.1 Software Architectural Integration

Often departments have had the freedom for a "programming in the wild" with no clear architectural concept for integration. This freedom helps to rapidly develop applications from the scratch. But when the number and complexity of applications rises, maintenance and integration of application become more and more difficult. In such a diverse field, like communication infrastructure, a lacking integration concept means not only to run into problems with integrating communication technologies, but also in integrating the various programming languages, object models, services, access variants to shared resources, etc. Many organizations, like the enterprise in this case study, react by creating an external department that should impose standards over information system development. Often these standards tend to be loaded with severe constraints. Therefore, often the standards are either ignored by the developers or they lead to monolithic systems that are hard to maintain.

These problems are similar in many companies, but they are more pressing in the enterprise context, than in small companies. There are several reasons. I.e., the enterprise has more different applications and technologies that have to be integrated. Applications have to be deployed to more hosts and middleware products have to be purchased earlier. Departments that impose standards have more likely limited personal contact to all affected developers and, therefore, developers are less involved in key technological decisions.

From our experience, an early technological study should give guideline and communication assistance for technological decisions to the concrete applica-

tion's software architect rather than to make premature decisions in her/his place. Therefore, in this paper we will try to show a way to avoid such rather random impositions. Nevertheless, the paper comes to concrete results from managerial and technological viewpoints for the integration of object-oriented middleware in a very early project phase.

1.2 Roadmap

So far we have sketched a problem field, which seems on the first glance to have no clear solution. Management demands for a clear basis for decision and for an architectural perspective, but the field is much too diverse to provide a simple answer. However, simplicity and transparency of decisions are of central importance. Both the involved managers and developers have to be able to understand the decisions and the reasonings behind them directly. The acceptance of decisions relies on the solution's ability to cover the technical realm of the application in focus. In early evaluation studies this is very hard, because the application specifications do not exist when the study is launched. In enterprise-scale studies it is even harder, because a wide range of applications has to be covered.

Another problem we cannot ignore is subjectiveness. If people have to decide for key technologies personal experiences, predilections, opinions, and prejudices come into the decision. Our experience is that no pseudo-objective decision process can change this as long as not enough "hard" criteria can be found.

Our approach relies on the simple idea – which is not so simple in its realization – to use the right tool for a given task and then seek for technical solution for integration of these tools. However, the goal to find a company-wide integrated solution may cause damages that outweigh the benefits by far. In contrast to the concrete applications, the technological requirements for integration are quite concrete even in early stages, because the technologies themselves are already existing. We will see, that in this domain we face recurring problems, like object system integration, finding of a suitable component concept, bridging between technologies, etc. It is important that these integration decisions are performed at a detailed technological level. Otherwise the evaluation case study bears little to no technological substance and is unlikely to be accepted by concrete application developers.

In detail, we firstly have discussed and communicated the technological field, the available technology types, and the available products with the stakeholders. The outcome was a document describing all these aspects in detail and a taxonomy of the middleware technologies/products. Thus we have come to consensus on all these aspects covering the objective *and* the subjective knowledge about the middleware technologies/products. We have created a "common language" for further discussion. These issues are discussed in the Sections 2 and 3.

With this "common language" we have then begun to discuss concrete application scenarios, that the stakeholders characterize as typical for the company. From our experience the outcome is nearly always a diverse set of technologies. Therefore, two questions arise directly: How do we integrate the different appli-

cations built with different technologies and how do we find the best technology from the taxonomy for a concrete application?

We provided a concrete technological concept for the integration task, which we introduce briefly in Section 4. As a guidance for concrete application decision we use a scenario-based process. In order to illustrate this process, we exemplify it with the typical application scenarios, which we had used earlier throughout the discussions. In Section 5 we present one such example of a letter distribution center information system.

2 Middleware

At the enterprise level the expectations are high and are often beyond the functionalities of existing products. At an early stage of a project it is hard to determine which technologies meet the requirements of an enterprise. Software development depends on several changing technologies. One of the most complex technology areas – faced in today's technology decision processes – is middleware [3]. In the context of this paper we see a middleware as following:

> *A middleware extends the platform with a framework comprising components, services, and tools for the development of distributed applications. It aims at the integration, the effective development, and the flexible extensibility of the business applications.*

Middleware comprises several different technologies provided by different vendors, which conform to a different extent to a great number of partially overlapping standards. Middleware abstracts from the network layer and from direct network access. Applications access networking functionalities through a well-defined interface of the middleware and communicate virtually on top of the middleware among each other. The middleware implements the details of network access (as illustrated in Figure 1). It provides a software layer between operating system functionalities and applications [17]. Thus it is often seen as an extension of the traditional notion of the platform.

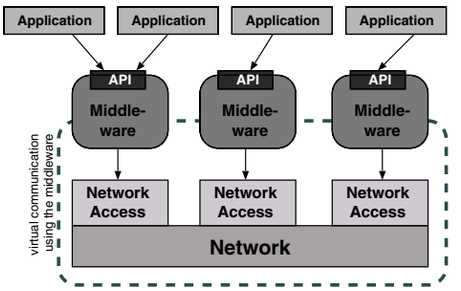


Fig. 1. Middleware

Choosing a key technology, like middleware, has severe impact on the *software architecture* of the enterprise's information systems. The software architecture consists of software components and the relations among them. Therefore, the component (and object) model of the middleware has to be integrated, adapted, and/or adopted to the component models used throughout design and implementation. Limitations of the middleware's component model can restrict the expressive power accessible in design/implementation. But the middleware technology can also introduce new concepts and architectural means not accessible in the used design/implementation languages. E.g., the new CORBA 3 component model introduces a distinct component model into languages that offer none (like C) or Orbix Filters [9] introduce interception techniques into non-interceptible languages (such as Java).

Different middleware products (and technologies) offer a different spectrum of *services*. Services are central for the usage of middleware in the enterprise. Generally each service can also be developed by the enterprise itself (or by a third party). But standardized services allow the applications to be developed faster and more cost effective and they provide a higher interoperability among applications, that need such a service. Important service areas are messaging service, transaction service, security service, and naming (directory) service.

Even the *software development processes* are strongly influenced by middleware technologies. E.g., a clear component/interface model enables software development in separate teams. Therefore development with a component model enforces another development process than the development of a monolithic piece of software. For these reasons a positive impact of software architecture, a suitable set of services and tools, and the ability to enforce a promising development process are central requirements for a middleware technology. Besides these central requirements other non-technical requirements have to be considered, like costs of licenses, education of developers and designers, vendor politics, standards, existing education of the SW developers, designers, etc.

3 Technologies and Taxonomy

In the preceding sections we have tried to define middleware and to name important aspects when choosing a middleware for an enterprise. These aspects are chosen in order to be able to explain the impact of middleware technology onto distinct areas of software development. We have used these aspects to develop together with the stakeholders in the enterprise a common framework in order to characterize and compare different products in different categories. The specific framework was:

- Interoperability:
  - (Standardized) communication over the network,
  - Support for various programming languages,
  - Support for various platforms (i.e., platform independence).
  - Integrated component model.

- Services:
  - Messaging service,
  - Transaction service,
  - Security service,
  - Naming (directory) service.
- Scalability.
- Performance.
- Standardization.
- Marketablility of the products.

Note, that even this quite generic evaluation framework is subjective and company-specific. In other companies some aspects would probably more or less prominent, probably different service areas would have been chosen as important, etc. Every enterprise has to build its own framework and taxonomy when making an important technological decision. For different settings other central aspects, as for instance application deployment, have to be considered as well. In general the enterprise's set of important quality attributes has to be mapped onto a set of criteria that distinguishes the technologies clearly (e.g., when deciding about database technology the same technique may be applied).

The mapping can only be found through ongoing discussion with the stakeholders throughout the study, since they know their business case the best. Afterwards the enterprise can use the resulting taxonomy for their concrete applications. The reason for using a taxonomy is transportation of knowledge to the system's stakeholders, like management, developers, customers, etc. A middleware expert will know without building a taxonomy when to apply CORBA and when to use an RPC approach. But with a taxonomy it is easy to communicate such decisions, if the taxonomy is, on the one hand, based upon the central quality attributes of the enterprise and does, on the other hand, represent characteristic properties of the technologies.

3.1 Middleware Technologies

After we agreed with the stakeholders in the enterprise upon the framework that is able to express the relevant technologies and captures the important quality attributes of the company, we decided which technologies/products had to be investigated. These were:

- RPC Mechanisms, like: Sun RPC, OSF's DCE.
- Distributed Object Systems (based on the RPC principle):
  - CORBA - ORBs, like Orbix, Visibroker, TAO.
  - DCOM,
  - Enterprise Java Beans (EJB) and Java RMI,
- Application Server:
  - EJB-based servers, like WebLogic, Oracle Application Server, WebSphere.
  - Scripted web-servers, like Vignette V/5, AOLServer, Ajuba2, WebShell, Zope.

- Transaction Processing (TP) Monitors, like: Tuxedo, Encina.
- Message Oriented Middleware (MOM), like: MQSeries, Tibco.
- Mobile Code Systems (MCS), like: Aglets, Voyager, Telescript, Jini.

Finally we gave a two/three pages description of each technology and each product. The descriptions consist of a brief description, an illustrating figure, and an textual evaluation of each item of the taxonomy framework. Here, we just give a heavily abbreviated discussion of the CORBA technology as an example of the style of presentation (see Figure 2).

3.2 Results of the Assessments

The result of our assessments was a rather technical evaluation of the named middleware technologies and the different products in the various technological fields. Note, that these findings are *not* a generalizable view on middleware technology, which could be converted without change to another company. Instead these finding are strongly influenced by the process of discussing a suitable middleware in the realm of the actual company. The assessments reflect a lot of objective knowledge about middleware, but also a lot of subjective aspects, such as personal predilections, special experiences in the company, company politics, etc.

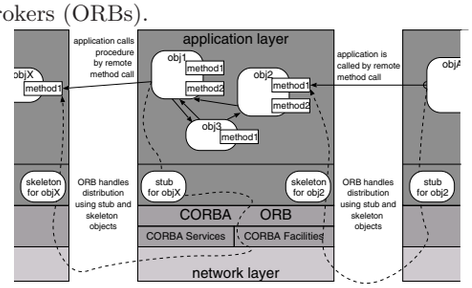
In summary, all participants in the discussion came to the opinion, that all technologies provide enhancements in the fields of the other technologies. TP monitors have their strength in database connection/transaction management. Application servers are superior in representing different business logics (like an additional web representation). But both technologies have only limited capabilities in other domains. Therefore – in the enterprise context – they should not be used as a sole middleware solution, if a part of the application (or even an expected future change) requires additional functionalities. Mobile code technologies can be superior to all other technologies in certain applications (e.g., when high configuration/customization needs on the server side are paired with low network bandwidth), but no marketable products are existing for the enterprise scale with all required middleware services.

That means for these three technologies we have hard (or better: merely objective) criteria for several applications whether they should be used or not. But even for these technologies it is most often not obvious at the first sight if the application benefits from one of these technologies more than from another. With all other technologies the decision for a technology is even harder. With other words: at the enterprise scale and under consideration of the diversity in the middleware field, there can be no a priori decision for one specific product that suits all application needs.

This point is a key problem of this work: How can we – as technical consultants – deal with a situation, when a managerial decision requires a foundation, but the technical field is so unwieldy or complex or entangled, that an objective statement is merely impossible. On the other hand, the manager requires a basis

Common Request Broker Architecture (CORBA)

Description CORBA [18] is a distributed object system with the aim to realize distribution of object communication across different machines, vendors and software systems. Object method calls are invoked with the same principle as RPC from object stub (placeholder or proxy at the client side for an remote object on the server) to skeleton (interface implementation at the server side). Relevant functions are interface definition (with an IDL), localization and activation of distributed objects, communication among client and object, and distribution transparency. These issues are handled by object request brokers (ORBs).



Interoperability ORBs utilize the IIOP (internet inter-ORB protocol) in order to connect ORBs. The protocol on the TCP layer is designed to let all ORBs use the same protocol. The design of CORBA is generally aimed at language and platform independence. A common IDL lets components be specified through their interface without knowledge of internal implementation details. A disadvantage is that languages are broken down to a common denominator (partially solved by the data type any). The CORBA IDL is mapped to great variety of languages, like C, C++, Java, OLE, Eiffel, Smalltalk, etc. CORBA ORB implementations exists on nearly any commonly used platform.

Services Initially the CORBA messaging service and the event service were based on a simple push/pull model for message exchange through event primitives. Now the OMG specifies a more robust messaging service. Meanwhile several CORBA ORBs have implemented their own protocols or they can be combined with professional MOM products, like MessageQ, MQSeries, etc. The CORBA transaction service supports flat and nested transactions. Heterogeneous ORBs and (procedural) non-ORB applications can take part in a transaction. Some vendors even offer support for integration with commercial transaction monitors, like Tuxedo. The CORBA security services specification is one of the most detailed security specification existing, covering nearly every aspect of security, like integrity, authentication, access control, etc. It is achievable in three levels (0-2) from no to full security. Support for these services varies in different ORB implementations. The CORBA naming service enables to search for objects using their object name. It wraps several different traditional directory services. Some ORBs offer a fault-tolerant naming service.

Scalability ...

Performance ...

Standardization ...

Marketablility of the products ...

Fig. 2. Technology Description: CORBA

for decisions, so this statement alone is not a sufficient answer for her/him. Moreover, in any such complex decision field, we have to deal with a lot of subjective opinions and prejudices.

Table 1. Subjective, Company-Specific Taxonomy Overview for Middleware Technologies

	RPC	CORBA	DCOM	EJB	AS	TP	MOM	MCS
Interoperability:								
Network communication	+	+ +	+	+	+	+	+	+
Programming language independence	-	+ +	+	- -	-	-	o	o
Platform independence	o	+ +	- -	+	+	+	+	+
Integrated component model	-	+	+	+	+	-	-	+
Services:								
Messaging services	-	o	-	o	o	-	+ +	+
Transaction services	o	+	o	o	o	+ +	-	- -
Security services	o	o	o	-	o	+	-	+
Naming (directory) services	o	o	o	o	o	o	-	-
Scalability	- -	+	-	+	+	+ +	+	o
Performance	o	+	o	-	-	+	o	o
Standardization	o	+ +	o	+	o	- -	- -	o
Marketability of available products	o	+ +	o	o	+	+ +	+ +	- -

Therefore, we think an enterprise has to check for every application which middleware technology or combination of technologies suits best. Throughout the process of building a taxonomy the members of discussion get a feeling for the technologies. An enterprise should identify a key middleware technology as an *integration base*, which integrates applications using different middleware technologies. This key technology should be the technology which is presumably used for the most applications. E.g., some enterprises use CORBA as their key technology, because it offers platform and language independence and a mature component model. Others use an application server as their key technology because the majority of their applications are web-based e-commerce applications.

This outcome is obvious to the technology expert, but not for the manager. The ideal outcome for a manager would be, that one or a very limited set of products could be chosen. But at the enterprise scale it is not realistic that one product (or one specific product combination) serves best for all application requirements. A very detailed document describing the various properties, advantages, and disadvantages of technologies/products in detail is a good backdrop for specific discussion, but it gives a bad overview.

Therefore, we have added a rather simplistic overview table (similar to Table 1 – here we only summarize the technologies). The table is only meant as a starting point for discussions and for making/communicating a pre-selection. We use the following simple scale for rating of the taxonomy aspects: support for

aspect is outstanding (+ +), support for aspect is good (+), support for aspect is available (o), support for aspect is not ready for the enterprise scale (-), and support for aspect is not or nearly not available (- -).

Note, that the simplicity of the table is provoking. And it is intended to be provoking, because this helps to start a discussion. E.g., an RPC user may heavily object, that RPC technologies are nearly not scalable. If the RPC user can argue for the technology and can argue that it is less work to build a bridge to the integration base than to use the integration base itself, the concrete application project, will probably use RPC. Afterwards, the taxonomy can be updated according to the experiences with that project.

Such considerations heavily depend on the company and the involved developers. If a department has several RPC experts and an RPC library framework, several of the aspects may require a quite different evaluation than in a company that has never used RPC before.

4 Integration and Coping with Change

So far, we have discussed how to make pre-selections for middleware technologies in very early stages of projects for single applications. One outcome was that no single technology can serve all requirements. Therefore, we need to come to a decision for the concrete application and we require an integration of (a) the technologies (and their services, tools, and processes) and (b) the involved (slightly) different paradigms. A special interest lies on the changeability at the technology seam, since it is a hot spot of the application.

4.1 Key Technology Decision and Integration Base

The software architecture is the first artifact in the development of a software system, that enables us to set priorities among competing concerns of the different stakeholders of the system. These concerns may be expressed in terms of quality attributes, as in [1], like performance, security, availability, modifiability, portability, reusability, integrateability, or testability. It is obvious that no architecture can maximize all of these quality attributes at once. The architect has to analyze the relevant requirements in terms of quality attributes. Influences for an architect are the architect's experience, the technical and organizational environment, and the stakeholders (like customer, end user, developing organization) of the system, who can be interviewed to find relevant scenarios with methods like SAAM [1] or [2]. The architect has to actively gather these information from the stakeholders by interviews and circulation of the results.

Our taxonomy does not lead to a distinct recommendation for one middleware solution. It just helps an architect of a concrete application to select an appropriate middleware solution from the possible alternatives. None of them is absolutely superior to other solutions, and unfortunately, each application demands different quality attributes. Therefore, the architects of any software

system have to choose the appropriate solution for their application. This outcome is very unsatisfying regarding the aim to find an integration base technology/programming language at a very early stage of a project.

However, after understanding of the business cases for the software systems and elicitation/understanding of sufficient number of requirements (using scenario-based techniques if appropriate, sometimes other techniques, like formal requirement specifications, are necessary), an integration base can be identified. A sufficing number of applications is reached when the domain experts are sure that examples of most characteristic applications are investigated. One technology combination is chosen as an integration base. Concrete application developers are free in their choice of a technology, but the architecture must contain an interface, that conforms with the integration base. These interfaces are Facades [5] that shield the application from direct access to the internals. The interfaces offer the application's services to client's based on different technologies.

Both integration base and concrete applications can be found by firstly performing a pre-selection of technologies, e.g., using the taxonomy. The result is a brief evaluation which technologies can not satisfy the requirements sufficiently. Afterwards larger, characteristic examples are investigated and candidate architectures for the examples using the different technologies are developed. These are evaluated for their architectural advantages/liabilities by comparison of the solutions and development/evaluation of (expected) change scenarios using software architecture analysis [1,8] (see Section 5 for an example).

The integration base is one kind of interface to which all applications offer their service and can comprise for instance an integrating technology and its component concept. To gain architectural flexibility in the integration base and in the software architectures of the applications the decision of each used technology should be performed stepwise on basis of change scenarios that are found and evaluated in interviews and circulation with the stakeholders. Figure 3 illustrates the various influences (ovals) and the derived artifacts (rectangles) in this process of finding a key technology. Dotted lines represent aspects which are evolving in distinct studies/implementations, while solid lines indicate continuously evolving aspects.

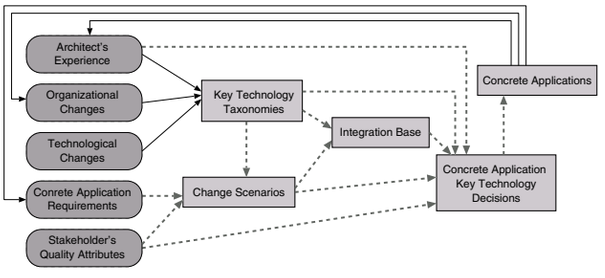


Fig. 3. Influences and Artifacts in Key Technology Decision

With the presented approach we give the application developers the freedom to develop applications with the technology that fits the application domain the best. And we have not ignored the subjectiveness in the technological decision process. Such ignorance makes developers feel uncomfortable and thus produces bad results. However, the developing department has to care for building a bridge to the integration base, if it is not available in the company already.

4.2 Object System Layer for Paradigm Integration

The open questions are, how to integrate another key technology with the integration base, how much efforts integration takes, and how much complexity the integration adds to an application system. In this domain we can present a quite concrete, technical solution, which relies on the component concepts from [6] and the Object System Layer architectural pattern [10].

We can see each subsystem as an opaque black-box, that is accessed via a Facade component. The Facade component includes a Facade object for the used middleware technology and wrapper objects that implement the calls of the Facade to the subsystem (if necessary). E.g., if we have a C legacy subsystem that uses RPC calls, we would extract a component with one distinct interface to the subsystem using RPC. Now we build an Adapter that is a second Facade to the subsystem and that has no other task than adaptation of calls using the integration base technology to the RPC interface.

In summary we add to each independent subsystem a component that explicitly defines the component's export interface. This interface is only way for other systems parts to access the subsystem and it is built with the middleware technology chosen for the subsystem. A simple Adapter integrates the integration base technology.

This simple approach lets us split up an existing system into self-contained subsystems and components. Thus we can build a component-oriented structuring for an existing legacy system in a piecemeal way. Therefore, our approach also provides a clear, piecemeal way for migration to the new middleware technology. In [7] we present a larger case study of such a migration process for a document archive system with the presented concepts.

Often paradigms of various programming languages and key technologies have to be integrated in a single component (especially in the Facade component of a subsystem). Often these models have to be integrated with concepts and languages that do not offer a notion of a certain paradigm at all, as the object-oriented paradigm that is introduced into languages, like C, by middleware, like CORBA, MOM, or TP monitors. For the enterprise context this integration of object/component concepts is especially important, since a large number of different models has to be integrated with the integration base technology/language.

This problem of integrating a foreign object system into a base technology/language can be solved by various approaches. We have documented the general underlying solution in the Object System Layer architectural pattern [10]. The solution builds or uses an object system as a language extension in the

target language and then implements the design on top of this Object System Layer. It provides a well-defined interface to components that are non-object-oriented or implemented in other object systems. It makes these components accessible through the Object System Layer and then the components can be treated as black-boxes. The Object System Layer acts as a layer of indirection for applying changes centrally. There are several implementation variants of the Object System Layer pattern. Examples of popular Object System Layers are object-oriented scripting languages, libraries implementing an object-system, and object systems of key technologies.

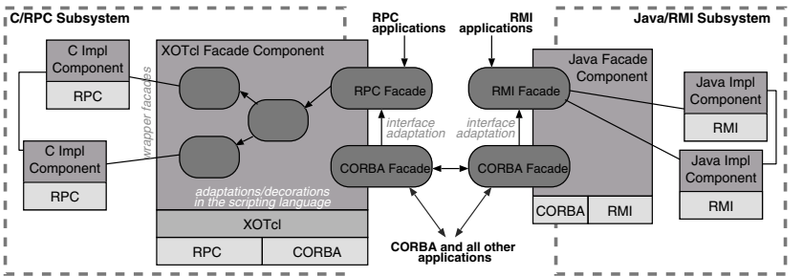


Fig. 4. CORBA Integration Base with Java/RMI and C/RPC Subsystem

Figure 4 shows the C/RPC subsystem with a CORBA integration base. Here, we must integrate the C procedural paradigm with the object system of CORBA. We propose to use an scripting languages, like XOTcl [11], as an Object System Layer to integrate with the integration base technology, because the flexible and dynamic language means of the scripting language allow us to easily integrate Adapters/Decorators at the Facade component. Rapid customizability of the Facade interfaces and the connection to the subsystem are important, because the interface section of black-box components are hot spots of distributed systems and the scripting language's component concept allows us to build compound components from several base language components. An (existing) Java/RMI subsystem can integrated with the C/RPC subsystem by giving it a similar CORBA Adapter to the RMI Facade.

Note, the similarity and symmetry of the scripting language solution with XOTcl and the Java solution. In both cases the Facade component has the task to serve as a component glue. This is the basic language design issue of scripting languages, like Tcl, which is designed as a glue for C or C++ components. Here, we re-build the same architecture in Java to have a glueing component that shields the subsystem. This way we have a clear stepwise way to migrate existing subsystems or subsystems build with another technology than the integration base into the enterprise's information system.

5 Distribution Center Example

In order to show how to apply the found results to concrete applications, we have concluded our study with several illustrative examples from various application fields of the enterprise. Here, we present one of these examples very briefly: a middleware solution of distribution centers for letters. A distribution center sorts letters by destinations. Various centers are connected to exchange letters and logistic data. Each center sorts letters for other centers. Inside of a center letters move around in standardized baskets. These baskets are sorted several times with sorting machines which are equipped with computers that are running non-standard operating systems. Non-standard letters have to be additionally sorted by human operators at a special sorting place. Before and after sorting the baskets are weight in order to control whether all letters have made it through the sorting machine or not. The balances for weighing the baskets are special peripheries which have to integrated into the system. Several NT and Unix workstations collect the data from the sorting process and compare the results. Furthermore various workstations are used for character recognition by human operators. Central computers are Leitstand, PPS, communication systems. They directly interact with central databases. The example can be seen in Figure 5.

First, we take a look at the requirements of the system. A middleware for the system should be capable of integrating the various platforms and languages used in the legacy systems. Since the system is a large, continuously working system, legacy applications have to be integrated and the migration to the middleware solution should be incremental. It should transparently encapsulate the special peripheries, like the balances. An important issue is scalability. On some days in the year (e.g., before Christmas) there is a considerable higher demand for sending letters than in other periods of the year. The system has to be integrated with other distribution centers and with the management's information systems for exchange of statistical data.

Now we map these requirements to our taxonomies' criteria. The system requires reliable network communication, programming language, and platform independability. An integrated component model should integrate various applications, legacy components, and special peripherals. The system would benefit from a transaction processing monitor or transaction service, because the database connectivity has to handle a larger number of transactions. Since the system is a very large system and should survive a long time, standardization is important in order to be independent of vendor politics. Finally the products must have proven a high reliability in an enterprise context, because failures in the system can cause considerable costs and severe damage to the image of the enterprise.

In the next step we make a pre-selection of candidate technologies. RPC approaches are not a superior solution, since their scalability properties are weak and they are poorly standardized. DCOM suffers from very limited platform independence and from its low-level scalability functionalities. Enterprise Java Beans and RMI are not programming language independent, what makes it hard to incorporate existing legacy applications, their performance is weaker than in

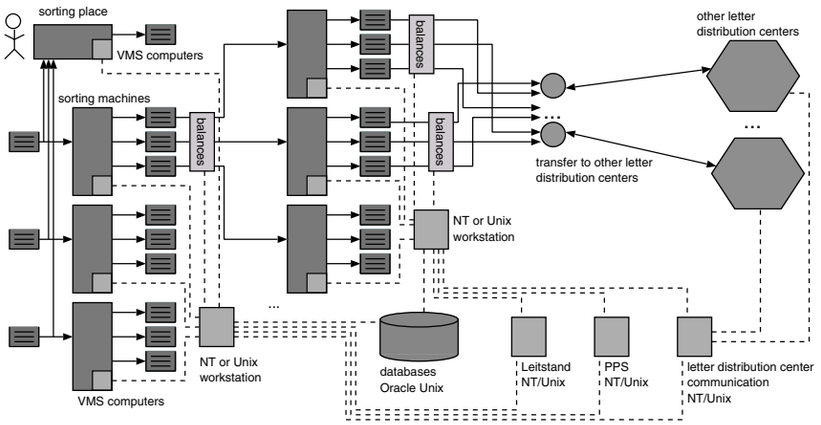


Fig. 5. Letter Distribution Center Example

other approaches. For both DCOM and EJB/RMI it is questionable if the technologies are ready for the scale of the application. A transaction monitor alone has a weak component concept, weak language independability, and is not standardized. Mobile code systems would be the good paradigm for this application, because the code can move around with the baskets, each basket's procedure can easily be customized, mobile agents can gather statistical data locally and convey it to the management department by migrating to the management places, etc. But the current approaches do not seem to be ready for the scale of the application.

Three middleware technologies are candidates after the results of the initial discussion: CORBA, application servers, and MOM. Furthermore combinations are possible. CORBA meets most of the requirements: it is platform independent and language independent, offers an integrated component model, has high scalability and performance, is standardized, and the products are marketable. An application server has its weaknesses in programming language independence and probably in performance, if the application requires client-to-client interaction. In the given application example, most parts of the application are clients and servers at the same time. E.g., a PPS computer is a client to the sorting machines, when it gathers information, but it is a server in providing information about PPS decisions. If not every machine is wrapped behind its own application server, it is unclear how to cluster machines to application servers. The main drawback for message-oriented middleware in the given application is its missing standardization and the overhead of asynchrony, which is negative for net load and performance. It may result in further investments in stronger hardware.

After performing a comparison on the criteria of the taxonomy, we perform a software architecture analysis. For space reasons, here we just investigate three scenarios as examples in Table 2 very briefly. We find the relevance of the scenarios by interviews with the stakeholders. We assume that a change scenario

that was often named is more likely to happen than one which is named only seldomly.

Table 2. Change Scenario Description

Scenario Description	**CORBA**	**Application Server**	**MOM**
What happens if a new platform is introduced?	With CORBA it is quite easy to add a new platform support, if the platform is supported by any CORBA vendor, since the basic CORBA functionalities are quite compatible across vendor implementations.	The change depends on the support of the middleware vendor of the concrete product. With EJB servers it also depends on the availability of Java on the platform.	The change depends on the support of the MOM vendor of the concrete product. It may expensive and exotic platforms may not be supported.
What happens if licensing costs of a technology rise dramatically (e.g., through changes in vendor's pricing policy)?	Using CORBA the effort to change to another vendor depends to what extent vendor specific extensions/services are used. Generally a change is possible with foreseeable costs.	Application server specific application parts have to be re-written in order to change to a new vendor. Programming language depend parts normally, may be reused since for all prevalent languages (like Java, C, Tcl) several similar products are existing.	MOM products can cause significant problems in changing of the vendor, since their models are heavily vendor dependent and since there in not a great variety of comparable products at the market.
What happens if the database technology is changed?	All three technologies offer capabilities for encapsulation of legacy components/database wrappers. With a good design (a database access layer) it should be easy to change the database with all technologies. However, the abstraction from client interaction logic of a transaction monitor would serve better. Runtime means of adaptation (offered by some vendors) also help to transform one database wrapper to another.		
...	...	...	...

Overall we have built a list of all scenarios in the same style. Then we have given marks for each product in each scenario for evaluation. These were found by a discussion of the scenario descriptions with the stakeholders. Upon these marks we have made the concrete choice for the application. In the concrete example a CORBA based architecture was chosen, because it deals with most of the quality attributes better than its competitors, it is a good integration base for integration with other applications, and it handles most relevant change scenarios sufficiently. Nevertheless, for other applications or other enterprises different technologies or combinations could serve better.

6 Related Work

The presented approach deals with the decision for key technologies. Firstly, a general overview and a subjective taxonomy are built. Then we mime example architectures, to find relevant scenarios. Finally, we find very concrete integra-

tion base technologies and explain concrete architectural solutions of legacy integration. Despite the earliness of the study, we provide quite concrete outcomes, without ignoring subjective experiences or company/application-specific aspects. There are several approaches, especially in the field of software architecture, that deal with parts of this process.

In [1] the software architecture analysis method (SAAM) is introduced. Several case studies are presented, including a case study of the CORBA architecture and the architecture of the web. Influences of software architecture on quality attributes and stakeholders of an organization are described in great detail. Other scenario-based architecture analysis methods are described in [2] and [8]. These approaches concentrate on the flexibility of architecture analysis in very early stages of software projects. This methods may be used as a part of our approach in order to find and evaluate scenarios. Generally, these approaches rather concentrate on more concrete architecture and not on very early evaluations and are, therefore, alone not suitable for an early key technology evaluation. They are not accompanied by a clear architectural vision for object-system or paradigm integration.

Architectural styles and patterns rather deal with the last part of the process discussed in this work. In [15,4,14,1] architectural styles and patterns are discussed. In [12] the influence of the styles imposed by middleware technologies is investigated with the conclusion that no style serves best and, therefore, that different application have different middleware needs. Middleware induced styles should be made explicit in form of a style map, that can possibly be defined formally by a architecture description language. The case study in this work shows the entanglement of key technology decision and integration solutions. Therefore, these works are an important companion to the present work. With a clear knowledge of relevant styles and pattern it is easier to explain the integration of the technologies, object systems, and paradigms.

Brown discusses in [3] obstacles and important issues in applying and understanding middleware systems. He identifies a set of central aspects software managers have to adopt in order to successfully use and understand the benefits of middleware technology. Besides the aspects covered by our approach, for every application it has to be investigated, what the additional necessities for the usage of the technology with a concrete application are. Additional boundaries evolving with the usage of the technology have to be considered. Time and costs of adapting to the technology should accompany a final decision. These issues could be taken into concern in form of change scenarios. The approach is not as detailed as our approach and lacks a discussion of the integration problem.

A set of smaller studies solely provide comparison frameworks for middleware technologies (similar to our taxonomy). These approaches lack a discussion of the broader selection process and of the integration task. Raj [13] compares the middleware technologies CORBA, DCOM, and Java RMI very detailed at a implementational level. On a larger example he compares the benefits/liabilities by source code comparison. In [16] a detailed comparison of COM and CORBA technologies for distributed computing is given in form of a decision framework.

From its intention it strongly resembles our taxonomy approach. Unfortunately, none of the named works gives a full fledged overview and a systematic comparison of all relevant middleware technologies.

Thompson [17] defines and positions middleware similarly to our work and propose a four step based approach to select a middleware technology. First the approach identifies the communication types within a business and between businesses. Then the underlying communication models of these types are classed into five communication models, which are conversational, request/reply, message passing, message queuing, and publish/subscribe. On the basis of these models middleware technologies are identified and finally evaluated on candidate architectures. The general steps are similar to our approach, but we doubt that the communication models are a detailed enough characterization of middleware technologies. Any organization of enterprise-size will most likely require all kinds of communication models. Most current middleware technologies implement more than one communication model. For both reasons the communication models can not serve as a good critierion for distinction. Moreover, in early studies when business requirements are not fully known, it can also be hard to find the relevant communication types.

7 Conclusion

Key technologies, like middleware, have significant influence on the software architecture of an information system. The software architecture, in turn, has a severe impact on the realization of quality attributes of a company. Often evaluation studies on key technologies have to performed in early stages of projects for a large number of application. On the enterprise scale the application's requirements are in most cases too diverse to let an imposed, upfront key technology decisions over all applications seem sensible. An early study can identify the relevant requirements/quality attributes, map them in a taxonomy to the technology's properties, and identify an integration base. On example systems technology decisions and integration with existing system can be exemplified. With such a partial result that leaves a lot of freedom for the application designers, an early key technology study can make sense from the technological viewpoint. Rather simplistic overviews of the results, like taxonomies or change scenarios, serve as good starting points for discussion with non-experts in the technological field, if they are clearly mappable to their concerns. The best way to achieve such a mapping is an ongoing discussion with the stakeholders throughout the study.

References

1. L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Reading, USA, 1998. 17, 18, 24
2. P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Proceedings of the International Conference of Software Engineering (ICSE'99)*, Los Angeles, USA, 1999. 17, 24

3. A. W. Brown. Mastering the middleware muddle. *IEEE Software*, 16(4), 1999. 11, 24
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996. 24
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 18
6. M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct. 2000. 19
7. M. Goedicke and U. Zdun. Piecemeal migrating of a document archive system with an architectural pattern language. to appear, 2000. 19
8. N. Lassing, D. Rijsenbrij, and H. van Vliet. Towards a broader view on software architecture analysis of flexibility. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999. 18, 24
9. I. T. Ltd. The orbix architecture, 1993. 12
10. M.Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000. 19
11. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000. 20
12. E. D. Nitto and D. S. Rosenblum. On the role of style in selecting middleware and underware. In *Proceedings of the ICSE'99 Workshop on Engineering Distributed Objects*, Los Angeles, USA, 1999. 24
13. G. S. Raj. A detailed comparison of CORBA, DCOM and Java/RMI. http://www.execpc.com/ gopalan/misc/compare.html, 1998. 24
14. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000. 24
15. M. Shaw. Some patterns for software architecture. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, pages 271–294. Addison-Wesley, 1996. 24
16. O. Tallman and J. B. Kain. COM versus CORBA: A decision framework. *Distributed Computing*, Sep-Dec 1998. 24
17. J. Thompson. Avoiding a middleware muddle. *IEEE Software*, 14(6), 1997. 11, 25
18. S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneos environments. *IEEE Communications Magazine*, 14(2), 1997. 15

An Architecture Proposal for Enterprise Message Brokers

Jörn Guy Süß[1] and Michael Mewes[2]

[1] Technical University Berlin
Einsteinufer 17, D-10587 Berlin / Germany
jgsuess@cs.tu-berlin.de
[2] Fraunhofer Society for the Promotion of Applied Research,
Institute for Software and Systems Engineering
Mollstraße 1, 10178 Berlin / Germany
michael.mewes@isst.fhg.de

Abstract. In large enterprises, asynchronous communication and messaging are gaining importance as integration solutions between applications. The concept of a message broker has been proposed as a universal mediator at the center of business. This paper gives a distinct definition of a message broker by enumerating enterprise critical criteria and describes a reference architecture to meet these criteria. To arrive at the relevant criteria, the message broker is positioned with respect to other middleware solutions like CORBA and MOM/DAD, and limitations and advantages are pointed out. From this comparison the catalogue of critical criteria is deduced and examples are given of how commercial broker products fulfill or fail these criteria. Finally, a reference architecture based on Java, XML and XSL(T) is described that meets the criteria with respect to configuration, execution and extensibility.

1 Limitations of Object-Oriented Middleware Regarding EAI

Middleware, as envisioned by its promoters, primarily views the enterprise as a large, distributed, object-oriented or component oriented system interacting on the basis of synchronous remote procedure calls: Business is modeled as binary objects, talking to each other over a transparent network. This situation comprises three implicit fundamental assumptions:

- The two partners must know of each other, i.e. their interfaces must be well published and well defined. This implies that interfaces have to be global, stable metadata.
- The two partners must be able to communicate. This requires compatibility to a common network medium and a common means of binary communication.
- The two partners must be available; i.e. transactions must be possible. This requires availability of all interfaces at all time or significant effort invested in exception handling for transaction failures.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 27-42, 2001.

Apart from touching on the political implications of data ownership, which arise in every integration effort, the aforementioned factors are also the main obstacles of transferring traditional middleware development to the enterprise scale. Projects are characterized by having local scope, local network or platform and local transactions. As a consequence, the systems that result, although they are distributed to increase performance and availability, are in fact local solutions. The following sections discuss the three aspects of the local character of middleware applications.

1.1 Local Scope

The interfaces for distributed systems are normally architected in a project context, which emphasizes the delivery of end-to-end functionality. The middleware, being "invisible" from both the database and the user interface, and therefore outside of the scope of the process owner, often has its design time reduced to deliver project scope functionality more quickly. As a result, interfaces are narrowed to the task the system in question is being built for. Even if there is a consensus to build cross-company reusable interfaces, the results are complex and unwieldy to use and suffer from the inclusion of too much detail as shown by typical “Jack-Knife-Classes” [1]. For this reason global interface repositories are very rare.

1.2 Local Network or Platform

The middleware approach also lightly assumes that there is such a thing as a binary and network compatible platform. This assumption does not hold for the large number of business critical batch systems, which lack a notion of networks, let alone TCP/IP. The assumption is still hard to cover for the "open platforms", which make up the rest of the business critical systems in the enterprise. Components and ORBs vary, sometimes even if bought from the same vendor, for different platforms or in different versions. And they are regularly not interoperable when provided by different vendors. Since problems in critical interoperability could not be attributed to any one vendor, decision-makers are faced with a stark choice: Either the company opts to choose one product vendor only. This means costly vendor lock-in. Additionally it requires rigid discipline to enforce the standard. Or the company leaves the choice of the middleware product to the project team, so that each system uses its own middleware product. Most companies sensibly opt for the second choice.

1.3 Local Transactions

Finally, to create a business application from objects or components spread out across an enterprise necessitates that the underlying structure can provide distributed transactions spanning several objects or components on different levels. This feature causes a major overhead if implemented. Or it eliminates the structure of the object oriented interfaces if left out, leading to a flat TP-light, two-tier solution. Initially, middleware packages did not even include transactions. Even today, programming transactional middleware in an open manner is cumbersome at best. For this reason, middleware architectures often consist of a well-architected non-transactional information system, and a one-layer transaction system to perform updates, which maps methods 1:1 to stored procedures of the underlying database.

2 Message-Based Architectures are Better Suited to Provide EAI

The asynchronous messaging paradigm known as message oriented middleware and distributed architecture design (MOM/DAD) offers a different approach. It involves modeling the interaction of system elements as a message exchange: messages are placed into a transaction-safe queue, which reliably delivers them to a recipient.

2.1 Advantages: Simple Semantics and High Reliability

Its greatest advantages are the ease in creating parallelism and the simplicity with which discernable, low-communication tasks can be modeled. Integration can also be undertaken on the basis of text messages, which have explicit semantics. The solution is also technologically simple, as it is sufficient to transfer a text (file) from system to system as soon as a connection becomes available. Reliable file transfer capabilities (like OFTP), as opposed to the binary connections required by remote procedure calls, are available for most host systems.

2.2 Disadvantages: Complex Synchronization and Brittle Units of Work

However, the asynchronous paradigm suffers from the complexities of request-response modeling, which is difficult to visualize and cannot be programmed easily, as it involves programming different threads of execution. It is also hampered by an overhead created by small and smallest units-of-work.
For these reasons, message oriented middleware was not successful *within* system development projects, but gained a lot of popularity as a connection medium *between* systems. Many enterprises opted for integration by message queue and had considerable success, because of good vendor support on the ubiquitous host systems and the simplicity of the approach.

2.3 Size Limitation: The N²-Problem

The application of this approach is limited by the N²-problem, described in [7] as „Order (N X N) Interfaces“, and as „Enterprise Chaos“ in [6]. Without well architected common interfaces the number of interfaces that have to be maintained develops towards N², as can be seen on the left in figure 1. Each connection is the equivalent of an individual, highly specialized contract. The cost is incurred through the time that has to be invested to negotiate an individual vocabulary every time, and to enforce data quality with respect to the contract. Because of this factor the ancestor of electronic business, the electronic document interchange standard (EDI), which was based on dialects of standard contracts provided by the United Nations, never gained much importance.

3 Message Brokers Offer a Solution to the N²-Problem of EAI

The message broker can help to solve the N²-Problem, because it serves to accommodate changes in interfaces, to create interfaces dynamically and in such a way to

change business without ever interrupting it. Interfaces in the sense of the broker are simply document types of common interest to the business. Every system is seen as a publisher of messages, irrespective of the subscriber. This allows every system owner to start out with a well-defined set of messages representing the information of the system that are considered to be relevant to overall business. In setting up and making known the message type the system owner has to explain the business relevance of the items to the person handling the broker. In this way, the services running the broker become business experts by exploring the source systems semantic information. Other systems' owners, when requiring some information from within the business can seek their expert knowledge. Applying transformation rules the broker can then supply systems with outputs formatted for their individual needs.

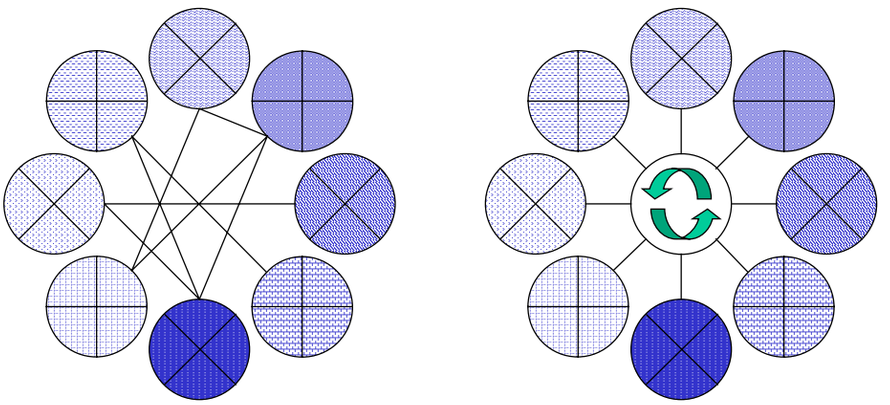


Fig. 1. N^2-problem of integration and the message broker solution. Maintenance effort increases to square order, if individual system-based contracts are created. With semantic contracts effort only grows in linear order

If a message broker is used in this way, it

- assures reliable automated delivery of business data formerly exchanged in tedious manual procedure. This is the argument fielded by traditional, bilateral EDI exchanges, however it is strengthened by the internet as an ubiquitous network which has radically lowered communication cost in comparison to classical EDI VANs.
- builds a corpus of documents, implicitly standardizing the company and preparing it for B2B exchanges. This differs from the basic situation in EDI because XML as a description standard provides legible, understandable and immediately usable documents, which can be created with low overhead. As a result, *businesses* as opposed to *government organizations* are now getting involved in the publishing of DTDs (compare efforts like xml.org) and adapt them to their changing business.
- builds an understanding of systems and allows for limited centralized supervision via observation of the queue length on each of the systems, according to [3].

However, the two limitations described in the preceding sections, incurred through the underlying messaging architecture, still apply:

- Failure through Brittle Units of Work
- Failure through Complex Synchronization

3.1 Failure through Brittle Units of Work

To avoid suffering from the overhead, system integration by message broker should be based on larger units-of-work, which should describe discernable, logical business items and facts. Larger units-of-work make better documents. To facilitate reusability even across widely different platforms, data should be formatted as XML-documents. It is necessary to have meaningful document-types, which can (and should) be reused as is, and do not need a lot of integration or interpretation. As a rule of thumb, a message broker should only deal with document messages that a human would want to read also. This ensures semantic soundness.

3.2 Failure through Complex Synchronization

Middleware-based enterprise systems should not be mixed up with a message broker, but connected to it. The message broker is a hub between systems, which in turn should be built around the most fitting technology applicable to the task. The attempt to use a message broker *inside* a synchronous system leads to three main negative effects:

- Development is complicated, as an asynchronous development paradigm is enforced where it may not be applicable.
- The quality of the system's design is lowered, as brittle messages are created, which expose system internals.
- Centralized disarray is created, as the operators of the broker have to understand and deal with all internal messages of all different systems.

The message broker is an extension for intra-business and inter-business communication and integration. It is not suited as a tool for system development, and it does not compete with function-based, object-oriented or component-based middleware.

3.3 Intuitive View of the Message Broker

In a simplified way, the message broker can be viewed as a number of transformation services running in parallel, invoked and surrounded by the publish and subscribe middleware. This configuration is shown in figure 2.

When a system submits a message, the message is stored in a queue. The broker receives the message from the queue and transforms it according to the document type of the message. The results are then placed into the queues of the subscribers by the publish-subscribe-middleware (hereafter referred to as PSM).

4 Message Brokers must Meet Enterprise Critical Criteria

Message brokers should provide businesses with enterprise critical services. In our research we found that, in order to arrive at a conclusive definition of the term mes-

sage broker, it is useful to define the "Enterprise Critical Criteria" that must be mapped onto the architecture. In [8] and [9] we use these criteria to examine four commercial products: TIBCO's TIB/MessageBroker, TSI's Mercator, Microsoft's BizTalk Server 2000, and STC's e*Gate. We chose the range of products aiming to provide a realistic picture of the development scene with its traditional EDI tools (TSI, STC) and platform-based newcomers (TIBCO, Microsoft). IBM's MQS Integrator V.2 was not available to us, as the company only wanted to present the product in the lab, but not submit it to any tests. The first test series was based on case studies and a standardized scenario to explore the handling. To facilitate the examination we grouped the criteria into three categories, which are at the same time categorical criteria: Configuration, Execution and Extensibility.

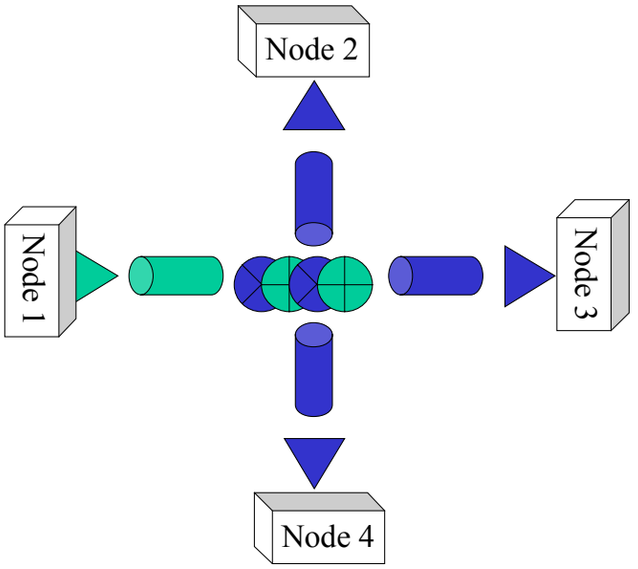


Fig. 2. Intuitive view of the message broker. A publisher submits a semantic message to the system, which gets transformed into all derivable semantic business messages. The results are sent to all subscribers

- **Configuration:** This refers to the ability of the broker to create integration solutions not by programming but by configuration. The configurability of the broker therefore is the first category.
- **Execution:** After the configuration of the message broker has been fixed it has to be executed. The process of execution is the second category of the examination.
- **Extensibility:** The goal of the message broker is to provide a fast and cost-efficient integration solution. Therefore it has to accommodate many different system environments, a necessary condition that is hard to accomplish for systems by many different vendors. Message brokers must be simple to install and to extend in every system environment.

Through the analysis of the application scenario and using the intuitive view of the message broker, a catalogue of criteria was deduced from these three categories.

Based on this catalogue (see Table 1) we have further examined the message broker products in specific second 'torture test' series which explicitly probed the characteristics in the execution category. A third series, which will establish a benchmark is under development.

Table 1. Catalog of examination criteria. The factors described should reflect the requirements on a core enterprise service. Factors may be interdependent

Configuration	*Execution*	*Extensibility*
Rule definition	Monitoring	Supplied adapters
Rule execution	Performance	Adapter programming
Dynamic configuration	Availability	Function library
Prioritization	Reliability	Function programming
Parallel execution	Scalability	
	Platform independence	

The following sections adhere to this structure of organization. Within each section the criterion is explained and the negative effect described if the criterion is not met. We point the reader to [8] and [9] for closer information on the results, where these sections can be found in full detail.

4.1 Configuration

The examination of configuration parameters can be subdivided into the examination of rule definitions, execution of rules, the possibility to make changes at runtime, prioritization of messages, and the order of execution as well as the ability to match messages.

Rule Definition is the central means of configuration in a message broker. In our definition, rules are a denotation of transformations between context-free languages.

- Without a standardized rule definition language, the user is tied to the broker product. Consequently, no standard package of rules can be developed independently of the broker, and be sold for a domain (e.g. SWIFT compliant transformations for banking).

Rule Execution describes the application order of commands. The examination of the execution order of the rules provides insight into the efficiency of rule execution and points to possibilities for the optimization of the process.

- If rule execution is not explicitly controlled or described, predictions about performance are difficult to make. This may lead to bad estimates and increased costs for hardware to assure that enough resources are available for critical processes.

Dynamic Configuration is the possibility to configure hard- and software without having to deactivate it, thus enabling changes at runtime. Depending on the use case the message broker is subject to relatively frequent changes. E.g. new applications must be connected to it, changes to message formats accommodated.

- If a broker cannot be configured at runtime, it has to be stopped to accommodate the change. All transformations running on this instance of the broker will also stop and all business performed by these transformations will cease.

Prioritization is the ability to assign an order of execution according to rules. For business reasons it can be sensible to give priority to certain messages.

- If this feature is not available, transformations cannot be ordered according to the importance of business items. This means that additional hardware has to be used to ensure that critical transformations are performed in time.

Parallel Processing describes the ability to run two or more applications concurrently. This implies a performance gain in the processing of each message.

- If parallel processing is not available, the response times add up as a sum and not as a statistical mean.

4.2 Execution

The configuration of the execution can be analyzed in terms of the execution supervision or monitoring, the speed of or performance inhibitions in execution, the likeliness of interruptions during execution, and the possibility to distribute execution of the message broker as well as available platforms for execution.

Monitoring describes the supervision of active soft- and hardware. Since the whole flow of communication is being piped through the message broker, it can be used as a central node for supervision of all peripheral equipment. Device management data can be used to create statistics. By employing these data the flow of communication and the processing of rules can be optimized and error sources can be localized.

- If monitoring is not available, the performance parameters of the surrounding systems cannot be supervised.

Performance describes the speed of processing in software and hardware. The message broker's performance must accommodate all communication between applications. If a relevant number of applications is connected to the message broker they will cause a data volume in messages of such a size that the message broker must be highly efficient to provide the routing and mapping rules for each message in an acceptable amount of time.

- If performance is insufficient, communication via the broker will cease once queuing resources are exhausted, and the broker will be unavailable.

Availability - If the message broker can only process a limited number of messages the sender has to be notified once no more messages can be accepted. In this case the broker is not available. Ideally the broker should always be available and applications should not have to know about its existence.

- If the broker cannot notify submitting applications of its unavailability, messages will be lost.

Reliability and Fault Tolerance describe the ability of soft- and hardware to catch errors, which would lead to a crash or system failure, and to recover from such a failure. If a message broker becomes unavailable because of a system crash the whole communication between all connected applications will fail.

- If recovering the broker takes long, downtime cost hurts the business. If the broker does not recover cleanly, messages will be lost.

Scalability and Distribution describes the ability to provide a sufficient function if the soft- or hardware or the application context is changed in range or volume. As the message broker is a central node providing the whole communication of the business, its performance must be sufficient to process all messages in an acceptable time frame. One possibility to deal with this is to distribute the message broker itself. For example there could be several message broker instances on several node computers.

- If the broker cannot be distributed or scaled, the maximum size of the broker is limited to the largest available hardware of all platforms on which the broker is implemented. This implicitly limits the maximum size of the business.

Platform Independence describes the independence of software and hardware of its system environment (hardware, operating system, etc.). All applications in the business including the message broker itself should not be limited to one system environment. It seems sensible that the message broker should not only run under Windows, but also under UNIX, VMS and other environments.

- If the broker is limited to one platform or can only connect to a limited number of platforms, vendor lock-in occurs.

4.3 Extensibility

Extensibility means availability of adapters or functions that can be embedded in the rules. Both kinds of extensibility can be viewed under the aspect of pre-included features as well as new, self-designed features. The examination of message broker extensibility includes a check on the number of available adapters and functions, as well as an examination of the possibilities to develop and incorporate new adapters and functions.

Available Adapters - The message broker must be able to address many different applications. Applications are attached to the broker via adapters. The range of available adapters is the number of the adapters delivered with the broker.

- If the number of available adapters is low, it is more likely that adapters for specific applications will have to be created before a project can be started. This increases the cost of the project.

Adapter Programming describes the possibilities for the user to write additional adapters. Regarding the innumerable number of applications in the field it is understandable that no message broker can provide an adapter for each application. For this reason a message broker must allow for additional adapters and for addressing applications via these adapters.

- If it is not possible to program additional adapters, the use of a broker is limited to the available adapters in the package, and therefore the impact of the broker is limited.

Function Library - In order to define rules, operators and functions are necessary. The more operators and functions are available, the more powerful are the possibilities to define complex rules.

- If the function library does not provide the necessary functions for the business scope in question, transformations involving such functions can not be performed by the broker, but must be realized in auxiliary applications or be provided manually.

Function Programming describes the possibility to add additional functions to the software by implementing them. If the operators and functions supplied with the broker are insufficient, it should be possible to implement functions and operators oneself and integrate them into the broker.

- If it is not possible to extend the function library, the scope of functions in the broker transformations can only be extended by external auxiliary applications.

5 Ability of Commercial Broker Products to Meet the Enterprise Critical Criteria

Our examinations showed, that the commercial broker's available to us

- performed well on most user interface, system interface and extension related points like rule definition, monitoring and function library access and extension, the available adapters and adapter programming;
- were less well designed with respect to architecture aspects like rule execution, dynamic configuration, prioritization and parallel processing;
- were consequently lacking under aspects of performance, availability, reliability and fault tolerance, scalability and distribution;
- were proprietary in their format of rule definition, adapters, function interface formats and choice of platforms.

The brokers were either new designs like lightweight Java-based implementations and COM servers, or extensions of existing EDI solutions.
We subjected brokers to tests…

- …where a “bottom function”, that went into recursion, was called as a side effect from transformation. As a result rule processing ceased and often the host computer of the broker instance was stalled.
- …where two rules were connected in the form of a pipe, but the output parameters of the preceding rule did not match the input parameters of the following rule.
  As a result, transformation generally caused an error at runtime, which was not detected at design time.

- …where a function was deliberately programmed to raise an exception or cause an error code, when called as a side effect from transformation. We found that the exceptions were handled very differently and could not be used to control the processing of the broker (e.g. rolling back the transformation as a transaction, or continuing).

Because of these experiences we assume that most broker products today do not meet Enterprise Critical Criteria, because they are not architected to deliver scalability, availability, reliability, platform independence and open interfaces. We see these points as crucial for the enterprise application of the message broker.

6 Reference Architecture Addresses the Enterprise Critical Criteria

We will now describe the reference architecture for a simple message broker as seen in figure 3, which aims to fulfill the enterprise critical criteria defined above. We will describe the broker scenario and thereby give a design description, which shows how the features fulfill the criteria explained above. Finally, we will describe which features often found in or demanded from broker products we have deliberately left out of the design and why we have decided to do so.

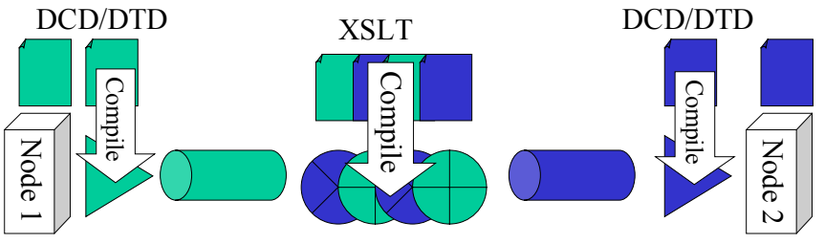


Fig. 3. Reference architecture for a message broker

To describe the features of the architecture we will track the message from generation to consumption.

Message Creation by Adapter The message is created as a side effect of a function call in a connected system, described here as node one. The conversion of a function call or an event into a message is achieved by the use of adapters. The resulting message is always an XML document of a type, which has to be registered with the broker. The notion of adapters is based on research of the DaimlerChrysler technology center described in [5]. They are dynamically compiled from a description language (documented in [10]) using two XML documents for each system: platform and instance description. Data types are based on CORBA. The adapters themselves are Java programs. The concept of automatic generation of adapters allows quick connection of systems to the broker, as the system owners only have to identify well-documented and relevant function calls or events.

criteria: dynamic configuration, parallel processing, platform independence, adapter programming.

Enlisting the Message into the Queue After receiving the equivalent of the message from the connected system, the adapter attempts to enlist the message into the queue. If the adapter does not succeed to finish this transaction…

- In the case of a nontransactional system it raises an exception in the local system monitor and logs the error to a local file. If local file space is sufficient, messages will be queue locally on disk.
- In the case of a transactional system and in the context of a transactional call it votes to abort the transaction as the queue is transparently enlisted as a resource.

criteria: monitoring, reliability, and platform independence.

Delivering the Message to the Publish-Subscribe Middleware and DTD-Repository When the message arrives at the center, it is submitted to a publish-subscribe-middleware. A document type repository enhances this system. The repository is used to identify the document type and optionally validates the document. It then delivers the document to all registered subscribers of the type. There are two types of subscribers:

- Subscribing systems are sinks and consume messages.
- Subscribing transformations are rules, which convert the messages and enlist them again in the queue to the PSM.

Since there is the potential for recursive transformations and cycles heuristic algorithms must be used to traverse the graph of the document type repository. The detection of cycles is a NP-complete problem.

criteria: dynamic configuration, prioritization, monitoring, scalability and distribution, platform independence.

Transformation Engine The transformation engine is the core of the broker. It is architected like the Java Server Page engine. To enlist a new transformation in the repository, it is described by an XSL style sheet. Each newly enlisted style sheet is compiled into a runnable class of Java code, which can either be started as a stand-alone process or enlisted in a thread group in a host process. Each style sheet can be characterized by a property as system load, provider load, demand load. Each style sheet is assigned a timeout after which the transformation must be finished.

Defining Rules via XSL The main function of the broker is to add different markup to documents being processed, and to rearrange the elements contained in the document into a new semantic form. The extensible style sheet language is perfectly suited for this purpose. These transformation descriptions can be translated into executable binary objects, which efficiently translate one document type into the other. This translation occurs whenever a new style sheet is enlisted. Below is a short excerpt from the XSLT V1.0 specification, which defines the XSL notion of templates. Then we have an example snippet from an XML source document, an XSL translation document, its description and the resulting XML-snippet to explain the principle:

```
<!-- Category: top-level-element -->
<xsl:template
match = pattern
name = qname
priority = number
mode = qname>
<!-- Content: (xsl:param*, template) -->
</xsl:template>
```

A template rule is specified with the `xsl:template` element. The `match` attribute is a Pattern that identifies the source node or nodes to which the rule applies. … The content of the `xsl:template` element is the template that is instantiated when the template rule is applied
Now the short example on how transformation is defined and used in the context of the broker:

If we confirm an order to a customer, it might playfully look like this:

```
<confirm>
Dear wealthy credit card customer,
You successfully ordered <quantity>42</quantity>
<stockitem>wombat widgets</stockitem> for <grandto-
tal>19.99</grandtotal> which will arrive at your door-
step within 24 hours.
Regards, digitally
The WidgedCorp. Message Broker
</confirm>
```

The broker would convert that into a mail to our packaging personal using this fragment:

```
<xsl:template match="confirm">
<package>
<xsl:text> Hi, would you kindly package </xsl:text>
<xsl:value-of select="quantity"/>
<xsl:text> of our </xsl:text>
<xsl:value-of select="stockitem"/>
<xsl:text> for pickup in the afternoon? </xsl:text>
<xsl:text> Regards, Messy, the Broker </xsl:text>
</package>
</xsl:template>
```

The actually generated mail to the packager would then contain this:

```
<package>Hi, would you kindly package 42 of our wombat
widgets for pickup in the afternoon? Regards, Messy,
the Broker<package>
```

criteria: rule definition, dynamic configuration, prioritization, and platform independence

Starting Rules According to Their Importance Depending on whether the style sheet is characterized as system load, provider load or demand load its associates its executable is loaded at different times:

- System load executables are loaded when an instance of the broker starts on a node. These rules cannot be removed unless the whole instance is stopped. These rules are always available and provide fastest processing.
- Provider load executables are loaded when a queue associated with a certain message type connects to the broker. The repository has to derive all dependent provider load executables by graph traversal and activate them. These rules can be removed and configured, if all providers of all associated types disconnect from the instance. These rules are good balance between configurability and availability.
- Demand load executables will be loaded, when a document of the respective input type is available. After the transformation, the process is terminated. These rules can be configured dynamically but have a high overhead at runtime, as processes are loaded to and removed from memory.

criteria: Rule Execution, dynamic configuration, prioritization, parallel processing, monitoring, performance, availability, reliability, scalability, platform independence

Connecting to Middleware via XSLT-Extension Functions While in its simplest form XSL describes a mapping from one document format (DTD) to the other, XSLT also allows the introduction of procedural elements expressed externally via a namespace. This allows connecting to arbitrary middleware which thus can be tied into the transformation. As a description for the middleware, the same language that is used for the adapters can be applied.

This change makes transformations nondeterministic and opens up the possibility for locking, as function calls to middleware do not return. Because of these grave implications, function extensions should be applied with extreme care and only reliable, context-free components chosen as targets for integration. Since there is no means to determine if a function will ever return, the transformation engine as the controlling instance will abort transformations exceeding the timeout defined in their stylesheet and report the failure in the log.

criteria: platform independence, available adapters, function programming.

6.1 Delivering the Resulting Message

The resulting message is delivered either through an adapter to the subscribing systems or to a file system, which a native XML client can access. If the results cannot be put into the queue or the queue cannot deliver the result, an exception is raised at the PSM console of the broker.

criteria: monitoring, availability, and reliability

6.2 Features Deliberately Left Out

Many brokers offer additional features, which are left out in this design to accommodate the enterprise critical criteria, which we perceive as primarily important.

No Matching Many brokers offer to combine several messages into one message or to match questions to answers by means of a session. This results in the design of transactional stores in the broker and violates the criteria of dynamic configuration, parallel processing, monitoring, performance, reliability, scalability. We see combination of messages as an explicit service of external applications (like data warehouses). Applications are best suited to handle transactions. The complexity N:1 matching and session handling creates in the broker is prohibitive, because processes must remain simple and transparent at all time.

No Proprietary Programming Model Many brokers store rules in proprietary formats. This leads to difficulties in porting and debugging broker applications and violates the criteria of monitoring, performance, reliability, scalability and platform independence. The internal operations of the broker thus cannot be analyzed using standard tools.

7 Conclusion and Outlook

Message brokers and message broker methods are a convenient vehicle to provide integration inside and across enterprise boundaries. They do not compete with, but complement existing function-based, object-oriented, or componentized middleware. Internally they enable the choice of the most fitting platform for every project.
The application of a method broker is a *process* based on a *tool*. Like the OO-paradigm at its onset required new tools and methods to be discovered, so does the messaging paradigm now. We anticipate that the architecture described in this paper will be close to the outline of forthcoming basic messaging tools. For this reason it is sensible to build our exploration of methods on this model. We believe it is sensible to look into methods for the application of message brokers, because messaging by its simplicity in large granular transaction represents well the processes of business across division and enterprise boundaries. Consequently, we see the possibility to grow the standards of communication evolutionarily and to preserve the freedom of choice over the form of contract as unprecedented strong points. We expect that electronic document interchange will become public domain within the next two years.

References

[1] W. J. Brown, R.C. Malveau, H.W. McCormick III and T.J. Mowbray, *Anti Patterns.* (John Wiley & Sons, Inc., New York, 1999).
[2] J. M. Carroll, M. Beth Rosson, G. Chin and J. Koenemann, *IEEE Transactions on Software Engineering* **, 24,** 1156--1170 (1998).
[3] C. F. Goldfarb und P. Prescod, *The XML Handbook.* (Prentice-Hall PTR, 1998).
[4] Gartner Group, Application Integration: Better Ways to Make Systems Work Together. (Stamford, Conn, 1998).

[5] K. Hergula and T. Härder, Eine Abbildungsbeschreibung zur Funktionsintegration in heterogenen Anwendungssystemen. (Berlin, 1999).
[6] D. Linthicum, *Enterprise Application Integration.* (Addison-Wesley, 1999).
[7] T. J. Mowbray and R. C. Malveau, *CORBA Design Patterns.* (John Wiley& Sons Inc., New York, 1998).
[8] M. Mewes, Anwendungsbezogener Vergleich von Message-Brokern und Konzeption eines optimalen Message-Brokers. (Berlin, 2000).
[9] J. G. Süß, M. Mewes and E. Emilov, Benchmarking Message Brokers: Examination and Definition based on Enterprise Critical Criteria. (Berlin, 2000).
[10] K. Wissmann, Erweiterte Konzepte zur Funktionsintegration. (Universität Ulm, Ulm, 2000).
[11] H. Weber, in *Dagstuhl Seminar #98092* H. Müller, ed. (IBFI, Schloß Dagstuhl, 1998), .

Resource Management

Stoney Jackson and Prem Devanbu

Dept. of Computer Science, University of California
Davis CA 95616
{jacksoni,devanbu}@cs.ucdavis.edu

Distributed object systems have several clear advantages. They can be used to provide reliability, accessibility, durability, and scalability. They can also take advantage of computational parallelism. Distributed object programming has been popularized by the Web, and made feasible across heterogeneous systems by middleware technology. While these technologies make distributed object programming viable, they providing meager assistance for designing, deploying, and maintaining distributed object systems.

The following papers present two approaches addressing the design, deployment, and maintenance of distributed objects. These approaches use information about a system's resources as a primary concern in a distributed objects deployment. Felix provides a practical way of capturing a system's resources and the objects' characteristics as annotations. Using these annotations, he demonstrates a deployment plan can be constructed, denoted, and visualized. Hector's work takes a more formal approach developing a model for model for a distributed system's resources and the objects to be distributed. He demonstrates how the model can be used to reason about the deployment of those objects.

Several questions are still unanswered. What constitutes a system resource? What resources are worth considering when? There may be other aspects besides a system's resources that are worth considering in distributed object design and implementation.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, p. 43, 2001.

The Importance of Resource Management in Engineering Distributed Objects

Hector A. Duran-Limon and Gordon S. Blair

Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK.
Tel: +44 1524 593141 Fax: +44 1524 593608
{duranlim,gordon}@comp.lancs.ac.uk

Abstract: Middleware technologies such as CORBA and DCOM have been developed as a means of tackling heterogeneity and complexity problems inherent in distributed systems. However, more work still need to be done to develop methodologies for the construction of distributed objects. In addition, little attention has been paid to the development of methodologies for the configuration of computational resources among distributed objects. This paper introduces a resource configuration description language (RCDL) for the specification of the resource management of distributed systems. This language is based on both a resource model and a task model. The former offers various levels of abstraction for resources, resource factories and resource mangers. The latter then provides a fine- and a coarse-grained approach to allocate resources to both application services and middleware services by breaking such services into task hierarchies. Finally, we use reflection as a principled means to obtain a clear separation of concerns between the functional and non-functional behaviour (e.g. resource management) of distributed systems.

1 Introduction

Object-oriented middleware technologies, such as CORBA and DCOM, have recently been developed as a means to solve complexity and heterogeneity problems inherent in distributed systems. These technologies have succeeded in providing standards for a common programming model that addresses the problems mentioned above. However, there is still a lack of modelling and analysis methodologies for the construction of distributed objects. Although there is some work in this area, there are some important aspects that have often been ignored such as that of resource management. Object-oriented real-time distributed applications require timeliness guarantees to work properly. Therefore, resource management plays an important role in the process of engineering such kind of applications. There has been some work on this area [OMG99a, Pal00, Schmidt97], however, most of these approaches end up producing complex code which is difficult to maintain, evolve and reuse. Such a complexity is the result of tangling the functional behaviour of real-time distributed applications with non-functional aspects. In addition, some of these solutions usually model resource management for single client-object interactions which does not

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 44-60, 2001.

represent any problem for small applications. However, for large applications the code complexity increases considerably.

This paper introduces a *resource configuration description language* (RCDL) for the specification of the resource management of distributed systems. Such a language is based on both a hierarchical resource model and a task model. The former provides various levels of abstraction for resources, resource factories and resource managers. Therefore, both fine- and coarse-grained resource management are feasible within this model. The latter then models both application and middleware services in terms of task hierarchies. That is, such services are broken into tasks which can in turn be recursively split into sub-tasks. Tasks are then associated with top-level resource abstractions. Such an association determines how both resources and resource management policies are allocated to these services. In addition, we make use of reflection as a means of achieving a separation of concerns of the functional and non-functional behaviour (e.g. resource management). Reflection is a means by which a system is able to inspect and change its internals in a principled way [Maes87]. Basically, a reflective system is able to perform both self-inspection and self-adaptation. To accomplish this, a reflective system has a representation of itself. This representation is causally connected to its domain, i.e. any change in the domain must have an effect in the system, and vice versa. A reflective system is mainly divided into two parts: the base-level and the meta-level. The former deals with the normal aspects of the system whereas the latter regards the system's representation.

Importantly, both the task and the resource model offer a high-level of resource management which eases the programming of adaptation strategies in charge of attaining the contracted level of quality of service (QoS). The resource model offers various levels of resource abstraction whereas the task model provides resource management of coarser-grained interactions. Moreover, the use of reflection diminishes code complexity by providing a clear separation of concerns. That is, functional aspects are defined in a base-level program whereas non-functional aspects are defined in a meta-level program. Hence, real-time systems may be configured according to both the application requirements, i.e. quality of service specifications of the application, and the platform of deployment, i.e. the capacity of the computational resources. For instance, consider the specification of a distributed audio application related to different levels of QoS guarantees (e.g. hard, soft and best-effort guarantees). These specifications are then processed to build the code of the application, thus, alleviating the application developer of the burden of dealing with the details of the hard-wiring the desired level of QoS. The work presented here is focused on distributed multimedia systems. In concrete, we are mainly addressing distributed multimedia communication services. However, our approach is not constrained to this area and can be extended to cover other domains.

The paper is structured as follows. Section 2 introduces the overall approach of our middleware platform. Section 3 then focuses on the description of both the resource model and the task model. The RCDL is introduced in section 4. Following this, section 5 describes the implementation of a pre-processor of the RCDL specifications and section 6 comments on some related work. Finally, some concluding remarks are presented in section 7.

2 Middleware Architecture

The base-level of the middleware platform concerns the programming structures of the services provided by such a platform. These services are accessed through the base-level interface. In contrast, the meta-level provides the reification of the non-functional properties of this platform, e.g. the resource management of the objects involved in a particular service. The meta-level provides a means to inspect and modify the implementation of the middleware by accessing a well established meta-interface. For instance, the computational resources involved in a service may be reconfigured by manipulating the operations defined at the meta-level.

We adopt the RM-ODP computational model [ISO95] for both the base- and the meta-level. Objects within this model have several interfaces. Not only operational interfaces but also stream and signal interfaces are supported. Moreover, explicit bindings are supported which offer more control over the communication path between object interfaces.

In addition, the meta-space is structured, but not restricted, as a set of four orthogonal meta-models (compositional, environmental, encapsulation and resources.) This approach was first introduced by AL-1/D [Okamura92] in order to simplify the meta-interface by maintaining a clear separation of concerns among different system aspects. Each meta-model is introduced in turn below.

The *compositional meta-model* deals with the way a composite object is composed, i.e. how its components are inter-connected, and how these connections are manipulated. There is a compositional meta-model per object. The *environmental meta-model* is in charge of the environmental computation of an object interface, i.e. how the computation is done. It deals with message arrivals, message selection, dispatching, marshalling, concurrency control, etc. The *encapsulation meta-model* relates to the set of methods and associated attributes of a particular object interface. Methods scripts can be inspected and changed by accessing this meta-model.

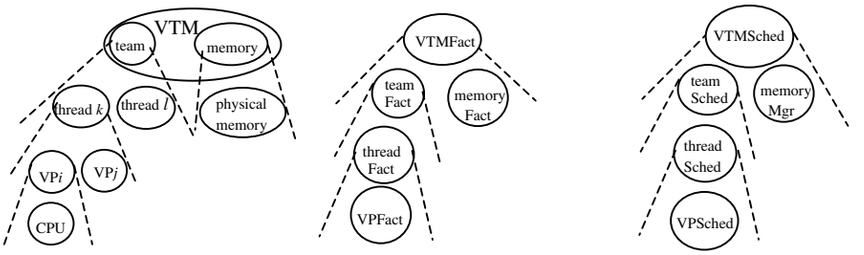
Finally, the *resource meta-model* is concerned with both the resource awareness and resource management of objects in the platform. A more comprehensive description of the resource meta-model is presented below. Further details of the overall architecture can be found in the literature. For example, detailed descriptions of the four meta-models can be found in [Costa00a].

3 Modeling Resources for Middleware

3.1 Resource Model

The most important elements of the resource model are *abstract resources*, *resource factories* and *resource managers* [ReTINA99, Blair99b, Duran00a, Duran00b]. Abstract resources explicitly represent system resources. In addition, there may be various levels of abstractions in which higher level resources are constructed on top of lower level resources. Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher level resources on top of lower level resources. Furthermore, *resource schedulers* are a specialisation of managers

and are in charge of managing processing resources such as threads or virtual processors (or kernel threads). Lastly, the main duty of resource factories is to create abstract resources. For this purpose, higher level factories make use of lower level factories to construct higher level resources. The resource model then consists of three complementary hierarchies corresponding to the main elements of the resource model. Importantly, *virtual task machines* (VTMs) are top-level resource abstractions and they may encompass several kinds of resources (e.g. CPU, memory and network resources) allocated to a particular task.



(a) A hierarchy of abstract resources (b) A factory hierarchy (c) A manager hierarchy

Fig. 1. A Particular Instantiation of the Resource Meta-model

As an example, a particular instantiation of the framework is shown in figure 1 (note, however, that the framework does not prescribe any restriction in the number of abstraction levels nor the type of resources modelled). At the top-level of the resource hierarchy is placed a VTM, as shown in figure 1 (a), which encompasses both memory buffer and a team abstraction. The team abstraction in turn includes two or more user level threads. Moreover, a user level thread is supported by one or more virtual processors (VPs), i.e. kernel level threads. At the bottom of the hierarchy are located physical resources. In addition, a VTM factory is at the top of the factory hierarchy and uses both a memory and a team factory. The team factory then is supported by both the thread and the virtual processor factory as depicted in figure 1 (b). Finally, the manager hierarchy is shown in figure 1 (c). The team scheduler and the memory manager support the VTM scheduler to suspend a VTM by temporally freeing CPU and memory resources respectively. The thread scheduler in turn allows the team scheduler to suspend its threads. Finally, the VP scheduler supports the preemption of virtual processors (although, this should be used with caution as a VP may support the processing of more than one VTM). Conversely, this hierarchy also provides support for resuming suspended VTMs.

3.2 The Class Hierarchy of the Resource Framework

The class hierarchy of the resource framework is shown in figure 2. Such a class hierarchy may be used to build on top of it a particular instantiation of the resource framework. The `Common` class is a mixin class for the three class hierarchies (i.e. resources, factories and managers). The interface of this class provides operations to

traverse any of the hierarchies by recursively accessing either the lower- or the higher-levels. For instance, the resource hierarchy may be traversed by applying the `getLL()` operation at the top-level, i.e. the VTM. This operation would be later applied to the lower-level resources, and so on. Access to both the manager and factory of an abstract resource can be obtained through the interface defined by the class `AbstractResource`. In addition, *machines* are capable of performing some activity [ReTINA99, Blair99b], that is, they receive messages and process them. Thus, machines may be either abstract processing resources (e.g. threads) or physical processing resources (e.g. CPUs). In contrast, *Jobs* [Blair99b] only refer to abstract processing resources since they inherit from the abstract resource class. Both abstract resources and jobs are created by factories as shown in figure 2. In addition, abstract resources are managed by managers. However, since jobs are processing resources, they are managed by schedulers instead.

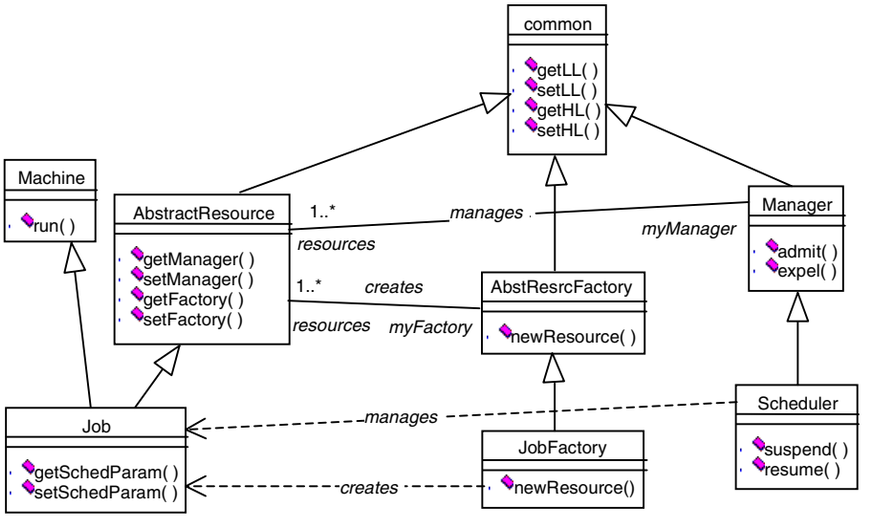


Fig. 2. Resource Framework Classes

The class hierarchy of factories is rather simple. The AbstResrc`Factory` class interface basically exposes an operation for the creation of abstract resources and associates a management policy with it. In case of creating processing resources, a job factory is used and the scheduling parameters are also indicated.

Finally, in respect to the manager hierarchy class, the `manager` class interface basically exposes two operations, `admit()` and `expel()`. The former determines if an abstract resource not managed by this manager can be accepted for being managed. The latter excludes an abstract resource for continuing being managed by this manager. Moreover, the scheduler allows us to suspend and resume an abstract processing resource as shown by its interface.

3.3 The Task Model

The main feature of the task model is that it offers a high-level of resource management by modeling coarse-grained interactions. That is, a task may spawn the boundaries of an object and even those of an address space.

The task model allows us to define both how system resources are allocated in a distributed system and the resource management policies that are used. This model prescribes the level of quality of service for services provided by such a system. For instance, for different services of the same type, such as audio transmission, more than one service offering different level of QoS may be defined.

We define *tasks* from two points of view. From the application point of view a task is an activity performed by the system, e.g. transmitting audio over the network or compressing a video image. In contrast, from the programmatic point of view a task is a method invocation sequence which has an amount of resources assigned for its execution. The task model is concerned with both application services and middleware services. Thus, we take a task-oriented approach for managing resources in which services are broken into tasks and are accommodated in a task hierarchy. Top-level tasks are directly associated with the services provided by a distributed system. Lower levels of this hierarchy include the partition of such services into smaller activities, i.e. *sub-tasks*. This approach offers a fine-grained modeling of resource management. Tasks are not necessarily disjoint and may be interconnected. For instance, an object running one task may invoke another object concerned with a different task. Such a method invocation represents a *task switching point*. A *distributed task* then spawns the boundaries of an address space and is associated with a distributed VTM. Hence, distributed tasks are split into local tasks to facilitate their management.

In addition, there is a one-to-one mapping between tasks and VTMs within an address space. Hence, a VTM represents a virtual machine in charge of supporting the execution of its associated task. VTMs also represent a higher-level of resource management. They are an encapsulation of the pool of resources allocated for the execution of their associated tasks. Thus, a task switching point corresponds to a change in the underlying resource pool to support the execution of the task that has come into play. VTMs isolate the resources that a service uses to have a better control of the desired level of QoS provided by it. That is, the resources a task uses for execution are localised. Hence, it is straightforward to access the resources of a task when its is under-performing to reconfigure its resources.

4. Specification of Resource Management for Middleware

4.1 A Description Language for the Specification of Resource Management

As mentioned earlier, a *resource configuration description language* (RCDL) is used to specify the resource management of the services provided by a distributed system. The language is basically characterised by three sections. Firstly, application and middleware services are defined in a *service template* as shown in figure 3. Within a

service type, several services may be configured whereby each one of them is associated with a QoS region, the task that supplies the required computational resources to provide the given level of QoS and the (composite) object that implements the service. The QoS region refers to the range of QoS values delineated by the minimum and maximum QoS levels offered by a service.

```
Service Template:
      Def service type service_a:
              Def service service_a_V_d:
                      qos_region: qos region of this service
                      task: the task associated with this service
                      object: the object class that implements the service
              Def service service_a_V_e:
                      . . .
      Def service type service_b:
              . . .
      . . .
```

Fig. 3. Service Template

Secondly, a *task template* is the specification of the tasks a service is broken into. Each of these tasks is related to a VTM in the task template. Furthermore, as shown in figure 4, a task template specifies the associated tasks, the abstract resources with their related management policies and the importance of each VTM. A high importance value is assigned to critical tasks whereas lower importance values are assigned to tasks where contention does not have a drastic impact in the system. In addition, sub-tasks inherit importance values from their super-tasks. There is also a *default VTM* that includes all those activities that are not represented in the resource model. That is, such activities would use the resources defined by the default VTM. The mapping of QoS values on to resource parameter values may be achieved by both mathematical translation and trial-and-error estimations as described in [Nahrstedt98]. For instance, the description of CPU resources would include the definition of a team abstraction in terms of maximum CPU usage, number of threads, the management policy and their scheduling parameters.

Any type of resource may be shared between two or more VTMs as long as they live in the same address space. This approach is useful in cases where, for performance reasons, it is needed that tasks share resources but still have certain resources only allocated to them. An example of such a case is a protocol stack in which each layer corresponds to a single task. Thus, each layer has a memory buffer allocated for its own use. However, it uses the same thread along the stack to avoid thread switches in order to achieve a good performance of the system. Allocation of shared resources is defined within the task template as shown in figure 4. Thus, the shared resources clause specifies both a list of the shared abstract resources and a list of the VTMs sharing these resources.

Finally, an *object template* maps object operations on to tasks as shown in figure 5. An object interface may have both exported methods and imported methods [Andersen00]. The former concerns operations that are implemented by the interface's object whereas the latter refers to operations implemented by the object to be bound. A method exported by an object may be imported by another object, thus allowing the latter to access this method locally through its interface. Imported methods can then be considered as delegate operations since they delegate the execution of an operation to its corresponding exported method. Hence, when two

object interfaces are bound, imported methods must match the name and parameters of their related exported methods. Only exported operations may have associated tasks.

```
Task Template:
      VTM h:
               Task: task l
               Abstract resources: description of abst. resources of this vtm
               Importance: m

      VTM i:
                 . . .
             . . .

      Default VTM:
               Abstract resources: description of abst. resources of this vtm
               Importance: n

      Shared Resources j:
                Abstract resources: description of the shared abst. resources
               Vtms: list of vtms sharing these resources

      Shared Resources k:
               . . .
```

Fig. 4. Task Template

In addition, exported methods may determine *multiple task switching points* which define the transition from one task to another. To achieve this, an exported method is associated with a list of tasks. A task is selected from this list according to the current running task. For that purpose, "if" statements are introduced within the definition of an exported method in an object template as shown in figure 5. For instance, task *f* is selected if task *g* is the current running task that invokes the imported method *z*.

```
ObjectTemplate o:
    Interface p:
         operation y: task t
         operation z: task f if task g
                      task u if task v
               . . .
    Interface q:
               . . .
        . . .
```

Fig. 5. Object Template

It is worth mentioning that an object may be associated with several tasks and that there are two cases whereby this may happen. The first case is when an object is shared in different object configurations. As an example consider a task graph of a video stream connection v*a* which includes a filter object bound to a compressor object and these objects are part of task t*y* and t*z* respectively. Similarly, there is another video stream connection v*b* that uses the same instances for both the filter and the compressor, although they are associated with t*i* and t*k* respectively. Thus, the compressor will switch to task t*z* if the filter was executed as part of task t*y*, otherwise it will switch to task t*k*. Task switching may be achieved by defining "if" statements. In the second case, each of the methods of an object may be associated with a different task. For instance, consider a stub object with the marshal and unmarshal

methods. These methods may be associated with different tasks since they are concerned with different activities, namely sending messages and receiving messages.

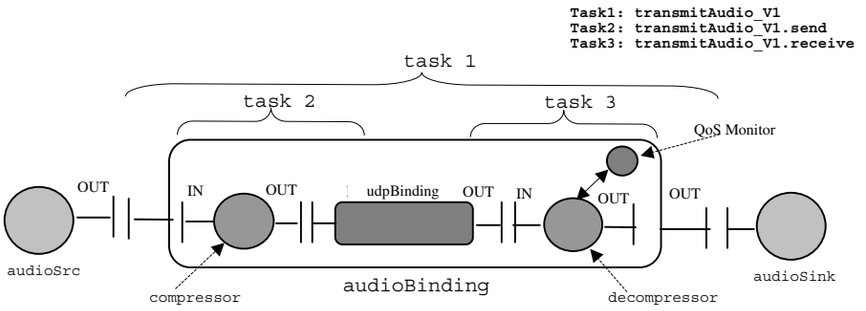


Fig. 6. An Instance of an Audio Stream Binding

4.2 An Example

As an example consider a middleware platform in charge of providing the service of an audio stream binding as depicted in figure 6. Task 1 regards the activity of transmitting audio over the network. In addition, task 2 and task 3 are sub-tasks of task 1 and are concerned with sending and receiving audio streams respectively. Resources of this service are then configured by the RCDL as shown in figure 7 (for the sake of brevity we only show the object templates of the object `audioBinding` and the object `compressor`).

The service `AudioCommService_V1` has associated a QoS region which is supported by the task `transmitAudio_V1`. The specification also defines that the object class `AudioBinding` implements the service. The task template specification defines the task `transmitAudio_V1,` which encompasses both roles transmission and reception of data streams. In addition, it breaks this task into finer-grained tasks whereby task 1 is mapped on to `vtm_1` whereas task 2 and task 3 are mapped on to `vtm_2` and `vtm_3` respectively. The importance metric of `vtm_2` and `vtm_3` are inherited from `vtm_1`. As a result, `vtm_1` is a distributed VTM whose abstract resources include `vtm_2` and `vtm_3`. In contrast, both `vtm_2` and `vtm_3` have associated each a bundle of resources including their management policies. Finally, the object templates map object operations on to tasks which in turn are associated with VTMs. For instance, the operation `send()` of the interface `IN` that belongs to the compressor object is mapped on to the task `transmitAudio_V1.send`. As a consequence, the execution of this operation is supported by `vtm_2` as specified by the task template. The resource configuration description of the audio communication service and the application code are then compiled to obtain the initial resource configuration of the system.

After being instantiated, the application is able to provide services as they were configured. Hence, the binding service is then requested as follows:

```
binding = AudioCommService(audioSrc.OUT, audioSink.IN, qos_spec)
```

The parameters passed to the request of the service are the interfaces to be bound and the desired level of QoS for that binding. The QoS manager then selects the most adequate service within the requested service type. In this case we assume that the selected service is `AudioCommService_V1`. As a consequence the QoS manager identifies if there are enough resources to provide the required level of QoS by inspecting the spare resources in the VTM associated with the task `transmitAudio_V1`. In an affirmative case, a new audio binding is created.

```
Service Template:
      Def service type AudioCommService:
               Def service AudioCommService_V1:
                        qos_region: qos_region_x
                        task: transmitAudio_V1
                        object: AudioBinding


Task Template:
 VTM 1:
      Task: transmitAudio_V1
      Abstract resources: vtm_2, vtm_3
      Importance: 5
 VTM 2:
      Task: transmitAudio_V1.send
      Abstract resources: description of resources of task 2

 VTM 3:
      Task: transmitAudio_V1.receive
      Abstract resources: description of resources of task 3


Object Template AudioBinding:
      Interface IN:
               send: transmitAudio_V1.send
      Interface OUT:
               receive: transmitAudio_V1.receive
Object Template Compressor:
      Interface IN:
               send: transmitAudio_V1.send
```

Fig. 7. An Example of RCDL Definitions.

As an example of dynamic resource reconfiguration consider that the QoS monitor has detected that packet loss has increased over its limit due to traffic congestion. To tackle this situation the compressor and decompressor objects might be replaced with a more aggressive scheme to reduce the use of network bandwidth. However, the QoS manager soon realises that the VTM supporting the execution of the sender side does not have enough processing resources to deal with the new scheme since the thread that operates on the compressor belongs to a scheduling class that has already reached its maximum capacity. Hence, the QoS manager moves this thread into another scheduling class with spare resources. For this purpose, the QoS manager, which happens to be located in the receiver side, performs the following operations:

```
taskDict = encapsulation(binding.OUT).getAttribute('taskDict')
vtm_3 = vtmFact.getVTM(taskDict.getTask('receive'))
```

```
vtm_1 = vtm_3.getHL()
vtm_2 = vtm_1.getLL()['VTMs'][0]
proxy = vtm_2.getProxy()
team_id = proxy.getLL(vtm_2.get_id())['TEAM']
thread_id = proxy.getLL(team_id)[0]
proxy.setManager(thread_id, scheduler_id, schedParam)
```

The first line reifies the encapsulation meta-object concerning the interface `OUT` of the `binding` object and consequently the task dictionary of such interface is obtained. This dictionary maps the interface methods on to tasks. The VTM that supports the execution of the operation `receive()` (i.e. `vtm_3`) is then retrieved from the VTM factory which keeps the mapping of tasks on to VTMs. The higher-level resources of this VTM are later inspected through the `getHL()` operation. As a result, the distributed VTM supporting the execution of the binding is obtained, i.e. `vtm_1`. The remote VTM, i.e. `vtm_2`, is then accessed and its proxy is obtained. The lower-level resources of the `vtm_2` are later inspected through the `getLL()` operation of the `proxy`. As a result, a team of threads is obtained. Similarly, the team resource is inspected and a particular thread is obtained. Finally, the scheduling policy of this processing resource is changed with another scheduling class calling the `setManager()` operation of the `proxy`. This operation encompasses for the thread the process of both being expelled by the old scheduler and later admitted by the new one. The sequence of all the described operations and their implementation are defined in a meta-level program. As a consequence, the resource management aspect is clearly separated from functional aspects of the middleware code.

5. Implementation

5.1 Processing of Resource Management Specifications

Resource configuration is performed according to the resource specifications defined by the RCDL. A pre-processor of the description language is then in charge of setting up the initial configuration of the underlying resources of the system. Therefore, the pre-processor takes the description of the resource configuration and the application code as its input and produces the instantiation of the three hierarchies of the resource model, i.e. the abstract resource hierarchy, the factory hierarchy and the manager hierarchy. The instantiation of such hierarchies involve two important issues. Firstly, the pools of resources (i.e. VTMs) for the specified services are allocated. Secondly, the mechanism for the wiring of method call interception "hooks" is placed to redirect object invocation calls to the appropriate pool of resources when a service is accessed. These "hooks" are installed, on a per-object basis, at run-time when object classes are instantiated.

The mechanism for the wiring of these "hooks" involves the *methodVStask* and *taskVSvtm* records, maintained by the VTM Factory. These records contain information about the mapping between methods and tasks (defined by object templates) and the mapping between tasks and VTMs (defined by task templates) respectively. In addition, such a factory also maintains the *services* record that

contains information about each defined service. Such information regards the level of QoS, the task and the object class that are associated with a service. Such a record is inspected by the QoS manager to select the service that best matches the required level of QoS when a service type is requested. As a result, an instance of the class that implements the service is created if its associated VTM has enough resources to support this level of QoS. The "hook" wiring mechanism then maps tasks on to object instances at object creation time. For this purpose, objects within the system inherit from the `Object` class. Every time an object is created, the constructor of the `Object` class asks the VTM factory, by invoking the `mapTasks()` method of the factory, to map tasks on to the object that is being created. Such a mapping is accomplished on a per method basis, that is, one or more tasks (in case the method participates in more than one task) are assigned to each method. Such assignments are placed in a record named `taskDict` located in the encapsulation meta-model of the object interface the method belongs to. The purpose of this record is to provide a means for accessing the resources (i.e.VTMs) a method is related to.

Since the implementation of the "hooks" for method call interceptions are out of the scope of this paper, we will only briefly mention how the interception process is done. Dynamic allocation of resources is performed on a per method invocation call basis when a task switch occurs. That is, the environmental meta-model of the callee intercepts a method invocation call if such a method is a task switching point. In such a case, it inspects its encapsulation meta-model to obtain the task associated with the invoked method. If the current task is the same as the method's task, the execution of the method continues using the same resources as the caller's. Otherwise, the VTM associated with the method's task is used to process the invocation call.

5.2 Integrating the Resource Framework into an ORB

We have integrated the particular instantiation of the framework, described in section 3.1, into an implementation of OpenORB [Andersen00] as shown in figure 8. VTMs are top-level processing resources that include both abstract resources (i.e. buffer and network resources) and abstract processing resources (i.e. user- and kernel-level threads). A *distributed VTM* encompasses one *local VTM* and two or more *remote VTMs*. Remote VTMs are abstractions that keep information of the VTM residing in a remote site in terms of both the id of such a VTM and a *proxy* (instance of the `ResourceServerProxy` class) of the *resource server* that resides in the remote capsule. The interface of the resource server exposes operations that delegate the actions of inspection and reconfiguration of resources to the appropriate entities (e.g. VTMs, managers, etc). Therefore, any of the three hierarchies of the resource meta-space may be remotely accessed by the proxy (see section 4.2 for an example). In addition, objects within this ORB may have several interfaces which can be bound to other matching interfaces whereby the exported methods correspond to the imported methods of the adjacent interface. Remote bindings permit connecting interfaces residing in different capsules. Three styles of binding are provided. *Operational bindings* are adequate for remote method invocation whereas *signal bindings* are suitable for one-way notification. Lastly, *stream bindings* provide support for stream communication. In addition, *capsules* represent an address space and allows objects to

invoke other objects residing in a different capsule as if they were in the same capsule. The *name server* allows the user to register both interfaces and capsules and, as a consequence, to access them remotely. Finally, the *node manager* allows several capsules to coexist within the same node. We are aiming for next generation middleware and interoperability with current standards is not a primery concern for this research. However, interoperability with CORBA, for instance, may be obtained by translating (with appropriate tool support) RCDL definitions into Real-time CORBA IDL descriptions. In addition, CORBA interceptors could be used as hooks to implement task switching points. That is, object methods representing such points would have CORBA interceptors in charge of switching to the adequate pool of resources to process the method invocation.

The RCDL pre-processor, the resource framework and the ORB are implemented in Python. We selected Python as the prototype programming language because this language offers some reflective facilities, thus making it easier to implement reflective systems. Python is also an ideal language for rapid prototyping. Similar to Java, Python modules are translated into byte-codes, either manually or the first time they are imported, to speed up the interpretation process. Moreover, since Python can easily be extended with C programs, Python's applications can dramatically improve their performance by programming the most critical modules in C.

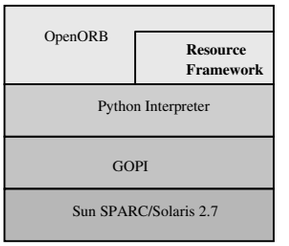


Fig. 8. Implementation of the Resource Framework

Additionally, the Python interpreter was layered on top of GOPI [Coulson98] which is a micro-ORB, implemented in C, that offers support for distributed multimedia applications. In particular, we make use of the thread and memory management features offered by GOPI. User-level threads run on top of kernel threads in the *two-level thread model* provided by this micro-ORB. Priority propagation of user-level threads is achieved by the integration of common schedulers defined at the middleware level. However, further work has still to be done to deal with priority propagation of kernel-level threads. In addition, *user-level threads may be suspended and resumed*. Furthermore in GOPI*, the scheduler of a thread may be dynamically changed with another*. A fixed-priority and an EDF scheduler have been implemented so far. Finally, *memory buffer pools may be accessed*. Therefore, GOPI provides low-level support for CPU and memory management.

6. Related Work

There has been some interest in the development of resource models for distributed object systems. The QuO project [Pal00] provides a framework for the specification of QoS of CORBA object interactions. QoS aspects are specified in QDL which specify QoS aspects in various description languages. A code generator is used to produce the code of contract objects, delegate objects and system objects. The former represent operating regions and application service requirements. Delegate objects support adaptive behaviour upon method call and return. System condition objects interface to resources, mechanisms and ORBs in the system. However this work focuses on client-object interactions whereas our work additionally covers resource management of coarser-grained interactions. The ERDoS project [Chatterjee99] offers a generic and comprehensive resource framework. It provides an extended CORBA IDL that partially captures the resource framework. Similar to our work, resource management is performed at the middleware level. A distributed system is described within three base-level models whereas our approach is based on a multi-model reflection framework. ObjectTime [Lyons98] is a tool that implements UML for real-time constructs [Selic98]. Specifically, the modelling structure of this tool comprises three elements, namely, capsules, ports and connectors. The former are *actors* (i.e. active objects) that communicate with each other through ports. Such ports are objects that implement a particular interface. Finally, connectors represent signal-based communication channels. However, no means is provided to model resources. Recently, a generic framework for modelling resources with UML [Selic00] has been developed and a proposal [OMG00] has been submitted for the OMG adoption. Both, physical and logical resources at various levels of abstractions may be represented within this model. QoS management is modelled in terms of QoS contracts, resource managers and resource brokers.

Other efforts in this area include that performed by the OMG which has recently adopted a real-time specification for CORBA [OMG99a]. This specification presents a platform independent priority scheme and defines a propagated priority model in which CORBA priorities are propagated across address spaces. In contrast with our work, this adopted standard does not support dynamic scheduling. Similar to our work, real-time CORBA introduces an *activity* as a design concept rather than as part of the implementation. However, our framework additionally provides explicit entities (i.e. VTMs) for the support of the execution of such activities In addition, real-time CORBA tangles the application code with the scheduling of activities whereas in our work such scheduling remains hidden from the base-level application code.

Finally, the approach followed by the XMI and the MOF [OMG99b] focus on the interchange and management of the meta-information of a system respectively. Complementary to the work presented here, the latter standard has been used in [Costa00b] to define a type system for a repository of middleware configurations.

7. Concluding Remarks

There is a need of software engineering methodologies for the construction of object-oriented real-time distributed applications. We have presented a description language for the specification of resource management of distributed systems. Computational resources are modelled within a hierarchical resource model with various levels of abstraction. In addition, both application and middleware services are partitioned into tasks and associated with a bundle of resources. A finer-grained control of the resource management of services is achieved as a result of the task partitioning. The resource model also allows us the run-time reconfiguration of resources through a well-defined meta-level interface. In addition, reflection permits us to separate functional aspects from non-functional ones, thus, leading to clearer and more reusable code.

The work has mainly been focused on distributed multimedia systems. However, our approach is not constrained to this area and can be extended for other application domains. For instance, storage resources may be modeled for both database systems and persistence services. In contrast, other higher-level middleware services, such as the transaction service, represent a more complex issue and would require further study.

We are currently developing an ADL for specifying distributed real-time systems [Duran00c]. In contrast with the description language presented here, the ADL will provide a more comprehensive specification of middleware services by embracing other aspects such as component configurations and QoS management structures.

To evaluate the platform, ongoing work is being carried out on the implementation of an audio application on top of the RM-ODP ORB. In addition, we are also addressing the issue of broadening the number of computational resources managed by the system. Future work will also address the integration of the resource model with the QoS management framework defined in [Blair99a].

Acknowledgements

Hector A. Duran-Limon would like to thank his sponsor, the Autonomous National University of Mexico (UNAM). Particular thanks are also due to Jean-Bernard Stefani and his group at CNET. We also thank Rui Moreira for helpful comments on early drafts of the paper.

References

[Andersen00] Andersen A., Gordon S. Blair, Frank Eliassen. “A Reflective Component-Based Middleware with Quality of Service Management”. In Proceedings of the Conference on Protocols for Multimedia Systems (PROMS2000), Cracow, Poland, October, 2000.

[Binns96] P. Binns, M. Engelhart, M. Jackson and S. Vestal “Domain-Specific Software Architectures for Guidance, Navigation, and Control”, Int’l J. Software Eng. And Knowledge Eng., Vol. 6, no. 2, 1996.

[Blair99a] Blair, G.S., Andersen, A., Blair, L., Coulson, G., “The Role of Reflection in Supporting Dynamic QoS Management Functions”, Internal Report MPG-99-03,

Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., 199 January 1999.

[Blair99b] Blair, G.S., Costa, F., Coulson, G., Delpiano, F., Duran-Limon, H., Dumant, B., Horn, F., Parlavantzas, N., and Stefani, J-B. "The Design of a Resource-Aware Reflective Middleware Architecture", In Second International Conference on Reflection and Meta-level architectures (Reflection'99), St. Malo, France, July 1999.

[Chatterjee99] S. Chatterjee, B. Sabata, and M. Brown, "Adaptive QoS Support for Distributed, Java-based Applications". In the Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), St-Malo, France, May 1999.

[Costa00a] Fabio M. Costa, Hector Duran-Limon, Nikos Parlavantzas, Katia B. Saikoski, Gordon Blair, and Geoff Coulson. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. Reflection and Software Engineering, Walter Cazzola, Robert J. Stroud and Francesco Tisato, June, 2000. p. 79-99.

[Costa00b] Costa, F., Blair, G.S., "The Role of Meta-Information Management in Reflective Middleware", Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'00 Workshop Reader, Springer-Verlag, 2000.

[Coulson98] Coulson, G., "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol. 21, No. 9, pp 802-818, July 1998.

[Duran00a] Duran-Limon H. and Blair G. A Resource Management Framework for Adaptive Middleware. In *3th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'2k)*, Newport Beach, California, USA, March 2000.

[Duran00b] Duran-Limon H. and Blair G. Configuring and Reconfiguring Resources in Middleware. In the 1st International Symposium on Advanced Distributed Systems (ISADS'2000), Gdl, Jalisco, Mexico, March, 2000.

[Duran00c] Duran-Limon H. and Blair G. Specifying Real-time Behaviour in Distributed Software Architectures. In the 3th Australasian Workshop on Software and System Architectures, Sydney, Australia, November, 2000.

[ISO95] ISO/ITU-T. Reference Model for Open Dist. Processing. International Standard ISO/IEC 10746, 1995.

[Lyons98] Lyons A., "Developing and debugging Real-Time Software with ObjecTime Developer. Available at http://www.objectime.com

[Maes87] Maes, P., "Concepts and Experiments in Computational Reflection", In Proceedings of OOPSLA'87, Vol 22 of ACM SIGPLAN Notices, pp 147-155, ACM Press, 1987.

[Magee95] J. Magee, N, Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures", Proc. Fifth European Software Eng. Conf. (ESEC'95), Sept. 1995.

[Medvidovic96] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style", Proc. ACM SIGSOFT'96: Fourth Symp. Foundations Software of Eng. (FSE4) pp. 24-32, Oct. 1996.

[Nahrstedt98] Klara Nahrstedt. Hao hua Chu, and Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. Journal of High-Speed Networking, Special Issue on Multimedia Networking, 7:227-255, 1998.

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

[OMG00] Joint Initial Proposal for a UML Profile for Schedulability, Performance, and Time. http://www.omg.org/, 1999, Object Management Group.

[OMG99a] Real-Time CORBA 1.0 Specification, http://www.omg.org/, 1999, Object Management Group.

[OMG99b] Meta Object Facility Specification. http://www.omg.org/, 1999, Object Management Group.

[Pal00] P. Pal, J. Loyall, R. Schantz, J. Zinky, R Shapiro and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In *3th IEEE*

International Symposium on Object-oriented Real-time Distributed Computing (ISORC'2k), Newport Beach, California, USA, March 2000.

[ReTINA99] ReTINA, "Extended DPE Resource Control Framework Specifications", ReTINA Deliverable AC048/D1.01xtn, ACTS Project AC048, January 1999.

[Schmidt97] Schmidt, D., D. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker", Computer Communications Journal, Summer 1997.

[Selic98] Selic B., Rumbaugh J., "Using UML for Modeling Complex Real-Time Systems", available at http://www.objectime.com

[Selic00] Selic B., "A Generic Framework for Modeling Resources with UML", IEEE Computer, Special Issue on Object-Oriented Real-time Distributed Computing, edited by Eltefaat Shokri and Philip Sheu, June 2000.

[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.

Towards Designing Distributed Systems with ConDIL

Felix Bübl

Computergestützte InformationsSysteme (CIS)
Technische Universität Berlin, Germany
fbuebl@cs.tu-berlin.de
http://cis.cs.tu-berlin.de/~fbuebl

Abstract. Designing and maintaining a distributed system requires consideration of dependencies and invariants in the system's model. This paper suggests expressing distribution decisions in the system model based on the system's context. Hence, UML is enriched by two new specification techniques for planning distribution: On the one hand, *'Context properties'* describe dependencies on the design level between otherwise possibly unrelated model elements, which share the same context. On the other hand, *'context-based distribution instructions'* specify distribution decisions based on context properties. The distribution language 'ConDIL' combines both techniques. It consists of four layers introduced informally via examples taken from a case study.

Keywords: Designing Distributed Systems, System Evolution, Design Rationale, Resource Management

1 Introduction

1.1 Distribution Needs 'Design for Change'

The context for which a software system was designed continuously changes throughout its lifetime. *Continuous software engineering* is a paradigm discussed in [11,17] and in KONTENG[1] to keep track of ongoing changes and to adapt legacy systems to altered requirements. The system's design level must support these changes - it must be prepared for changes. It must be possible to safely transform the system model in *consistent modification steps* from one state of evolution to the next without unwanted violation of existing dependencies and invariants.

This paper suggests taking basic distribution decisions into account right from start of the software development process and expressing these decisions in a distribution language on the design level.

[1] This work was supported by the German Federal Ministry of Education and Research as part of the research project KONTENG (Kontinuierliches Engineering für evolutionäre IuK-Infrastrukturen) under grant 01 IS 901 C

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 61–80, 2001.

1.2 Distribution Needs 'Design for Implementation'

An increasing number of services and data have to be spread across several distributed sites due to modern business requirements. Arranging the distribution of services and data is a crucial task that can ultimately affect the performance, integrity and reliability of the entire system. In most cases, this task is still carried out only in the implementation phase. According to [7], this approach is inefficient and results in expensive re-engineering of the model after discovering the technological limitations. This frequently leads to adaptations directly on the implementation level alone. Thus, verification of system properties is made more complicated or even impossible due to outdated models. Section 2 discusses typical design errors, which can be avoided, if essential distribution aspects are already considered in analysis and design.

1.3 Case Study: Government Assisted Housing

The results presented here are based on a case study in cooperation of the Senate of Berlin, debis and the Technical University of Berlin. In this case study, distribution across offices in 23 districts of a complex software system for housing allowance administration was designed.

2 Problem: Distribution Needs to be Considered from the Start

The main problem addressed here is how to reach overall performance and availability while distributing a complex software system in a way that eases later modification. Typical design problem are discussed here. First section 2.1 examines the need to consider distribution during the analysis phase. Afterwards distribution in the design phase is observed in section 2.2. Then evolution in distributed systems is addressed in section 2.3, before the goals of this paper are summarized in section 2.4.

2.1 Preparing Distribution in the Analysis Phase

Already throughout the analysis phase key aspects, which determine the distribution of a software system, should be addressed. The process of developing a software system starts with a *requirements engineering* stage where functional and non-functional requirements of the system are identified. Functional requirements determine a system's functionality. Non-functional requirements describe the quality that is expected from the system. Two typically aspects ignored so far will be now analysed. On the one hand, existing infrastructures and resources – hardware, software or organisational – might determine distribution and, therefore, must be reflected in analysis phase. On the other hand, a rough prediction of the most frequently used services and data and their usage should be given:

1. Which workflows are most commonly used and how often is each of them executed?
2. Which objects are heavily used and what quantity of each of them will have to be managed by the system?
3. Along which basic contexts shall the system be cut? What are the key factors that determine the allocation of services or data onto a node?

Application area experts can often answer these questions easily. They can estimate these numbers or name the most frequently occurring workflows already in the analysis phase. In most cases, this important information is not investigated sufficiently. Without considering a given infrastructure and the crucial system load it cannot be determined whether a software system is well balanced, scalable or reliable later on. The abovementioned analysis results need to be reflected in the subsequent design phase.

2.2 Planning Distribution in the Design Phase

Up to now, a distribution decision has typically been taken in the implementation phase. But the rationale for distribution decisions may be ignored later on if they are already not expressed in the model. For instance, by writing down the distribution requirements '*all data needed by the field service must be stored on the field worker's laptops*' at the design level, the developers will not be allowed to remove certain data from the laptops in future modifications. One distribution requirement may contradict others. The field service example may contradict a requirement stating that '*personal data must not be stored on laptops*'. In order to reveal conflicting distribution requirements and to detect problems early they should be written down in the beginning of the development process, not during implementation. Fixing them during implementation is much more expensive.

While defining the *structure* of a system in the design phase the assignment of an attribute to a class[2] can complicate or even inhibit distribution.

Each class is used in several *contexts*. For instance, a system's context is a company with both headquarters and field service. The notion of 'context' is explained in section 3.1. Up to now the context of a class has not been considered in the design phase. Thus, it cannot be determined during design whether one class contains attributes from different contexts or not. If, e.g., a class belonging to the context 'headquarters' also contains attributes of the context 'field service' the instances of this class have to be available in both contexts and therefore be replicated or remotely invoked. Unnecessary network load can be avoided during design if this class is split into classes that only contain attributes of one context.

Multimedia data or other large binary data may obstruct replication. For instance, the class '*Movie*' has a large attribute '*Moviefile*' that stores a large MPEG movie. And it contains additional normal attributes like '*Movietitle*', '*Producer*' and '*Last time when this movie was accessed*'. If this class is replicated, high network traffic occurs each time when the '*Last time when this movie*

[2] In the case of designing a non object-oriented system please substitute 'class' with 'entity-type' throughout this paper

was accessed' attribute changes. This attribute is modified each time the movie is watched, and each time the whole changed instance of this class including the many megabytes large '*Moviefile*' must be copied over the network. This error could be recognised already on the design level, if replication would be considered during the design phase.

The *dynamic* behaviour of a system must also be reflected in distribution decisions. But dynamic models can become highly complex. This paper suggests taking only the names of the most often used workflows into account for planning distribution. These names can be investigated in the analysis phase and play an important role here. Up to now a system model doesn't show, which classes are needed by which workflow or database transaction. Designers should attempt to allocate all the classes needed by one workflow onto the same node to avoid network or system overload. There is no technique yet available to assist distribution decisions according to essential workflows or database transactions.

One distribution requirement can apply to several model elements, which can be part of different views. It should take dependencies between model elements into account. Current specification techniques for dependencies do not allow for model elements which are not directly connected or related, or are not even part of the same specification or view.

Considering distribution on the design level allows for the early prediction of problems and facilitates modifying the system model as discussed in the next section.

2.3 Modifying a Distributed System

Today *many* people develop *many* parts in *many* languages of *many* views of **one** distributed software system. This leads to system models that contain huge numbers of different model elements, like classes in class diagrams or transitions in petri nets. It gets increasingly difficult to understand such complex 'wallpaper'. Besides other problems, two important tasks become hard to fulfil:

Distribution Decisions: In deciding upon the allocation or replication of one model element, other distribution decisions and dependencies between related model elements must be considered.

System Evolution: The modification of one model element should take existing dependencies and distribution instructions into account in order not to violate them, and it should be propagated to all other concerned model elements. As analysed in [14], existing design techniques typically concentrate only on forward systems management.

New techniques for denoting dependencies and distribution instructions are needed to reduce the high costs of rearranging a distributed system model.

2.4 Goal: To Plan Distribution on the Design Level

This paper suggests enriching a UML model by two new specification techniques that facilitate decisions about a system's distribution and support consistent modification steps:

'Context properties' describe dependencies on the design level between otherwise possibly unrelated model elements.

'Context-based distribution instructions' are invariants on the design level. They specify distribution requirements for sets of model elements that share a context. Thus, they facilitate the preservation of distribution constraints at runtime or during model modifications.

The distribution language 'ConDIL' combines both techniques to express essential distribution requirements. It consists of four layers introduced informally via examples taken from a case study. Before introducing ConDIL in section 4, the following section presents the new techniques in general.

3 Context-Based Instructions

The essence of 'ConDIL' is writing down key distribution constraints. Before explaining why the Object Constraint Language OCL is not used, the two other techniques applied instead are sketched here. Context properties are initially discussed in section 3.1. They establish a basis for the context-based instructions introduced in section 3.2.

3.1 Describing Indirect Dependencies via Context Properties

System models contain elements, like classes in class diagrams or transitions in petri net diagrams. Model elements can depend on one another. Modification of a system model must not violate dependencies between model elements, and distribution decisions should take these dependencies into account. In order to consider dependencies, they must be specified in the model. Model elements can relate to each other even if an association does not directly link them. A new technique for describing such correspondences was introduced in [3]: Context properties allow the specification of dependencies between otherwise unrelated model elements that share the same context – even across different views or specifications.

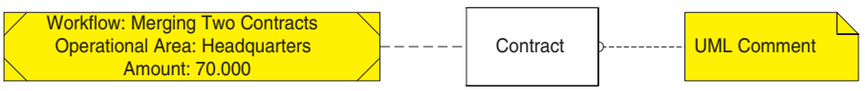


Fig. 1. Enhancing UML via Context Properties

A graphical representation and informal definition is indicated in figure 1. The context property symbol resembles the UML symbol for comments, because both describe the model element they are attached to. The context property

symbol is assigned to one model element and contains the names and values of all the context properties specified for this model element: the context property named *'Workflow'* in figure 1 has the value *'merging two contracts'* for the class *'Contract'*. Thus, it connects a static class to dynamic workflow that may be specified elsewhere - e.g. in a petri net.

A context property has a name and a set of possible values. Both are investigated in the analysis phase. Due to the limited length of this paper, methodical guidance is not discussed. The examples in the following sections distinguish between functional and non-functional context properties and give hints on how to use them. The 'context properties' used in this paper on design level differ from the 'context properties' used in Microsoft's COM/.NET on implementation level. A future paper will deal with context properties on the implementation level, but this paper focuses on the design level only.

Context properties are a technique that allows handling the results of analysis phase in the subsequent design phase. This general purpose grouping mechanism leads to better-documented system models and improves their overall comprehensibility. Knowing background information about elements enhances understanding of the model. It allows to focus on distributing or modifying within *subject-specific, problem-oriented views*. For example, only those model elements belonging to the workflow *'merging two contracts'* are of interest in a distribution or modification decision. Knowing about a shared context is necessary in order not to violate existing correspondences while distributing or modifying a model.

The primary benefit of enriching model elements with context properties is revealed in the next section, where they are used to specify a new type of invariants.

3.2 Introducing Context-Based Instructions

In the previous section context properties were introduced as technique for describing dependencies between otherwise unrelated model elements. This section suggests expressing decisions based on these correspondences as '*instructions*' which are invariants on the design level and specify requirements. There are many different kinds of *implementation requirements* during different phases of developing a software system. This paper focuses on distribution decisions. Before describing a language for distribution instructions in section 4, two examples are discussed here to promote their general benefits.

In first example, the system's context is a company with both headquarters and field service. Then, according to this context, there is a distribution instruction saying, that certain components have to run on the travelling salesman's laptop without being in connecting to the server. This distribution instruction is valid for all model elements, whose context property 'operational area' has the value 'field service' – it is a *context-based instruction*. This example is one answer to the question 'along which basic contexts shall the system be cut?' raised in section 2.1 by cutting the system into operational areas. In the case of a context change, like an alteration in the company's privacy or security policy the

system must be adapted to the new requirements without violating the already existing distribution instruction. In this case, either no component necessary for working offline should be removed from the field worker's laptop for security or privacy reasons, or the distribution instruction must be adapted. Therefore, modifying the model must take already existing distribution instructions into account in order not to violate them.

Another example demonstrates how ensure distribution requirements via context-based distribution instructions. In order to develop a smoothly operating distributed system every workflow[3] should be able to run locally on a single node without generating network traffic. A frequently executed workflow should be described in a context property 'workflow' with the value 'merging two contracts' to all model elements needed by this workflow, and than state a distribution instruction allocating everything needed by this workflow onto the same node.

Classical load balance calculations need non-functional context properties instead of functional ones like 'workflow'. Some of the examples in section 4 illustrate how to use context-based instructions in predicting and establishing an evenly load balance in distributed systems.

Fundamental choices about how to distribute a model should reflect the system's context and should be preserved and considered in a modification step. Therefore they must be expressed on the design level. Up to now, there has not been a method or technique for describing reasons for distribution decisions. A language for specifying distribution requirements is proposed in the next section.

4 The Distribution Language 'ConDIL'

The **con**text-based **D**istribution **I**nstructions **L**anguage 'ConDIL' consists of four layers or views enhancing UML:

The enhanced UML class diagram identifies the classes to be distributed.
The net topology layer indicates the hardware resources available in the distributed system.
The distribution instructions layer states distribution decisions for sets of classes of the enhanced class diagram.
The enhanced, generated UML deployment diagram displays the results of the other layers' specifications.

The layers are introduced informally via simplified examples taken from the case study. Before describing each layer the following section addresses limitations of ConDIL.

[3] In the case of designing a distributed database system please substitute 'workflow' with 'transaction' throughout this paper

4.1 Designing Distributed Components or Databases with ConDIL?

The short descriptions of the four layers given above speak about distributing 'elements' of the enhanced class diagram. It depends on the type of distributed system to choose a more concrete term: In the case of a *component-based system* 'components' 'classes' or 'objects' are distributed, while in designing a *database*, it is intended to distribute 'entity-types' or 'entities'. The general version of ConDIL proposed here neither fits all the needs of component-based systems nor of distributed databases.

Neither case is discussed in this paper due to space limitations. Forthcoming papers will study each case separately. This paper restricts itself to allocating 'classes' on 'nodes' without going into detail about whether this means allocating a class in a component's interface or in a local schema of a distributed database system. Among the other unexplored topics of interest also are, e.g., heterogeneity, the choice of distribution technologies or the allocation of model element other than classes.

The general version of the distribution language ConDIL allows the specification of any kind of distributed system because it only expresses the most basic distribution requirements necessary.

4.2 ConDIL's Enhanced UML Class Diagram

The previous section explained why ConDIL has been restricted to allocating classes to nodes. Classes are specified in an enhanced UML class diagram. The example illustrated in figure 2 describes a system where poor people can apply for a state grant that helps them to pay their rent. The housing allowance is an n:m association between the applicant and the apartment and lasts only for a certain time. The following concepts have been added to the UML Class Diagram:

Association Qualifiers & Classes are standard UML techniques which are used here to specify *foreign keys* for each association. Up to now foreign keys were not specified in the conceptual design. This well-known relational database concept is necessary to enable cutting an association when the classes at its ends are allocated onto different nodes. The concept of foreign key attributes for 1:n or n:m associations is explained in standard database literature. As illustrated between 'Apartment' and 'Owner', the arrow at 1:n-associations is required to give their qualifiers a clear semantic. An n:m association like between the classes 'Applicant' and 'Apartment' needs an *association class* – 'Housing Allowance' – to hold two foreign key attributes: in the qualifier of the association class for both foreign keys the name of the class referred to is followed by the name of the foreign key attribute.

Context Properties were introduced in section 3.1. Each context property has a name, e.g., 'Operational Area' and values, e.g., 'Headquarters'. A legend shows all the valid values for each context property. The default value of each context property is underlined. In the case the context property has only the

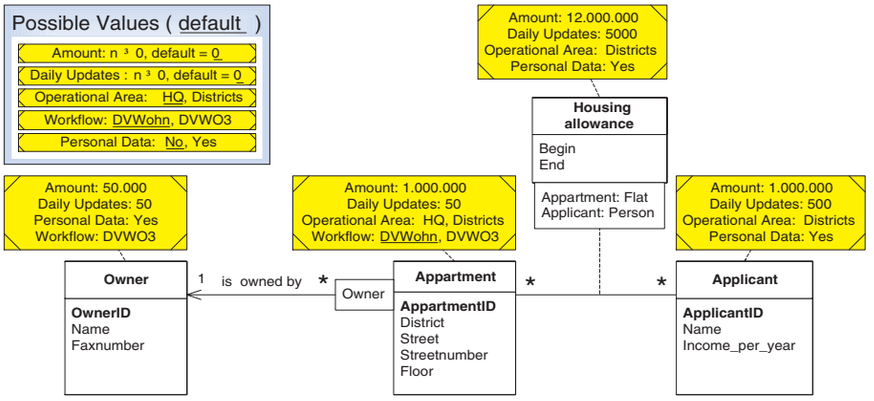


Fig. 2. Extract of an Enhanced UML Class Diagram Showing the Case Study

default value for a class, it can be left out in the context property symbol of this class. In this example, the context property 'Workflow' has the default value 'DVWohn'. As all classes in figure 2 belong to this workflow, none of them needs to specify it in its context property symbol. In the next section, figure 3 shows an alternative approach, where – for improved comprehensibility – the designer spelled out the values of all context properties for each class even if it is the default value.

The context properties suggested here may be useful for planning a distributed system in general. When designing a particular system some of these proposed context properties may be ignored and some unmentioned ones may be added by the developer, as needed. Figure 2 exemplifies both functional and non-functional properties:

'Workflow' describes the functionality – it is a context property of *functional type*. It reflects the most frequent workflows or use cases and enables the designer to write down distribution requirements for these workflows. For instance, in a distributed system all of the model elements needed by a certain workflow should be allocated to the same computer in order to be able to execute this workflow without connecting to the network. This requirement can be verified by marking all of the concerned model elements accordingly. Thus, *static aspects of system behaviour* can be expressed.

'Operational Area' is another example for managing distribution via a functional context property. It enables the writing down of the distribution decisions for certain departments or domains. Functional context properties provide an organisational perspective and thereby facilitate software design as indicated by [12].

'Personal Data' signals when a class contains data that must not be distributed due to its intimate content. It is of functional type and exemplifies how to model roles or authorization via context properties.

'Amount' is of *non-functional (qualitative) type*. Its value holds the estimated number of instances of each class.

'Daily Updates' allows to distinguish between frequently changing and rather static data. This non-functional information is needed to estimate the network load later on.

Hiding avoidable dynamics by only considering *static aspects of behaviour* is the primary benefit of using functional context properties. They allow consideration of methods, services, sequence or operations and others details to be left out in distribution decisions. Otherwise the model complexity would increase rapidly. The goal of this paper is to keep distribution decisions as straightforward as possible.

Section 4.5 give a demonstration how to use qualitative (non-functional) context properties for calculating metrics in order to facilitate system assessment.

After having only slightly extended the standard UML diagram up to now, two totally new visual languages are proposed in the next two sections.

4.3 ConDIL's Net Topology Layer

Taking distribution decisions demands awareness of hardware resources. The net topology layer depicts the hardware resources. It identifies nodes and connections as demonstrated in figure 3:

A node is denoted by a symbol resembling a computer. By working with symbolic names like 'RAID-Server' the same net topology diagram can be deployed for different customers and cases, and it does not have to be changed if the hardware is replaced. Symbolic names can describe hardware requirements or services like 'database', 'sensor', 'router' or 'printer'. A constraint stated for *one* symbolic node, e.g. 'Laptop', applies to *all* nodes of this kind. By using symbolic names, the network topology diagram must not be modified if the number of laptops changes.

A Connection links nodes and is drawn with the standard UML deployment diagram symbol for connections.

Both nodes and connections are more closely described via context properties. In the net topology diagram mostly 'non-functional' context properties are used. They are needed later on for establishing reasonable load balance. For instance, the non-functional context property 'Speed' can be helpful in taking a distribution decision. Improving a system's load balance via ConDIL is discussed in section 4.5.

The net topology diagram will feature additional symbols like 'table space' for databases or 'container'for Enterprise Java Beans. These specific symbols will be published in articles concentrating on certain platforms.

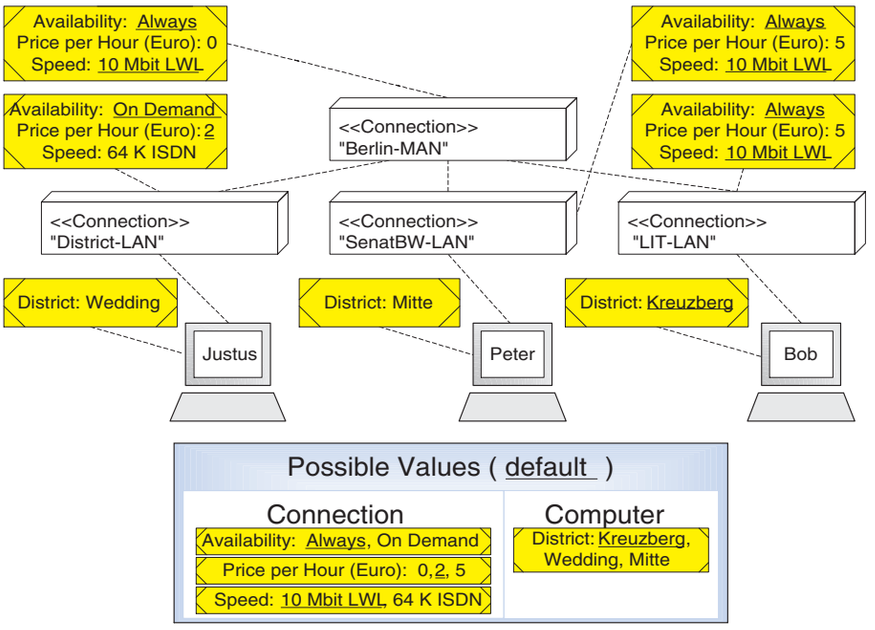


Fig. 3. The ConDIL Net Topology Layer

It would be possible to include distribution instructions in the net topology diagram, but there are already several symbols associated with one node, and there will be even more in future enhancements of this layer. Distribution instructions are better depicted in their own layer, which is introduced now.

4.4 ConDIL's Distribution Instructions Layer

In this central ConDIL layer two different types of distribution instructions control either *allocation* or *replication* decisions. As shown in the next figures, a distribution instruction is drawn as follows: an arrow points from the *symbol specifying the elements involved* that shall be distributed to the *symbol specifying the target* where the elements involved shall be allocated. The allocation instructions are introduced first.

Context-Based Allocation Instructions specify allocation constraints for more than one target. The basic assumption for discriminating between allocation and replication instructions is that there is *exactly one master copy* of each instance of a class. In the case of assigning one class to several nodes, only one node holds an original instance. All other copies of this instance on other nodes are replicated from this master copy. This notion enables the underlying middleware or database to ensure consistency.

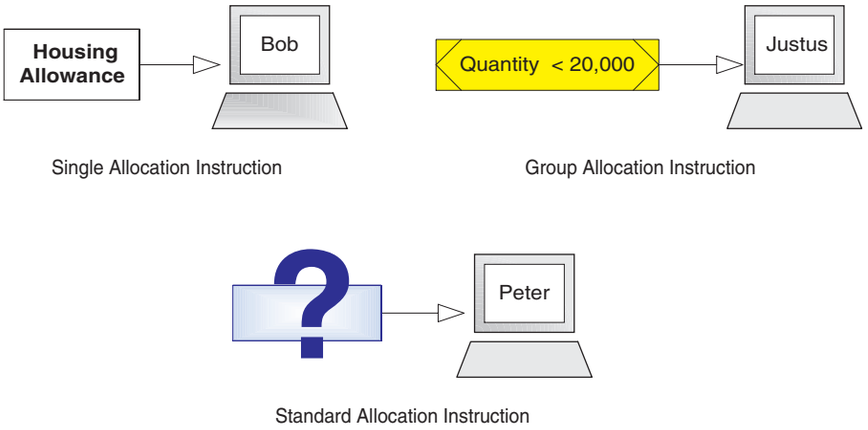


Fig. 4. Allocation Instructions of the ConDIL Distribution Instructions Layer

You shouldn't confuse classes with instances in this section. The general version of ConDIL introduced in this paper describes distribution on the class level only. It enforces all original instances of one class to be allocated on the same node. The upcoming version of *ConDIL for databases* will stick to the class level as well. In contrast, the future version of *ConDIL for components* will deal with allocation on instance level and will allow original instances of the same class on several computers.

In order to assure the allocation of a class on only one node different priorities are given to each kind of allocation instructions. They are exemplified in figure 4 from left to right:

Single allocation instructions are demonstrated on the left side of figure 4. They have the highest priority – no opposing set allocation instruction applies for a class, if a single allocation instruction for it exists. It is not allowed to spell out contradicting single allocation instructions for the same class.

Set allocation instructions are illustrated in the middle part of figure 4. They apply for every class that fits to the *context condition* and is not allocated via a single allocation instruction. In the example given, all classes having the context property 'Quantity' with values of less than 20,000 are allocated onto the node 'Justus'. Being able to give set instructions is the main benefit of ConDIL. Grouping classes, which share the same context, allows taking distribution decisions based on essential requirements. If a class is assigned to different nodes via contradicting set allocation instruction, one assignment becomes the master copy, and all others become synchronous replicates of this master copy.

One default allocation instruction as shown on the right in figure 4 must be specified in a system model. It has the lowest priority and applies only to classes where no other allocation instructions apply.

Context-Based Replication Instructions are exemplified in figure 5. The symbols are explained now from left to right:

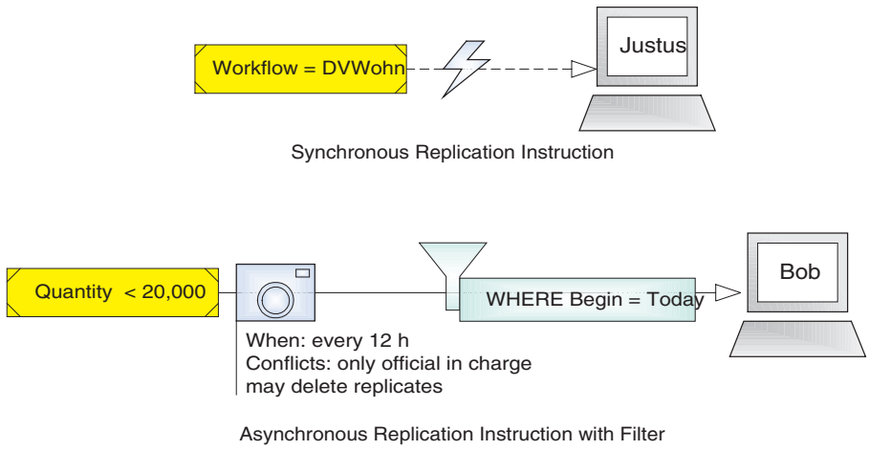


Fig. 5. Replication Instructions of the ConDIL Distribution Instructions Layer

A camera represents asynchronous replication. It needs additional attributes that are placed directly under the camera symbol. The attribute 'When' must be indicated in any case because asynchronous replication cannot be implemented without knowing when to replicate. The attribute 'Conflicts' must be stated only in the case of writeable or deleteable replication and describes what to do in the case of update or deletion conflicts. This information may be omitted if default rules are given for all asynchronous replication instructions when to replicate or what to do in the case of conflicts.

The dotted arrow of a replication instruction indicates *write only* replication.

A lightning bolt represents synchronous replication as demonstrated on the top of figure 5.

Set instructions , again, are the most powerful application of ConDIL. Available database products possess fast group replication mechanisms as Oracle's 'refresh groups' for the implementation of set replication instructions. Contrary to allocation instructions, there are no different priorities for replication instructions, and there is no default replication instruction.

The Filter symbol is needed to express views, where only some of the instances of the source class(es) shall be replicated. Normally filters only make sense for single replication instructions, because they use an SQL WHERE clause that names attribute of the selected class(es). In a filtered set instruction, like in figure 5, all classes selected by the context condition 'Quantity less then

20,000' must have the attribute 'Begin'. This could happen in historical data warehouses, but in general not all classes selected by the context condition have the attribute(s) needed by the filter condition.

Overall, this section introduces the key concepts in an informal way. Both the details of the context-based distribution language ConDIL and adequate methodical guidance are topics of future research. The next section suggests automatically generating an enhanced UML deployment diagram showing the results of the requirement specifications.

4.5 ConDIL's Enhanced, Generated UML Deployment Diagram

Without illustrating the results of a specification stated in the other three layers, a designer cannot verify if the goals of designing a distributed system have been reached: reliability, scalability and load balance need to be confirmed in an overview diagram showing all the details. ConDIL uses an enhanced UML deployment diagram for validation of the distribution goals and for *architectural reasoning* as discussed in [15]. The enhanced deployment diagram can automatically be generated based on the specifications in the other three layers.

Figure 6 shows one possible result of ConDIL distribution design. In this example, not all of the goals have been accomplished yet. The classes have been successfully allocated and replicated in a way that the workflows or database transaction on 'Justus' and 'Bob' can be executed locally. Thus, both computers can continue working even if the network fails. But in ensuring reliability, the other aims (scalability and load balance) are not achieved. In the example, both the connection 'District-LAN' and the computer 'Bob' show overload.

The numbers representing the system load in figure 6 are derived from the non-functional context properties given in the other layers. They can either be automatically calculated via given formulas, or they can be intellectually estimated via common sense. Some standard context properties have been suggested, but standard formulas for load balance calculation are a topic of future research. At the EDO symposium three precision levels for optimising load balance were discussed:

1. Estimation based on common sense and experience is the quickest and most vague way to figure out load balance.
2. Simulating an UML model as proposed by [9] turns out better load numbers, but needs an arbitrary detailed modelling of the system behaviour.
3. The best but most expensive load prediction results from prototypical benchmarking.

The enhanced deployment diagram assists in detecting problems before implementation phase. Figure 6 demonstrates the need to change the other ConDIL layers until a satisfactory trade off between the contradicting goals of distribution design is reached.

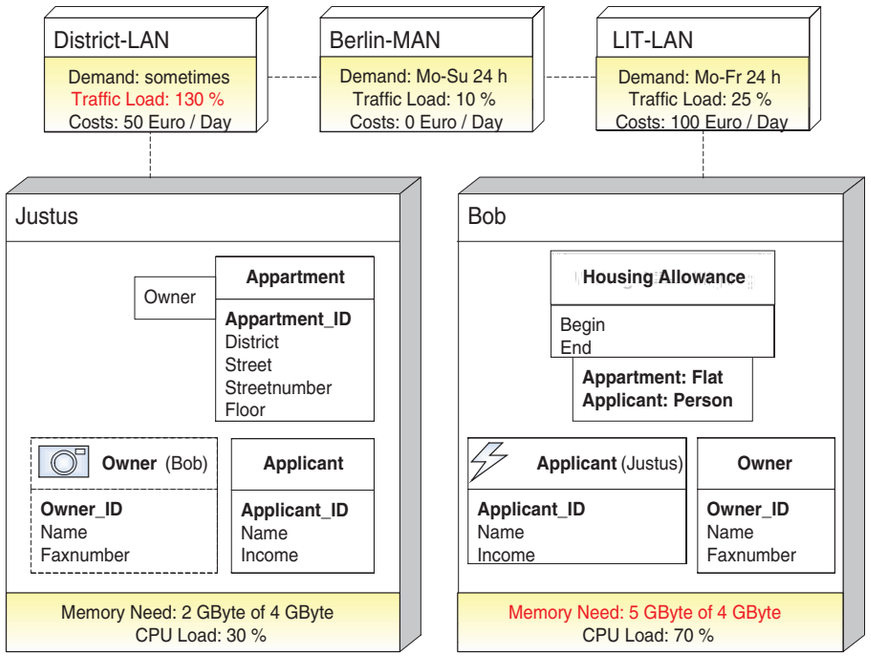


Fig. 6. Displaying a System Overview with ConDIL's Generated Enhanced UML Deployment Diagram

5 Related Research

5.1 Planning Distribution

After distributed applications became popular and sophisticated in the 80is, developers needed techniques to support their development. According to [1] over 100 new *programming languages* specifically for *implementing* distributed applications were invented. But hardly anyone took distribution into consideration already on the *design* level, whereas ConDIL does. Research concentrated on dealing with parallelism, communication, and partial failures on implementation level. On the contrary, ConDIL describes distribution on the design level.

For instance, the *Orthogonal Distribution Language (ODL)* proposed in [5] supplements the programming language C++ for distributed processing and, thus, does not support distribution decisions during the design phase. Roughly the same applies to *aspect oriented languages*, which, in principle, resemble the idea of context properties. D^2AL as one of these aspect oriented languages, however, differs in that it is based on the system model, not on its implementation. Though the closest to ConDIL, the differences will now be discussed.

According to [2], D^2AL enables the programmer to specify the distribution configuration based on UML. Whereas ConDIL states a distribution instruction for a group of classes which share the same context, D^2AL groups collaborating objects that are directly linked via associations. Objects that heavily interact must be located together. D^2AL describes in textual language in which manner those objects interact which are connected via associations. This does not work for objects that are not directly linked like 'all objects needed by the field service'. In contrast, ConDIL is a visual approach based on a shared context instead of object collaboration.

For companies, staying competitive means meeting continuously changing business requirements. As presented in [18], especially business process dominated systems demand flexible models. This paper proposes enriching system models via distribution decisions, which can be based on important business processes. In this paper the context property 'Workflow' was used to increase the adaptability of a model by enriching it via invariants according to its business processes.

One way in which we cope with large and complex systems is to abstract away some of the detail, considering them at an architectural level as composition of interacting components. To this end, the variously termed *Coordination, Configuration and Architectural Description Languages* facilitate description, comprehension and reasoning at that level, providing a clean separations of concerns and facilitating reuse. According to [8], in the search to provide sufficient detail for reasoning, analysis or construction, many approaches are in danger of obscuring the essential structural aspect of the architecture, thereby losing the benefit of abstraction. ConDIL is a concise and simple language explicitly designed for describing architectural distribution structures.

Darwin (or 'δarwin') is a 'configuration language' for distributed systems described in [13] that makes architecture explicit by specifying the connections between software agents. Instead of describing explicitly connections between distributed objects, ConDIL expresses vague dependencies via contexts properties and writes down instructions for model elements that share a context.

In [12] *policy driven management for distributed systems* integrates organisational engineering and distributed databases. Replication models are described within the organisation model. This way, consistency of replication policies can be inferred from the organisational model. In ConDIL, distribution instructions write down replication requirements. ConDIL can assist policy driven management for distributed systems.

There used to be a lot of interest in machine-processed records of *design rationale*. According to the many authors of [10] the idea was that designers would not only record the results of their design thinking using computer-based tools, but also the process that they followed while designing. Thus, the software designer would also record a justification for why it was as it was and what other alternatives were considered and rejected. Context-based instructions are one way to record design decisions. The problem is that designers simply don't like to do this. The challenge is to make the effort of recording rationale worthwhile

and not too onerous for the designer. In return for writing down design decisions via context-based instructions they harvest several benefits sketched below.

5.2 ConDIL and UML

In the long term ConDIL proposes becoming part of UML. This section inspects each new ConDIL concept, whether it can be realised via standard UML extension mechanisms or not:

Context properties can either be based on UML tagged values or on UML comments. Using tagged values is the concept closest representing context properties, because both are key-value pairs. Typically one class is characterized by several context properties. Each tagged value floats around its class separated from the other tagged values belonging to the class. This can be confusing when numerous tagged values belong to the same model element. Therefore, a specialisation of the UML comment symbol is proposed in this paper to place all of the context properties and their values of one class *into one single symbol*.

No stereotypes or packages: In order to write down distribution instructions for *sets of classes* it is necessary to *group* classes that share a context. ConDIL describes the context of classes via the new technique 'context properties' rather than grouping classes of the same context via existing UML mechanisms. Usually several context properties exists with each of them having multiple values. In [3] mechanisms, like *UML stereotypes* or *UML packages*, are rejected, because they cannot handle several overlapping contexts. Usually quite a few context properties exist where each has multiple values. For instance, if your system has n different values for 'Workflow', you would need $2^n - 1$ different stereotypes to classify your classes. Considering an additional context property, e.g. 'Operational Area' would result in an even more confusing number of stereotypes.

The enhanced class layer of ConDIL can be implemented via standard UML extension mechanisms.

The net topology layer is not part of the UML at present and calls for enhancements of the UML metamodel.

The context-based distribution instruction layer cannot be derived from contemporary UML as well. Context-based instructions are constraints associated with context properties (tagged values). The *Object Constraint Language OCL* summarized in [16] is the UML standard for specifying constraints like invariants, pre- or post-conditions and other constraints in object-oriented models. OCL was not chosen for stating context-based distribution instructions for several reasons. The model should serve as a document understood by designers, programmers and customers and, therefore, should use such simplified specification techniques. ConDIL is an easily comprehendible, straightforward visual language separated into layers and reduced to the bare minimum of what's needed for designing distribution. A major distinction is that *one* OCL constraints refers to *one* model element,

while *one* context-based instructions refers to *many* model elements. Even the OCL 1.4 draft does not permit one constraint for several model elements.

The enhanced deployment diagram can be realized via standard UML extension mechanisms. UML deployment diagrams are not widely considered to be attractive – e.g. [6] describes them as 'informal comic'. ConDIL improves their expressiveness and demonstrates their reasonable usage in developing distributed systems.

6 Conclusion

6.1 Limitations of ConDIL

Up to now ConDIL is only used on design level. An upcoming paper will examine the influence of ConDIL on the handling of distributed objects at runtime. ConDIL is restricted to *static aspects of system behaviour*. Reducing behaviour to the names of frequent workflows ignores a lot of the information that may have an impact on distribution decisions. For instance, the ConDIL layers do not show if and how workflows depend on each other. But such detailed modelling of dynamics would hardly improve the load balance calculation or the verification of the other design goals. The quality of the load balance prediction wouldn't fairly increase by accurate consideration of behaviour, because all non-functional numbers are estimated anyway and won't turn out a precise prediction. On the contrary, ignoring dynamical aspects provides some of the benefits that are summarized in the next section.

6.2 Benefits of ConDIL

ConDIL supports the design of distributed systems from the start of the development process. Key distribution requirements can now be expressed at the design level. It detects problems already during design. And it eases the identification of essential dependencies and invariants and thus improves the readability of the model, facilitates distribution decisions and helps to prevent their violation in later modification steps.

ConDIL's enhanced deployment diagram assists in establishing a trade off between load balance, reliability and scalability because it provides for the consideration of relevant aspects, such as names of frequent workflows or existing hardware resources. Detailed modelling of dynamical system behaviour is not needed for assessment of distribution decisions. Fewer iterations in the development process are necessary when distribution requirements were already taken into account during design.

When one model element changes, other related elements might also have to be adapted. Maintenance is a key issue in *continuous software engineering*. ConDIL helps ensure consistency in system evolution. A ConDIL instruction serves as an invariant and thus prevents violation of distribution requirements in a modification step. It helps detect when design or context modifications compromise intended functionality. The dependencies and invariants expressed via

ConDIL can prevent unanticipated side-effects during redesign, and they support collaborative distributed design. It is changing contexts which drive evolution. ConDIL's instructions are context-based and are therefore easily adapted to context modifications.

6.3 ConDIL Roadmap

Currently a CASE tool capable of ConDIL concepts is implemented at the Technical University of Berlin by extending the tool 'Rational Rose'. A first prototype is available for download at `http://cis.cs.tu-berlin.de/~fbuebl/ConDIL`.

The following subjects will be addressed in future research:

Enhancing ConDIL: Two versions of ConDIL will be published for designing either distributed component-based systems or distributed databases.

Method: Proposing a new technique is pointless without developing an adequate method for its appropriate use. For instance, both context properties and context-based instructions could be acquired during the reengineering process. Recovering basic dependencies and invariants can outline the distribution architecture of a legacy system. Furthermore, guidelines will be developed how to combine ConDIL and ConCOIL ([4]). – the **Con**text-based **C**omponent **O**utline **I**nstruction **L**anguage introduced in [4]. It describes the logical architecture of a component-based system.

Algorithm & tool for consistent modification: An algorithm for maintaining consistency in a modification step by considering existing context-based instructions will be developed. In a first step, this algorithm will serve to generate the enhanced deployment diagram. As 'proof of concept' a tool for evolution support will be implemented. This tool won't be based on the already existing Rational Rose extension, because ConDIL turned out to overstrain Rose's extensibility.

ConDIL at runtime: An upcoming paper will examine whether a scheduler can exploit ConDIL constraints for the sake of an optimal load balance.

References

1. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989. 75
2. U. Becker. D^2AL - a design-based distribution aspect language. Technical Report TR-I4-98-07 of the Friedrich-Alexander University Erlangen-Nürnberg, 1998. 76
3. F. Bübl. Context properties for the flexible grouping of model elements. In H.-J. Klein, editor, *12. GI Workshop 'Grundlagen von Datenbanken'*, Technical Report Nr. 2005, pages 16–20. Christian-Albrechts-Universität Kiel, June 2000. 65, 77
4. F. Bübl. Towards the early outlining of component-based systems with ConCOIL. In *ICSSEA 2000, Paris*, December 2000. 79
5. M. Fäustle. An orthogonal distribution language for uniform object-oriented languages. In A. Bode and H. Wedekind, editors, *Parallel Comp. Architectures: Theory, Hardware, Software and Appl.*, LNCS, Berlin, 1993. Springer. 75

6. M. Fowler and K. Scott. *UML Distilled*. Object Technologies. Addison-Wesley, Reading, second edition, 1999. 78
7. P. Koletzke and P. Dorsey. *Oracle Designer Handbook*. Osborne/McGraw-Hill for Oracle Press, Berkeley, second edition, 1999. 62
8. J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination 97, Berlin*, pages 18–31, 1997. 76
9. M. d. Miguel, T. Lambolais, S. Piekarec, S. Betgé-Brezetz, and J. Pequery. Automatic generation of simulation models for the evaluation of performance and reliability of architectures specified in UML. In V. Gruhn, W. Emmerich, and S. Tai, editors, *Engineering Distributed Objects (EDO 2000)*, LNCS, Berlin, 2000. Springer. 74
10. T. P. Moran and J. M. Carroll, editors. *Design Rationale : Concepts, Techniques, and Use (Computers, Cognition, and Work)*. Lawrence Erlbaum Associates, Inc., 1996. 76
11. H. Müller and H. Weber, editors. *Continuous Engineering of Industrial Scale Software Systems*, Dagstuhl Seminar #98092, Report No. 203, IBFI, Schloss Dagstuhl, March 2-6 1998. 61
12. A. R. Silva, H. Galhardas, P. Sousa, J. Silva, and P. Sousa. Designing distributed databases from an organisational perspective. In *4th European Conference on Information Systems*, Lisbon, Portugal, 1996. 69, 76
13. D. Spinellis. *δ*arwin reference manual. Technical report, Dept. of Computing, Imperial College, London, 1994. 76
14. S. Tai. *Constructing Distributed Component Architectures in Continuous Software Engineering*. Wissenschaft & Technik Verlag, Berlin, Germany, 1999. 64
15. F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. Architectural reflection realising software architectures via reflective activities. In V. Gruhn, W. Emmerich, and S. Tai, editors, *Engineering Distributed Objects (EDO 2000)*, LNCS, Berlin, 2000. Springer. 74
16. J. B. Warmer and A. G. Kleppe. *Object Constraint Language – Precise modeling with UML*. Addison-Wesley, Reading, 1999. 77
17. H. Weber. IT Infrastrukturen 2005 - Informations- und Kommunikations-Infrastrukturen als evolutionäre Systeme. White Paper, Fraunhofer ISST, 1999. 61
18. H. Weber, A. Sünbül, and J. Padberg. Evolutionary development of business process centered architectures using component technologies. In *Society for Design and Process Science, IEEE International Conference on Systems Integration IDPT*, 2000. 76

Architectural Reasoning

Wolfgang Emmerich

Dept. of Computer Science, University College London
Gower St., London WC1E 6BT, UK
w.emmerich@cs.ucl.ac.uk

When we summarized the session on architectural reasoning, we developed the mindmap shown in Figure 1. It identifies the key issues that are involved with reasoning about the architectures of distributed object systems and provides the outline for the summary of this introduction.

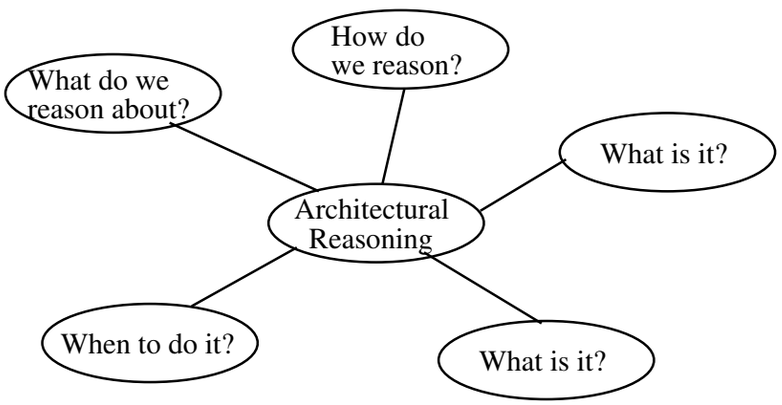


Fig. 1. Architectural Reasoning Mindmap

Architectural reasoning was defined by Bass, Clements and Kazman as the *reasoning about structural and behavioral properties of the solution space*. The workshop participants largely agreed with this definition.

The papers that were allocated to this session identify several approaches for how to reason about architectures. The paper "Automatic Generation of Simulation Models for the Evaluation of performance and reliability of Architectures specified in UML" by de Miguel et al. uses simulation as the basic method of reasoning. Nima Kaveh's paper proposes the use of model checking to identify potential deadlocks in distributed object systems. Other approaches that were discussed include scheduling analysis and slicing. The methods for architectural reasoning often assume some basic mechanisms of distributed object systems, such as reflection as outlined in Tisato et al's paper on "architectural reflection" and meta data as discussed in Orso et al. paper on "component meta data".

Reasoning about the distributed object-based architectures can be used to attain several goals. We discussed approaches to reason about the flexibility of

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 81–82, 2001.

a design. De Miguel's paper identifies architectural reasoning as a mechanism to identify the potential for reuse of a component. Most importantly, however, architectural reasoning was seen as a method for early prediction of potential behvioural problems, such as liveness or safety property violations as outlined by Nima Kaveh. Finally, architectural reasoning can be used to better understand the models of architectural abstraction that we produce at early stages of a distributed object system development effort.

Architectural reasoning can be performed at different stages in the development process of a distributed object system. It can be used at the time when components are designed to identify the properties that components contribute to architectures. Architectural reasoning may also be used at the time when distributed objects or components are assembled into an architecture in order to reason about the emergent properties of that architecture. Finally, we can also reason about architectures at the time when objects or components are executed.

Acknowledgements

The material for preparing this session summary was gathered at the end of the workshop with the help of Miguel de Miguel, David Rosenblum and Andrea Savigni.

Automatic Generation of Simulation Models for the Evaluation of Performance and Reliability of Architectures Specified in UML

Miguel de Miguel[1], Thomas Lambolais[1], Sophie Piekarec[3], Stéphane Betgé-Brezetz[3] and Jérôme Péquery[2]

[1]Thomson-LCR
Domaine de Corbeville
91404 Orsay, France
{Miguel.DeMiguel,Thomas.Lambolais}@lcr.thomson-csf.com
[2]SOFTEAM
8, rue Germain Soufflot
78184 Saint-Quentin Yvelines, France
jpequery@objecteering.com
[3]ALCATEL-CRC
Route de Nozay
91461 Marcoussis, France
{Stephane.Betge-Brezetz,Sophie.Piekarec}@ms.alcatel.fr

Abstract. Non-functional requirements are especially critical in real-time and distributed systems. UML is progressively becoming a standard of object-oriented analysis and design of systems, it pays attention to software architectures specification, but it does not take into account their evaluation, and the specification of resource restrictions and non-functional requirements. In this paper we introduce an approach for the evaluation of non-functional requirements in UML models, using simulation techniques. Simulation models are generated automatically, and their execution provides results to evaluate the UML architectures. The simulation statistics generated allow the evaluation of some non-functional requirements like resources usage, objects and classes activity and availability, restoration times of errors and throughputs. We associate these results to objects, classes, states, operations, actors, system resources and other UML elements. UML semiformal semantics have associated problems that we reduce with UML extension techniques.

1 Introduction

The Unified Modeling Language (UML) is progressively becoming a standard for object-oriented analysis and design modeling of software systems [1]. The UML is being used to specify a variety of enterprise applications including telecom and real-time systems. At the same time, the UML standard defined by the OMG keeps evolving in order to fix diverse remaining issues and to include new features based on in-

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 83-101, 2001.

dustrial and academic contributions. The UML metamodel specification and its extension facilities [9] allow the adaptation of the modeling elements for specific purposes. Unfortunately, the semantics of the metamodel is not formally defined, and because of that the user has no means to define precise and unambiguous extensions and consistent models.

We can use UML for different modeling purposes: analysis specification, architectural modeling, capturing e-business and technology requirements, etc. The objective of our work is the evaluation of UML architectures and detailed designs of distributed and real-time systems. UML architectures include the set of significant decisions about the organization of software system, the selection of structural elements and their behavior specification, the identification of their usage, and the specification of functionality, performance and other non-functional characteristics [1]. In the project OURAL we have studied the evaluation of UML architectures based on analysis [3] and simulation techniques. In this document we are going to introduce the evaluation based on simulation models.

The execution of simulation models serves for different purposes: the evaluation of system outputs results, the animation of model execution, the evaluation of properties, etc. We use OPNET [10] as a simulation kernel for the evaluation of performance, resource capacities, and reliability in software architectures. OPNET is a simulator based on discrete events, oriented to network simulations, which includes facilities for the construction of general-purpose simulations. The simulation results are useful for the evaluation of software architectures, comparison of alternative solutions, and identification of architectural risks. This is especially complex in distributed software architectures, where we need to identify and select physical elements such as hardware nodes, networks, links and hardware topologies in general.

We simulate distributed architectural models and provide statistic results of nodes, networks and UML elements. The simulation models update the statistics depending on the model behavior and system architecture. Examples of these types of statistics are: i) occupation of system resources (CPU, memory, network) and UML elements (busy states of classes, instances, states, etc.), ii) temporal parameters of some UML elements and system resources (response time of classes and instances, response time of operations, execution time of operations, frequency of memory allocation, response time of actors' invocations, deliberation time of messages), iii) modification or access to class and instance attributes, iv) frequency of error occurrences in UML elements and networks, v) temporal distribution of network error restoration, and object instances restoration, vi) temporal distribution of availability of objects, vii) and length of some waiting queues (CPUs, networks and state machines). The values of statistics include the discrete occurrence of their events, percentiles, deviation, media and number of values. These values can be evaluated using different types of filters. All these values are the basic criteria for the evaluation of non-functional requirements in software architectures, and allow for the evaluation of Quality of Service (QoS) constraints such as resource usage, response time bounds, delay jitters, availability of services, and failures distributions.

Simulation techniques and automatic code generation have been employed largely for the evaluation of UML models. Raphsody [13] and Rose Real-Time [14] are

commercial tools that provide facilities for the execution and animation of UML models. These tools support automatic code generation for executing models, but do not use simulation techniques. This does not facilitate the evaluation of UML deployment elements like nodes and networks, and they do not take into account distributed aspects of architectures in general. They provide services for the animation of models and for the evaluation of functional results, but they do not evaluate non-functional parameters. The Permabase project [11] uses simulation techniques based on SES Workbench [15], paying special attention to distributed executions. However, the project does not take into account modeling elements like state machines, and the simulation techniques do not provides the detailed simulations of network elements.

The rest of this paper introduces the different phases of evaluating architectural models, and discusses problems related to UML specifications (Section 2). Section 3 describes the automatic generation of simulation models. Section 4 describes the types of evaluation results, and section 5 presents some conclusions.

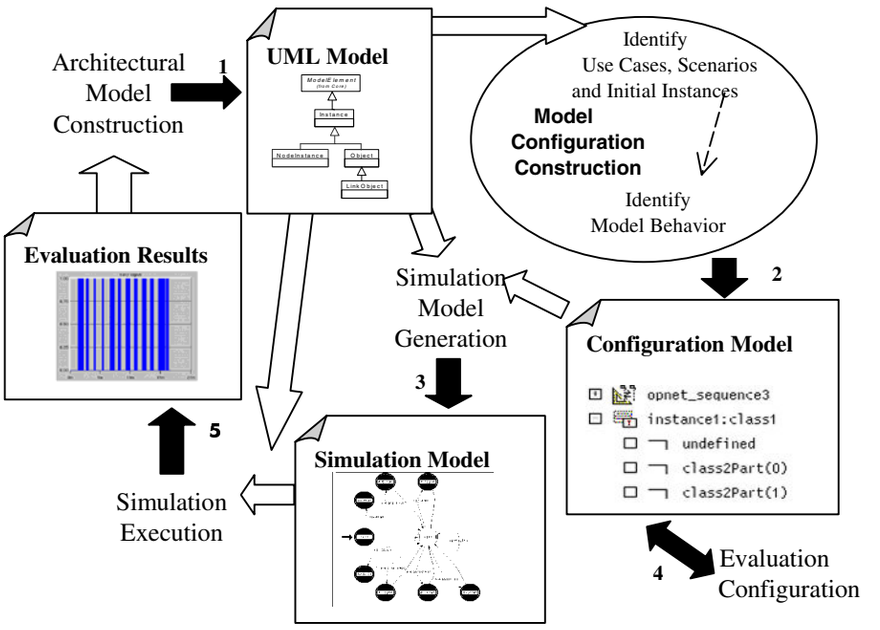


Fig. 1. Evaluation Phases

2 Evaluating UML Architectures with Simulation Models

The techniques that we are going to present provide results useful in the selection and evaluation of architectural software solutions. Their results allow the identification of problems not taken into account during the model construction; we can use them as rollback during the architecture construction. We can compare architecture solutions, evaluate scenarios of execution, or analyze UML behaviors (we can select alternatives of state machines or other behavior descriptions associated to UML elements). If we want to evaluate our model architecture to select the best alternative, we can investigate different evaluation configurations, and compare their responses. These evaluations facilitate the configuration of hardware, middleware, and structural architectures; this is especially interesting in the evaluation of distributed software architectures.

Figure 1 represents the five basic phases of the evaluation sequence. These phases are:

1. **Architecture Modeling.** We start the process by building the *architectural model*. We do this using a CASE tool. Once the model is completed, we export the model information using XMI generator, or using proprietary CASE tool exporting facilities. This information will be used in the simulation scenario construction and in the simulation execution.
2. **Simulation Configuration.** The second step is the *configuration* of a simulation model. In general, a simulation evaluation only considers certain elements of UML models and selects specific model alternatives. A configuration includes basically two concepts: *initial instances* to be included in the simulation model, and *behaviors* that will be used in the execution of application elements.
   - **Initial Instances.** The model that represents our architecture can include different types of object instances that identify types of execution scenarios. For example, we can have a model that represents two different execution scenarios using two sequence diagrams. The two sequence diagrams will include different object instances, but in general we will not execute the system with all instances of those scenarios: we can do the execution with the first scenario, with the second, or with both at same time. The identification of initial instances allows the construction of an initial simulation scenario, which includes the distribution of the objects. [4] introduces a script language for the description of the initial configuration. In our solution, we can do this configuration with the explicit identification of instances (actor instances, objects, and node instances), and with the identification of scenarios based on sequence, collaboration and deployment diagrams. The diagrams selected include the instances that the simulation models will represent. The instances identified explicitly or with diagrams can make the indirect inclusion of other instances, the instances that they reference. The simulation will include those instances, and any UML element needed for their execution, that means, any model element that they reference directly (for example the instance classifier) or indirectly (for example the classifier operations). The initial value of attributes or associations depends on the attribute occurrences and association occurrences defined for the instance. Those initial values are defined explicitly in the UML instance, and represent the initial state of the instance.

- **Behavior Configuration.** The *state machine selection* identifies the behavior of modeling elements when we have different alternatives. An UML modeling element (e.g. operation, classifier, use case) can have several state machines associated. These state machines can represent alternative behavior, or model the management of different sets of events.

3. **Simulation Generation.** The configuration finishes with the simulation model generation. Section 3 includes a detailed description of the generation phase.
4. **Evaluation Configuration.** During this configuration we select the statistics that we want to evaluate, we chose the filter that we want to use in the evaluation, and we configure other simulation parameters like the amount of simulation time.
5. **Results Evaluation.** After the model execution, we can analyze the statistics. During the analysis we can compare different values, represent them graphically, do their comparison based on displacements, evaluate statistics values, etc. These analysis results can be used in the reconfiguration of UML architecture, or can be used as criteria for the architecture selection. We use OPNET facilities to visualize results.

The simulation model represents the distribution of UML elements (objects, operations, signals, states machines, and other UML elements) to node instances of the configuration.

2.1 UML Semiformal Specification

The mapping between UML elements and simulation models is the most critical aspect of the evaluation method. The UML semantics are semiformal. They are based on the UML metamodel [7], which includes structure diagrams that formalize the UML elements associations, and the constraints of these elements and associations. But this is a structural specification that does not pay special attention to the behavior of most UML elements (except behavior modeling elements). The behavioral specification is the most critical specification, if we want to do a simulated interpretation of UML models. The *pUML* group [12] has been created to study the problems of semiformal specification of UML.

UML modeling does not pay attention to some modeling aspects, which are critical in the non-functional evaluation of software architecture. Examples of these aspects are the system resources and their specification (amount of memory, CPU, system buffers, execution times, networks, etc.) and other non-functional constraints [3].

In the following, we introduce examples of incomplete or ambiguous UML specifications, and present the solutions that we have adopted. We found these to be especially important for the generation of simulation models.

- **Behavior Specification Based on State Machines.** The state machine is the most common behavior description method of UML. It is used in the specification of classifiers and other modeling elements, but there are some problems in their interpretation, because of their different semantics for different modeling elements. [4,2] describe some interpretations of associations of state machine to UML elements, especially classifiers and operations. In this first approach, we only con-

sider the association of state machines to operations and classifiers. Other possible associations are *packages* and *use cases*. Problems with the state machines description are:

1.1. If we use the state machines to model behavior of classifiers, the UML semantics interprets this association as the specification of classifier instances behavior. We interpret the state machines associated to operations as the behavior specification of operation invocations. When a classifier is associated with more than one-state machines, or its operations are associated to state machines, which state machine defines the behavior? (More than one state machine can recognize the same event). Figure 2 is a collaboration diagram that represents a metadata model (model of instances of UML metamodel), it includes instances of metaclass *class, operation* and *state machine*, and links representing association instances. `BankOffice` is a class, `transfer` is an operation, and `officeBehavior`, `officeBehavior2` and `transferParallel` are state machines. The class has associated two state machines, and the operation one. The operation is a feature of `BankOffice` class. Which one is the state machine that handles the *Call* events of operation *transfer*?

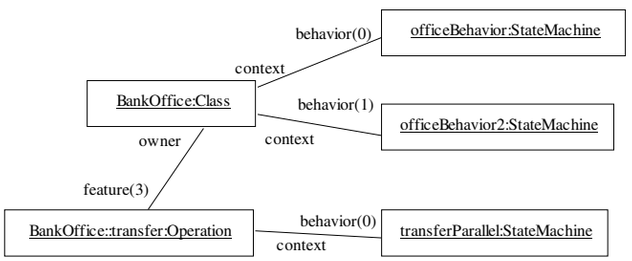


Fig. 2. Example of UML 1.3 Meta Data with Class and Operation, and Multiple State Machines

Solution: In our interpretation, when we select more than one state machine in the configuration of a classifier, or we include the state machine of a classifier and state machines associated to the classifier operations, all these state machines will handle all events occurred in the instance. For the operation state machines, UML specifications associate the operation *Call* event to the transitions from the initial state (UML 1.3 specifications introduces the last association).

1.2. Another example of indetermination with UML specification is the following: What is the behavior of an operation, when neither the operation nor its classifier are associated with a state machine? UML methods for behavior description are message sequences of collaboration and sequence diagrams, state machines and activity diagrams. Other methods are notes that include pseudo-code or source code.

Solution: We will consider four types of behavior description of application elements: state machines, resource consumption[1], an UML 1.3 extension of action sequences, and sequences of messages of collaboration and sequence diagrams. If an operation has not associated any of them, in our interpretation, its behavior will be the consumption of 0 instant of CPU (the operation execution can have associated a non 0 response time, the response time will depend on the CPU load, the CPU scheduling, and the scheduling parameters).

1.3. Whenever we create a new instance of a classifier that has associated state machines, we create a new instance of the state machine. Multiple instances of the same classifier can handle events in parallel, but one instance of state machine can only handle one event at the same instant (to introduce the problem, we do not consider parallel states or *fork* pseudo states). In case of state machines associated to operations, this will be different. In this case we create a new instance, whenever there is an operation invocation. This means that we can execute the same operation in parallel, in the same or different instances, and in the same operation state machine. The different interpretation of state machines in operations and classifiers implies a different interpretation of final state. When we enter in a final state of a top state, the entire state machine terminates, implying the termination of the entity associated with the state machine. When the entity is a classifier instance, this represents the instance termination (in our solution we destroy the object instance and the state machine instance too). When the entity is an operation this semantic is not specified.
Solution: When we enter a final state of an operation state machine, we terminate the operation flow execution. One type of UML action, executable in transitions and states, is the *Return* action. If we execute the *Return* action of the operation in a transition other than the transitions to the final states, the synchronous *Call* action that has started the operation will be unlocked, and the sequence of the call action will continue. If we do not execute the return action before entering in a final state, the *Call* action will continue blocked until the execution of a *Return* action for the *Call* event. We can execute the *Return* action, when we are handling another event (in the same or in different machines).

1.4. UML signals are associated with significant events in state machines; the signals are classifiers; the *Send* action is associated with a signal. UML semantics do not include details about the instantiation of signal classifiers; it does not introduce any detail about the instant of signal instance creation. The specification of the *Send* action describes aspects of sending and receiving, but not classifier instantiation and handling. The *Signal* event of UML specification describes the signal reception event, but does not introduce the classifier or instance handling.
Solution: We consider the signals and their events but we have not defined any specific behavior associated to the signals as classifiers. When we execute a *Send* action, the signal classifier instance must be provided as action parameter, and the signal destruction must be described in the application model.

1.5. UML specifications [7] and other references [4,18] introduce policies in state machine refinements associated to classifier inheritances. These rules are appli-

[1] Resource consumption is described with UML extensions techniques

cable when we use classifier state machines, but their application when we combine operation and classifier state machines is not evident. Figure 3 includes instances of the UML 1.3 metamodel; it represents a superclass, a subclass, a polymorphic operation, and four state machines that describe the behavior of classes and operations. In this figure, which are the state machines that must handle the *Call* event of operation `oneFeature` for instances of `subClass` and `superClass` classes?

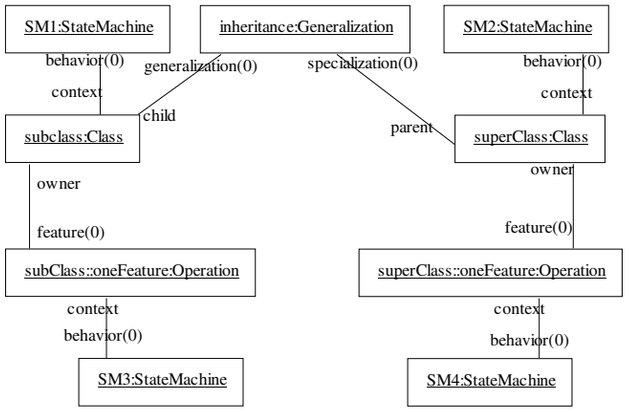


Fig. 3. Example of UML 1.3 Meta Data with Inheritance and Multiple State Machines

Solution: In our interpretation, there is no automatic generation associated to state machine behavior inheritance. Only the classifier of an instance can identify the classifier instances behavior, but different classifiers can refer to the same state machine. If an operation has associated a state machine, and another classifier inherits this operation and the subclassifier has associated a state machine, then both state machines will handle the invocation event. If the operation is redefined and a new state machine describes the behavior in the subclass, the operation behavior of superclass will not handle the event for subclass instances. In the previous example, state machines `SM3` and `SM1` will handle events of `subClass` instances and `SM2` and `SM4` will handle events of `superClass` instances. We do not consider consistence rules of superclass and subclass state machines during the generation.

- **Semantics of Actions.** The UML actions are described in the UML metamodel. They are a hierarchy of metaclasses that describe the different types of actions. The transition and states of state machines are described with these actions. Problems of action semantics are:

  2.1. We have considered all actions but the *Uninterpreted* action (which has no interpretation and we can consider it as an empty action) and the *Terminate* ac-

tions (which has semantics similar to the transition to final states). The *Return* action can only be executed as a response to a *Call* action of a synchronous operation. In this first approach we do not consider the recurrence and the asynchronous execution of actions. UML actions do not include actions to represent resource usage, or attribute access.

Solution: We have defined new actions to represent the CPU and memory consumption too. The action *New* and the final states modify the memory occupation too. We have actions to access and modify local and remote attributes, and actions to access and modify state machine local variables.

2.2. Our action language can include arithmetic, Boolean and string comparison expressions, provides access to parameters, and includes some specific variables. UML specification does not include details about time expressions, or how we can execute a *Return* action when we are handling a second event. [8] includes extensions for timing expression representation.

Solution: We have included three predefined variables in the action language: i) actual simulation time, ii) the event that is processed, and iii) the actual instance. The first is used in time expressions with a set of operators; the second is used when to return *Call* events handling another event. And the last variable provides the same functionality as *this* and *self-variables* of object-oriented programming languages.

2.3. The specification of UML states includes three execution phases: *entry*, *do*, and *exit* activities. The *exit* activity must be executed when the state is exited, and the execution time of this action is unpredictable (e.g., *Call* actions of synchronous operations of remote instances). One type of transition events is time events. These transitions define the maximum amount of time that we can stay in the state or the absolute instant when we must exit. When we arrive to that instant, we will execute the transition, if another transition has not been enabled yet. If we must execute the *exist* action of a state, the timed transition can be executed after the expiration time, and other transitions can be enabled between the deadline and the end of *exit* action. Figure 4 includes a state with empty *do* and *entry* actions; *exit* action needs x instants of simulation time to be executed (it is a synchronous *Call* action to a remote object). We enter in the state at instant t; the timed transition will be enabled at instant $t+4$, and at instant $t+y$ the event *operation1* arrives. When should we schedule the events? If $x < y$, only the timed transition can be enabled when *exit* action is finished, but if $x \geq y$, then any transitions can be enabled.

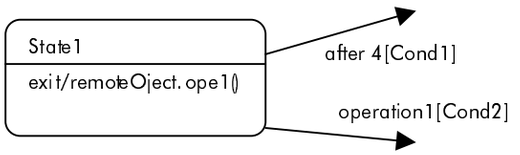


Fig. 4. Example of UML 1.3 State with Exit Action and Temporal Transition

Solution: We use a priority based scheduling *before* to start the *exit* activity. When there are more than one enabled transition with the same priority, we do a non-deterministic selection. Another solution would be to do the selection at the *end* of the *exit* action, but we must take into account the guards before and after the *exit* action and this can create inconsistencies. In the previous example, if the condition `Cond1` is `true` at instant *t+4*, our solution will start the exit at instant *t+4,* and at instant *t+x+4* the timed transition. In the alternative solution the other transition could be scheduled. We use an UML tagged value to specify priorities in instances, operations, classes, states, transitions and other UML elements. In the previous example, the transitions do not modify the priorities and will have the same priority as the transition that entered in the state.

- **Behavior Specification and Class Features.** In UML 1.3 the *Feature* metaclass specifies when the scope of a feature is a classifier or an instance. The state machines of classifiers represent the instance behavior. When we define classifier features (operations or attributes), they cannot be associated to state machine events. This creates some problems:

  3.1. The events of classifier state machines must be associated to instance features. How can we describe the behavior of class operations?
  Solution: In this case we use operation state machines, but this limits the communication between class operations and instance state machines. This communication is limited to class attributes. A class operation can only access to class attributes.

  3.2. How can we access to classifier attributes, or execute *Call* and *Send* actions to classifier operations and signals?
  Solution: In our implementation, each classifier has associated a pseudo-instance. This is an instance with semantics similar to Java or Smalltalk class objects. These pseudo-instances are identified with the class name, and are used in the invocation of class operations, and to access to class attributes. There is not state machines instance for these pseudo-instances.

- **Deployments Elements Behavior.** The description of some UML elements, especially the deployment diagrams, is incomplete for the generation of simulation models. We use extension mechanisms (tagged values and stereotypes) to do a detailed specification of these deployment diagrams. The detailed specification allows the identification of parameters like bandwidth, protocols, speed, error probabilities and sizes. Those parameters specify hardware elements like CPU, networks and memories. We include details of these extensions in Section 3.1. UML simplicity creates other problems:

  4.1. If we connect two node instances with more than one node link sequence, we will need to select a link to arrive to the other node, when we simulate a remote message deliberation.
  Solution: We reuse routing facilities of OPNET when it is possible, but in some cases we need to define a routing table to determine the link that we use in a node instance to arrive another node. We use these tables in the simulation model execution. We are working on these extensions looking for better solutions adapted to dynamic routing protocols.

4.2. Queue policies are not considered in UML specifications. Simulations associate queues to different UML elements, for example communication links, state machines and some resources like CPUs.
Solution: We use the UML tagged value *Priority* and policy constraints to make deterministic queue policies. The policy constraint is limited to the policies that we implement in the simulations.

- **Other Problems** are:

5.1. When an initial instance is considered in the configuration of the simulation, the instance can have associated initial values of attributes (*attribute occurrences*) and it can have associations to other instances (*association link occurrences*). Often, the *association link occurrences* are related to classifier associations. The class association can have an unlimited multiplicity in the association end. During the simulation we need specific notation to identify these links and the instances that they reference.
Solution: In general, we do not know all *association link occurrences* of an association classifier for a specific instance; they can be created dynamically. We use the notation *associationEndRoleName(index)* to identify these links during simulation and in the collaboration diagrams. The index is optional, when the multiplicity is 1 or 0.1. The value *index* can be unlimited, but we will have an error if we access to non-initialized links or if the index is out of range. Figure 2 and 3 include examples of this notation for the UML 1.3 metamodel associations. If we include links with a different notation in collaboration diagrams, they will not represent the instances of the association. In our interpretation, the link between two objects allows to send messages between objects. All link and attribute values initialized in the model will be initialized when the simulation starts.

5.2. Another general problem is the behavior when exceptions occur. For example, the instance of an action *Call* does not exist (or has been destroyed). What can we do with these exceptions? Should we define predefined signals? Should we stop the execution? Should we return error codes?
Solution: In the actual solution we return error codes.

3 Automatic Generation of Simulation Models

Figure 5 includes the general components of the system architecture. In general, we use Objecteering 4.3 as the UML editor [17], but we have studied the adaptation of the generator architecture to other UML/XMI tools [6].

The communication between the UML Editor and *Simulations Shaper and Generator* (SSG) is the *MetaData* component that includes the description of application elements based on a metamodel notation. The generator reuses a library of simulation models. They are generic simulation models that represent most of the UML metaclasses. The generator customizes them depending on the UML application. The customization is based on OPNET attributes and on the interconnection of these submodels.

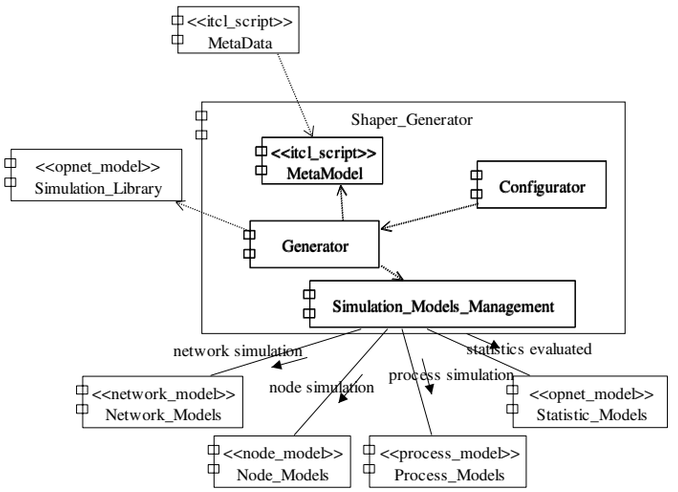


Fig. 5. Components of Simulation Shaper

SSG component includes four basic subcomponents:

1. ***MetaModel***. This is a hierarchy of classes and packages that represents the UML metamodel. It includes all UML metaclasses and the super metaclass *MetaMetaClass*, which is the supermetaclass of all metaclasses and includes a set of metadata operators of general purposes.
2. ***Simulation models management***. This component allows the representation of OPNET models and objects, and supports high level operations to handle them. These operations construct new models reusing and combining existing models, and update the simulation objects.
3. ***Configuration of simulation***. This component is a GUI used to configure the simulation generation. It is based on three UML element hierarchies. One hierarchy represents the metadata information. It is a browser of application elements. A second hierarchy represents the diagrams of the selected element (each UML element can have associated a set of diagrams) and the elements included in those diagrams. We can select instance elements included in the first and second hierarchy and diagrams included in the second. This selection defines the instances that will be used during the simulation generation, the state machines that represent the element behavior, and the behavior of actor instances based on the sequence and collaboration diagrams. The selections are included in the third hierarchy. This hierarchy has associated a command for the invocation to the generator. Another hierarchy includes the selected statistics that will be evaluated during the simulation. Each UML element has associated a set of statistics that can be selected in the evaluation configuration phase.
4. ***Generator.*** The generator implements the simulation generation. It extends the metamodel implementing the specialization of the same operation for each meta-

class, and other specific metaclass generation methods. The specializations use the simulation library models that represent the simulation behavior of metaclasses. Each specialization does the copy and paste of different simulation models, and modify its attributes (name, models used, speeds, sizes, etc.) to customize the generic model to the specific metadata instances. Those operations depend on the metaclass semantic, and the application properties.

3.1 Stereotypes Hierarchies for the Detailed Description of Deployment Diagrams

UML provides some extension solutions that allow the identification of domain specific modeling elements, and the definition of specific domain properties [16]. The extensions are based on stereotypes, tagged values and constraints. Each stereotype is associated to a metaclass; they define new metamodel constructors. The tagged values identify new parameters or information associated to the modeling elements. These parameters allow the representation of domain specific information. The constraints represent specific restrictions of modeling elements, with linguistic notations.

If we want to do the detailed modeling of deployment diagrams and hardware elements (nodes, networks, connections, etc.), we have found the UML notation is insufficient and imprecise. We have created hierarchies of stereotypes (stereotypes with inheritance relationships) to provide a more detailed description. The hierarchy depends on the network and node entities that we can represent in the simulation models. Each stereotype has associated certain tagged values that represent the stereotyped element parameters. Tagged values are inherited between stereotypes. We apply these stereotypes in deployment diagrams elements and we reuse that information in the simulation models generation.

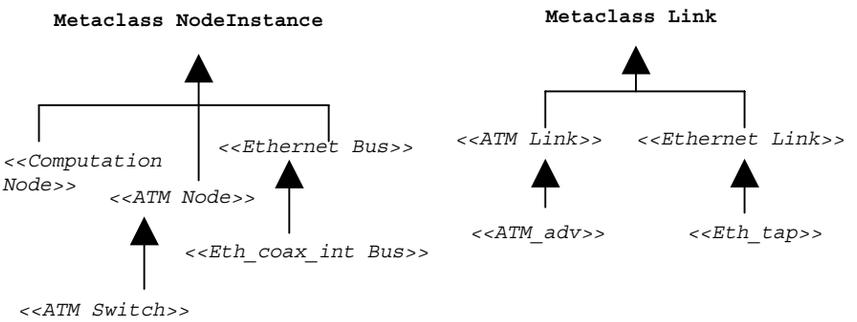


Fig. 6. Stereotypes Hierarchies of Deployment Elements

Figure 6 includes some examples of stereotypes of nodes and links of deployment diagrams. They are the stereotypes applied in the deployment diagram of Figure 7 and their superstereotypes. These stereotypes represent computation nodes, buses and communication links, and they have associated tagged values that specify parameters

like *data rate, queue policy,* and *QoS policy*. The stereotypes define different abstraction levels. For high levels we generate default configurations with default values, and when we need a more precise specification we use specialized stereotypes. The hierarchy that we interpret in the generators allows the specification of node, buses and links of ATM and Ethernet networks. Examples of node specialization are ATM and Ethernet switches, routers and bridges. Other networks included in OPNET that we have not implemented yet are BGP, Token Ring, and FDDI. Figure 7 includes computation nodes and some network elements that interconnect them. Object instances are located in the computation nodes, and the behavioral and structural elements specify their properties. The tagged value *location* specifies object locations. The network elements of this diagram include an Ethernet bus, ATM links, and an ATM switch. Tagged values of buses and links specify the data rate, background utilization, propagation speed, and error model. The ATM switch has associated specific attributes as switching speed and delays.

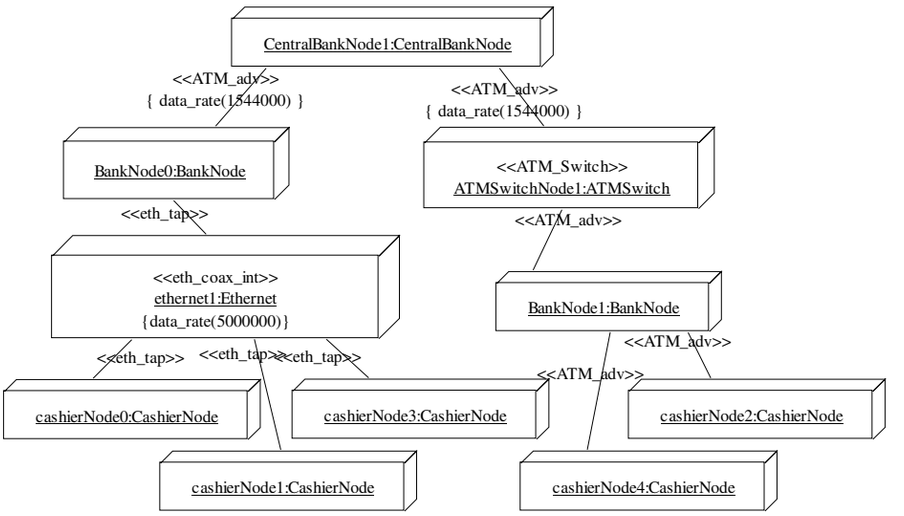


Fig. 7. Example of Stereotyped Deployment Diagram

4 Evaluation of Quality Parameters

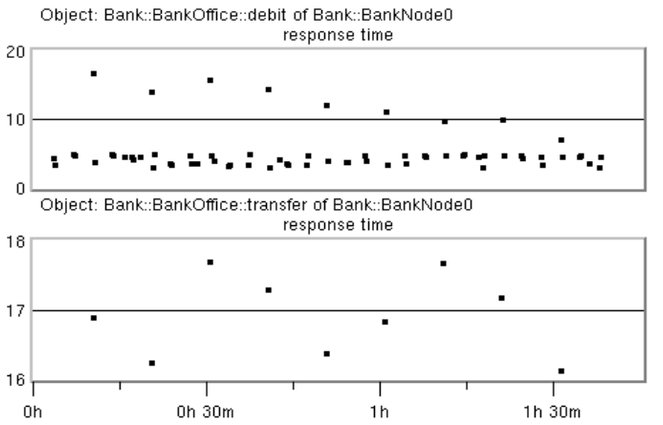
After the selection of statistical results and the simulation execution, we will have a set of statistical results and we can do the architecture evaluation. In the configuration we have chosen diverse filters for the evaluation of statistical result. These filters define the statistical methods for the evaluation of the different statistical parameters.

Examples of statistical results that we can get include: i) graphical representation of statistical vectors, ii) number of values of statistical vectors, iii) minimum and maximum values, iv) expected value (this value is evaluated by time average filter and provides information about the average statistic), v) variance, and vi) confidence intervals (these results provide information about the distribution of different values). The evaluation depends on the type of statistics provided; we have identified parameters for the evaluation of performances, resources capacities and reliability. The types of parameters that we have identified are:

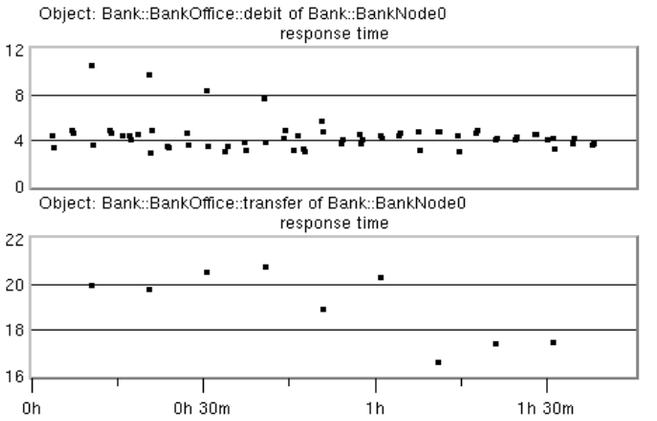
- **Resources capacities.** These parameters include the identification of resources' states (busy and idle), the distribution of resource requests, size of resource queues, response time of resource request, and other specific resource parameters like preemption times and network collisions. The resources evaluated are *CPU*, *memory* and *networks*. Examples of application are the evaluation of frequencies of memory allocation, network access and CPU consumption. Some real-time systems include limitations of size and frequency of memory allocations, CPU consumption, and network access; those limitations allow the identification of garbage collection and memory management execution times, preemption, and network blockages. If we want to evaluate two alternative architectural solutions, these statistics will identify the distribution frequency of both solutions.
- **Performance.** Some UML elements like object instances, classes, instances of state machines, and states have associated statistical parameters that identify their idle and busy state. Other statistics identify the invocation of methods and signals in those elements, the invocations of other objects and classes, and the access and modification of their attributes. Elements like operations, signals and state machines include statistics of their invocation and their responses. Those statistics allow the evaluation of worst case response time of operations and state machines invocations during the simulation execution, and their comparisons for different models. Some specific performance parameters are the ATM QoS attributes and other network dependent parameters.
- **Reliability.** Some events like operation invocation of deleted objects, and network access can have associated error responses. We can model the error occurrence in object behavior and in their disposition, and we can specify attributes of network error occurrences. An object can be destroyed and created again with the same identity, this can be used in combination with statistics like object *availability*, *time to repair* object availability, *and time to failure.* Other statistics provide information about error occurrences, erroneous responses and network availability. The application behavior can detect the error occurrences and handle these errors; this is useful in the representation of reliable behaviors.

[2] includes a practical example of behavior modeling in a banking system. We have modeled and extended this example with static diagrams, deployment diagrams (Figure 7 is the deployment diagram of this example), state machines for classes and operations, collaboration diagrams for the initial configuration of the system (initial objects and links that represent associations instances), sequence diagrams that describe the behavior of actors, and actions that represent the CPU consumption. [2]

considers two alternative behaviors to handle *transfer* operation (it represents the transfer between accounts of different banks).



(a) Response Times for Sequential Solution



(b) Response Times for Parallel Solution

Fig. 8. Response Times of Operations

Figure 2 models the alternative behavior and the state machines that represent the behaviors that we will consider. We can represent the behavior of a bank with a state machine that handles all operations sequentially (`officeBehavior` in Figure 2). Or we can represent the behavior with two state machines, the first execute all operations

sequentially, but *transfer* (`officeBehavior2` in Figure 2), and an operation state machine represents the behavior of *transfer* operation (`transferParallel` in Figure 2). *Transfer* operation has the longest response time, and communicates four nodes in Figure 7 (*BankNode0, CentralBankNode1, ATMSwitchNode1,* and *BankNode1*). In the second alternative we can execute any number of *transfer* operations in parallel with the rest of operations. The second alternative provides better response times in *credit* and *debit* operations, but can create inconsistencies if we liquidate a bank when the transfer has not finished yet, and this can create *availability* errors.

Figure 8 is an example of statistic results for both behavior configurations. The figure includes the *response times* of *debit* and *transfer* operations of class *BankOffice* in node *BankNode0*. In this simulation, external actors call the operations periodically, for different accounts, with different types of distributions (constant, exponential and uniform) and parameters. In our model, the actors communicate with object instances included in *cashier* nodes, and the cashier nodes communicate with the banks. In Figure 8 (b) we can see that maximum response time of *debit* operation in the parallel configuration is less than in the sequential. In sequential solution (Figure 8 (a)), the sample mean of *transfer* operation is 16.9249 s, the maximum value is 17.6996 s. The sample mean of *debit* operation is 5.2612 s, and the maximum value is 16.7950 s. In parallel solution (Figure 8 (b)) the sample mean of *transfer* operation is 19.0781 s, the maximum value is 20.7834 s. The sample mean of *debit* operation is 4.4728 s, and the maximum value is 10.6698 s. The configuration do not generate *availability* errors after 6000 s of simulation.

5 Conclusions and Future Work

Simulation techniques give support for evaluating UML models. We can complement simulated evaluations with other types of techniques, for example analysis and code generation. UML provides a notation for the description of distributed object-oriented systems, but the notations are imprecise and the semantics are semiformal. We must use UML extension techniques to provide a detailed description, especially in deployment elements. If we use any interpretation technique for those models, we must identify certain imprecision of UML specification and determine our specific interpretation. The simulation methods provide flexible support for the evaluation of performance and reliability qualities of software architectures. Performance and reliability are important QoS, but we must precise the specific parameters used in their evaluation; these parameters depend on the type of systems. In general, the performance evaluation of general distributed systems is based on throughput, but the evaluation of hard real-time systems is latencies oriented. The types of parameters in both cases are different. In case of reliability, distributed systems evaluate the availability of elements, and hard real-time systems evaluate the deadline fulfillment. We are considering the extensions of statistics and simulation models, and their application in the specification of fault-tolerant systems.

Currently we are working on the extension of the deployment stereotypes hierarchy to include other types of networks, routers and switches, and the representation of middleware dependencies in simulation models. Those simulation models represent CORBA elements and a proprietary middleware based on CORBA [5]. They extend the basic system models that we use actually (CPU, its scheduling policy, memory, network access and some basic network protocols). We will use those extensions in the evaluation of CORBA systems modeled with UML. We are extending our generators to take into account some UML elements like components, and represent their behavior in simulation models.

Acknowledgements

This work is partially supported by RNTL Oural-99, and is developed in Alcatel/Thomson-CSF Common Research Laboratory (LCAT).
We sincerely thank Stefan Tai for improving this paper with his careful reading and invaluable comments.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson: UML User Guide. Addison-Wesley (1999)
2. R. Breu and R. Grosu. "Relating Events, Messages, and Methods of Multiple Threaded Objects". Journal of Object-Oriented Programming. Vol. 12 No. 8 (January 2000)
3. M. de Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec: "UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models". In Proceedings 2nd Workshop on Software and Performance (WOSP2000). ACM (September 2000)
4. D. Harel, and E. Gery: "Executable Object Modeling with Statecharts". COMPUTER IEEE (July 1997)
5. J. Maisonneuve, S. Chabridon, and P. Leveillé: "The PERCO Platform". In Object-Oriented Real-Time Distributed Computing (ISORC99) *(*May 1999)
6. Object Management Group: XML Metadata Interchange (XMI). OMG. (October 1998)
7. Object Management Group: OMG Unified Modeling Language Specification. OMG (June 1999)
8. Object Management Group: UML Profile for Scheduling, Performance and Time. Join Submission. OMG (August 2000)
9. Object Management Group: Meta Object Facility (MOF) Specification. OMG (June 1999)
10. OPNET: OPNET Manual. MIL 3, Inc (1999).
11. Permabase: http://www.cs.ukc.ac.uk/projects/permabase (September 2000)
12. PUM: http://www.cs.york.ac.uk/puml (September 2000)
13. ilogix Raphsody Modeler: http://www.ilogix.com (September 2000)
14. Rational. Rational Rose Real-Time: http://www.rational.com/products/rosert (September 2000)

15. SES/Workbench. http://www.hyperformix.com/lowres/products/workbench/workbench.htm (September 2000)
16. S. Si Alhir: "Unified Modeling Language: Extensions Mechanisms". Distributed Computing (December 1998)
17. Softeam: "Objecteering CASE Tool". Softeam Europe. http://www.objecteering.com (September 2000)
18. M. Stumptner and M. Schrefl: "Behavior Consistent Inheritance in UML". In Proceedings of the 19th International Conference on Conceptual Modeling. ER2000 (October 2000)

Architectural Reflection

Realising Software Architectures via Reflective Activities

Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio

D.I.S.Co. – Università di Milano-Bicocca. Milan, Italy
{tisato,savigni,cazzola,sosio}@disco.unimib.it

Abstract. Architectural reflection is the computation performed by a software system about its own software architecture. Building on previous research and on practical experience in industrial projects, in this paper we expand the approach and show a practical (albeit very simple) example of application of architectural reflection. The example shows how one can express, thanks to reflection, both functional and non-functional requirements in terms of object-oriented concepts, and how a clean separation of concerns between application domain level and architectural level activities can be enforced.

1 Introduction

Software architecture is an infant prodigy. On the one hand, it is an extremely promising field, and these days no sensible researcher could deny its prospective importance. On the other hand, it is an extremely immature subject; as a matter of the fact, there is little (if any) agreement even on its definition (see [26] for an extensive, but probably incomplete, list of such attempts).

A major problem in this field is the semantic gap between architectural concepts and concrete implementation. Many architectural issues, in particular those related to non-functional requirements, are realised by mechanisms spread throughout the application code or, even worse, hidden in the depth of middleware, operating systems, and languages' run-time support. This is what we call the implicit architecture problem, which is especially hard for distributed objects systems, where clean objects representing application domain issues rely on obscure system-dependent features related to architectural concepts (static and dynamic configuration, communication strategies, Quality of Service, etc.).

The goal of this paper is to show how a systematic approach based on computational reflection i.e., *architectural reflection*, may help filling this gap by reifying architectural features as meta-objects which can be observed and manipulated at execution time. This lifts up to the application level the visibility of the reflective computations the system performs on its own architecture and ensures a proper separation of concerns between domain-level and reflection-level activities. The proposal derives from the authors' experience both in related research areas and in the development of real, industrial projects where the key ideas presented in the paper have been developed and exploited.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 102–115, 2001.

The paper outline is the following. Section 2 presents an example that will be employed as a reference throughout the paper. Section 3 introduces the problem of implicit architecture. Section 4 presents the fundamental concept of architectural reflection. Section 5 sketches, in an ideal scenario, how the requirements of the example can be fulfilled via architectural reflection, while Sect. 6 discusses how the ideas can turn into the practice of real-life systems. Section 7 compares the approach with other work. Finally, Sect. 8 presents the current state of the work.

2 An Example

The following example, which is a simplified version of a real-life problem in the area of on-line trading, will be the basis for the discussion.

In a virtual marketplace one or more feeders (the information providers) provide on-line stock exchange information to a set of customers (the information consumers i.e., basically the clients). Such a system has three basic requirements:

1. the marketplace must ensure that local views of information, held by the clients, are kept aligned with a reference image of the information itself, maintained by the feeders;
2. a stock broker may place buy or sell orders;
3. the marketplace evolves through several different phases. There is an opening phase, during which privileged users (not discussed here) define the initial prices. Then there is a normal phase, which is the only one during which customers may buy or sell. Finally, there is a suspension phase, during which customers can only observe the prices.

The system has three more requirements:

4. customers may dynamically join or leave the marketplace, and an overall supervision of which customers are connected must be ensured. In addition, the marketplace must be able to disconnect a client e.g., for security purposes, should any doubt arise as to the client's actual identity;
5. each customer must be capable of selecting the alignment strategy for their local image: on request (a.k.a. *pull*), on significant changes (a.k.a. *push*), or at fixed time intervals (a.k.a. *timed*);
6. the system must be flexible with respect to the number and physical deployment of the feeders. Adding or removing a feeder should *not* imply any change in the client code.

All this looks quite simple. However, as soon as we go through the analysis and design process (for instance using UML [3]), we recognise that requirements 1, 2, and 3 (let us call them "functional requirements") can be easily expressed in terms of well-defined domain classes. 1 and 2 can be expressed via simple class and interaction diagrams, and 3 via state diagrams.

Things are not as simple for requirements 4, 5, and 6 (let us call them "non functional requirements"). Regarding 4, any distributed platform provides mechanisms for dynamic connections to services. However, in most cases it is not easy to monitor who is connected. Such information exists somewhere inside the middleware, but it can hardly be observed and relies on platform-dependent features. The result is that the application is not portable. We would like to provide the end user (e.g., the manager of the marketplace) with a clean and platform-independent visibility of the status of the connections.

A similar, but harder, problem holds for requirement 5, whose fulfilment implies the definition of an application-level protocol whose behaviour can be dynamically selected according to users' taste. Even if at the analysis stage the protocol is well specified (e.g., as a state machine), at the design and implementation stages it is split into a specification of individual components' behaviour and then implemented by components' code. This code (implementing an architectural choice) will be intermixed with architecture-independent, functional code. In current practice, most architectural choices follow this fate and get dispersed in the components' code in implemented systems. Moreover, the implementation relies on elementary transport mechanisms (RPC or asynchronous messages or the like). Therefore, a clean and platform-independent visibility of the communication strategies is not provided at the application level.

Finally, point 6 is just an example that raises the most important issue dealt with in this paper i.e, the implicit architecture problem, whereby information about the system architecture is embedded in application-level code. As a matter of the fact, the number and actual deployment of the feeders is an architectural issue, that should *not* be embedded in application-level code.

3 Implicit Architecture

The example highlights that, while object-oriented technology provides a sound basis for dealing with domain-related issues, we still lack adequate notations, methodologies, and tools for managing the *software architecture* of the system [24]. The architecture of a software system based on distributed objects is defined by stating:

- how the overall functionality is partitioned into *components*;
- the *topology* of the system i.e., which *connectors* exist between components;
- the *strategy* exploited in order to co-ordinate the interactions and, in particular, how connectors behave.

Existing notations for composing software modules are usually limited to expressing some small subset of these issues e.g., topology alone or basic communication paradigms such as RPC or asynchronous messages. Many architectural concepts, even if well specified at the design stage, are no more clearly visible at the implementation stage and at execution time. Moreover, many architectural issues (such as scheduling strategies, recovery actions, and so on) are tied to the concrete structure and behaviour of the underlying platform. Therefore,

such issues are spread throughout the (opaque and platform-dependent) implementation of OS and middleware layers. In other words, this means that these concepts are confined at the *programming-in-the-small* level [8].

Ultimately, there is no separation of concerns among domain, implementation, and architectural issues, and the latter ones are not clearly visible and controllable at the application level. This is what we termed the *implicit architecture problem (IAP)* in a previous paper [5].

Designing and building systems with implicit architectures has several drawbacks: most notably, it hinders components' reuse due to the architectural assumptions components come to embed [10]; it makes it infeasible to reuse architectural organisations independent of the components themselves; it makes it overly complex to modify software systems' architecture; and it is also cause of the undesirable, yet empirically observed fact that architectural choices produced by skilled software architects are most often distorted and twisted by implementers [15]. A more detailed discussion of the IAP and its consequences can be found in [4].

The IAP is especially serious because in most real systems architectural issues must be dealt with at execution time. The topology may dynamically change either by adding new components or by modifying their connections, in order to extend or modify system functionality, to add new users, to enhance availability, to perform load sharing, etc. The strategy too may change in order to meet changing user requirements and timing constraints, to ensure a given average rate and reliability for data transfer, and so on. The need for dynamic management of architectural issues often arises from non-functional requirements i.e., configurability, availability, performance, security and, in general, Quality of Service.

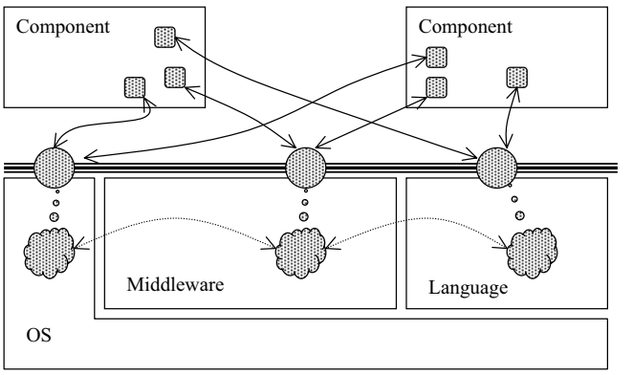


Fig. 1. The Implicit Architecture Problem

Figure 1 sketches a typical situation in terms of concrete run-time architecture (i.e., in terms of well distinguished objects which exist at execution time). Even if components exist at run-time as (distributed) objects, the connectors'

implementation is heavily spread. Inside the components there are code fragments which rely on application-level interfaces such as programming language constructs, middleware APIs, or OS primitives. Such interfaces are implemented inside the underlying platforms via hidden mechanisms which, in turn, interact in mysterious ways. Unfortunately, such mechanisms implement architectural issues (mainly those related to non-functional requirements) which can be hardly observed or controlled.

Expressing functional requirements in terms of application domain concepts modelled as classes and objects provides the basis for building well-structured systems which exploit domain-level classes and objects to fulfil the functional requirements. Accordingly, *expressing non-functional requirements in terms of architectural concepts modelled as classes and objects should be the basis for building well-structured systems which exploit architectural classes and objects aimed, in particular, at fulfilling non-functional requirements*. As a matter of the fact, the IAP mainly arises from the fact that architectural information is spread throughout application and platform code, and is not properly modelled in terms of well-distinguished classes and objects.

4 Architectural Reflection

All of the above issues imply that a running system performs computations about itself. More practically, there exist somewhere data structures representing system topology and system behaviour, plus portions of code manipulating this information. Architectural reflection (AR) basically means that there exists a clean self-representation of the system architecture, which can be explicitly observed and manipulated.

In its general terms, computational reflection is defined as the activity performed by a software agent when doing computation on itself [14]. In [5,4] we defined architectural reflection as the computation performed by a system about its own software architecture. An architectural reflective system is structured into two levels called architectural levels: an *architectural base-level* and an *architectural meta-level*.

The base-level is the "ordinary" system, which is assumed to be designed in such a way that it does not suffer from the IAP; in the sequel we shall discuss how to achieve this goal. The architectural meta-level maintains causally connected objects reifying the architecture of the base-level.

According to the concept of domain as introduced in [14], the domain of the base-level is the system's application domain, while the domain of the architectural meta-level is the software architecture of the base-level.

Architectural reflection can be further refined into *topological* and *strategic* reflection. Topological reflection is the computation performed by a system about its own topology. With regard to the example, topologically reflective actions include adding or removing customers to the marketplace.

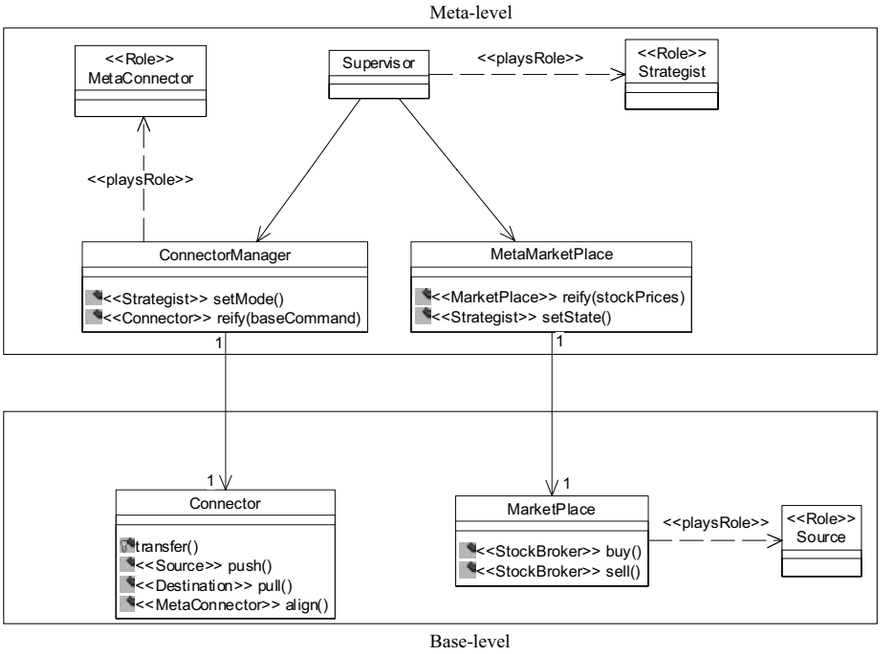


Fig. 2. An example of an ideal system

Strategic reflection is the computation performed by the system about its own strategy; for example, dynamically changing the strategies for the alignment of customers' local images of the stocks.

Architectural reflection has the desirable effect of lifting architectural issues up to the programming-in-the-large level.

5 An Ideal Scenario

Architectural reflection is quite a general concept, and assumes only a IAP-free base-layer. This section briefly describes the ideal approach in building both the base- and the meta-level. The on-line trading example first encountered in Sect. 2 will be used as a (simplistic) case study (see Fig. 2).

Note that in this example, roles are treated as first-class entities, and represented as classes with a `Role` stereotype. A class can play one or more roles (via the `playsRole` stereotype). Operations can be restricted to one or more roles only. This means that only instances of classes playing that role can call those operations. This selective operation export is represented by tagging the operation with the stereotype bearing the name of the role the operation is exported to (e.g., `<<Source>>`). In some cases, such as operation `reify()` in class `ConnectorManager`, the operation is directly exported to class `Connector` for the sake of simplicity; however as a general rule, operations are never exported

directly to object classes in order to avoid hard-coding in a class the dependency to a particular context (see [19] for further details).

5.1 Base-Level

Both components and connectors in the base-level should be realised as passive entities. This means that they should not embed any activation strategy. In this way, they can be reused under completely different strategies.

Connectors embed all communication issues, such as relationships with the underlying middleware (if any), network protocols, and so on. They export to the above levels a uniform interface, which is strictly independent on the underlying implementation details. In this way, when composing a system one can reason at a higher abstraction level than that allowed for by common middleware (such as CORBA), in that one can actually ignore distribution issues. This is not the case when interacting directly with middleware.

In other words, the goal of our approach is to *separate* distribution from the other issues, definitely not to ignore distribution. This is a very controversial point, about which a lot of discussion is taking place (see for example [11]). One very popular approach (adopted for example by CORBA) is to *mask* distribution; every method call is accomplished in the same way regardless of the actual location of the target object (this is the largely touted "distribution transparency"). This approach has the undisputed advantage of greatly fostering reuse, as components are independent of distribution. However, many people argue that issues such as latency and partial failure, typical of distributed systems, make it impossible in practice to systematically ignore distribution, especially in real-time systems.

We believe that many of these problems stem from an excessive urge to encapsulate. While encapsulation is undoubtedly a key achievement in software engineering, it should not be abused. Issues that are of interest to the rest of the system should definitely be visible through the encapsulating shield. In addition, in some cases distribution is meaningful even at the analysis level (the so-called "intrinsic" – or should we call it "analysis-level" – distribution, as opposed to "artificial" – or should we call it "architecture-level" – distribution useful e.g., for fault tolerance purposes). All of this can be ascribed to the very in issue of Quality of Service, that we do not mean to address here. Our approach to distribution tries to separate distribution from other issues, not to ignore it. If distribution is kept well separate from other issues (computation, strategy, etc.), it can be properly dealt with in one place and ignored by the rest of the system.

In the example, the `MarketPlace` class exports only two operations, both to the `StockBroker` role (in the base-level). The idea is that only application-domain activities are represented at the base-level. So for example, in the `Marketplace` class there is only information about the current value of stocks, and there is no way at this level to influence the policies of the stock market (such as opening or closing the negotiations).

Similarly, the `Connector` class only offers the means to push and pull information; the corresponding operations are exported to the `Source` and `Destination`

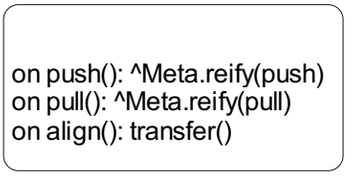


Fig. 3. The state diagram for class `Connector`

roles respectively. Consequently, the corresponding state diagram is trivial[1](see Fig. 3).

The `align()` operation can only be called from the meta-level, and is implemented by the protected `transfer()` method, which actually performs the information transfer.

5.2 Meta-Level

In the meta-level reside meta-objects for both components and connectors. These encapsulate the *policies*, both application-related and communication-related. So for example, the `MetaMarketPlace` class encapsulates all the policies that are usually enforced by the financial authorities (such as withdrawing stocks from the market). Similarly, the `MetaConnector` class manages the communication policies (push, pull, timed), as shown in Fig. 4.

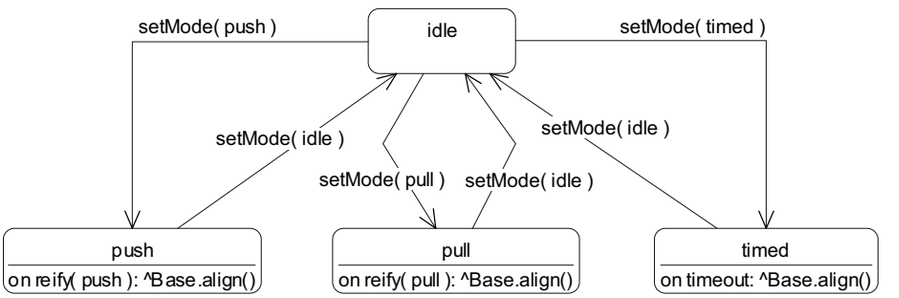


Fig. 4. The state diagram for the `MetaConnector` class

In addition, in order to meet requirement 4 in Sect. 2, a topologist must be introduced that manages, at execution time, the architecture of the software system, by instantiating components and/or connectors. For further details, see [7,6].

[1] Due to space limitations, throughout the example we only detail communication-related classes.

6 Turning Ideas into Practice

The approach outlined in the previous chapter, albeit simplistic, sets the ideal properties of an architectural reflective system. However, it is widely known that a software system is almost never built from scratch; more often than not, the prevalent activity in software construction lies in integrating existing pieces of software into a new product. Therefore, no discussion of a practical software construction paradigm would make any sense without turning to the real world and discussing the practical applicability of the paradigm.

Architectural reflection is no exception; in this section we briefly set forth some requirements that existing systems must meet in order to be good candidates for being integrated into an architectural reflective system.

6.1 COTS

COTS should not embed architectural issues. Apart from architectural reflection, this seems to be a general requirement (see also [10]). In other words, COTS should simply provide functionality without falling into the IAP.

For them to be part of an architectural reflective system, they should provide:

- hooks for connectors. Such hooks should be visible in the component's interface;
- visibility of their internal state, so as to allow meta-level entities to operate on them;
- most important, they should embed no activation strategy. In other words, they should be passive entities.

6.2 Legacy Systems

It is often said that any running system can be thought of as a legacy system. Due to this variety, integrating legacy systems into a new product is often close to impossible. Such systems are more often than not poorly documented or not documented at all, and are often built out of spaghetti code. In these systems it is often extremely hard even to understand *what* the system does, let alone *how* it does it.

The solution usually employed to integrate this class of systems is to wrap them in a modern, most often object-oriented shield (this is for example the approach used to integrate them into a CORBA environment). However, since it's hard to fully grasp their functionality, usually just a subset of this is actually exposed by the shield.

Having said that, the most basic requirement for integrating legacy systems is that systems should embed no activation strategy. Since this can be an overly strong requirement, it can be weakened into the following: embedded strategies are allowed as long as they do not affect domain requirements. This basically means that embedded strategies should not prevent an external strategist to *examine* the basic policies and *modify* them when needed. If the system is completely opaque and allows no insight into its strategy, then it just cannot fit into an architectural reflective system.

6.3 One Remark about Conceptual and Practical Approaches

Sometimes it may be impossible or impractical to build an actually layered system such as the one outlined in Sect. 5. However, it should be noted that, even when *actually building* such a system is impossible, one can always follow the *conceptual approach* described in this paper, possibly implementing the system using conventional means.

So for example it may be impractical to implement connectors with two classes (`Connector` and `ConnectorManager`, see Fig. 2). In that case, one can merge the two classes into one, obtaining a state diagram such as the one shown in Fig. 5.

In this case, there is no explicit distinction between the two layers. However, a nice separation of concerns between domain-related activities (information transfer) and architectural activities (selection of policies) still holds; in fact the `on xxx: yyy()` clauses represent the former, while actual state transitions represent the latter.

In other words, a conceptual approach is useful even if it does not turn into actual design and code. This is like implementing a well-designed, object-oriented system in assembly language for optimisation purposes; the level of reuse is certainly not the same as using an OO language, but most of the benefits of a good design are preserved.

6.4 One Remark about Architectural Languages

A lot of discussion is going on in the software architecture community about what is to be considered an architectural language and what isn't. As a matter of the fact, OOPSLA 99 hosted a Panel session entitled "Is UML also an Architectural Description Language?"

Clearly, in the conventional, "flat" (i.e., non-reflective) approach neither design languages nor programming languages offer sufficient means to express architectural issues. Thus, the need arises to extend the programming paradigms with architectural/non-functional issues. Examples include introspection à-la JavaBeans, communication primitives with timeout, priorities, etc.

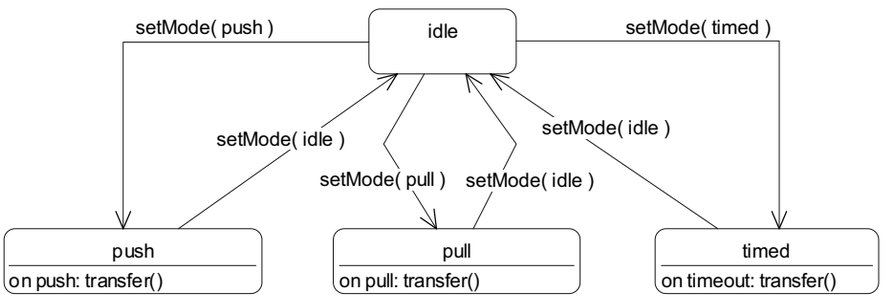


Fig. 5. The state diagram of a merged `Connector` class

As far as the UML in particular is concerned, the answer to the above question is certainly: "no, using the flat approach. Yes of course, provided a reflective approach is employed instead." As a matter of the fact, one of the key features of reflection is that the same language can be used for both base-level and meta-level, provided that the language includes a mechanism (reification and reflection) for causal connection. In (architectural) reflective terms, switching from domain level to architectural level simply means changing domain (again in the sense used in [14]). This does not imply a change in language. In other words, architectural objects can be treated as first class objects (architectural objects) in the reflective level, thus achieving the fundamental goal of separation of concerns.

7 Related Work

The idea of enforcing a separation of concerns between basic computational blocks and the entities governing their overall behaviour and cooperation periodically reappears in different branches of information technology under different guises and formulations. The seminal paper by DeRemer and Kron [8] proposed using a different notation for building modules and for gluing modules together, yet this latter notation could only convey simple define/use relationships. A whole family of coordination languages, termed control-driven [17], and especially the Manifold language [2], are also based on the idea of separating computation modules (workers) from "architectural" modules (managers) at run-time. While this is quite close to the concept of explicit architecture on which this proposal builds, it is not the intent of Manifold (and other coordination languages) to allow the encapsulation of architecture in the broadest sense of the term. Nevertheless, it must pointed out that Manifold also provide constructs to address dynamic architectural change via managers, which also has similarities with AR. Other recent proposals focus on run-time connectors (explicit run-time representation of cooperation patterns) and include Pintado's gluons [18], Aksit et al.'s composition-filters [1], Sullivan's mediators [25], and Loques et al.'s R-Rio architecture [13].

Several authors have confronted with the problem of modifying architecture at run-time, for reconfiguration or evolution purposes. Kramer and Magee [12] discuss an approach to runtime evolution that separates evolution at the architectural and application level. Architectural reconfiguration is charged to a configuration manager that resembles AR's meta-entities. In their approach, nevertheless, the meta-level has a limited visibility of the "base-level" state (i.e., it only perceives whether base-level entities are in a "quiescent" state). Oreizy et al. [16] propose a small set of architectural modification primitives to extend a traditional (non-dynamic) ADL, and also exploit connectors to let architectural information be explicit in running systems. With respect to AR, their approach is more related to defining what operations are useful at the meta-level than to devising how the meta-level and base-level should interact (which is the main focus of this paper).

Very few works insofar have pointed at the advantages of a reflective approach for the design of systems with dynamic architecture. One of those few proposals is that of Ducasse and Richner, who propose introducing connectors as run-time entities in the context of an extended, reflective object model termed FLO [9]. FLO's connector model is very rich and interesting, and has several similarities to ours. Nevertheless, FLO is based on a simpler component model which does not include a behavioural component specification.

8 State of the Work

The approach described above is far from a speculative vision. On the contrary, it's being employed under many forms in several practical situations.

A somewhat simplified version of the framework is the base for the Kaleidoscope reference architecture [21,23,22], which is being employed in several industrial projects in the areas of traffic control and environmental monitoring. Kaleidoscope is also undergoing major improvements in the form of an object-oriented framework and an associated methodology, all heavily influenced by the reflective concepts described above.

The idea of separating functional from non-functional, and in particular management, issues has been successfully exploited in the design and implementation of a platform which integrates heterogeneous devices and applications for traffic control at the intersection level [20].

On a more theoretical, long-term perspective, we are working on a more formal definition of reflective architecture. In particular we are evaluating UML (possibly augmented with OCL [27]) as a meta-level architectural language, which is giving sound and promising results. In particular, the concept of role as described in Sect. 5 is proving very interesting and powerful (see [19] for further details).

References

1. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of Object-Based Distributed Programming (ECOOP94 Workshop)*, number 791 in LNCS, pages 152–184. Springer-Verlag, Jul 1994. 112
2. F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and Its Implementation. *Concurrency: Practice and Experience*, 50(1):23–70, Feb 1993. 112
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. 103
4. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. A Fresh Look at Programming-in-the-Large. In *Proceedings of The Twenty-Second Annual International Computer Software and Application Conference (COMPSAC '98)*, Vienna, Austria, Aug 13-15, 1998. 105, 106
5. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2^{nd} Euromicro Conference on Software Maintenance and Reengineering and 6^{th} Reengineering Forum*, Florence, Italy, March 8-11 1998. 105, 106

6. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection: Concepts, design, and evaluation. Technical Report RI-DI-234-99, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1999. 109
7. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of Automated Software Engineering – ASE'99 14th IEEE International Conference*, pages 263–266, Cocoa Beach, Florida, USA, Oct 12-15 1999. 109
8. F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-small. *Transactions on Software Engineering*, SE-2:80–86, June 1976. 105, 112
9. S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC'97*, LNCS 1301, pages 483–500. Springer-Verlag, 1997. 113
10. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of XVII ICSE*. IEEE, April 1995. 105, 110
11. R. Guerraoui and M. E. Fayad. OO Distributed Programming Is *Not* Distributed OO Programming. *Communications of the ACM*, 4(42):101–104, 1999. 108
12. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transaction on Software Engineering*, SE 16(11), Nov 1990. 112
13. O. Loques, A. Sztajnberg, J. Leite, and M. Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 191–210. Springer-Verlag, Heidelberg, Germany, June 2000. 112
14. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA87, Sigplan Notices*. ACM, October 1987. 106, 112
15. G. C. Murphy. Architecture for Evolution. In *Proceedings of 2nd International Software Architecture Workshop*. ACM, 1996. 105
16. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings ICSE 98*, pages 177–186, Kyoto, Japan, Apr. 1998. 112
17. G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46, 1998. 112
18. X. Pintado. Gluons: A Support for Software Component Cooperation. In *Proceedings of ISOTAS'93*, LNCS 742, pages 43–60. Springer-Verlag, 1993. 112
19. A. Savigni. RoleJ. A Role-Based Java Extension. In *Proceedings of ECOOP 2000 (The 14th European Conference for Object-Oriented Programming)*, Cannes, France, June 12 – 16 2000. Poster. To appear. 108, 113
20. A. Savigni, F. Cunsolo, D. Micucci, and F. Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8th Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 11 – 14 2000. 113
21. A. Savigni and F. Tisato. Kaleidoscope. A Reference Architecture for Monitoring and Control Systems. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, USA, February 22-24 1999. 113
22. A. Savigni and F. Tisato. Designing Traffic Control Systems. A Software Engineering Perspective. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the*

International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT), Roma, Italy, September 11–14 2000. 113

23. A. Savigni and F. Tisato. Real-Time Programming-in-the-Large. In *Proceedings of ISORC 2000 The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 352 – 359, Newport Beach, California, USA, March 15 – 17 2000. 113
24. M. Shaw and D. Garlan. *Software Achitecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. 104
25. K. J. Sullivan, I. J. Kalet, and D. Notkin. Evaluating the Mediator Method: Prism as a Case Study. *IEEE Transactor on Software Engineering*, 22(8):563–579, August 1996. 112
26. S. E. I. C. M. University. How Do You Define Software Architecture? `http://www.sei.cmu.edu/architecture/definitions.html`. 102
27. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1999. 113

Using Model Checking to Detect Deadlocks in Distributed Object Systems*

Nima Kaveh

Dept. of Computer Science, University College London
London WC1E 6BT, UK
n.kaveh@cs.ucl.ac.uk

Abstract. We demonstrate how the use of synchronization primitives and threading policies in distributed object middleware can lead to deadlocks. We identify that object middleware only has a few built-in synchronization and threading primitives. We suggest to express them as stereotypes in UML models to allow designers to model synchronization and threading of distributed object systems at appropriate levels of abstraction. We define the semantics of these stereotypes by a mapping to a process algebra. This allows us to use model checking techniques that are available for process algebras to detect the presence or absence of deadlocks. We also discuss how the results of these model checks can be related back to the UML diagrams.

Keywords: Software Architecture, Object Middleware, Model Checking

1 Introduction

An increasing number of applications now use a distributed system architecture. If designed properly, these architectures can be more fault-tolerant due to replicated components, can achieve better response times if user interface components are executed on powerful desktop machines or workstations, and they may achieve cost-effective scalability by using several relatively cheap hosts to execute replicated components rather than one central server or mainframe, which is usually more expensive. The construction of such distributed systems by directly using network operating system primitives, such as TCP or UDP sockets, is rather involved. To reduce this complexity, software engineers use *middleware* [5], which resolves the heterogeneity between distributed hosts, the possibly different programming languages that are being used in the architecture and provides higher level interaction primitives for the communication between distributed system components. There are many different forms of middleware, including transaction monitors, message brokers and distributed object middleware, which encompasses middleware specifications such as the Object Management Group's Common Object Request Broker Architecture (CORBA), Microsoft's Component Object Model (COM) or Java's Remote Method Invocation

* This paper is an extended version of [6]

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 116–128, 2001.

(RMI). We note that distributed object middleware offers the richest support to application designers and incorporates primitives for distributed transaction management and asynchronous message passing. From the set of distributed object middleware approaches, we concentrate on CORBA [5] in this paper because it offers the richest set of synchronization and threading primitives.

An example scenario is used throughout this paper, which we will use to demonstrate our ideas and methods. This example involves the remote monitoring of patients which have been retired from the hospital to their homes. Sensor devices are attached to patients and information is communicated between the sensor devices and a central server in a health care centre. Additionally each patient is equipped with an alert device used in case of an emergency. This example is an inherently distributed system. The different approaches have in common that they enable distributed objects to request operation executions from each other across machine boundaries. We refer to this primitive as an *object request*. For this example, we will use object requests to pass diagnostic information about patients that are gathered by sensor devices to a centralized database where the diagnostic data are evaluated, and if necessary alarms are generated.

Distributed objects that reside on different hosts are executed in parallel with each other. In our example, this means that several different patient monitor hosts gather patient data at the same time. To handle the situation where several of them send data concurrently to a server, distributed object middleware supports different threading policies, which determine the way in which the middleware deals with concurrent object requests. A *single-threaded* policy will queue concurrent requests and execute them in a sequential manner, whereas a *multi-threaded* policy can deal with multiple requests concurrently. A common method of implementing multi-threaded policies is to define a thread pool, from which free threads are picked to process incoming requests and requests are queued if the pool is exhausted.

Object requests need to be synchronized, because client and server objects may execute in parallel. Object middleware support different synchronization primitives, which determine how client and server objects synchronize during requests. *Synchronous* requests block the client object until the server object processes the request and returns the results of the requested operation. This is the default synchronization primitive not only in CORBA, but also in RMI and COM. *Deferred synchronous* requests unblock the client as soon as it has made the request. The client achieves completion of the invocation as well as the collection of any return values by polling the server object. With a *oneway* request there is no value returned by the server object. The client regains control as soon as the middleware received the request and does not know whether the server executed the requested operation or not. *Asynchronous requests* return control to the client as soon an invocation is made. After the invocation the client object is free to do other tasks or request further operations. The result of the method invocation is returned in a call back from the server to the client. We note that CORBA supports all these primitives directly. In [5], its shown

how the CORBA primitives can be implemented using multiple client threads in Java/RMI and Microsoft's COM.

The main contributions of this paper are firstly an identification of an important class of liveness problems in distributed object systems. We use the example scenario to demonstrate how particular combinations of synchronization primitives and threading policies in CORBA can lead to deadlocks. Secondly, we exploit the fact that object middleware only has a few built-in synchronization and threading primitives and express these as stereotypes in dynamic UML models. Thirdly, we define the semantics of these stereotypes by mapping stereotyped UML models to a process algebra. Finally, we show how model checking techniques available for these process algebra notations are able to detect the possibility of deadlocks and how their results can be related back to the UML models.

Application developers need to verify their design specifications for absence of any deadlock situations. We aim to develop a CASE tool that takes in such design specifications, from which we generate a process algebra specification which is then analysed for deadlock. This approach has the advantage of detecting deadlocks in the design stage of development compared to the traditional way of attempting it during the later testing phase. Early indications of potential deadlock situations will make the process of design and implementation modifications more efficient.

In Section 2, we will define UML stereotypes to express both the threading policies and the synchronization primitives of distributed object middleware. In Section 3, we explain informally how a deadlock occurs in the running example. We then define the semantics of our threading and synchronization stereotypes using FSP, a process algebra representation [9] in Section 4. We explain in Section 5 how we use this semantics definition to generate an FSP process model from a UML Sequence Diagram. Section 6 discusses how compositional reachability analysis can be used to check for presence or absence of deadlocks in our Sequence diagrams. Section 7 concludes our paper by summarizing the main results and indicating future directions of this research.

2 Modelling Distributed Object Interactions

Rather than proposing to use new or complex notations and tools we have chosen the Unified Modelling Language to model object interactions and their class structures. UML is widely accepted and used in industry and allows us to enrich its notation to accomodate the extra semantics required for model checking. In this section we will look at modelling the described example, with an aim of using it for deadlock detection later in the paper.

Stereotypes provide designers with the means of augmenting basic UML models to include the semantic information required to model the synchronization behaviour in an application design. We have separated the stereotypes into two main groups. The first group deals with the synchronization primitives used by a client to request the services of a server object. The second group involves

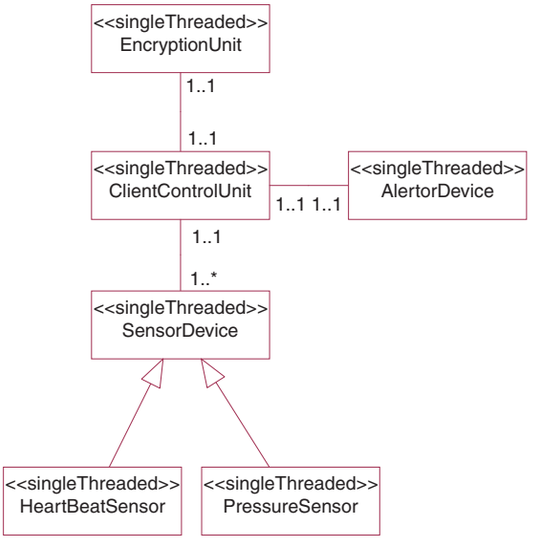


Fig. 1. UML class diagram of the Client

threading policies used on the server-side. These policies determine how server objects deal with multiple concurrent requests.

The ⟨⟨`Synchronous`⟩⟩ stereotype represents the synchronous request primitive request, whilst the ⟨⟨`DeferredSynchronous`⟩⟩ stereotype is used to indicate a deferred-synchronous request being made on a server object. ⟨⟨`Asynchronous`⟩⟩ is used to indicate an asynchronous client request and a ⟨⟨`OneWay`⟩⟩ stereotype represents a oneway request. Similarly on the server-side, we have defined the ⟨⟨`singleThreaded`⟩⟩ stereotype to indicate that a particular server object uses a single threaded policy to deal with incoming service requests and the ⟨⟨`multiThreaded`⟩⟩ stereotype shows that the server object handles multiple service requests by using multi-threading techniques.

The class diagrams in Figures 1 and 2 show the main object types involved in gathering patient data and communicating it to a health care centre database. Note that the design diagrams mainly address distributed communication. Issues such as the user interface need to be looked at separately and are of no concern here.

`SensorDevice` is an abstract type for all types of medical sensor devices attached to patients. `HeartBeatSensor` and `PressureSensor` are two concrete object types inheriting from the `SensorDevice` class. The `AlertorDevice` represents the device that a patient activates to get medical attention. The `ClientControlUnit` is used by sensor devices to send updates and alert messages to the health care centre server. `EncryptionUnit` ensures a secure communication channel as well as providing a non-repudiation service, which will be used when charging patients for services.

Fig. 2. UML class diagram of the Server

All patient data are stored in a central database at the health care centre. All incoming data from patients are logged and appropriate updates are made to the central database. `ServerControlUnit` is the main co-ordinator class on the server side. This type is responsible for communicating information between the patients and the health care centre, and its capable of servicing several patients simultaneously. In order to obtain this parallelism, it uses a multi-threading technique. The `DBThreadPool` represents a fixed-size collection of `DBDriverThread` objects, which it manages. For each time that the `ServerControlUnit` receives a message from a patient it asks the `DBThreadPool` object to provide a free `DBDriverThread`, to which it delegates the task of processing the message. The delegated `DBDriverThread` object makes any required amendments to the central database and sends back any messages through the `ServerControlUnit`. Upon completion of a task the `DBDriverThread` object reports back to the `DBThreadPool` objects and is deemed free once again. In the case of having no free `DBDriverThread` objects the `ServerControlUnit` will store all new incoming messages in the `AccessQueue` object. As the `DBDriverThread` objects start becoming free again, messages can be dequeued from the `AccessQueue` object and serviced, in a first-in-first-out order. In the event of all `DBDriverThread` object being occupied and the `AccessQueue` being at full capacity the `ServerControlUnit` will have to reject any messages which it receives.

Class diagrams are used to describe the structure and hierarchy in a design, thus containing static information. Whilst a sequence diagram represents a given scenario of how instances of classes interact with each other, thus containing dynamic information. The sequence diagram in Figure 3 describes the interactions of a `HeartBeatSensor` object to update information of a health care centre. Due to

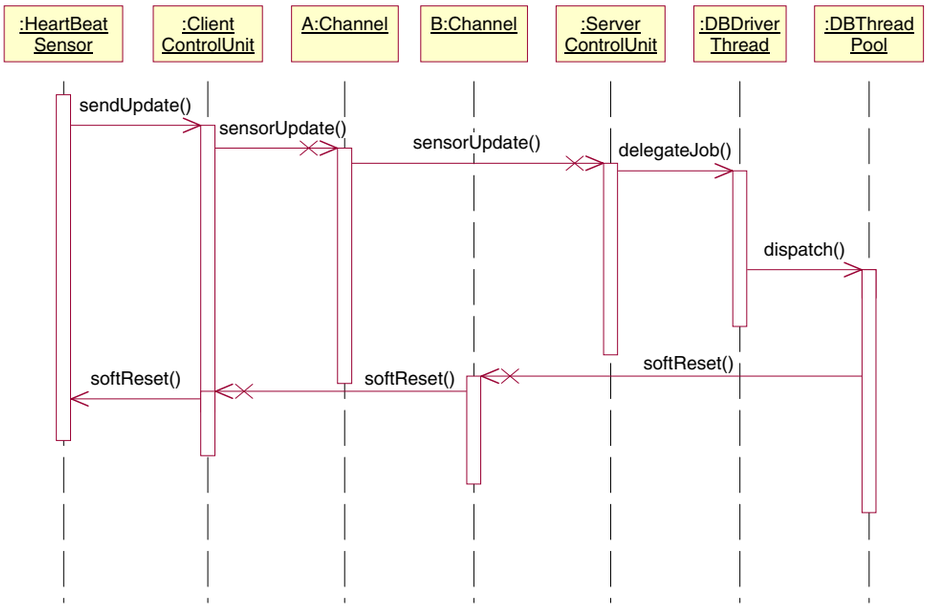


Fig. 3. Sequence diagram of a routine sensor update

lack of space the `EncryptionUnit` has not been included in the sequence diagram, but its exemption does not alter the behaviour.

3 Deadlock in Distributed Object Systems

The source of a deadlock is often a cyclic wait-for relation between communicating components. The complex communication patterns between software components and a need to control the way their shared resources are accessed via methods such as mutual exclusion, gives way to deadlock vulnerabilities. This coupled with the inherent parallelism present in distributed systems, makes deadlock situations a likely and difficult problem to resolve. In fact, the default synchronization primitive and threading policy used in middleware systems, namely the synchronous request and the single threaded policy, are the most likely combination to bring about deadlock.

The sequence diagram shown in Figure 3 actually results in a deadlock. First the `HeartBeatSensor` device sends an update through the `ClientControlUnit`, in the form of a synchronous request. Upon arrival on the server-side a `DBDriverThread` instance is assigned to deal with this message. Control is not returned back to the client until the `DBDriverThread` has finished processing the request. The `DBDriverThread` concludes that a soft reset of the sensor device is required and so it sends the reset command by invoking a synchronous request to the `HeartBeatSensor` object. Thus we have a case where the Client is blocked

waiting for a response from the `DBDriverThread` and vice versa, thus causing a deadlock chain. This deadlock is not easily spotted, because as mentioned before the threading behaviour is determined at a type-level of abstraction and the synchronization behaviour is modelled at an instance level of abstraction. Only the combined knowledge of the two allows designers to consider the liveness issues. In order to demonstrate the idea we kept the interaction small. But the reader should note that such a detection would have been a lot more difficult in a real world industrial case, where the number of objects involved in an interaction may be considerably larger.

Deadlocks are inherently difficult to detect due to the large number of factors affecting the probability of their occurrence. Factors such as varying hardware resources and a wide range of possible user inputs creates a large number of scenarios to run an application. The conventional testing approach creates a test case for each of the likely scenarios in which the application is thought to be used under. The test cases are then executed and their results are compared with predefined expected results. Moreover, distributed applications make the task of testing more difficult by adding new dimensions of complexity. Prime examples of such complexities are hardware heterogeneity, a lack of global memory and physical clock and absence of bounds on transmission delays. We argue that the exponential growth in likely scenarios makes the conventional methods of testing much less effective and scalable. We argue that model checking provides a suitable method of overcoming this complexity dilemma as well tackling it with a rigour and thoroughness that cannot be expected from a human being.

4 Semantics of Stereotypes

A process algebra was chosen to define the semantics of the stereotypes ahead of alternatives such as denotational and axiomatic models, since it provides a more powerful mathematical model of concurrency. Process algebra operators offer direct support for modelling the inherent parallelism in distributed systems. The syntax allows for hierarchical description of processes, a valuable feature for compositional reasoning, verification and analysis.

Figure 4 further demonstrates the reasoning behind choosing a process algebra for modelling the semantics of stereotypes. We are specifically referring to the FSP [9] process algebra. We would like to generate FSP specification from stereotyped UML models. There are liveness properties which the designer would like to have in these models such as deadlock safety, which are directly supported by the liveness properties experessed in FSP.

The CORBA Notification Service [13] uses an architectural element called an Event Channel which allows messages to be transferred between suppliers and consumers of events. This service offers added capabilities such as being able to choose a level of Quality of Service and event filtering at the server-end. All client/server interactions in Figure 3 are taking place through such Channel objects. The generated FSP must exactly follow the semantic behaviour of the synchronization primitives and threading policies as outlined in Section 1.

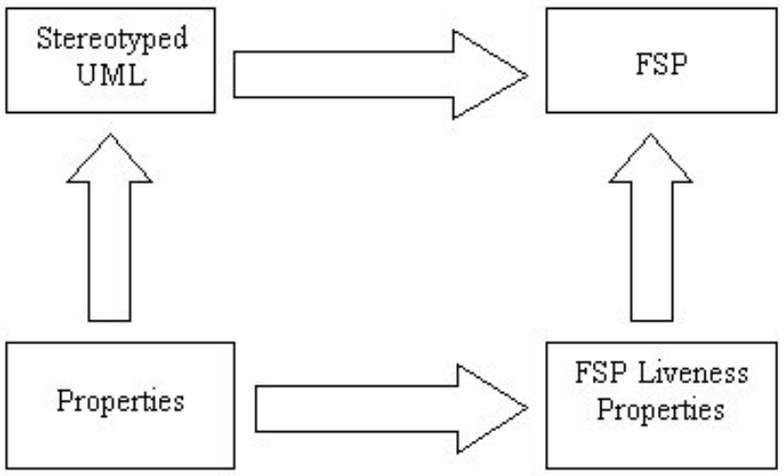


Fig. 4. Relation of FSP to design modelling

4.1 Synchronization Primitives

The process algebra model in Figure 5 defines the ⟨⟨`Synchronous`⟩⟩ stereotype semantic of requests. The `Client` process engages in an action `SendRequest` and does not return until it receives a reply using the `ReceiveReply` action. By using relabelling we have synchronized the Client `SendRequest` with the Channel `ReceiveRequest` and the Client `ReceiveReply` with the Channel `SendReply`. So by making the Channel process engage in a SendReply action only after receiving a reply from the server, we define the ⟨⟨`synchronous`⟩⟩ stereotype.

```
Client=(SendRequest->ReceiveReply->Client).

Channel=(ReceiveRequest->RelayRequest->ReceiveReply->
SendReply->Channel).

||System=(c:Client || a:Channel)
/{c.SendRequest/a.ReceiveRequest,
  c.ReceiveReply/a.SendReply}.
```

Fig. 5. Process Algebra Definition of Synchronous Stereotype

The process algebra model in Figure 6 defines the ⟨⟨`DeferredSynchronous`⟩⟩ stereotype request semantic. The `Client` process invokes a request by engaging in action `push_sendRequest` which is synchronized with the `push_ReceiveReply` action. The `WaitTime` constant defines the number of time units that the `Client`

process continues executing before blocking to receive any results from the server. The `Client` is unblocked when the `Channel` process engages in action `push_sendReply`, which is called only when the server returns a result to the `Channel`.

```
const WaitTime=3
range T = 0..WaitTime

Client = (push_SendRequest->Client[0]),
Client[i:T]= if (i<WaitTime) then (execute->Client[i+1])
     else (push_ReceiveReply -> Client).

Channel=(push_ReceiveRequest->push_SendRequest->push_ReceiveReply->
push_SendReply->Channel).

||System=(c:Client || a:Channel)
/{c.push_SendRequest/a.push_ReceiveRequest,
  c.push_ReceiveReply/a.push_SendReply}.
```

Fig. 6. Process Algebra Definition of the Deferred Synchronous Stereotype

4.2 Threading Policies

The FSP representation in Figure 7 defines the semantics of a server that uses a thread pool policy to handle multiple concurrent requests. The total number of slave threads and queue slots are specified as constants at the beginning. The server-side is composed of four processes, representing the slave thread, thread pool, queue and the server. The processes have synchronization points where they share the same action name. The `Server` process uses two variables to keep track of the current size of the queue and the number of threads currently in use. The server `ReceiveRequest` action indicates the arrival of a client request, if there are any available threads the synchronised action `getFreeThread` is taken which starts the `ThreadPool` process. This further causes the `Thread` process to be initiated using the shared `delegateTask` action. Once the request has been serviced the responsible `Thread` process engages in a `ReceiveReply` action which is shared with the `Channel` process, causing the results to be sent back to the client. If the number of used has reached the maximum the server attempts to add the message to the queue. This `addToQueue` succeeds if there are free queue slots left, otherwise the message is being rejected.

5 Generating FSP Models from UML Diagrams

We have identified a fixed number of synchronization primitives and threading policies used in mainstream object-oriented middleware systems. From these we

```
const PoolSize=16
const QueueSize = 10
range T=0..PoolSize
range Q=0..QueueSize

Channel=(
  push_ReceiveRequest->push_SendRequest->push_ReceiveReply
  ->push_SendReply->Channel).

Thread=(delegateTask->taskExecuted->push_ReceiveReply->Thread).

ThreadPool = ThreadPool[0],
ThreadPool[i:T] = if (i<PoolSize) then
                    (getFreeThread->delegateTask->ThreadPool[i+1]
                   | taskExecuted -> ThreadPool[i-1])
  else (noFreeThreads -> ThreadPool[i]).

Queue = Queue[0],
Queue[j:Q] = if (j<QueueSize) then (
        inspectQueue-> if(j>0) then (dequeueMessage-> Queue[j-1]
                                   | addToQueue[j] -> Queue[j+1])
                       else (addToQueue[j] -> Queue[j+1]))
             else (rejectMessage -> Queue[j]).

Server = Server[0][0],
Server[i:T][j:Q]=(
  push_ReceiveRequest->
  if (i<PoolSize) then (
     getFreeThread-> Server[i+1][j])
  else
     (noFreeThreads->
     if (j<QueueSize) then (addToQueue[j]->Server[i][j+1])
     else (rejectMessage-> Server[i][j]))).

||System=(a:Channel||s:Server||s:ThreadPool||s:Thread||s:Queue)
        /{a.push_SendRequest/s.push_ReceiveRequest,
          a.push_ReceiveReply/s.push_SendReply}.
```

Fig. 7. Semantics Definition of ThreadPool Stereotype

obtain a fixed number of combinations in which they can be formed. We have defined the FSP specification for the semantics of each synchronization primitive and threading policy as demonstrated in section 4. The CASE tool will take as input, UML models enriched with stereotypes and translate them into a FSP specification. In order to achieve this we have to absorb information from two levels of abstraction, namely the type level and the instance level. The threading behaviour are specified in class diagrams with the aid of stereotypes whereas the

synchronization behaviour is modelled in the interaction diagrams. The interaction between clients and server objects will involve a combination of synchronization primitives and threading policies. Thus the corresponding FSP specification will need to be formed from combining specification of a specific synchronization primitive with that of a threading policy. For example the FSP specification for the interaction between the `Channel` object `A` and the `ServerControlUnit` object in Figure 3 is formed by combining the specification in Figures 5 and 7. XMI [14] will be used as the intermediate form, for the transition of input UML models into FSP specification. Research implementations for the UML to XMI transition [12] are well under progress and will benefit us.

6 Detecting Deadlocks by Model Checking

Once we have derived the FSP specification, we can use a model checker to do an exhaustive search for deadlocks. The Labelled Transition System Analysis tool that is available for FSP performs a compositional reachability analysis [2] in order to compute the complete state space of the model. This tool operates by mapping the specification into a Labelled Transition System [11]. A deadlock is detected by looking for states with ingoing but no outgoing transitions.

In the case of a deadlock detected, the LTSA will provide us with a trace of actions leading to the deadlock. From this trace we can single out the starting and ending link in the deadlock chain. In FSP terminology these links are processes and within each process we can find the actual action statement leading to deadlock. Figure 8 shows the output produced by the LTSA when processing the FSP specification of the example we have been discussing through out this paper. This FSP specification is formed by combining the specifications in Figures 5 and 7. As you can see the composition time is fairly quick, however the state space of the output is very large and its rate of growth is well above a linear relationship.

```
State Space:
 4 * 4 * 4 * 385 * 33 * 9 * 21 * 3 = 461039040
Composing
 potential DEADLOCK
States Composed: 10 Transitions: 9 in 10ms
```

Fig. 8. Output of the LTSA for the discussed example

7 Related Work

Process algebra representations, such as CSP [8], CCS [10], the π-calculus [11] or FSP [9] can be used to model the concurrent behaviour of a distributed system.

Tools, such as the Concurrency workbench [3] or the Labelled Transition System Analyzer available for FSP can be used to check these models for violations of liveness or safety properties. The problem with both these formalisms and tools is, however, that they are difficult to use for the practitioner and that they are general purpose tools that do not provide built-in support for the synchronization and activation primitives that current object middleware supports.

Many architecture description languages support the explicit modelling of the synchronization behaviour of connectors by means of which components communicate [15]. Wright [1], for example uses CSP for this purpose. A main contribution of [4] is the observation that connectors are most often implemented using middleware primitives. In our work, we exploit the fact that every middleware only supports a very limited set of connectors, which can be provided to practitioners as stereotypes that are very easy to use.

In [7] CCS is used to define the semantics of CORBA's asynchronous messaging. The paper however, fails to realize that the synchronization behaviour alone is insufficient for model checking as deadlocks can be introduced and resolved by the different threading policies that the object adapters support.

Acknowledgements

I would like to thank Wolfgang Emmerich for his continual feedback and useful suggestions. I am also indebted to Jeff Magee for expressing his views on an earlier version of this paper.

References

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997. 127
2. S.-C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–7, 1999. 126
3. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993. 127
4. E. di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proc. of the* 21^{st} *Int. Conf. on Software Engineering, Los Angeles, California*, pages 13–22. ACM Press, 1999. 127
5. W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Apr. 2000. 116, 117
6. W. Emmerich and N. Kaveh. Model Checking Distributed Objects. In B. Balzer and H. Obbink, editors, *Proc. of the* 4^{th} *International Software Architecture Workshop, Limerick, Ireland*, 2000. To appear. 116
7. M. Gaspari and G. Zavattaro. A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service. In *Proceedings of the* 13^{th} *European Conference on Object-Oriented Programming, ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, pages 495–518. Springer, 1999. 127

8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 126
9. J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999. 118, 122, 126
10. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1995. 126
11. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999. 126
12. C. Nentwich, W. Emmerich, A. Finkelstein, and A. Zisman. Browsing Objects in XML. Research Note RN/99/41, University College London, Dept. of Computer Science, 1999. 126
13. Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.0*. 492 Old Connecticut Path, Framingham, MA 01701, USA, July 1995. 122
14. Object Management Group. *XML Meta Data Interchange (XMI) – Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF)*. 492 Old Connecticut Path, Framingham, MA 01701, USA, Oct. 1998. 126
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. 127

Component Metadata for Software Engineering Tasks

Alessandro Orso[1], Mary Jean Harrold[1], and David Rosenblum[2]

[1] College of Computing, Georgia Institute of Technology
{orso,harrold}@cc.gatech.edu
[2] Information and Computer Science, University of California Irvine
dsr@ics.uci.edu

Abstract. This paper presents a framework that lets a component developer provide a component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks throughout the software engineering lifecycle. The framework is defined in a general way, so that the metadata can be easily extended if new types of data have to be provided. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. We motivate the untapped potential of component metadata by showing the need for metadata in the context of testing and analysis of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application: generating self-checking code and program slicing.

Keywords: Components, component-based systems, distributed components, metadata.

1 Introduction

In recent years, component-based software technologies have been increasingly considered as necessary for creating, testing, and maintaining the vastly more complex software of the future. Components have the potential to lower the development effort, speed up the development process, leverage other developers' efforts, and decrease maintenance costs. Unfortunately, despite their compelling potential, software components have yet to show their full promise as a software engineering solution, and are in fact making some problems more difficult. The presence of externally-developed components within a system introduces new challenges for software-engineering activities. Researchers have reported many problems with the use of software components, including difficulty in locating

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 129–144, 2001.

the code responsible for given program behaviors [4], hidden dependences among components [4,5], hidden interfaces that raise security concerns[12,17], reduced testability [18], and difficulties in program understanding [4]. The use of components in a distributed environment makes all the above problems even more difficult, due to the nature of distributed systems. In fact, distributed systems (1) generally use a middleware, which complicates the interactions among components, and (2) involve components that have a higher inherent complexity (e.g., components in e-commerce applications that embody complex business logic and are not just simple GUI buttons).

Several of the above problems are due to the lack of information about components that are not internally developed. Consider an application developer who wishes to use a particular component by incorporating it into her application, either by using it remotely over a network or by interacting with it through middleware such as CORBA [6]. The application developer typically has only primary interface information supporting the invocation of component functions. In particular, she has no source code, no reliability or safety information, no information related to validation, no information about dependences that could help her evaluate impacts of the change, and possibly not even full disclosure of interfaces and aspects of component behavior. When the task to be performed is the integration of the component, information about the component interface and its customizable properties can be all that is needed. Other software engineering tasks, however, require additional information to be performed on a component-based system.

In this paper, we present a framework that lets the component developer provide the component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks. The idea of providing additional data together with a component is not new: It is a common feature of many existing component models, albeit a feature that provides relatively limited functionality. In fact, the solutions provided so far by existing component models are tailored to a specific kind of information and lack generality. To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

The framework that we propose is defined in a general way, so that the metadata can be easily extended to support new types of data. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. Because the size and complexity of common component-based software applications are constantly growing, there is an actual need for automated tools to develop, integrate, analyze, and test such applications. Several aspects of the framework that we propose can be easily automated through tools. Due to the way the metadata are defined, tools can be implemented that support both the developer

who has to associate some metadata with his component and the user who wants to retrieve the metadata for a component she is integrating into her system.

We show the need for metadata in the context of analysis and testing of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application. The first task is in the context of self-checking code. In this case, the metadata needed to accomplish the task consist of pre-conditions and post-conditions for the different functions provided by the component, and invariants for the component itself. This information is used to implement a checking mechanism for calls to the component. The second task is related to slicing. In this case, the metadata that the developer needs to perform the analysis consist of summary information for the component's functions. Summary information is used to improve the precision of the slices involving one or more calls to the component, which would otherwise be computed making worst-case assumptions about the behavior of the functions invoked.

The rest of the paper is organized as follows. Section 2 provides some background on components and component-based applications. Section 3 presents the motivating example. Section 4 introduces the metadata framework and shows two possible uses of metadata. Section 5 illustrates a possible implementation of the framework for metadata. Finally, Section 6 draws some conclusions and sketches future research directions.

2 Background

This section provides a definition of the terms "component" and "component-based system," introduces the main technologies supporting component-based programming, and illustrates the different roles played by developers and users of components.

2.1 Components

Although there is broad agreement on the meaning of the terms "component" and "component-based" systems, different authors have used different interpretations of these terms. Therefore, we still lack a unique and precise definition of a component. Brown and Wallnau [2], define a component as "a replaceable software unit with a set of contractually-specified interfaces and explicit context dependences only." Lewis [10] defines a component-based system as "a software system composed primarily of components: modules that encapsulate both data and functionality and are configurable through parameters at run-time." Szyperski [16] says, in a more general way, that "components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."

In this paper, we view a *component* as a system or a subsystem developed by one organization and deployed by one or more other organizations, possibly

in different application domains. A component is open (i.e., it can be either extended or combined with other components) and closed (i.e., it can be considered and treated as a stand-alone entity) at the same time. According to this definition, several examples of components can be provided: a class or a set of cooperating classes with a clearly-defined interface; a library of functions in any procedural language; and an application providing an API such that its features can be accessed by external applications. In our view, a *component-based* system consists of three parts: the user application, the components, and the infrastructure (often called middleware) that provides communication channels between the user application and the components. The user application communicates with components through their interfaces. The communication infrastructure maps the interfaces of the user application to the interfaces of the components.

2.2 Component Technologies

Researchers have been investigating the use of components and component-based systems for a number of years. McIlroy first introduced the idea of components as a solution to the software crisis in 1968 [13]. Although the idea of components has been around for some time, only in the last few years has component technology become mature enough to be effectively used. Today, several component models, component frameworks, middleware, design tools, and composition tools are available, which allow for successful exploitation of the component technology, and support true component-based development to build real-world applications.

The most widespread standards available today for component models are the CORBA Component Model [6], COM+ and ActiveX [3], and Enterprise JavaBeans [7]. The CORBA Component Model, developed by the Object Management Group, is a server-side standard that lets developers build applications out of components written in different languages, running on different platforms, and in a distributed environment. COM+, OLE, and ActiveX, developed by Microsoft, provide a binary standard that can be used to define distributed components in terms of the interface they provide. The Enterprise JavaBeans technology, created by Sun Microsystems, is a server-side component architecture that enables rapid development of versatile, reusable, and portable applications, whose business logic is implemented by JavaBeans components [1]. Although the example used in this paper is written in Java and uses JavaBeans components, the approach that we propose is not constrained by any specific component model and can be applied to any of the above three standards.

2.3 Separation of Concerns

The issues that arise in the context of component-based systems can be viewed from two perspectives: the component developer perspective and the component user (application developer)[1] perspective. These two actors have different knowl-

[1] Throughout the remainder of the paper, we use "component user" and "application developer" interchangeably.

edge, understanding, and visibility of the component. The component developer knows about the implementation details, and sees the component as a white box. The component user, who integrates one or more components to build a complete application, is typically unaware of the component internals and treats it as a black box. Consequently, developers and users of a component have different needs and expectations, and are concerned with different problems.

The component developer implements a component that could be used in several, possibly unpredictable, contexts. Therefore, he has to provide enough information to make the component usable as widely as possible. In particular, the following information could be either needed or required by a generic user of a component:

Information to evaluate the component: for example, information on static and dynamic metrics computed on the components, such as cyclomatic complexity and coverage level achieved during testing.

Information to deploy the component: for example, additional information on the interface of the component, such as pre-conditions, post-conditions, and invariants.

Information to test and debug the component: for example, a finite state machine representation of the component, regression test suites together with coverage data, and information about dependences between inputs and outputs.

Information to analyze the component: for example, summary data-flow information, control-flow graph representations of part of the component, and control-dependence information.

Information on how to customize or extend the component: for example, a list of the properties of the component, a set of constraints on their values, and the methods to be used to modify them.

Most of the above information could be computed if the component source code were available. Unfortunately, this is seldom the case when a component is provided by a third party. Typically, the component developer does not want to disclose too many details about his component. The source code is an example of a kind of information that the component developer does not want to provide. Other possible examples are the number of defects found in the previous releases of the component or the algorithmic details of the component functionality. Metadata lets the component developer provide only the information he wants to provide, so that the component user can accomplish the task(s) that she wants to perform without having knowledges about the component that are supposed to be private.

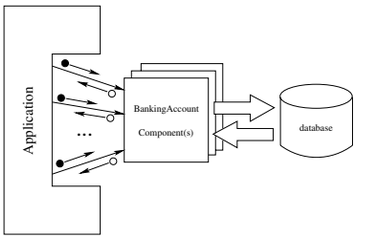
To exploit the presence of metadata, the component user needs a way of knowing what kind of additional information is packaged with a given component and a means of querying for a specific piece of information. The type of information required may vary depending on the specific needs of the component user. She may need to verify that a given component satisfies reliability or safety requirements for the application, to know the impact of the substitution of a component with a newer version of the same component, or to trace a given

execution for security purposes. The need for different information in different contexts calls for a generic way of providing and retrieving such information.

Whereas it is obvious that a component user may require the above information, it is less obvious why a component developer would wish to put effort into computing and providing it. From the component developer's point of view, however, the ability to provide this kind of information may make the difference in determining whether the component is or can be selected by a component user who is developing an application, and thus, whether the component is viable as a product. Moreover, in some cases, provision of answers may even be required by standards organizations — for instance, where safety critical software is concerned. In such cases, the motivation for the component developer may be as compelling as the motivation for the component user.

3 Motivating Example

In this section, we introduce the example that will be used in the rest of the paper to motivate the need for metadata and to show a possible use of this kind of information.



(a)

```
public class BankingAccount {
   ...
   BankingAccount(String code) {...}
   ...
   public void open()
             throws CantOpenException,
             InvalidPINException {...}
   public float getBalance() {...}
   public float withdraw(float amount) {...}
   public float deposit(float amount) {...}
   public float moveFunds(BankingAccount dest,
                          float amount) {...}
};
```

(b)

Fig. 1. (a) High-level view of the application. (b) Interface of the `BankingAccount` component

The example consists of part of a distributed application for remote banking. The application uses one or more instances of an externally-developed component to access a remote database containing the account-related information. Figure 1(a) provides a high-level view of the application, to show the interaction between the user code and the component(s). Figure 1(b) shows the subset of the `BankingAccount` component interface used by the application. We assume the common situation in which the component user is provided with the interface of the component together with some kind of user documentation.

Figure 2 shows a fragment of the application code. The code is part of a method whose semantics is to move a given amount of money from a checking account to a savings account. The first two parameters of the method are two strings containing the codes of the checking account and of the savings account,

```
...
public boolean checkingToSavings(String cAccountCode,
                                 String sAccountCode,
                                 float amount) {
1.  BankingAccount checking(cAccountCode);
2.  BankingAccount saving(sAccountCode);
3.  float balance, total;
    ...
4.  checking.open();
5.  saving.open();
    ...
6.  balance = checking.moveFunds(saving, amount);
    ...
7.  total = balance + additionalFunds;
    ...
}
...
```

Fig. 2. Fragment of the application code

respectively. The third parameter is a number representing the amount of the funds to be moved. Note that, for the sake of the presentation, we have simplified the example to make it smaller, self contained, and more understandable.

4 Metadata

When integrating an externally-developed component into a system, we may need to perform a set of tasks including, among possible others, the gathering of third-party certification information about the component, analyses and testing of the system, and assessment of some quality of the resulting application. These tasks require more than the mere binary code together with some high level description of the component's features. Unfortunately, the source code for the component is generally unavailable, and so is a formal specification of the component. Moreover, we are not simply interested in having a specific kind of information about the component, as a specification would be, but rather we need a way of providing different kinds of information depending on the context. This is the idea behind the concept of metadata: to define an infrastructure that lets the component developer (respectively, user) add to (respectively, retrieve from) the component the different types of data that are needed in a given context or for a given task. Obviously, metadata can also be produced for internally-developed components, so that all the components that are used to build an application can be handled in an homogeneous way.

This notion of providing metadata with software components is highly related to what electrical engineers do with hardware components: just as a resistor is not useful without its essential characteristic such as resistance value, tolerance, and packaging, so a software component needs to provide some information about itself to be usable in different context. The more metadata that are available from or about a component, the fewer will be the restrictions on

tasks that can be performed by the component user, such as applicable program analysis techniques, model checking, or simulation. In this sense, the availability of metadata for a component can be perceived as a "quality mark" by an application developer who is selecting the components to deploy in her system.

Metadata range from finite-state-machine models of the component, to QoS[2]-related information, to plain documentation. In fact, any software engineering artifact can be a metadatum for a given component, as long as (1) the component developer is involved in its production, (2) it is packaged with the component in a standard way, and (3) it is processable by automated development tools and environments (including possibly visual presentation to human users).

As stated in the Introduction, the idea of providing additional data — in the form of either metadata or metamethods returning the metadata — together with a component is not new. The properties associated with a JavaBean [1] component are a form of metadata used to customize the component within an application. The *BeanInfo* object associated with a JavaBean component encapsulates additional kinds of metadata about the component, including the component name, a textual description of its functionality, textual descriptions of its properties, and so on. Analogously, the interface *IUnknown* for a DCOM [3] component permits obtaining information (i.e., metadata) about the component interfaces. Additional examples of metadata and metamethods can be found in other component models and in the literature [14,20,8]. Although these solutions to the problem of how to provide additional data about a component are good for the specific issues they address, they lack generality. Metadata are typically used, in existing component models, only to provide generic usage information about a component (e.g., the name of its class, the names of its methods, the types of its methods' parameters) or appearance information about GUI components (e.g., its background and foreground colors, its size, its font if it's a text component). To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

To show a possible situation where metadata are needed, let us assume that the component user that we met in Section 3 had to perform two different software engineering tasks on her application: implementation of a run-time checking mechanism and program slicing. We refer to the system in Figures 1 and 2 to illustrate the two tasks.

4.1 Self-checking Code

Suppose that the component user is concerned with the robustness of the application she is building. One way to make the system robust is to implement a run-time checking mechanism for the application [9,15]. A run-time check mechanism is responsible for (1) checking the inputs of each call prior to the actual invocation of the corresponding method, (2) checking the outputs of each call

[2] Quality of Service

after the execution of the corresponding method, and (3) suitably reacting in case of problems.

It is worth noting that these checks are needed even in the presence of an assertion-based mechanism in the externally-developed component. For example, the violation of an assertion could imply the termination of the program, which is a situation that we want to avoid if we are concerned with the robustness of our application. Moreover, according to the design-by-contract paradigm, a client should be responsible for satisfying the method pre-condition prior to the invocation of such method.

```
public class BankingAccount {
  //@ invariant ( ((balance > 0) || (status == OVERDRAWN)) && \
  //@             ((timeout < LIMIT) || (logged == false)) );

  public void open() throws CantOpenException,
                            InvalidPINException {
    //@ pre (true);
    //@ post (logged == true)
  }

  public float getBalance() {
    //@ pre (logged == true);
    //@ post ( ((return == balance ) && (balance >= 0)) || \
    //@        (return == -1.0) );
  }

  public float withdraw(float amount) {
    //@ pre ( (logged == true) && \
    //@       (amount < balance) );
    //@ post ( (return == balance' ) && \\
    //@        (balance' == balance - amount) );
  }

  public float deposit(float amount) {
    //@ pre (logged == true);
    //@ post ( (return == balance' ) && \\
    //@        (balance' == balance + amount) );
  }

  public float moveFunds(BankingAccount destination, float amount) {
    //@ pre ( (logged == true) && \
    //@       ((amount < 1000.0) || (userType == ADMINISTRATOR)) && \
    //@       (amount < balance) );
    //@ post ( (return == balance' ) && \
    //@        (balance' == balance - amount) );
  }
};
```

Fig. 3. Fragment of the component code

The run-time checks on the inputs and outputs are performed by means of checking code embedded in the application. This code can be automatically generated by a tool starting from a set of pre-conditions, post-conditions, and invariants compliant with a given syntax understood by the tool. As an alternative, the checking code can be written by the application developer starting from the same conditions and invariants. A precise description of the way conditions and invariants can be either automatically used by a tool or manually used by a programmer is beyond the scope of this paper. Also, we do not dis-

cuss the possible ways conditions and invariants can be available, either directly provided by the programmer or automatically derived from some specification. The interested reader can refer to References [9] and [15] for details.

The point here is that, if the application developer wants to implement such a mechanism, she needs pre- and post-conditions for each method that has to be checked, together with invariants. This is generally not a problem for the internally-developed code, but is a major issue in the presence of externally-developed components. The checking code for the calls to the external component cannot be produced unless that external component provides the information that is needed. Referring to the example in Figure 1, what we need is for the `BankingAccount` component to provide metadata consisting of an invariant for the component, together with pre- and post-conditions for each interface method.

Figure 3 provides, as an example, a possible set of metadata for the component `BankingAccount`.[3] The availability of these data to the the component user would let her implement the run-time checks described above also for the calls to the externally-developed component. The task would thus be accomplished without any need for either the source code of the component or any other additional information about it.

4.2 Program Slicing

Program slicing is an analysis technique with many applications to software engineering, such as debugging, program understanding, and testing. Given a program P, a program *slice* for P with respect to a variable v and a program point p is the set of statements of P that might affect the value of v at p. The pair $< p, v >$ is known as a *slicing criterion*. A slice with respect to $< p, v >$ is usually evaluated by analyzing the program, starting from v at p, and computing a transitive closure of the data and control dependences.

To compute the transitive closure of the data and control dependences, we use a slicing algorithm that performs a backward traversal of the program along control-flow paths from the slicing criterion [11]. The algorithm first adds the statement in the slicing criterion to the slice and adds the variable in the slicing criterion to the, initially empty, set of *relevant variables*. As the algorithm visits a statement s in the traversal, it adds s to the slice if s may modify (define) the value of one of relevant variables v. The algorithm also adds those variables that are used to compute the value of v at s to the set of relevant variables. If the algorithm can determine that s definitely changes v, it can remove v from the relevant variables because no other statement that defines v can affect the value of v at this point in the program. The algorithm continues this traversal until the set of relevant variables is empty.

Referring to Figure 2, suppose that the application developer wants to compute a slice for her application with respect to the slicing criterion $<$ `total`$, 7 >$. [4]

[3] For sake of brevity, when the value of a variable V does not change, we do not show the condition "V' = V" and simply use V as the final value instead.

[4] Also assume that the omitted part of the code are irrelevant to the computation of the slide.

By inspecting statement 7, we can see that both `balance` and `additionalFunds` affect the value of `total` at statement 7. Thus, our traversal searches for statements that may modify `balance` or `additionalFunds` along paths containing no intervening definition of those variables. Because statement 6 defines `balance`, we add statement 6 to the slice. We have no information about whether `checking` uses its state or its parameters to compute the return value of `balance`. Thus, we must assume, for safety, that `checking`, `saving`, and `amount` can affect the return value, and include them in the set of relevant variables. Because `balance` is definitely modified at statement 6, we can remove it from the set of relevant variables. At this point, the slice contains statements 6 and 7, and the relevant variables set contains `amount`, `checking`, and `saving`.

When the traversal processes statement 5, it adds it to the slice but it cannot remove `saving` from the set of relevant variables because it cannot determine whether `saving` is definitely modified. Likewise, when the traversal reaches statement 4, it adds it to the slice but does not remove `checking`. Because the set of relevant variables contains both `checking` and `saving`, statements 1 and 2 are added to the slice and `cAccountCode` and `sAccountCode` are added to the set of relevant variables. When the traversal reaches the entry to `checkingToSavings`, traversal must continue along calls to this method searching for definitions of all parameters. The resulting slice contains all statements in method `checkingToSavings`.

There are several sources of imprecision in the slicing results that could be improved if some metadata had been available with the component. When the traversal reached statement 6 — the first call to the component — it had to assume that the state of `checking` and the parameters to `checking` were used in the computation of the return value, `balance`. However, an inspection of the code for `checking.moveFunds` shows that `saving` does not contribute to the computation of `balance`. Suppose that we had metadata, provided by the component developer, that summarized the dependences among the inputs and outputs of the method.[5] We could then use this information to refine the slicing to remove some of the spurious statements.

Consider again the computation of the slice for slicing criterion $<$ `total`, 7 $>$, but with metadata for the component. When the traversal reaches statement 6, it uses the metadata to determine that `saving` does not affect the value of `balance`, and thus does not add `saving` to the set of relevant variables at that point. Because `saving` is not in the set of relevant variables when the traversal reaches statement 5, statement 5 is not added to the slice. Likewise, when the traversal reaches statement 2, statement 2 is not added to the slice. Moreover, because `saving` is not added to the slice, `sAccountCode` is not added to the set of relevant variables. When the traversal is complete, the slice contains only statement 1, 3, 4, 6, and 7 instead of all statements in method `checkingToSavings`. More importantly, when the traversal continues into callers of the method, it will not

[5] We may be able to get this type of information from the interface specifications. However, this kind of information is rarely provided with a component's specifications.

consider definitions of `sAccountCode`, which could result in many additional statements being omitted from the slice. The result is a more precise slice that could significantly improve the utility of the slice for the application developer's task.

5 Implementation of the Metadata Framework

In this section, we show a possible implementation of the metadata framework. The proposed implementation provides a generic way of adding information to, and retrieving information from, a component, and is not related to any specific component model. To implement our framework we need to address two separate issues: (1) what format to use for the metadata, and (2) how to attach metadata to the component, so that the component user can query for the kind of metadata available and retrieve them in a convenient way.

5.1 Format of the Metadata

Choosing a specific format suitable for all the possible kind of metadata is difficult. As we stated above, we do not want to constrain metadata in any way. We want to be able to present every possible kind of data — ranging from a textual specification of a functionality to a binary compressed file containing a dependence graph for the component or some kind of type information — in the form of metadata. Therefore, for each kind of metadata, we want to (1) be able to use the most suitable format, and (2) be consistent, so that the user (or the tool) using a specific kind of metadata knows how to handle it.

This is very similar to what happens in the Internet with electronic mail attachment or file downloaded through a browser. This is why we have decided to rely on the same idea behind MIME (Multi-purpose Internet Mail Extensions) types. We define a metadata type as a tag composed of two parts: a type and a subtype, separated by a slash. Just like the MIME type "application/zip" tells, say, a browser the type of the file downloaded in an unambiguous way, so the metadata type "analysis/data-dependence" could tell a component user (or a tool) the kind of metadata retrieved (and how to handle them). The actual information within the metadata can then be represented in any specific way, as long as we are consistent (i.e., as long as there is a one-to-one relation between the format of the information and the type of the metadatum).

By following this scheme, we can define an open set of types that allows for adding new types and for uniquely identifying the kind of the available data. A metadatum is thus composed of a header, which contains the tag identifying its type and subtype, and of a body containing the actual information. We are currently investigating the use of XML [19] to represent the actual information within a metadatum. By associating a unique DTD (Document Type Definition) to each metadata type, we would be able to provide information about the format of the metadatum body in a standard and general way. We are also investigating a minimum set of types that can be used to perform traditional

software engineering tasks, such as testing, analysis, computation of static and dynamic metrics, and debugging.

5.2 Producing and Consuming Metadata

As for the choice of the metadata format, here also we want to provide a generic solution that does not constrain the kinds of metadata that we can handle. In particular, we want to be as flexible as possible with respect to the way a component developer can add metadata to his component and a component user can retrieve this information. This can be accomplished by providing each component with two additional methods: one to query about the kinds of metadata available, and the other to retrieve a specific kind of metadata. The component developer would thus be in charge of implementing (manually or through a tool) these two additional methods in a suitable way. When the component user wants to perform some task involving one or more externally-developed components, she can then determine what kind of additional data she needs, query the components, and retrieve the appropriate metadata if they are available.

Flexibility can benefit from the fact that metadata do not have to be provided in a specific way, but can be generated on-demand, stored locally, stored remotely, depending on their characteristics (e.g., on their amount, on the complexity involved in their evaluation, on possible dependences from the context that prevent summarizing them). As an example, consider the case of a dynamically-downloaded component, provided together with a huge amount of metadata. In such a situation, it is advisable not to distribute the component and the metadata at the same time. The metadata could be either be stored remotely, for the component to retrieve them when requested to, or be evaluated on demand. With the proposed solution, the component developer can choose the way of providing metadata that is most suitable for the kind of metadata that he is adding to the component. The only constraint is the signature of the methods invoked to query metadata information and to retrieve a specific metadatum, which can be easily standardized.

5.3 Metadata for the Example

Referring to the example of Section 3, here we provide some examples of how the above implementation could be developed in the case of an application built using JavaBeans components.

We assume that the `BankingAccount` component contains a set of metadata, among which are pre-conditions, post-conditions, and invariants, and data-dependence information. We also assume that the methods to query the component about the available metadata and to retrieve a given metadatum follows the following syntax:

```
String[] component-name.getMetadataTags()
Metadata component-name.getMetadata(String tag,String[] params)
```

When the application developer acquires the component, she queries the component about the kind of metadata it can provide by invoking the method

`BankingAccount. getMetadataTags()`. Because this query is just an invocation of a method that returns a list of the tags of the available metadata, this part of the process can be easily automated and performed by a tool (e.g., an extension of the JavaBeans BeanBox). If the tags of the metadata needed for the tasks to be performed (e.g., `analysis/data-dependency` and `selfcheck/contract`) are in the list, then the component user can retrieve them. She can retrieve the invariant for the component by executing

`BankingAccount.getMetadata("selfcheck/contract", params)`,

where `params` is an array of strings containing only the string "invariant," and obtain the post-condition for `getBalance` by executing

`BankingAccount.getMetadata("selfcheck/contract", params)`,

where `params` is an array of strings containing the two strings "post" and "getBalance." Similar examples could be provided for the retrieving of the other information to be used for the tasks.

Our intention here is not to provide all the details of a possible implementation of the framework for a given component model, but rather to give an idea of how the framework could be implemented in different environments, and how most of its use can be automated through suitable tools.

6 Conclusion

In this paper, we have motivated the need for various kinds of metadata about a component that can be exploited by application developers when they use the component in their applications. These metadata can provide information to assist with many software engineering tasks in the context of component-based systems. We focused on testing and analysis of components, and with the help of an example discussed the use of metadata for two tasks that a component user might want to perform on her application: generating self-checking code and program slicing. In the first case, the availability of metadata enabled the task to be performed, whereas in the second case, it improved the accuracy and therefore the usefulness of the task being performed. These are just two examples of the kinds of applications of metadata that we envision for distributed component-based systems.

We have presented a framework that is defined in a general way, so to allow for handling different kinds of metadata in different application domains. The framework is based on (1) the specification of a systematic way of producing and consuming metadata, and (2) the precise definition of format and contents of the different kinds of metadata. This approach will ease the automated generation and use of metadata through tools and enable the use of metadata in different contexts.

Our future work includes the identification and definition of a standard set of metadata for the most common software engineering activities, and an actual implementation of the framework for the JavaBean component model.

Acknowledgments

Gregg Rothermel provided suggestions that helped the writing of the paper. The anonymous reviewers and the workshop discussion supplied helpful comments that improved the presentation. This work was supported in part by NSF under NYI Award CCR-0096321 and ESS Award CCR-9707792 to Ohio State University, by funds from the State of Georgia to Georgia Tech through the Yamacraw Mission, by a grant from Boeing Aerospace Corporation, by the ESPRIT Project TWO (Test & Warning Office - EP n.28940), by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061, and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

1. Javabeans documentation. http://java.sun.com/beans/docs/index.html, October 2000. 132, 136
2. A. W. Brown and K. C. Wallnau. Enginnering of component-based systems. In A. W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1996. 131
3. N. Brown and C. Kindel. *Distributed Component Object Model protocol: DCOM/1.0*. January 1998. 132, 136
4. R. Cherinka, C. M. Overstreet, and J. Ricci. Maintaining a COTS integrated solution — Are traditional st atic anallysis techniques sufficient for this new programming methodology? In *Proceedings of the International Conference on Software Maintenance*, pages 160–169, November 1998. 130
5. J. Cook and J. Dage. Highly reliable ungrading components. In *Proceedings of the 21st International Conference on Software Engine ering*, pages 203–212, May 1999. 130
6. The common object request broker: Architecture and specification, October 2000. 130, 132
7. Enterprise javabeans technology. http://java.sun.com/products/ejb/index.html, October 2000. 132
8. G. C. Hunt. Automatic distributed partitioning of component-based applications. Technical Report TR695, University of Rochester, Computer Science Department, Aug. 1998. Tue, 29 Sep 98 18:13:17 GMT. 136
9. N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990. 136, 138

10. T. Lewis. The next $10,000_2$ years, part II. *IEEE Computer*, pages 78–86, May 1996. 131
11. D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society Press, 1999. 138
12. U. Lindquist and E. Jonsson. A map of security risks associated with using cots. *IEEE Computer*, 31(6):pages 60–66, June 1998. 130
13. D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968. 132
14. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–14. The USENIX Association, 1999. 136
15. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995. 136, 138
16. C. Szyperski. *Component Oriented Programming*. Addison-Wesley, first edition, 1997. 131
17. J. Voas. Maintaining component-based systems. *IEEE Software*, 15(4):22–27, July–August 1998. 130
18. E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September–October 1998. 130
19. Extensible markup language (xml). http://www.w3.org/XML/, October 2000. 140
20. Xotcl - extended object tcl. http://nestroy.wi-inf.uni-essen.de/xotcl/, November 2000. 136

On Using Static Analysis in Distributed System Testing

Jessica Chen

School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
xjchen@cs.uwindsor.ca

Abstract. Reproducible testing is one of the effective ways to perform or repeat a desired test scenario in concurrent systems. This technique can be naturally applied to distributed environments as well. However, when it is applied to distributed systems where remote calls are used as communication facilities, new deadlocks may be introduced when we incorporate the test control mechanism to the execution of the program under test. In this paper, we present our work on using static analysis technique to solve this problem.

Keywords: Distributed Systems, Nondeterminism, Specification-based Testing, Automata, CORBA

1 Introduction

With the advances of modern computers and computer networks, *distributed concurrent* software systems are becoming more and more popular. We are concerned about the quality of these systems just as we are for sequential ones. Testing is still our primary device to determine the correctness of a software system nowadays. However, testing a distributed concurrent system is very often much harder than testing a sequential one, mainly due to the nondeterminism involved. Unlike traditional sequential systems, a given input sequence to the system may have several different execution paths depending on the interactions among different threads and/or different processes possibly running on different machines across the network.

Some research work has been done in the past to deal with the nondeterminism in testing *concurrent systems* [9,4,3,1,11,10,8,6,7]. Generally, there are two basic approaches. One is to execute the program many times with the same input and then examine the result. For example, Itoh et al. [7] have developed a method, together with some related monitoring tools, to measure the coverage with regard to k-tuples of concurrency statements in source code written by C. In another approach, for a given input, some additional information such as partial or total order of some statements in the program will be provided and the program is forced somehow to take certain execution path based on the additional information. Then the external observations are compared with the desired ones. However, forcing a concurrent system to take a particular execution

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 145–162, 2001.

path *manually* during the testing may be fairly difficult and tedious. Some kind of automated control mechanism is needed to help testers to accomplish that. For example, people have developed various techniques for replay control [9,3] and reproducible testing [11,10,2].

In replay control techniques, we record during the first run, the internal choices among the nondeterministic behavior and then re-run it with the same inputs together with some control mechanism to force the program to run with the same choices. Replay control techniques are especially important for regression testing. Reproducible testing differs from replay control mainly in that the controlled execution sequence can either come from the record of the previous run or from other sources, e.g. requirement documents, design documents or even the program code. A possible combination of these two approaches is proposed in [4], as a specification-based methodology for testing concurrent programs.

In general, forcing a concurrent system to take a particular execution path is very useful, especially for debugging. To achieve that effectively, two closely related components are needed.

- Firstly, some kind of test control mechanism needs to be integrated into the system during the test. The test control mechanism will interact with the Program Under Test (PUT) and force the system to take the desired execution path based on the given input and some additional information. The control is done by artificially blocking some threads/processes at certain points and letting other threads/processes proceed. The execution of the PUT is augmented by additional communications between the control mechanism and all the threads in the PUT. Each thread communicates with such control mechanism whenever it has to coordinate with other threads in its own process (via monitors etc.) or in other processes (via remote method calls, etc). This communication can be introduced in several ways, either via automatic code insertion or by altering the execution of the underlying execution environment (such as Java Virtual Machine for java programs). Here we consider the former approach, as discussed in [10,2,3,4]. With the added communication, a controller is able to decide whether a thread should proceed, wait for other threads, or resume from waiting state, based on the overall concurrency constraint for the system and the current status information of other threads.
- Secondly, for a given input, there should be some way to get the additional information to identify the desired execution path, such as when and where to block which threads/processes during the system execution. Usually, the nondeterminism for concurrent systems are caused by synchronization among different threads and processes, hence, as discussed in [9,4,3,1,11,10,7], the additional information is about the order of the threads and processes at the synchronization points. The test control mechanisms very often just block certain threads/processes at these points. In the following, we consider the additional information (called *test constraint*) as happen-before relation among *synchronization events* [3], i.e. accesses to shared objects.

For most of the network applications, a distributed system can be considered as a set of processes executed simultaneously, possibly on different machines. As we know, communications among processes can be realized via Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), stream sockets, (virtual) distributed shared memory, etc. When middleware layers like CORBA, DCOM, Java RMI etc. are used, we can actually consider remote procedural/method calls virtually as local calls. As a consequence, we can uniquely determine the external behavior of the system with given input by the orders of the accesses to shared objects. Thus, lots of discussions on testing concurrent systems in certain sense can be applied to distributed systems.

However, there are some new issues unique to testing *distributed* systems. One of the prominent problems is that the soundness of a test control mechanism may depend on the underlying implementation of process communications in the distributed architecture. As we know, when a process makes a *remote call*[1], an implicit separate thread on the server site may be used to handle it. How these implicit threads are managed depends on various thread models that may be used in the underlying implementations of process communications in the middleware layer. The concurrent threads on the server site for the remote calls usually are limited. With only limited threads available, remote calls will become the contention resources and thus, new deadlocks may be introduced when we execute a system under the control mechanism.

To avoid introducing new deadlocks, the order of remote calls needs to be coordinated together with the order of synchronization events. In this paper, we present our work on using a static analysis technique to provide a test control strategy that controls the order of both synchronization events and remote calls. In doing so, an abstract model of the possible behavior of the system, called *test model*, is constructed in terms of finite automata, according to the given test constraint and the thread model used in the underlying implementation of process communications. This test model is then used in the test control procedure to help the test controller to avoid deadlock. In this way, we can guarantee that the execution will never get into a deadlock state, provided that the given system itself does not contain any deadlock under the given test constraint.

In our experiment on specification-based distributed system testing, a prototype of an automated test control toolkit is developed to help users to realize some particular execution paths desired. A system under testing that we consider consists of a set of processes communicating through CORBA. Each process contains one or more Java threads. The solution introduced in this paper is used in our prototype implementation.

The rest of the paper is organized as follows. In Section 2, we give some detailed explanation on the possible deadlock problem due to the thread models used in the distributed architecture for process communications. This is exemplified by some possible implementations of the process communications in

[1] Here we use *remote calls* for both remote procedure calls and remote method calls (as in CORBA, RMI, etc.).

CORBA middleware. In Section 3, we present our technique on how to statically construct a test model, according to given test constraint and thread model. In Section 4, we will explain how to use the constructed test model to control a test procedure in order to avoid introducing new deadlocks. The last section is dedicated to conclusions and some final remarks.

2 Thread Models and Testing

In this section, we use CORBA middleware as an example to show the deadlock problem introduced by incorporating test control mechanism into the PUT.

As we mentioned above, a remote call may be handled by an implicit separate thread on the server side. Using CORBA middleware, there are a few underlying thread models to realize this. Typically, we have

- *thread-per-request*: Each time a remote call request arrives, a separate thread is created to handle it.
- *pool of threads*: A pool of threads is created to handle incoming remote call requests. The size of the pool is fixed.
- *thread-per-object*: One thread is created per remote object. Each of these threads accepts remote calls on one object only.

Although default thread model is used in the Object Request Broker (ORB), CORBA users are allowed to choose their own preferred model and modify the related part of the implementation.

Now we show a scenario when the test control introduces new deadlocks into the execution of the PUT.

Assume that we have defined a remote object o with which clients can call remote method m on it. The execution of m involves the access to a shared object r. Now, two clients c_1 and c_2 may call $o.m$ in arbitrary order. As we mentioned in the Introduction, a test constraint is expressed as a happen-before relation on synchronization events. In the test constraint, we require that we want to see an execution path in which client c_1 accesses shared object r before client c_2 does. In the real execution, client c_2 may call $o.m$ and reach the point to access r before c_1 does. The traditional way to handle this is to let o communicate with the test controller for permission to access r. In the case that c_2's request to access r arrives first, the test controller will delay its response until it knows that c_1's access to r is completed. Assume that *thread-per-object* model is used. In the above case, while c_2's execution to access r is suspended, waiting for the completion of c_1's access to r, c_1's access to r will never starts because remote object o is currently used for c_2 so c_1's remote call for m is suspended waiting for the completion of c_2's execution of m. So far the system gets into a deadlock state. Such a deadlock state is obviously introduced by our test control mechanism.

The solution used in our work is to control the order of remote calls together with the order of synchronization events given in the test constraint. In other words, the extended PUT should communicate with its controller not only for

synchronization events, but also for any remote calls. In the above example, the test controller should block c_2's remote method call, so that c_1's execution of m can be started first.

With this treatment, upon receipt of a request from a process to make a remote call, the test controller will decide whether the permission should be granted, taking into account both the current execution status and the underlying thread model adopted, in order to avoid leading the execution into a deadlock state. Having difficulty in finding a suitable algorithm for the test control in this regard, we propose a static analysis technique to be incorporated into the dynamic testing. We obtain an abstract *test model* from the given test constraint and the thread model used in the underlying implementation of process communications. The test model is given in terms of finite automata. It is constructed in such a way that it contains only those states from where there always exists at least one path to complete the test procedure. The test controller uses this statically obtained model to make its decision on whether or not to grant permission to a request for remote call or for accessing shared objects.

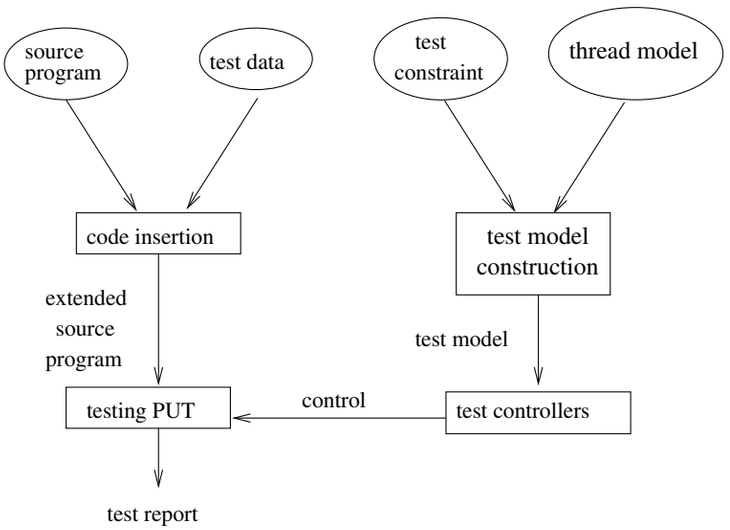


Fig. 1. Test control structure

Figure 1 illustrates part of the structure of our test control method for reproducible testing in distributed systems. A test constraint expresses the order of some synchronization events that we want to observe in a test scenario. According to given test constraint and the underlying thread model of process communications, we construct a test model. The constructed test model is used by test controller to control the execution of the PUT. In order to coordinate

with the test controller, the PUT should be slightly extended, augmenting the communications with the test controller during the execution.

In the following, we give detailed explanation on the constructions of test models. Without loss of generality, we present the rest part of the work only for those PUTs that follow the object oriented programming paradigm.

3 Test Model

We have mentioned before, that the underlying CORBA implementation of process communications can be user defined. Apparently, if for each request it is guaranteed that a separate thread on the server site for remote calls will be available to handle the request, then the implicit threads used on the server site will not be part of the contention resource. As a direct consequence, we do not have the above-mentioned problem if *thread-per-request* model is used. In the following, we consider *pool of threads* model and *thread-per-object* model. If *pool of threads* model is adopted, where the limit of the number of threads in a pool for process p is n, the controller will dynamically decide which (maximum n) remote calls should be allowed at each moment to be executed within p. If *thread-per-object* model is adopted, the controller will decide for each remote object, which remote call should be allowed at each moment to be executed.

3.1 Events and Test Constraints

As we have explained above, apart from synchronization events, a test controller should also control the order of assigning implicit threads for remote calls. When a process makes a remote call, it needs to send to the controller (i) a *remote request event* to get permission from its controller before the call; and (ii) a *remote completion event* to inform its controller once it has finished the execution of the body of the method. We call these two kinds of events *remote call events*.

Note that a synchronization event is in fact a request sent to the test controller to access a shared object. We also call it *synchronization request event*. Similarly as for remote call events, for each synchronization request event, the (extended) PUT has a corresponding *synchronization completion event* to inform the controller of the completion of obtaining the shared object.

In test models, we consider synchronization events (including synchronization request events and synchronization completion events) and remote call events (including remote request events and remote completion events). In general, an event is represented by a 7-tuple $(p, t, tp, o, na, no, ty)$ where

- p is the name of the process this event is from.
- t is the name of the thread this event is from.
- tp is the target process if the event is a remote call.
- o is the name of the object with which we call a method.
- na is the event name, e.g. the name of the remote method being called.
- no is the number of appearances of this e in thread t and process p.

- *ty* is the type of the event. We classify the events into four types:
  - *synReq* for a request to access a shared object;
  - *synCom* for a message of completion of obtaining a shared object;
  - *remReq* for a request to call a remote method;
  - *remCom* for a message of completion of executing a remote method.

As an example, let us consider a centralized arbiter system that resolves requests from users for a shared resource. The server program is in charge of assigning the shared resource to only one client at a time. The server provides remote object o_1 with method m within which, Java synchronized method *request* will be called by object o_2. Each client should call m in order to use the shared resource.

As we know, the Java language and runtime system support thread synchronization through the use of *monitors* originally introduced in [5]. Generally, the critical sections in Java programs are defined as a statement or a method (identified by the *synchronized* keyword), and Java platform uses monitors to synchronize the access to this statement/method on an object: each object with synchronized statement/method is a monitor that allows only one thread at a time to execute a synchronized statement/method of that object. Every object with synchronized statement/method has an associated *waiting queue* of threads. Thus, according to the above implementation of the server program, the first process who accesses method *request* will be allowed to execute the body of the method and use the shared resource. Before it has completed the method, all other calls on *request* will be put on the waiting queue of o_2.

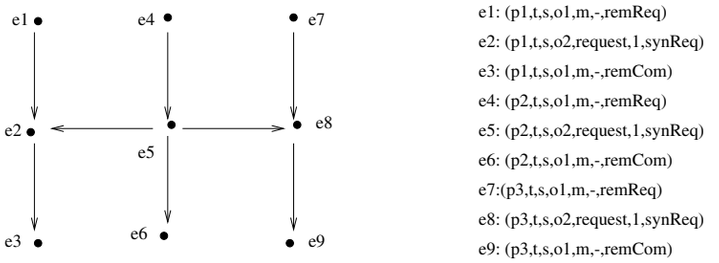


Fig. 2. Test constraint in Arbiter example

For simplicity, we assume that there are three clients with process p_1, p_2 and p_3 respectively. Each process has one thread t. The server process is s. Suppose we want to test a case when p_1, p_2 and p_3 each requires and uses the shared resource once. Figure 2 illustrates a test constraint together with the remote call events. Here the nodes denote events and arrows denote the happen-before relationship on events. We use "-" to denote the piece of information that we are not interested in. For instance, event $(p_1, t, s, o_1, m, -, remReq)$ denotes

a request from thread t in process p_1 for a remote call of method m on object o_1 in process s.

The test constraint is expressed by the happen-before relation $\{e_5 \rightarrow e_2, e_5 \rightarrow e_8\}$. It essentially says that process p_2 should obtain monitor o_2 before processes p_1 and p_3. e_1, e_3, e_4, e_6, e_7 and e_9 are remote call events. We have also added them in this figure with additional information about which synchronization events are within remote method calls. Such additional information is static and can be obtained in various ways.

3.2 Test Model with Pool of Threads

To construct a test model, we assume that we have, as Figure 2 illustrated, a test constraint together with remote call events and the relationships between each synchronization event and its corresponding remote call events.

Formally, we assume that we are given

- E: a set of events, including both synchronization events and remote call events.
- $R \subseteq E \times E$: a binary relation, including both the test constraint and the relationships between each synchronization event and its corresponding remote call events. $(e_1, e_2) \in R$ has the intuitive meaning that e_1 should happen before e_2.
- $P = Proc \rightarrow N$: a function from a set $Proc$ of process names to set N of natural numbers. $P(p) = n$ has the intuitive meaning that the maximum number of threads in the thread pool of process p is n.

With given E, R and P, we construct the test model in two steps: first we define a finite automaton with all possible paths according to E, R and P. Then we provide an operation on this automaton to prune those unwanted states that will anyhow lead to deadlock.

Note that the test controller works together with the concurrency control mechanism within the PUT, thus the (extended) PUT can send out the synchronization completion event either upon the occupation of the shared object or after it has released the shared object. Our model is generally defined in the sense that synchronization completion events are enabled at any time after the corresponding synchronization request events are executed. Based on this, we do not require that E contain any synchronization completion event. Those remote completion events, on the other hand, have to be given in E because during the construction of a test model, we need to statically determine the place where a remote call is completed and thus the related thread on the server site is released.

Now given E, R and P, we construct automaton $\langle S, E \cup E_C, \rightarrow, s_0, s_f \rangle$ where

– $E_C = \{(p, t, -, o, na, no, synCom) \mid (p, t, -, o, na, no, synReq) \in E\}$
contains all synchronization completion events for those synchronization request events in E.

- $S \subseteq E \times E_C \times \mathcal{T}$ is a set of states. Here $\mathcal{T}$ denotes the set of functions from set *Proc* of process names to set N of natural numbers. For $T \in \mathcal{T}$, $T(p) = n$ has the intuitive meaning that in current state, there are n threads in the thread pool of p available for remote method calls. In a state $\langle E_1, E_2, T\rangle$, E_1 contains all events in E that have been completed, E_2 contains all synchronization completion events that have not yet happened but their corresponding synchronization request events have been executed. T maintains the number of threads available in the thread pools for each process.
- $\rightarrow \subseteq S \times (E \cup E_C) \times S$ is a ternary relation. We will give more details below on how it is defined. $(s_1, e, s_2) \in \rightarrow$, also denoted as $s_1 \xrightarrow{e} s_2$, means that from state s_1 we can accept event e ending at state s_2.
- $s_0 = \langle \emptyset, \emptyset, P\rangle$ is the initial state.
- $s_f = \langle E, \emptyset, P\rangle$ is the final state.

The ternary relation $\_ \xrightarrow{\_} \_$ is defined as the least relation satisfying the following *structural rules*. All the structural rules have schema:

$$\frac{\text{ANTECEDENT}}{\text{CONSEQUENT}}$$

which is interpreted logically as:

$$\forall(\text{ANTECEDENT} \longrightarrow \text{CONSEQUENT})$$

where $\forall(\ldots)$ stands for the universal closure of all free variables occurring in $(\ldots)$[2].

In the following, $e, e' \in E$, $E_1 \subseteq E$, $E_2 \subseteq E_C$, $p \in Proc$ is a process name, and $T : Proc \rightarrow N$ is a function. We use $T[p]$ to denote T's return value on p, and $T[p/x]$ the new status of thread pools obtained from T by substituting $T[p]$ by x. Recall that an event is a 7-tuple $(p, t, pt, o, na, no, ty)$. We use $e(i)$ to denote the ith element of this tuple. So $e(3)$ is the remote process this event communicates with, $e(5)$ is the event name, and e(7) is the event type. Furthermore, let e be a synchronization request event, we use e_C to denote the corresponding synchronization completion event.

Rule (A1)

$$\frac{e \notin E_1 \wedge (\forall e'.\ e' \rightarrow e \in R \Rightarrow e' \in E_1) \wedge e(7) = synReq}{\langle E_1, E_2, T\rangle \xrightarrow{e} \langle E_1, E_2 \cup \{e_C\}, T\rangle}$$

This rule essentially says that for any non-executed synchronization request event e, if all events that should happen before it have already been executed, then this event is enabled. In the ending state, the corresponding synchronization completion event is added into E_2.

[2] Observe that, typically, ANTECEDENT and CONSEQUENT share free variables.

Rule (A2)

$$\frac{e_C \in E_2}{\langle E_1, E_2, T\rangle \xrightarrow{e_C} \langle E_1 \cup \{e\}, E_2 - \{e_C\}, T\rangle}$$

This rule says once a synchronized request event has happened, the corresponding synchronization completion event is enabled. After the execution of this synchronization completion event, we remove it from E_2 so that it becomes disabled.

Rule (B1)

$$\frac{e \notin E_1 \wedge (\forall e'.\ e' \rightarrow e \in R \Rightarrow e' \in E_1) \wedge e(7) = remReq \wedge e(3) = p \wedge T[p] > 0}{\langle E_1, E_2, T\rangle \xrightarrow{e} \langle E_1 \cup \{e\}, E_2, T[p/T[p]-1]\rangle}$$

This rule essentially says that for any non-executed event to request for remote method call, if all events that should happen before it have already been executed, and there is at least one thread available in the thread pool of the called process (p), then this event is enabled. In the ending state, the number of available threads in the thread pool of p is reduced by one.

Rule (B2)

$$\frac{e \notin E_1 \wedge (\forall e'.\ e' \rightarrow e \in R \Rightarrow e' \in E_1) \wedge e(7) = remCom \wedge e(3) = p}{\langle E_1, E_2, T\rangle \xrightarrow{e} \langle E_1 \cup \{e\}, E_2, T[p/T[p]+1]\rangle}$$

This rule expresses that for any non-executed event to inform the controller of the completion of a remote method call, if all events that should happen before it have already been executed, then it is enabled. In the ending state, the number of available threads in the thread pool of p is augmented by one.

Consider the Arbiter example (see Figure 2). We have

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$$

$$R = \{(e_1, e_2), (e_2, e_3), (e_4, e_5), (e_5, e_6), (e_7, e_8), (e_8, e_9), (e_5, e_2), (e_5, e_8)\}$$

Suppose that the maximum number of threads in the thread pool of the server process s is 2. I.e.

$$P(s) = 2$$

Figure 3(a) shows a partial view of the automaton constructed according to the above rules (The continuous dots denote an omitted part). Since $e_5 \rightarrow e_2$, $e_5 \rightarrow e_8$, when p_1 (or p_3) requests the controller to access synchronized method *request* before p_2 does so, it will be suspended by the controller. However, before sending request to the controller for permission to access synchronized method *request*, p_1 (or p_3) has already occupied a thread in the thread pool of s. When

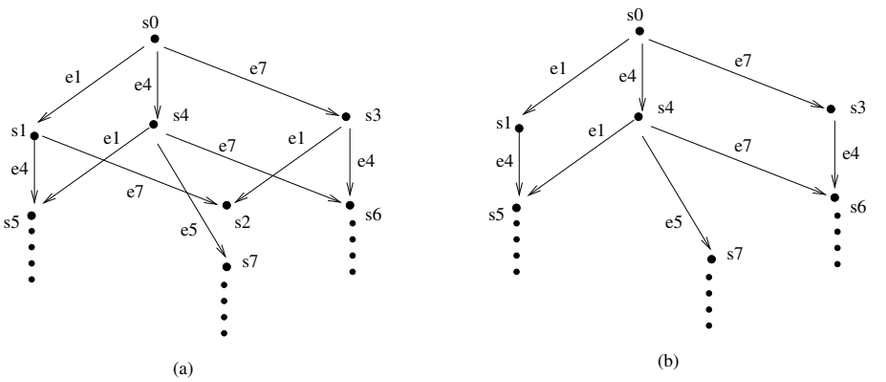


Fig. 3. Constructed automaton in Arbiter example

it is suspended by the controller, it will not release this thread. Now suppose both p_1 and p_3 have occupied the threads in the thread pool of s before p_2 does so. p_1 and p_3 will be blocked by the controller to access method *request* because p_2 has not yet done so. Then we are in a deadlock state because p_2 cannot obtain a thread in the thread pool of s in order to execute remote method m and thus complete the execution of *request*. This is shown in Figure 3(a) where s_2 is a deadlock state.

3.3 Automata with Thread-Per-Object

We have previously explained how to construct an automaton to be used in testing when *pool of threads* model is used. If *thread-per-object* model is used, we can similarly construct the automaton. In this case, we assume that we are given a set E, a binary relation R as we defined before, together with a set Obj of remote object names.

The configuration of the states in the constructed automaton has form $\langle E_1, E_2, O\rangle$, where $O: Obj \to \{true, false\}$ is a boolean function. $O(obj) = true$ has the intuitive meaning that the thread for remote method call on object obj is available. The initial and final state of the automaton is $s_0 = \langle \emptyset, \emptyset, O_0\rangle$ and $s_f = \langle E, \emptyset, O_0\rangle$ respectively. Here O_0 maps all remote object names to *true*.

The structural rules to define the ternary relation $\_ \xrightarrow{\_} \_$ can be similarly defined. Rules (A1) and (A2) are the same. We list the other two rules below.

Rule (B1')

$$\frac{e \notin E_1 \wedge (\forall e'.\ e' \to e \in R \Rightarrow e' \in E_1) \wedge e(7) = remReq \wedge e(4) = o \wedge O[o] = true}{\langle E_1, E_2, O\rangle \xrightarrow{e} \langle E_1 \cup \{e\}, E_2, O[o/false]\rangle}$$

Rule (B2')

$$\frac{e \notin E_1 \wedge (\forall e'.\ e' \rightarrow e \in R \Rightarrow e' \in E_1) \wedge e(7) = remCom \wedge e(4) = o}{\langle E_1, E_2, O\rangle \xrightarrow{e} \langle E_1 \cup \{e\}, E_2, O[o/true]\rangle}$$

3.4 Deadlock-Free Automata as Test Models

As we mentioned, in Figure 3(a), s_2 is a deadlock state. In general, given a finite automaton with final state s_f, a state s is deadlocked if s is not the final state, and there is no transition starting from s, i.e.

$$s \neq s_f \wedge \nexists e, s'.\ s.t.\ s \xrightarrow{e} s'$$

A test model to be used by test controller can be obtained from the above constructed automaton by pruning out all deadlock states.

Table 1. Algorithm 1: pruning deadlock states in automata

```
while ∃s ∈ S s.t. s ≠ s_f ∧ ∄e, s'. s.t. s --e--> s' do
  for all s s.t. s ≠ s_f ∧ ∄e, s'. s.t. s --e--> s' do
    remove s from S;
    for all e, s' s.t. s' --e--> s do
      remove s' --e--> s from →
```

Table 1 shows an algorithm to prune deadlock states in an automaton constructed from a test constraint. The outer *for loop* removes all deadlock state in the current automaton. It is worth mentioning that during the pruning procedure, some previously non-deadlock states may later on become deadlock ones. So the pruning procedure should continue until there is no more deadlock state. This is achieved by the *while loop*.

Let A be a given automaton. Let m and n be the number of states and number of transitions in A respectively. In the best case, i.e. when A is deadlock-free, the execution of Algorithm 1 (cf. Table 1) on A will visit each state and each transition of A once and only once to check the while-condition. So the execution takes $O(m)$ time. In the worst case, the while-loop will be executed *n-1* times. Each time the outer for-loop removes exactly one deadlock state and exactly one non-deadlock state becomes deadlock one. As the checking of while-loop and for-loop takes $O(m)$ time, the execution of the algorithm takes $O(mn)$ time.

The derived automaton, also called *test model*, is deadlock-free in the sense that along any existing path we will eventually arrive at the final state. Figure 3(b) shows the test model obtained from the automaton in Figure 3(a).

3.5 Advanced Pruning Procedure for Test Model

The above-mentioned automaton is generated according to only the synchronization events and remote call events, so the dimension of the automaton is relatively small compared with those models constructed to simulate the behavior of the whole system. Even though, we can still develop some techniques to considerably reduce its size. Here we present one of the possibilities.

Consider that test constraints are in reality, much smaller compared with test models. We should try to use only the test constraints whenever possible. So, if at certain moment during a test procedure, we can decide that the test constraint is sufficient for all the future decisions of the test controller without leading to any deadlock state, then we can ignore the test model immediately. We can make such a decision in a state s in an automaton, if all paths starting from s will end at the final state. We call such kind of states *successful states*. Successful states can be equivalently defined as below:

- the final state is a successful state;
- if $(\exists e, s' \text{ s.t. } s \xrightarrow{e} s')$ and $(\forall e, s'.\ s \xrightarrow{e} s'$ implies s' is a successful state), then s is a successful state.

When we meet a successful state in the automaton, the test controller can switch to the test constraint itself to make decisions. This indicates that we can actually remove all successful states in an automaton. Such pruning procedure to remove successful states can be followed by the pruning procedure to remove all deadlock states. Table 2 shows an algorithm on how to do it.

Table 2. Algorithm 2: obtaining a simplified deadlock-free automaton

mark the final state as a successful state;
while $(\exists s\ s.t.\ (\exists e, s'\ s.t.\ s \xrightarrow{e} s') \wedge (\forall e, s'.\ s \xrightarrow{e} s'$ implies s' is a successful state)) do
  for all $s\ s.t.\ (\exists e, s'\ s.t.\ s \xrightarrow{e} s') \wedge (\forall e, s'.\ s \xrightarrow{e} s'$ implies s' is a successful state)
    mark s as a successful state;
    remove $s \xrightarrow{e} s'$ from $\rightarrow$ for all $e, s'\ s.t.\ s \xrightarrow{e} s'$
  for any $s'\ s.t.\ \nexists e, s.\ s \xrightarrow{e} s'$
  remove s' from S
continue with the algorithm in Table 1

Let A be a given automaton. Let m and n be the number of states and number of transitions in A respectively. Similarly as in Algorithm 1, the pruning procedure to remove successful states takes $O(m)$ time in the best case, i.e. when the final state is the only successful state, and takes $O(mn)$ time in the worst case, i.e. when each while-loop identifies exactly one successful state.

Note that the procedure to remove successful states *must precede* the procedure to remove deadlock states. If we first remove all deadlock states, then all remaining states will become successful states.

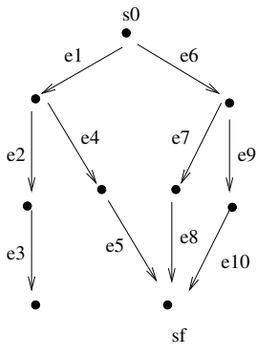


Fig. 4. An automaton with deadlock states

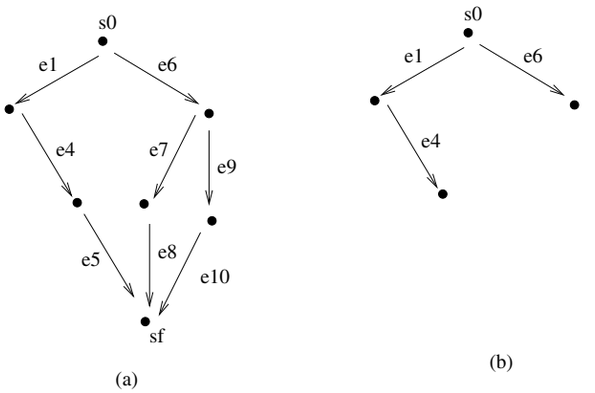


Fig. 5. Deadlock-free automaton and simplified deadlock-free automaton

Assume that the automaton given in Figure 4 is constructed from a given test constraint. Figure 5(a) shows the automaton obtained from it by removing all deadlock states, while Figure 5(b) shows the automaton obtained from it by first removing all successful states then removing all deadlock states.

4 Test Control with Test Model

Once all deadlock states are pruned away, we can use the derived test model to control the test procedure.

Given a (deadlock-free) test model, let pointer A be used to indicate the current state in it. At the beginning A points to the initial state s_0. An event e is *enabled* in current state s, i.e. the state pointed by A, if there exists s' such that $s \xrightarrow{e} s'$. We use waiting set W to maintain all suspended requests, i.e. those requests (including both remote call requests and synchronization requests) that

are disabled in current state. Initially W is empty. Now with a test model, a test controller works in this way:

- If a message arrives and the related event is not in $E \cup E_C$, then if it is a request, then grant the permission; otherwise just ignore it;
- If a message arrives and the related event e is in $E \cup E_C$, then check whether e is enabled in current state s.
  - If there exists s' such that $s \xrightarrow{e} s'$ then move pointer A to s' and grant the permission if e is a request;
  - If there is no s' such that $s \xrightarrow{e} s'$ then put e into the waiting set W.
- Whenever we have moved pointer A, check out in W all those that are enabled, move pointer A correspondingly and grant the permissions.

We consider only those PUTs that contain no infinite loops. With this assumption, since a test constraint contains only finite events, in the execution of a program, all the events in the test constraint will be triggered. Thus, the test procedure eventually terminates. If the automaton is obtained by removing only deadlock states, the test procedure will terminate when A points to the final state, the waiting set W is empty, and there is no more message from processes in PUT.

If the automaton is obtained by removing both successful states and deadlock states, then the test controller should also maintain a set R of all executed synchronization events in the test constraint. When pointer A reaches a non-final state which has no out-going transition, the test controller should continue the execution in such a way:

- For any remote call event, any synchronization completion event whose corresponding synchronization request event is not in the test constraint, and for any synchronization request event not in the test constraint, if it is a request, then grant the permission; otherwise just ignore it;
- For any synchronization request event in the test constraint, make its control decision based on the test constraint and set R of all synchronization events already executed.
  - If the event is enabled, grant the permission;
  - If the event is not yet enabled, suspend it in the waiting set W.
- For any synchronization completion event whose corresponding synchronization request event e is in the test constraint, add e to R. Then check out in W all those events that are now enabled, and grant the permissions.

In such a case, the test procedure terminates when the waiting set W is empty, R contains all events in the test constraint, and there is no more message from processes in PUT.

5 Conclusions and Final Remarks

Many methods and techniques on concurrent system testing can also be applied to distributed system testing. However, there are also some issues unique to the

latter. We have shown that depending on the thread models possibly used in the underlying implementation of process communications in the distributed architecture, deadlocks may be introduced by incorporating test control mechanism. As a solution, some remote calls should be controlled to keep the soundness of the test control mechanism. We have given our control strategy by way of static analysis using finite automata.

As we explained in the Introduction, the problem we addressed in this paper exists in the setting where implicit separate threads are needed on the server site to handle client's requests. This includes Remote Procedure Call, Remote Method Invocation, etc. The problem will not be raised when message passing (e.g. stream socket) or hand-shaking synchronization (as in Ada) is used as communication facilities, because no separate threads are used on the server's site.

Although our solution is given for PUTs that follow object oriented programming paradigm, similar discussions are straightforward for PUTs that follow other programming paradigms.

As final remarks, we would like to mention the following points:

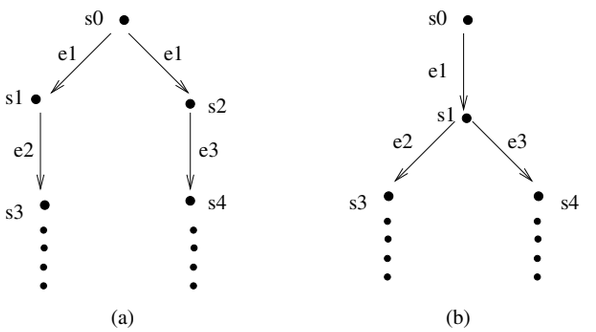


Fig. 6. Determinstic and nondeterministic test models

1. The problem we addressed here may appear only when we have control over the order of two events that are within remote calls. If in the given test constraint, there is no happen-before relationship between two events within remote calls, no execution of the remote calls will be blocked by the test controller, and thus no new deadlock will be introduced by incorporating test control mechanism. In the work of [10], it has been discussed reproducible testing for CORBA applications. The authors have used a sequence (total order) of *remote method calls* as input and discussed the control mechanism to force the execution to follow the given sequence. Since they consider only the order among remote method calls, there will be no new deadlock introduced in their context.

2. We have given three structural rules to construct an automaton from a given test constraint. Such an automaton should be *deterministic*. If it is nondeterministic, the test controller will have to make a nondeterministic choice on the next state when it moves the state pointer one step forward. This is error-prone when the two states to be chosen from are not *trace equivalent*. For example, in Figure 6(a), state s_1 and s_2 are not trace equivalent. Suppose in initial state s_0, the test controller received a message of event e_1, then it will move to either state s_1 or state s_2. Now if we are in state s_1 and the next coming event from a process is e_3, or we are in state s_2 and the next coming event is e_2, then the test controller will make a wrong decision. Making the automaton deterministic as Figure 6(b) illustrates, will smooth away this problem. Of course, there are many possible implementations to obtain deterministic automata from given test constraints and structural rules.
3. With our solution, it is guaranteed that no new deadlocks will be introduced into the test procedure. However, the system itself may contain deadlocks, and even if the system is deadlock-free, the given test constraint may be improper, in the sense that some executions according to such test constraint may, by nature, get into deadlock states. These problems cannot be resolved by our presented technique.
4. Static analysis technique has been widely used in analyzing systems behavior. Rather than considering the whole system behavior, here we have concentrated only on the part related to test events. Thus, the size of the generated automaton is relatively small. Also because of the fact that we consider system models of only test events, the static analysis we used here is just an auxiliary means for dynamic testing. The quality of the systems is still assumed to be improved by way of intensive testing rather than static analysis.

Acknowledgements

The author would like to thank the anonymous reviewers for helpful comments on the preliminary version of this paper submitted to EDO 2000. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References

1. P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, Feb. 1995. 145, 146
2. A. Bechini and K. Tai. Design of a toolset for dynamic analysis of concurrent java programs. In *The 6th International Workshop on Program Comprehension*, Ischia, Italy, June 1998. 146
3. R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 66–74, Mar. 1991. 145, 146

4. R. Carver and K. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, June 1998. 145, 146
5. C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. 151
6. M. Hurfin, M. Mizuno, and M. Raynal. Efficient distributed detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24(8), Aug. 1998. 145
7. E. Itoh, Z. Furukawa, and K. Ushijima. A prototype of a concurrent behavior monitoring tool for testing concurrent programs. In *Proc. of Asia-Pacific Software Engineering Conference (APSEC'96)*, pages 345–354, 1996. 145, 146
8. S. Kenkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, Feb. 1995. 145
9. T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, Apr. 1987. 145, 146
10. H. Sohn, D. Kung, and P. Hsia. State-based reproducible testing for CORBA applications. In *Proc. of IEEE Internaltional Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, pages 24–35, LA, USA, May 1999. 145, 146, 160
11. H. Sohn, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Reproducible testing for distributed programs. In *The 4th International Conference on Telecommunication Systems, Modeling and Analysis*, pages 172–179, Nashiville, Tennessee, Mar. 1996. 145, 146

Distributed Communication

Alfonso Fuggetta[1], Rushikesh K. Joshi[2], and Antonio Rito Silva[3]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
fugetta@elet.polimi.it
[2] Dept. of Computer Science and Engineering, Indian Institute of Technology
Powai, Mumbai - 400 076, India.
rkj@cse.iitb.ernet.in
[3] INESC/IST Technical University of Lisbon
Rua Alves Redol n°9, 1000-029 Lisboa, Portugal
rito.silva@inesc.pt

1 Abstractions and Integration Mechanisms

When looking at distributed communication in the context of software development, it is important to focus on two different aspects: the abstractions and the integration mechanisms.

An abstraction for distributed communication is a middleware-neutral and platform-independent model that describes a solution for distributed communication. In this model, distributed communication is perceived as a separated concern that can be dealt in isolation from other concerns as, for instance, replication or concurrency.

Each abstraction is characterized by different properties, for instance expressive power and modularity. Expressive power property determines the ability to model the different variations of distributed communication. Modularity property defines how the description of distributed communication can be isolated from other concerns such as replication and functionality issues. These properties determine its feasibility to be accepted by programmers of distributed applications. So, a distributed communication abstraction should be evaluated in terms of these properties.

An integration mechanism is an implementation tool that integrates code. Integration mechanisms can be classified in terms of their properties, for instance, compile-time integration versus run-time integration. Code generation and reflection are two common examples of integration mechanisms. There has been an intensive research on integration mechanism, for instance, aspect-oriented programming.

An abstraction can have several implementations. The implementation should consider which integration mechanism will be used.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 163–164, 2001.

2 Distributed Proxy

The *Distributed Proxy* pattern is an abstraction that decouples the communication between distributed objects by isolating distribution-specific issues from object functionality.

The expressive power of distributed allows both, transparent and non-transparent distributed communication, and it also permits the use of different distributed communication implementations, CORBA, JAVA or COM+.

3 Filters

Filtering is a low level communication abstraction that specifies an ability to perform intermediate actions. Filtering separates transparent intermediate processing (message control) from object's functionality (message processing). Basic filtering abstractions can be used to build richer transparent communications abstractions such as transparent message routing, transparent message repetition and transparent decoration.

Filtering abstractions may be realized through specific filtering paradigms such as Filter Objects, Composition Filters or CORBA Interceptors.

4 Towards a Software Engineering Discipline

Middleware is constantly changing, either for technical reasons or for commercial reasons. It seems not to be wise to base a software engineering discipline for the development of distributed and concurrent applications on the technology.

Technology captures some abstractions and integrates them using one or several integration mechanisms. Consider transactions for instance. Existing technology offers transactional support. However, ACID is the only transactional model that is supported. But, there are applications, like CSCW applications, that require relaxed transactions. Note that, even if the technology is extended to support some of these relaxed models the full support of these models is a distant goal.

The correct software engineering approach to deal with transactions for distributed and concurrent applications is to perceive them as a composition of several abstractions: synchronization, persistence, etc. That way a particular relaxed transactional model, required for a particular kind of applications, can be described as a specific composition of particular variations of each one of the involved concerns. Afterwards, and only then, the software engineering should look at the existing technology and decide how to use it to implement the required transactional model.

Thus, since abstractions are more permanent than rapidly changing technology, it is important to stress on abstractions rather than on technology in early stages of a software life cycle. Early stress on abstractions would improve the longevity of software models and their implementations, and would also ease technological migration. Focus on abstractions needs to be coupled with a a clear understanding of their mappings to appropriate technology.

Distributed Proxy: A Design Pattern for the Incremental Development of Distributed Applications

António Rito Silva[1], Francisco Assis Rosa[2], Teresa Gonçalves[2], and Miguel Antunes[1]

[1] INESC/IST Technical University of Lisbon
Rua Alves Redol n°9, 1000-029 Lisboa, Portugal
Rito.Silva@inesc.pt,
http://www.esw.inesc.pt
[2] HKS - Hibbitt, Karlsson & Sorensen, Inc
1080 Main Street, Pawtucket, RI 02860, USA

Abstract. Developing a distributed application is hard due to the complexity inherent to distributed communication. Moreover, distributed object communication technology is always changing, todays edge technology will become tomorrows legacy technology. This paper proposes an incremental approach to allow a divide and conquer strategy that copes with these problems. It presents a design pattern for distributed object communication. The proposed solution decouples distributed object communication from object specific functionalities. It further decouples logical communication from physical communication. The solution enforces an incremental development process and encapsulates the underlying distribution mechanisms. The paper uses a stage-based design description which allow design description at a different level of abstraction than code.

1 Introduction

Developing a design solution for distributed object communication is hard due to the complexity inherent to distributed communication. It is necessary to deal with the specificities of the underlying communication mechanisms, protocols and platforms. Moreover, the lack of performance measures at the beginning of the project and the existence of various distributed object communication technologies providing different features, recommend that choosing and introducing the technology should be delayed until performance measures are obtained during tests and simulations.

Herein, we propose an incremental approach for this problem which allows the transparent introduction of distributed object communication. In a first phase the application is enriched with logical distribution, which contains the distribution complexity in a non-distributed environment where debugging, testing and simulation is easier. This first step ignores the particularities of distributed

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 165–181, 2001.

communication technologies. In a second phase, the application is transparently enriched with physical distributed mechanisms. During this second step the distributed communication technology is chosen and the implementation is tuned for the specific application needs. The approach also allows an intermediate step where a quick implementation using a distributed communication technology is done to test the application in a real distributed environment before the final implementation. In this case the chosen distributed communication technology should allow a rapid prototyping.

Usually, object distributed communication involves the definition of proxies which represent remote services in the client space and encapsulate the remote object [1]. This way, remote requests are locally answered by the proxy which is responsible for locating the remote object and for proceeding with invocation, sending arguments and receiving results.

This paper presents a design pattern [2] for distributed object communication that uses the proxy approach. Design patterns describe the structure and behavior of a set of collaborating objects. They have become a popular format for describing design at a higher level of abstraction than code.

The rest of this paper is structured as follows. The next sections presents a design pattern for distributed communication using the format in [2]. Related work is presented and discussed in Sect. 10 and Sect. 11 presents the conclusions.

2 Intent

The *Distributed Proxy* pattern decouples the communication between distributed objects by isolating distribution-specific issues from object functionality. Moreover, distributed communication is further decoupled into logical communication and physical communication parts.

3 Motivation

3.1 Example

An agenda application has several users which manipulate agenda items, either private (appointments) or shared (meetings). A meeting requires the participation of at least two users. When an agenda session starts, it receives an agenda manager reference from which the agenda user information can be accessed. It is simple to design a solution ignoring distribution issues.

The UML [3] class diagram in Fig. 1 shows the functionalities design of the agenda application, where distribution issues are ignored.

Enriching this design with distribution is complex. For example we must consider different address spaces. In terms of our agenda application this means, that method `getUser` in `Agenda Manager` should return to the remote `Agenda Session` a `User` object across the network. Distributed communication implementation is another source of complexity. For instance, the communication between `Agenda Session` and `Agenda Manager` might be implemented using CORBA.

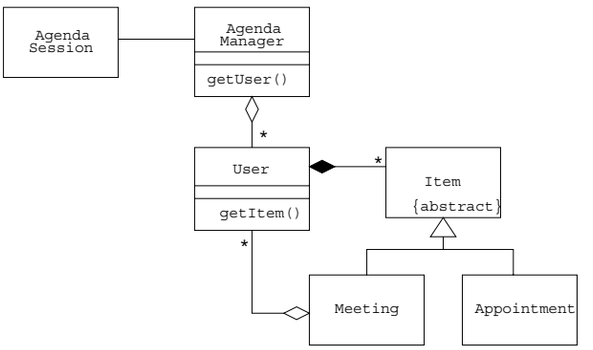


Fig. 1. Agenda functionalities design

3.2 Issues

The design solution for distributed object communication must consider the following issues:

- **Complexity.** The problem and respective solution is complex. Several aspects must be dealt with: the specificities of the distributed communication mechanisms; and the diverse name spaces.
- **Object distribution.** Object references may be transparently passed between distributed nodes.
- **Transparency.** The incorporation of distributed communication should be transparent for functional classes by preserving the interaction model, object-oriented interaction, and confining the number of changes necessary in functionality code.
- **Flexibility.** The resulting applications should be flexible in the incorporation and change of distribution issues. The distributed communication mechanisms should be isolated and it should be possible to provide different implementations.
- **Incremental development.** Distributed communication should be introduced incrementally. Incremental development allows incremental test and debug of the application.

3.3 Solution

Figure 2 shows a layered distributed object communication which constitutes a design solution for the previous problems. In this example the `Agenda Session` object invokes method `getUser` on `Agenda Manager`.

The solution defines three layers: functional, logical, and physical. The functional layer contains the application functionalities and interactions that are normal object-oriented invocations. At the logical layer, proxy objects are introduced between distributed objects to convert object references into distributed

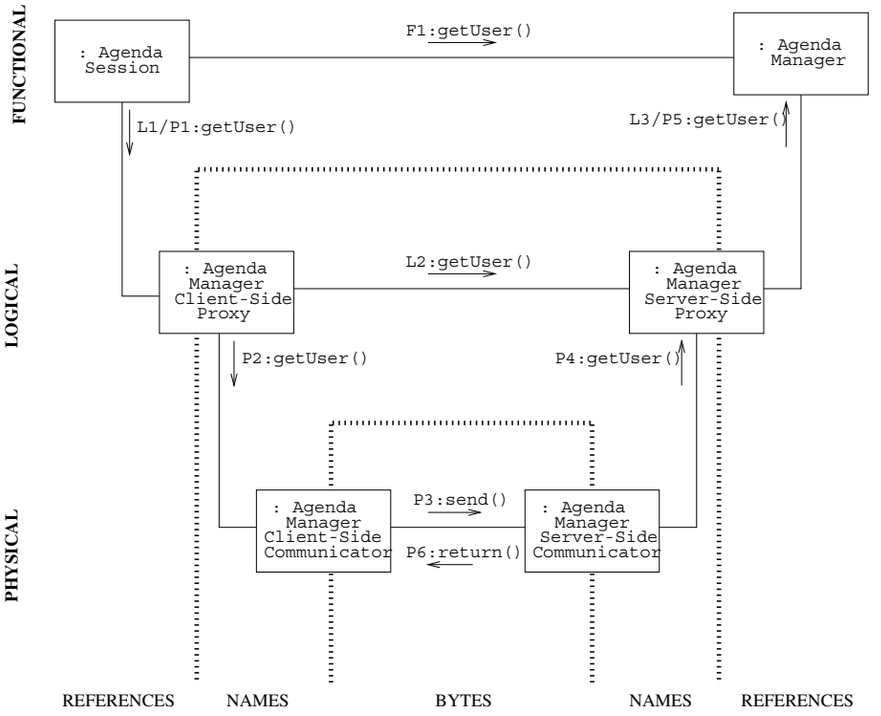


Fig. 2. Layered distributed object communication

names and vice-versa. This layer is responsible for the support of an object-oriented model of invocation, where distributed proxies are dynamically created whenever an object reference from another node is contained in a distributed message. Finally, the physical layer implements the distributed communication using the distributed communication mechanisms.

This solution takes into account the issues previously named:

- **Complexity** is managed by layered separation of problems. Logical layer supports name spaces and physical layer implements the distributed communication mechanisms.
- **Object distribution** is achieved because proxy objects convert names into references and vice versa.
- **Transparency** is achieved since logical and physical layers are decoupled from the functional layer. Functionality code uses transparently the logical layer, `Agenda Manager Client-Side Proxy` and `Agenda Manager` have the same interface.
- **Flexibility** is achieved by means of the physical layer, which contains the distributed communication mechanisms particularities, is decoupled from logical layer.

– **Incremental Development** is achieved since `Agenda Manager Client-Side Proxy` and `Agenda Manager` have the same interface, and the incorporation of the logical layer is done after the functional layer is developed. Moreover, `Agenda Manager Server-Side Proxy` and `Agenda Manager Client-Side Communicator` have the same interface, and the physical layer can be incorporated after the logical layer is developed. This way, the application can be incrementally developed in three steps: functional development, logical development, and physical development. In the same incremental way we define the interaction between the participating components of the pattern. First we define the interaction `F1` at the functional level, as if no distribution was present. Then, when adding the logical layer we define interactions `L1 - L3`. Finally, when implementing the physical layer we establish the interaction chain `P1 - P6`.

4 Applicability

Use the *Distributed Proxy* pattern when:

– *An object-oriented interaction model is required between distributed objects.* Distributed objects are fine-grained entities instead of large-grained servers accessed by clients.
– *Several distributed communication mechanisms may be tested.* Moreover, the communication mechanism can be changed with a limited impact on the rest of the application.
– *Incremental development is required by the development strategy.* Incremental testing and debugging should be enforced.

5 Structure and Participants

The UML class diagram in Fig. 3 illustrates the structure of *Distributed Proxy* pattern. Three layers are considered: functional, logical, and physical. Classes are involved in each layer: `Client Object` and `Server Object` at the functional layer, `Client-Side Proxy`, `Server-Side Proxy` and `Reference Manager` at the logical layer, and `Client-Side Communicator` and `Server-Side Communicator` at the physical layer. Two abstract classes, `Reference Interface` and `Data Interface`, define interfaces which integrate functional and logical layers, and logical and physical layers.

The pattern's main participants are:

– `Client Object`. Requires a service from `Server Object`, it invokes one of its methods.
– `Server Object`. Provides services to `Client Object`.
– `Client-Side Proxy`. It represents the `Server Object` in the client node. It is responsible for argument passing and remote object location. It uses

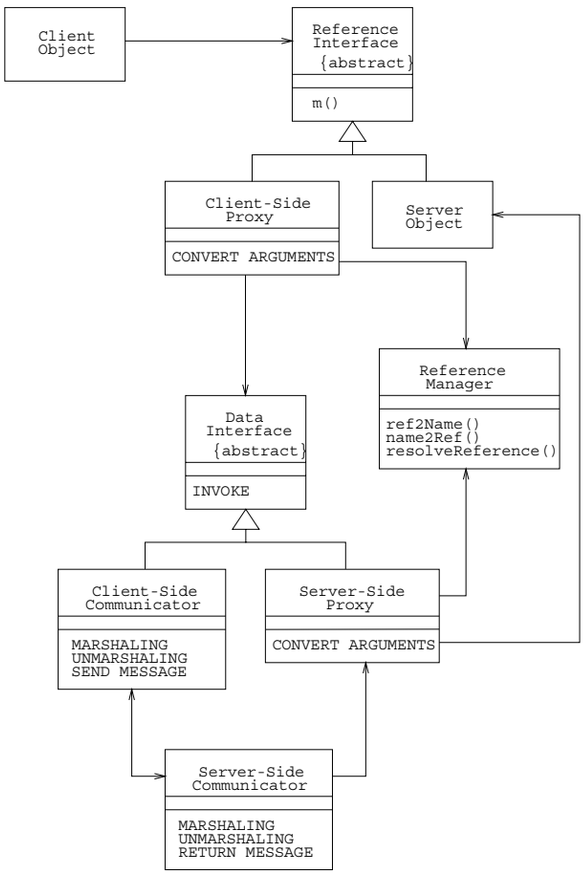


Fig. 3. Distributed proxy pattern structure

the `Reference Manager` to convert sending object references to node independent names (distributed names) and received distributed names to object references. It also uses the `Reference Manager` to obtain a `Data Interface` where it proceeds with the invocation.

- `Server-Side Proxy`. It provides distribution support for the `Server Object` in the server node. It is the entry point for remote requests to the `Server Object`. As `Client-Side Proxy`, it is responsible for supporting argument passing semantics.
- `Reference Manager`. It is responsible for associating object references, local and proxy, with distributed names and vice-versa. It creates new proxies, when necessary. Method `resolveReference` is responsible for returning to the `Client-Side Proxy` a `Server-Side Proxy`, when at the logical layer, or a `Client-Side Communicator`, when at the physical layer.

- `Distributed Name`. It defines an identifier which is valid across nodes. It is an opaque object provided from outside to the `Reference Manager`.
- `Client-Side Communicator` and `Server-Side Communicator`. They are responsible for implementing the distributed physical communication. For each called method it is responsible for `MARSHALING` and `UNMARSHALING` to, respectively, convert arguments to streams of bytes and vice-versa.
- `Reference Interface`. Defines an interface common to `Server Object` and `Client-Side Proxy`, an interface that supports method `m`.
- `Data Interface`. Defines an interface common to `Server-Side Proxy` and `Client-Side Communicator`, an interface that supports `INVOKE`.

6 Collaborations

Three types of collaborations are possible: functional collaboration, which corresponds to the direct invocation of `Client Object` on `Server Object`; logical collaboration, where invocation proceeds through `Client-Side Proxy` and `Server-Side Proxy`; and physical collaboration, where invocation proceeds through the logical and physical layers.

The UML sequence diagram in Fig. 4 shows a physical collaboration which includes the functional and logical collaborations.

After `Client Object` invokes `m` on `Client-Side Proxy`, arguments are converted. According to the specific arguments passing semantics, it converts object references to distributed names or it creates new objects which may include distributed names. To invoke on a `Data Interface`, the `Client-Side Proxy` obtains a `Client-Side Communicator` by using `resolveReference`. In the case of a logical collaboration `resolveReference` returns a `Server-Side Proxy`. The invocation on `Data Interface` is instantiated with the converted arguments.

When invoked, `Client-Side Communicator` marshals its arguments, and sends a message to `Server-Side Communicator` which unmarshals the message and invokes on `Server-Side Proxy`. `Server-Side Proxy` converts received arguments to object references using `Reference Manager` according to the specific argument passing semantics. Finally, `m` is invoked on the `Server Object`.

After invocation on the `Server Object`, three other similar phases are executed to return results to `Client Object`.

In this collaboration two variations occur when transparently sending an object reference: there is no name associated with the object reference in the sending node; and there is no reference associated with the distributed name in the receiving node. In the former situation the object reference corresponds to a local object, and `Reference Manager` is responsible for creating a `Server-Side Proxy` and associating it with a new distributed name. In the latter situation the distributed name corresponds to a remote object, and `Reference Manager` is responsible for creating a `Client-Side Proxy` and associating it with the distributed name. Note that in the physical collaboration proxy creation includes communicator creation.

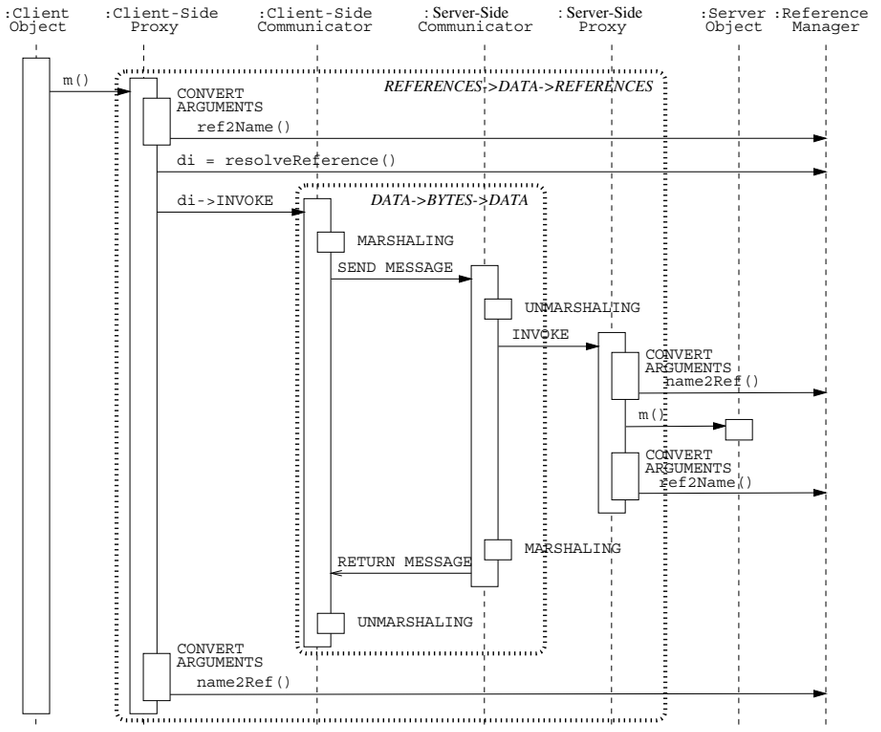


Fig. 4. Distributed proxy pattern collaborations

7 Consequences

The *Distributed Proxy* pattern has the following advantages:

- *Decouples object-functionality from object-distribution.* Distribution is transparent for functionality code and clients of the distributed object are not aware whether the object is distributed or not.
- *Allows an incremental development process.* A non-distributed version of the application can be built first and distribution introduced afterwards. Moreover, it is possible to simulate the distributed communication in a non-distributed environment by implementing communicators which simulate the real communication. Data can be gathered from these simulations to detect possible bottlenecks and decide on the final implementation architecture.
- *Encapsulation of distributed communication mechanisms.* Several implementations of distributed communication can be tested at the physical layer, e.g. sockets and CORBA, without changing the application functionalities. Portability across different platforms is also achieved.
- *Location transparency.* The conversion of distributed names into `Data Interface` objects, done by method `resolveReference`, gives location

transparency of remote objects. That way it is possible to re-configure the application and migrate objects.

This pattern has the following drawback:

– *Overhead in terms of the number of classes and performance.* Four new classes are created or extended for each distributed object depending on whether the implementation uses delegation or inheritance, respectively. The number of classes overhead can be reduced if they are automatically generated. The performance overhead due to indirections can be reduced if the implementation uses inheritance, communicators are subclasses of proxies.

8 Implementation

When implementing the *Distributed Proxy* pattern the following variations should be considered.

8.1 Arguments Passing

Arguments passing can have several semantics: (1) an object argument may be transparently passed between distributed nodes, a proxy is created in the receiving node; (2) an object argument may be copied (deep copy); (3) an object argument is copied but proxies are created for some of its internal references.

At the logical layer argument passing semantics are supported by `CONVERT ARGUMENTS` code blocks. For instance before sending a message to the physical layer, `CONVERT ARGUMENTS` code block should implement the argument passing semantics: (1) to support transparent object passing it interacts with `Reference Manager` to convert references to distributed names; (2) to support deep copy the object is passed to the physical layer where its data will be marshaled and recursively the data of all the objects it refers to; (3) to support partial copy a new object is created that contains the object data and associates distributed names with some of the objects it refers to. After receiving a message from the physical layer and before dispatching it to the functional layer, `CONVERT ARGUMENTS` code block is responsible for converting distributed names to references and data objects to objects with references.

8.2 Transparency

Transparency can be implemented if `Client-Side Proxy` and `Server Object` have the same interface: the interface defined by `Reference Interface`.

However, it can be the case that `Client Object` should be aware of distribution. For instance, the `Client Object` should deal with communication faults. In this situation, transparency can be relaxed by enriching the `Client-Side Proxy` interface according to `Client Object` distribution requirements.

Note that losing transparency does not imply a mix of the pattern layers but only a change of the interfaces between layers. These interfaces must express,

make visible, some distribution aspects of the communication. This means that even in the lack of transparency the pattern keeps the qualities resulting from the decoupling it defines.

8.3 Naming Policies

There are several possibilities when implementing `Reference Manager`: distributed nodes can share a single `Reference Manager` or have its own `Reference Manager`.

As described in [4] there are several naming policies. A distributed name is universal if it is valid, i.e. can be resolved in all the distributed nodes. A distributed name is absolute if it denotes the same object in all distributed nodes. A distributed name is an identifier if remote objects have a single distributed name and not two different remote objects with the same distributed name exist. A distributed name is pure if it does not contain location information. An impure name allows immediate invocation without previous resolution.

A distributed name can be sent to any distributed node if it is universal and absolute. Names with such properties can be supported by a single `Reference Manager` shared by all distributed nodes or by several cooperating `Reference Manager`s which enforce their properties.

Identifier distributed names can be supported if the `Reference Manager` only generates new distributed names and whenever generating a new distributed name verifies that the object does not already have another distributed name.

When performance is a requirement `Reference Manager`s can support impure names at the price of loosing object migration. Resolution of reference associated with a pure name requires that the `Reference Manager` collaborates with a name service that associates pure names with physical addresses where the invocation should occur. Name services are centralized or replicated entities. Impure distributed names avoid the need for a name service because during reference resolution the `Reference Manager` obtains the physical address from the distributed name.

8.4 Reference Resolution

Location of remote objects can either be at proxy creation time or at invocation time. The latter allows the remote objects to change their location.

Location of remote objects variations are supported because `resolveReference` can either be invoked only once or before each invocation. When performance is a requirement and remote objects do not migrate, reference resolution at proxy creation time can be used.

Location of remote objects may dependent on the name policies used. If names are impure then reference resolution at proxy creation time should be used.

8.5 Data Interface Implementation

The definition of the `Data Interface` is crucial since it establishes the physical layer interface which will be used by proxies. Two major variations are possible when defining `Data Interface` and they are related with the goals of using the *Distributed Proxy* pattern:

- **Technology Encapsulation**. Distributed object communication technology, such as DCOM, CORBA and JAVA RMI, is used to develop a distributed application without tangling the functional code with distribution code.
- **Technology Implementation**. The pattern can be also applied to implement a distributed object communication mechanism like a CORBA compliant ORB.

Using the technology encapsulation approach a `Data Interface` is defined for each `Reference Interface`. In the `Data Interface` all object references are replace by distributed names references. Concrete client communicators must be defined for each defined `Data Interface` (see Sect. 8.6). In this way the code that uses the communication technology will be encapsulated in the client and server communicators and each client proxy will use its corresponding `Data Interface` class.

Using the technology implementation approach `Data Interface` will define a fixed interface for the physical layer and all the client proxies will use that interface. That interface should offer methods for marshaling and unmarshaling and to send requests and receive results. In this approach the `Data Interface` will correspond to the `Forward` class of the *Forward-Receiver* pattern [5] such that the replacement of the underlying communication technology does not have repercussions on the proxies code. Different communication mechanism will be implemented by different pairs of client-server communicators. Each client communicator implements the `Data Interface`. In this way the implementation details of the underlying communication is encapsulated in the communicators. Also, by defining a fixed interface to the physical layer automatic generation of communicators is simplified.

It is also possible to use a hybrid solution where for some classes the communication is done using a commercial distributed object communication technology and for other classes the communication is done using a specific communication mechanism implemented by the application programmer itself. This may be useful in applications where some of the application's remote classes have specific requirements that are not covered by the distributed object communication technology being used.

All three implementations provide a clean separation of the physical layer from the logical layer, allowing programmers to change the communication mechanisms without having to change the proxy's code.

8.6 Implementation of Communicators

The implementation of the communicators depends on how `Data Interface` was defined.

Using the technology encapsulation approach the communicator implementation is very simple. Consider a CORBA implementation. CORBA IDL interfaces are defined for each `Data interface` classes. The `Server-Side Communicator` implements the IDL interfaces delegating to the `Server-Side Proxy`. `Client-Side Communicator`s contain a CORBA reference and invoke on the IDL interface. The code that interacts with CORBA is within communicators. Catching exceptions and manipulating the CORBA types (`CORBA::Any`, `CORBA::Octet`) code is encapsulated in the communicators and therefore separated from the functional code.

Using the technology implementation approach, for each type of communication, e.g., ISIS, TCP, UDP, IIOP, a concrete client/server communicator pair is defined. The complexity of communicators implementation will depends on the communication mechanisms. In this case design patterns such *Forward-Receiver* or *Acceptor*, *Connector* and *Reactor* [6] can be used.

9 Sample Code

The code below shows the distributed communication associated with method `getUser` of class `Agenda Manager` which given the user's name, returns a `User` object. The code emphasizes the logical layer of communication and the physical layer using the technology encapsulation approach for CORBA.

A client-side proxy of `Agenda Manager`, `CP_Agenda_Manager`, which returns client-side proxies of `User`, `CP_User`, is defined. A `CP_User` is a subtype of `Reference_Interface_User`. In this case it is not necessary to convert send arguments since the only entity sent, string `name`, is not an object. Before invocation, a `Data_Interface_Agenda_Manager`, where the invocation should proceed, is obtained from the `Reference Manager`. After invocation, `CP_Agenda_Manager` converts the received distributed name into a `Reference_Interface_User` reference. The down-casts are necessary because object `Reference Manager` manipulates objects of type `Data Interface` and `Reference Interface`.

```
Reference_Interface_User* CP_Agenda_Manager::
getUser(const String* name)
{
  // empty convert send arguments

  // get data interface
  Data_Interface_Agenda_Manager*
    diam = static_cast<Data_Interface_Agenda_Manager*>(
              referenceManager_->resolveReference(this));

  // invoke
  Distributed_Name* dn = diam->getUser(name);

  // convert received arguments
  Reference_Interface_User*
    riu = static_cast<Reference_Interface_User*>(
```

```
                referenceManager_->name2Ref(dn));

  // return result
  return riu;
}
```

A server-side proxy of `Agenda Manager`, `SP_Agenda_Manager`, which returns a distributed name of a `User`, is defined. The `SP_Agenda_Manager` is a subtype of `Data_Interface_Agenda_Manager` and `User` is a subtype of `Reference_Interface_User`. In this case it is not necessary to convert received arguments since the only entity received, string `name`, is not an object. After invocation, the `User` object is converted to a distributed name.

```
Distributed_Name* SP_Agenda_Manager::
getUser(const String* name)
{
  // empty convert received arguments

  // invoke
  User* user = agendaManager_->getUser(name);

  // convert send arguments
  Distributed_Name*
    dn = referenceManager_->ref2Name(user);

  // return result
  return dn;
}
```

The following code shows the CORBA IDL definition for the `Agenda Manager` server-side communicator.

```
interface SC_User
{
  // CORBA IDL interface for server-side
  // communicator of the User class
}

interface SC_Agenda_Manager
{
  SC_User getUser(in string name);
};
```

The client-side communicator implementation of the `getUser` method is straightforward. Each client-side communicator inherits from its corresponding `Data Interface` class and from `CorbaCComm` that defines behavior to all the CORBA client-side communicators. In the example below the class `CorbaCC_Agenda_Manager` inherits from `Data_Interface_Agenda_Manager`. It simply narrows its `CORBA::Object_ptr` of the corresponding server communicator to the correct interface type, and then delegates the execution to the CORBA proxy. The method `getDName`, defined in `CorbaCComm`, obtains a distributed name for the received remote reference. In this case the distributed name is just a container for a CORBA reference that also contains some type information that is used by the `Reference Manager` to create the correct proxies.

```
Distributed_Name* CorbaCC_Agenda_Manager::
getUser(const String* name)
{
  SC_Agenda_Manager_ptr access;
  SC_User_ptr user_corba_ref;

   // Get the concrete CORBA reference
   access = SC_Agenda_Manager::_narrow(obj_ptr_);
   // Remote invocation
  user_corba_ref  = access->getUser(name);
  return this->getDName(user_corba_ref);
}
```

Each server-side communicator is the implementation of a CORBA IDL interface. It inherits from the IDL generated class and from `CorbaSComm`. It contains a reference to its server-side proxy where it delegates the method execution. The method `getCorbaRef`, defined in `CorbaSComm`, given a distributed name returns a CORBA reference.

```
SC_User_ptr CorbaSC_AgendaManager::
getUser(const String* name)
{
  Distributed_Name *dn =
    static_cast<Data_Interface_Agenda_Manager*>
      (sp_)->getUser(name);

  return SC_User::_narrow(this->getCorbaRef(dn));
}
```

10 Related Work

In [1] the proxy principle is described: *"In order to use one service, potential clients must first acquire a proxy for this service; the proxy is the only visible interface of the service"*. The presented design applies and extends this principle by relaxing transparency and defining the logical layer. The former allows several argument passing semantics, transparency is preserved from a syntactic point of view because the server class and client-side proxy have the same interface, but different semantics occurs when communication duplicates non-constant objects. The latter allows incremental introduction of distribution with testing, debugging and simulation in a non-distributed environment.

The system presented in [7] also identifies the need of an interaction model which is independent of the transport protocol that is used to transmit messages between endpoints. This decoupling permits performance improvements by taking advantage of the facilities provided by the specific transport protocol. DeLine [8] also defines an approach that allows a component's functional and interactive concerns to be separated. DeLine emphasizes reuse, a component's interaction is captured in a packager, which may either be reused directly or automatically generated from an high-level description.

Distributed communication is addressed by technology like CORBA [9] and JAVA/RMI [10]. In CORBA the implementation of distributed communication is encapsulated by an IDL (Interface Definition Language) and object references are dynamically created and passed across nodes. JAVA RMI (Remote Method

Invocation) defines remote interfaces which can dynamically resolve distributed methods invocations. Both, RMI and most of the existing CORBA implementations apply the `Distributed Proxy` pattern or some of its variations.

The D Framework [11] defines a remote interface language, which allows the specification of several copying semantics. It is based on code generation. Due to the lack of level of detail provided by the interface language it is not possible to do optimizations at the physical layer.

The *Distributed Proxy* patterns is related to the follow design patterns:

- The *Proxy* pattern [2,5] makes the clients of an object communicate with a representative rather than to the object itself. In particular the *Remote Proxy* variation in [5] corresponds to the logical layer of *Distributed Proxy*. However, *Distributed Proxy* allows dynamic creation of new proxies and completely decouples the logical layer from the physical layer.
- The *Broker* pattern [5] defines a distributed architecture with decoupled components that interact by remote service invocations. The *Broker* architecture hides system- and implementation-specific details from the users of components and services. The *Distributed Proxy* pattern separates functional, logical and physical communications but allows programmers to write code in any of the layers.
- The *Client-Dispatcher-Server* pattern [5] supports location transparency by means of a name service. The *Distributed Proxy* pattern also provides location transparency when method `resolveReference` is invoked on `Reference Manager` before each distributed invocation.
- The *Forwarder-Receiver* pattern [5] supports the encapsulation of the distributed communication mechanisms. This pattern can be used to implement the *Distributed Proxy* physical layer.
- The *Serializer* pattern [12] streams objects into data structures and as well creates objects from such data structures. It decouples stream-specific issues, as backends, from the object being streamed. This pattern can be used to implement the `Communicators` `marshaling` and `unmarshaling` methods.
- The *Reactor* pattern [6], *Acceptor* pattern and *Connector* pattern [13] can be used in the implementation of the *Distributed Proxy* physical layer.
- The *Naming* pattern [4] describes an architecture centered on names, naming contexts and name spaces in which object denotation and identification is supported. The *Naming* pattern can be applied to implement class `Reference Manager` and its distributed naming policies.

11 Conclusions

This paper describes a design pattern for distributed communication. It defines three layers of interaction: functionality, logical and physical.

The design patterns allows an incremental development process. A functionality version of the application can be built first and logical and distribution introduced afterwards. The functionality version allows the test and debug of

application functionalities ignoring distribution issues. The logical layer introduces distributed communication ignoring distribution communication mechanisms and preserving the object-oriented communication paradigm. At the logical layer testing and debugging is done in a non-distributed environment. Moreover, simulation of the distribution communication mechanisms can also be done in a non-distributed environment by implementing proxies which simulate the real communication mechanisms. Finally, at the physical layer the application is enriched with distributed communication mechanisms. Note that data gathered from simulations at the logical layer may help to decide on the final implementation.

The *Distributed Proxy* pattern was defined in the context of an approach to the development for distributed applications with separation of concerns (DASCo) initially described in [14]. Distributed communication is a DASCo concern and the presented design pattern define a solution for it. Moreover, it is part of a pattern language for the incremental introduction of partitioning into applications [15] which also includes configuration [16], replication [17] and naming [4].

The *Distributed Proxy* pattern was experimented in the DISGIS project [18]. DISGIS aims at providing effective and efficient development of Geographical Information Systems, where functions and data are increasingly distributed. Due to the large amount of data that is transmitted the *Distributed Proxy* was applied to provide an object-oriented interaction model while allowing performance improvements by adapting the logical and physical layers independently of the communication technology.

References

1. Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *The 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass., 1986. IEEE. 166, 178
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. 166, 179
3. Martin Fowler and Kendall Scott. *UML Destilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997. 166
4. António Rito Silva, Pedro Sousa, and Miguel Antunes. Naming: Design Pattern and Framework. In *IEEE 22nd Annual International Computer Software and Applications Conference*, pages 316–323, Vienna, Austria, August 1998. 174, 179, 180
5. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996. 175, 179
6. Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Jim Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 529–545. Addison-Wesley, 1995. 176, 179
7. Nat Pryce and Steve Crane. Component Interaction in Distributed Systems. In *IEEE Fourth International Conference on Configurable Distributed Systems*, pages 71–78, Annapolis, Maryland, USA, May 1998. 178

8. Robert DeLine. Avoiding packaging mismatch with flexible packaging. In *22th International Conference on Software Engineering*, pages 97–106, Los Angeles, CA, USA, May 1999. 178
9. Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996. 178
10. Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. 178
11. Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, PARC Technical report, February 1997. 179
12. Dirk Riehle, Wolf Siberski, Dirk Baumer, Daniel Megert, and Heinz Zullighoven. Serializer. In Robert Martin, Dirk Riehle, and Frank Buschman, editors, *Pattern Languages of Program Design 3*, chapter 17, pages 293–312. Addison-Wesley, 1997. 179
13. Douglas C. Schmidt. Acceptor and Connector. In Robert Martin, Dirk Riehle, and Frank Buschman, editors, *Pattern Languages of Program Design 3*, chapter 12, pages 191–229. Addison-Wesley, 1997. 179
14. António Rito Silva, Pedro Sousa, and José Alves Marques. Development of Distributed Applications with Separation of Concerns. In *IEEE Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995. 180
15. António Rito Silva, Fiona Hayes, Francisco Mota, Nino Torres, and Pedro Santos. A Pattern Language for the Perception, Design and Implementation of Distributed Application Partitioning, October 1996. Presented at the OOPSLA'96 Workshop on Methodologies for Distributed Objects. 180
16. Francisco Assis Rosa and António Rito Silva. Functionality and Partitioning Configuration: Design Patterns and Framework. In *IEEE Fourth International Conference on Configurable Distributed Systems*, pages 79–89, Annapolis, Maryland, USA, May 1998. 180
17. Teresa Goncalves and António Rito Silva. Passive Replicator: A Design Pattern for Object Replication. In *The* 2^{nd} *European Conference on Pattern Languages of Programming, EuroPLoP '97*, pages 165–178, Kloster Irsee, Germany. Siemens Technical Report 120/SW1/FB, 1997, July 1997. 180
18. DISGIS. Esprit Project 22.084: DIStributed Geographical Information Systems: Models, Methods, Tools and Frameworks, 1996. http://www.gis.dk/disgis/Intro.htm. 180

Modeling with Filter Objects in Distributed Systems

Rushikesh K. Joshi

Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Powai, Mumbai – 400 076, India
rkj@cse.iitb.ernet.in

Abstract. Filtering is emerging as an important programming abstraction in distributed object systems. We discuss the modeling capabilities of a first class filter object model in the context of distributed systems. Filter objects are transparent objects that are dynamically pluggable and provide selective filtering of messages. Filters can be injected into a system to dynamically evolve the system. The method is demonstrated with the help of an example application, a Transparent Distributed Decorator. A notation for representing static (class and object) and dynamic (interobject interactions) models in presence of filtering abilities is also discussed.

1 Introduction

Filter Objects [5] are first class objects which can transparently and selectively filter messages to objects to which they are dynamically plugged. Filtering ability is a consequence of a *filter relationship* between two classes. An IDL-centric design and implementation of filter objects for CORBA based distributed systems is presented in [9]. In this model, filter objects, which themselves are full fledged distributed objects, can be specified and plugged to *filter aware* distributed server objects.

Figure 1 depicts a filter relationship between two classes (between two interfaces in the case of component systems). Class *Dictionary* exports two member functions which are accessed by class *Client*. Class *Cache* is in filter relationship with class *Dictionary*. Filter relationship is shown with a special filter symbol at the filter-client end. In this case, class *Dictionary* is the filter-client and class *Cache* is its corresponding Filter class. It is possible to specify multiple filter classes in filter relationship with a filter-client class.

The upward filter function *Cache::searchCache* in Figure 1 filters message invocation of *Dictionary::searchWord* in upward direction. This filter function has the capability to bounce a result on behalf of the destination intended by the source of the message. The upward filter function in this case bounces a result to the client in the case of a cache miss. The bounce is thus a special form of return. The downward filter function *Cache::updateCache* filters the return result from an invocation of *Dictionary::searchCache* and updates the cache.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 182–187, 2001.

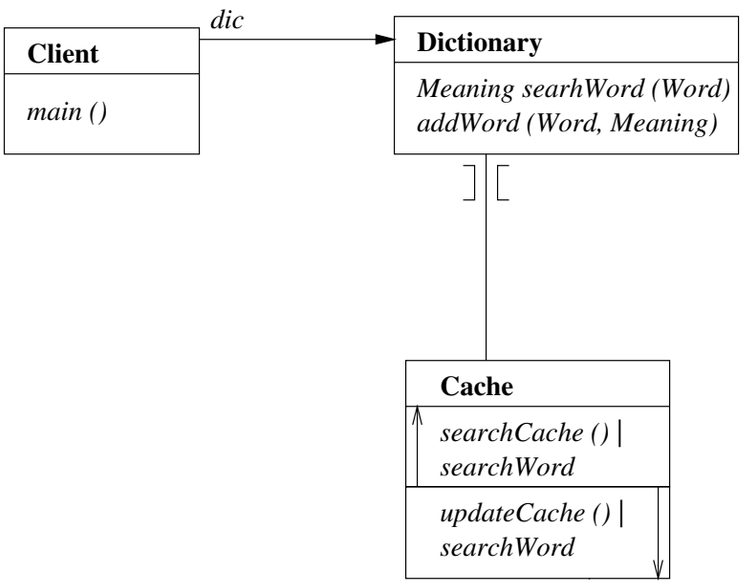


Fig. 1. Interclass Filter Relationship

Filter objects can selectively filter messages sent to their filter-clients. A filter function can be disabled and enabled during runtime. Filter objects are dynamically pluggable. For example, an instance of class *Cache* may be plugged to an instance of class *Dictionary* during runtime and replaced subsequently with another instance of a refinement of class *Cache* providing a better cache policy. Figure 2 models a filter relationship between instances. The filter notation also indicates the transparency of the filter object from the view of the client.

In this paper, we describe a transparent pattern based on filter objects for engineering distributed systems. The pattern is based on the filter configurations for distributed systems described in [4]. Implementations of Design Patterns based on conventional object oriented programming language features have been discussed extensively in literature. Filtering can be viewed as a meta-pattern introduced by Pree [8]. Apart from the hooks, templates and interclass relationships such as part-of, association and dependency, transparent filtering represents a distinct paradigm to engineer object systems.

2 Configurational Properties of Filter Objects

Filter objects transparently filter message sent to their corresponding filter clients. This makes client programs unaware of the filtering actions. Since filter objects are dynamically pluggable, new properties can be specified modularly in the path of messages between client and server objects. This ability is referred to as separation of message control from message processing in [5]. This

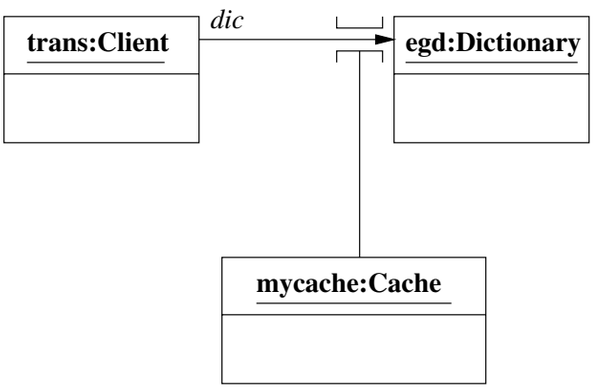


Fig. 2. Filter Relationship between Instances

makes it possible for an architecture to incrementally evolve over time, in some cases without changing the existing source code or even the code in execution. A network of filter objects or individual filter objects may be injected into an executing software at runtime for evolution.

Filter Objects can implement configurations such as logger, replacer, router, value transformer, message transformer and repeater [4]. These configurations may also be implemented with any specific programming paradigm for filtering which provides the required transparency capabilities. The Distributed Transparent Decorator (DTD) that we describe in the next session is based on the *logger* filter configuration which attaches a new responsibility on an existing one without the knowledge of the client.

3 The Distributed Transparent Decorator

The decorator dynamically attaches additional responsibilities to an object [2]. The decorator implementation in [2] requires that the client is given an explicit pointer to the decorator object. Thus, it is not possible to inject a decorator into an existing object oriented system in which clients keep pointed to their intended servers. Filter objects provide an interesting solution to this problem. A filter object can be injected into the path of messages and can attach additional responsibilities in both upward and downward direction. Figure 3 shows the design of the Distributed Transparent Decorator based on interclass filter relationship.

The Decorator class is modeled as a filter class that forms filter relationship with the component class. The client *knows* only of the component while a decorator instance may be dynamically plugged to an instance of the component class. The filter member function *Decorator::addedOperation()* in class *Decorator* filters *Component::operation()* in upward direction. The filter member

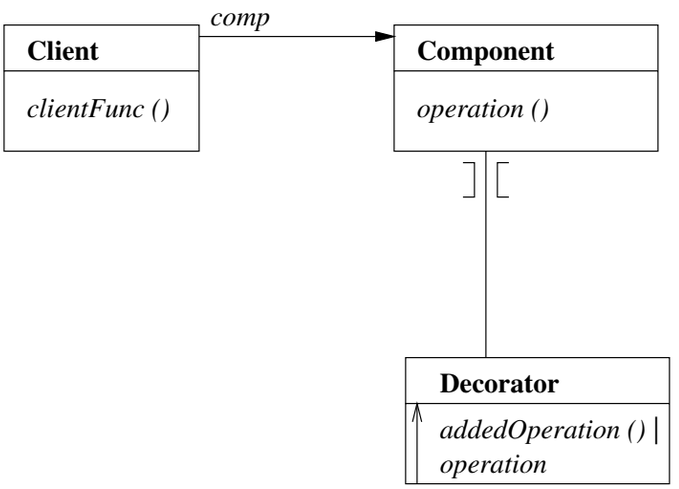


Fig. 3. DTD: Class Diagram

function can perform additional code and *pass on* the invocation to the intended destination.

An example decorator instance diagram is shown in Figure 4. The figure shows a logger as a decorator object that logs incoming requests to a local search engine from a query processor client. It can be noted that since the logger, which is a filter object, is itself an instance of a class, may be related with other classes in the system such as the class *LogFile* in the figure.

The solution to filter object based distributed decorator demonstrates an evolutionary distributed system that attaches new responsibilities in a message path dynamically and transparently, i.e. without the need to change the client or the server code. The dynamic behavior of the decorator implementation is modeled in Figure 5. The figure depicts an intermediate transparent filter object. The client object invokes a method *lse.getPage()*. The invocation in the figure is shown as an apparent invocation indicating the transparency of the filter object. The filter object captures this invocation *on-the-fly* and invokes a direct method on the log file object and subsequently passes the method on to the desired destination. The return result is not filtered in this case. It is also possible to *decorate* the return result with the help of the downward filtering action.

The following pseudo code represents an example decorator implementation. The filter member in the decorator performs its local computation before passing the message on to the intended destination. The filter interface that specifies filter member functions is not exported as a public contract, nor the methods

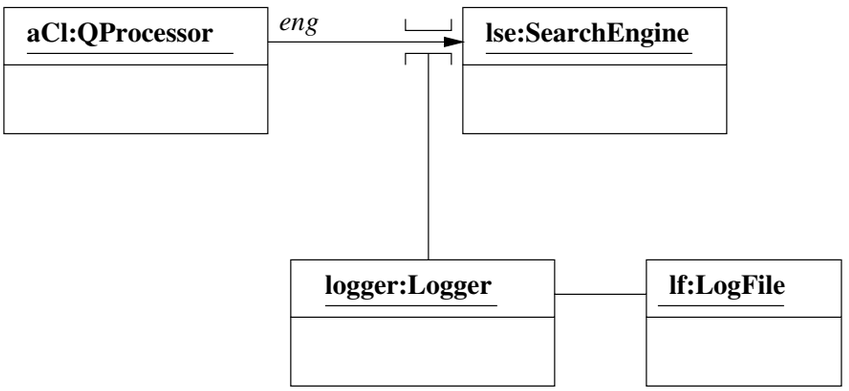


Fig. 4. DTD: Instance Diagram

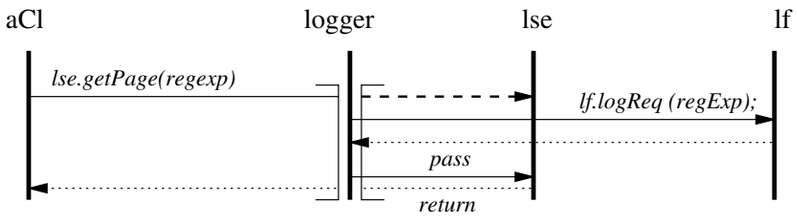


Fig. 5. DTD: Dynamic Model

are available locally for explicit invocation. Methods defined in the filter interface are invoked implicitly as filtering invocations.

```
Logger | SearchEngine {
...
filter interface:
  updateHits () upfilters SearchEngine::getPage (REGEXP rexp) {
     details = extractDetails (rexp);
     lf -> logReq (details);
     pass;
   }
}
```

4 Related Work and Conclusions

Filtering has been in use in software systems in various forms such as shell level filters, packet filters, mail filters etc. However, programming language support for filtering is not in abundance. Aksit et. al. discuss Composition Filters for language Sina [1]. Composition Filters provide for filter specifications of predefined filter types embedded into objects. Recent CORBA implementations

such as Orbix [3] support per-process and per-object filtering mechanisms. COM specification also discusses a filtering mechanism [6]. Context Relationships of Lieberherr et al. [10] is a related mechanism which allows context dependent code to be plugged to an object. Proxy pattern of Gamma et al. [2] provides a place-holder object for its corresponding server object, however the client *knows* the place-holder and must possesses an explicit handle to it.

Our approach to filtering in object oriented systems is based on an interclass filter relationship leading to transparency of filter objects [5]. Filter specifications are separated from their respective filter-clients. The unit of modularity for filters is class. In a distributed system, a filter object is just another distributed object with additional capabilities of filtering. Filtering member functions cannot be explicitly invoked. A filter object may however export a public contract and be a part of the system through conventional relationships. We discussed applicability of filter objects to distributed object systems engineering and demonstrated it through a transparent distributed decorator pattern with the help of a notation modeling filtering abilities.

References

1. M. Aksit, K. Watika, J. Bosch, L. Bergmans, A. Yonezawa.: Abstracting Object Interactions Using Composition Filters, Proceedings of ECOOP'93 Workshop, Springer Verlag, (1993) 152–184. 186
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995. 184, 187
3. IONA Technologies Ltd.: Orbix Advanced Programmer's Guide, 1995. 187
4. R. K. Joshi.: Filter Configurations for Transparent Interactions in Distributed Object Systems, (to appear in Journal of Object Oriented Programming). 183, 184
5. R. K. Joshi, N. Vivekananda, D. Janaki Ram.: Message Filters for Object-Oriented Systems, Software Practice and Experience, **27** (1997) 677–700. 182, 183, 187
6. Microsoft Corporation.: The COM Core Technology Specification, 1998. 187
7. Object Management Group.: CORBA Specifications, http://www.omg.org.
8. Wolfgang Pree.: Design patterns for object-oriented software development, Addison-Wesley, 1995. 183
9. G. S. Reddy, Rushikesh K. Joshi.: Filter Objects for Distributed Object Systems, (to appear in Journal of Object Oriented Programming). 182
10. Linda M. Seiter, K. L. Lieberherr.: Evolution of Object Behavior using Context Relations. IEEE Transaction on Software Engineering. **24** (1998) 79-92. 187

Advanced Transactions

Christoph Liebig[1] and Stefan Tai[2]

[1] Darmstadt University of Technology, Germany
chris@informatik.tu-darmstadt.de
[2] IBM T.J. Watson Research Center
New York, USA
stai@us.ibm.com

1 Introduction

The reliable and correct execution of programs is a key concern in the engineering of distributed systems. Transactions represent one of the mechanisms commonly used to address reliability and correctness. A transaction groups a set of actions (such as object requests) as one single unit-of-work for which reliability and correctness properties hold. The properties of *atomicity* (A) and *durability* (D) define the notion of reliability in the context of transactions: a set of actions is either executed as one atomic unit-of-work, or none of the actions are executed, and the committed state changes caused by the set of actions are guaranteed to be persistent. The properties of *consistency* (C) and *isolation* (I) define the notion of correctness in the context of transactions: a transaction transforms the system from one consistent state into another consistent state only, and any intermediate system states of an ongoing transaction are not visible to any other concurrent transaction. The two papers in this session describe research on transaction processing that each advance a certain kind of transaction model and transactional middleware technology. This session summary provides a short background on transactions first, and then discusses the presented research papers.

2 Background

A variety of transaction models and technologies supporting transaction processing have been proposed over the last years. These models and technologies differ in how they address and support some or all of the ACID properties. We can distinguish four major groups of kinds of transactions and transaction processing technologies:

- *Integrated database transactions*, as promoted by database management systems (DBMS),
- *Distributed object transactions*, as defined by the CORBA Object Transaction Service (OTS),
- *Component container-managed transactions*, as suggested by component technology such as Sun s Enterprise JavaBeans (EJB),

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 188–193, 2001.

– *Message-oriented transactions*, as promoted by message-oriented middleware such as IBM s MQSeries.

In a database context, the transaction model encompasses all of the ACID properties. DBMS use an integrated system architecture where the state of manipulated entities is under full control of the DBMS. This is the basis for integrated concurrency control and recovery. Transaction demarcation is either explicit, spanning several data manipulation statements, or implicit per statement. The transaction manager is typically closed, i.e. integrating external or remote transactional resources is not supported (besides the federation of multiple, possibly distributed, DBMS instances of the same brand). Distributed object computing systems typically address multicomponent application scenarios which are heterogeneous and open by nature. Following the paradigm of encapsulation, objects are characterized by their interface (or services provided) whereas state and persistence of state are considered to be implementation concerns. It is common practice to separate reliability and concurrency control concerns although these may not be considered to be independent matters. Transaction managers in X/Open DTP and CORBA OTS follow this principle and provide transaction context management and 2PC coordination as their primary services. Participating resources (or resource managers) are obliged to follow the OTS protocol and guarantee recoverability or durability with respect to the decision on transaction outcome. Transaction services in distributed object systems are open in that they make no further assumption on the implementation of the participating resources. As long as an object provides the required interfaces and interacts in a 2PC conforming way it may be involved in a distributed transaction. From a transactional client point of view, transactions are demarcated explicitly and transaction services are typically realized as first class objects. It is the application designer s responsibility to draw the transaction boundaries around logical units of work. In order to cope with the multitude of failure modes in distributed heterogeneous environments, atomicity of computations over distributed objects is a useful paradigm on its own. If additionally durability is required through persistence, this must be realized by the application programmer or typically by means of dedicated persistence services. The latter could be realized by traditional resource managers like databases or queues. But applications are not limited to the means by which they realize durability nor is the manipulation of state under control of the transaction service. The same applies for concurrency control. In fact, the concurrency control theory has been well researched for particular configurations like multi-database or multilevel systems. However, the general case for correctness and concurrency control in composite systems has been addressed in recent work, only, and is an area of ongoing research. Component technologies such as Sun s Enterprise JavaBeans (EJB) follow the distributed object computing model, but introduce and emphasize the concept of a "container" to manage the interactions between object instances and the server. A container is responsible for tasks such as creating new object instances, and it supports the mapping between objects and records in a database and generates code for persistent storage. With component container-

managed transactions, transactions also become the responsibility of the container. The container creates new or accepts existing client transaction contexts, and performs the invoked operations in those contexts. A major difference to distributed object transactions exist in the way a transaction is demarcated and managed. Explicit transaction demarcation as proposed by the CORBA OTS allows a client to fully control the scope and behaviour of transactions. However, the explicit model typically requires the use of complex APIs, and to write transactional code within the business logic. This may reduce the clarity and reusability of the code. Declarative transaction management as proposed with container-managed transactions separates transaction behaviour from business logic, thus promises for better reuse of the same business logic code in different transactional or non-transactional environments. Declarative transaction management simplifies the coding of the transaction, as all that is required is to set transactional attributes for a component s operations. These transactional attributes are set at deployment time (not development time) on the server side. However, this also implies that the transactional behaviour of a server can easily be overridden and changed at deployment, which may interfere with the transactional requirements and expectations of clients. Message-oriented transactions and transactional publish/subscribe include enqueueing/ dequeueing of messages or publishing/consumption of notifications in units of work with "all-or-nothing" semantics. Enqueueing/dequeueing of messages and publishing/ consumption of notifications is dependent on the overall transaction outcome and vice versa. The queue manager provides its own transaction manager and associated transaction demarcation API or it acts as a transactional resource on behalf of a distributed transaction. Note that message-oriented transactions are fundamentally different from the three other kinds of transactions discussed above. Message-oriented transactions group a set of messages that are to be delivered or consumed as a whole. Message-oriented transactions do not address the processing of data (as consequences of delivery or receipt of a message) and consistency of data transformations.

3 Advanced Transactions: X 2 TS and Bourgogne Transactions

The technologies presented so far all reflect the *traditional* transaction processing concepts. The traditional transaction concept has its limitations as *(i)* it lacks support for structuring multiple interdependent transactions and *(ii)* traditional concurrency control does not support cooperative transactions and long running units of work. In the literature, there have been various suggestions to extend the traditional concepts of transactions. The papers in this session present work that addresses such extended transaction management concepts in two different areas. Their goal has not been to introduce yet another transaction model but to synthesize some ideas of such extended transaction models on behalf of services for distributed object systems. They aim at better addressing problems commonly encountered in the engineering of distributed object systems. The first

paper in this session, "Integrating Notifications and Transactions: Concepts and X 2 TS Prototype" by Christoph Liebig, Marco Malva, and Alejandro Buchmann, falls into the category of advanced distributed object transactions. It addresses issues of relating event notifications and distributed object transactions. The work extends concepts which have initially been proposed in centralized and monolithic active database systems. X 2 TS presents a novel service-based approach for transaction-aware event-based interaction in distributed systems, CORBA in particular. While transaction services in current middleware only support request/reply interactions, X 2 TS focuses on event-based systems and allows to realize flexible transactional couplings between event producing and event consuming components. The application of the suggested service for process enactment is discussed. The prototype architecture is described and implications of distribution aspects on reliable event processing are discussed. The second paper in this session, Advanced Transactions in Enterprise JavaBeans by Marek Prochazka, looks at component container-managed transactions. It identifies limitations of the EJB model, in particular with respect to concurrency control, support for long-running and cooperative transactions, as well as transaction flow control. It proposes the adaptation of the ACTA specification framework to EJB transactions. A new transaction management API is proposed that allows to establish dependencies between ongoing transactions along the modes presented in ACTA. The approach is called "Bourgogne transactions". With respect to concurrency control, the programmer is given control over the visibility sets and delegation of objects that are affected by the transaction. Some of the issues discussed are not restricted to container-managed transactions, but also apply to distributed object transactions (JTA/JTS transactions for Java systems in particular).

4 Discussion

The two papers in this session raise a number of interesting and important questions for continued research. During the workshop, research issues relating to the question of integrating transactions and event notifications were primarily discussed. The paper by Liebig et al. tackles the problem of integrating the CORBA OTS with the CORBA Notification Service (NOS). While transaction management is well understood for procedural interactions, support for event-based interactions is restricted to message queue integrating transactions (and transactional publish/subscribe). In that case the visibility of notifications is bound to the successful commit of a transaction: immediate reactions as well as parallel actions triggered by events are not possible. In distributed object systems, there may be many different components that subscribe to events with differing reliability and processing demands. It is therefore suggested to extend the event-action paradigm with coupling modes that determine when events become visible, in which transaction context the reaction should run in, what the dependencies between triggering and triggered transaction are, and when events are considered to be delivered and consumed. Such integration of event

notifications and transactions is highly desirable, particularly in the context of process enactment and workflow management. During discussion, another reason for such an integration has been pointed out: many enterprise applications use both object-oriented middleware and message-oriented middleware in combination. Object middleware addresses reliability through the concept of transactions that comprise synchronous object requests only, while message middleware (or, messaging services for object middleware) addresses reliability through guaranteed message delivery even in the presence of system failures. An equivalent level of reliability for the use of both middleware in combination does not exist today. It is very desirable to advance the current state-of-the-art towards transaction processing that allows for both synchronous object requests and asynchronous messages to constitute a single transaction at the same time. When integrating event notifications and transactions we obviously need to understand to what extent the ACID properties are still satisfied, or are violated. We believe that the property of atomicity is actually "extended" in that the atomicity sphere of the event producer may depend - in various ways - on the atomicity sphere of the reacting event consumer. Thus a powerful mechanism to structure units of work into interdependent atomicity spheres is introduced that allows to realize advanced transaction concepts such as compensations, or pluggable reliable exception handling. Recovery in event-driven systems has not received much attention. X 2 TS supports forward recovery by replaying events in case of failures and provision of tuneable delivery and consumption guarantees. More research is needed to find design guidelines for a consumer in case of event recovery: which recovered notifications should be ignored or reacted to and in which manner, especially in case of complex composed events. The flexibility of reliable event-based interactions imposes an increased complexity of systems design on the software engineer. Therefore, declarative means for introducing reactive behaviour, in particular for specifying coupling modes should be introduced. In general, distributed event-based architectures lack adequate support by current software development methods. While an integrated approach may concentrate exclusively on the property of atomicity, the properties of consistency, isolation, and durability require particular attention as well. We need to define what these properties mean in an integrated context, and how the responsibility for assuring these properties is distributed or shared between the involved components: the (conventional) transaction service, the messaging service, integrated databases and resource managers, and the application objects. The paper by Prochazka discusses the EJB transaction model and identifies a number of weaknesses of it. The weaknesses stated fall in two categories. On the one hand, there are weaknesses that are inherited from existing transaction systems that EJBs need to collaborate with. On the other hand, the paper questions the appropriateness of conventional transactions for (EJB) component architectures and presents EJB-specific weaknesses. Most notably, the paper argues for more application-side flexibility and control of transactions, which could lead to conflicts with the concept of declarative transaction management of container-managed transactions. "Bourgogne transactions" introduce a new API, which suggests the preference

of an explicit transaction management model as opposed to a (more restricted) container-managed one. More research is needed to understand the differences between explicit and declarative transaction management in detail, and how to employ each of them (or, their combination) effectively for component architectures. Bourgogne transactions support the explicit management of visibility sets and delegation of object ownership at the Bean level. More research is needed, in how far the granularity of a Bean is appropriate for weakening isolation and to which extend the abstraction of a data object as presented in ACTA is compatible with the notion of an EJB component. In general, more practical experience as well as conceptual work is needed in the complex field of engineering concurrency control in component based distributed and heterogeneous systems.

Acknowledgements

This paper is based in part on the comments of the following workshop attendees: Wolfgang Emmerich, Kostas Kontogiannis, Marek Prochazka, David Rosenblum, Jrn-Guy S, and Stanley Sutton.

Integrating Notifications and Transactions: Concepts and X^{2}TS Prototype

C. Liebig, M. Malva, and A. Buchmann

Database and Distributed Systems Research Group
Darmstadt University of Technology

Abstract. Event-based architectural style promises to support building flexible and extensible component-oriented systems and is particularly well suited to support applications that must monitor information of interest or react to changes in the environment, or process status. Middleware support for event-based systems ranges from peer-to-peer messaging to message queues and publish/subscribe event-services. Common distributed object platforms restrict publishing events on behalf of transactions to *message integrating transactions*. We suggest that concepts from active object systems can support the construction of reliable event-driven applications. In particular, we are concerned with unbundling transactional reactive behavior in a CORBA environment and introduce X^{2}TS as integration of transaction and notification services. X^{2}TS features rich coupling modes that are configured on a per subscriber basis and supports the application programmer with coordinating asynchronous executions on behalf of transactions.

1 Introduction

Messaging and event-services are attractive for building complex component-oriented distributed systems [16, 22, 17]. Moreover, systems designed in an event-based architectural style [9] are particularly well suited for distributed environments without central control, to support applications that must monitor information of interest or react to changes in the environment, or process status. The familiar one-to-one request/reply interaction pattern that is commonly used in client/server systems is inadequate for systems that must react to critical situations and exhibit many-to-many interactions. Examples for such applications are process support systems and workflow management systems. They are best constructed using event-based middleware, realizing the control flow and inter-task dependencies by event-driven task managers [28]. In systems like weather alerts, stock tracking, logistics and inventory management, information must be disseminated from many publishers to an even larger number of subscribers, while subscribers must be given the ability to select and extract the data of interest out of a dynamically changing information offer [32, 4]. In related research as well as in de-facto standard message oriented middleware, reliability concerns are restricted to event-delivery using transactional enqueue/dequeue or transactional publish/subscribe [49]. Only minimal support is given to structure the dependencies between publisher and subscriber with respect to their transaction context. In distributed object systems, transac-

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 194-214, 2001.

tions are a commonly used concept to provide the simple all-or-nothing execution model. Transactions bracket the computations into spheres of atomicity. When using an event-based style this implies asynchronous transaction branches and dynamically evolving structure of atomicity spheres. Dependencies between publisher and subscriber can be expressed in terms of visibility - i.e. when should events be notified to the subscribers - in terms of the transaction context of the reaction and in terms of the commit/abort dependencies between the atomicity spheres of action and reaction.

The overall goal of the *Distributed Active Object Systems* (DAOS) project is to unbundle the concepts of *active object systems* [7], which are basic to active database management systems. An active object autonomously reacts to critical situations of interest in a timely and reliable manner. It does so by subscribing to possibly complex events that are detected and signalled by the active object system. To provide reliable and correct executions the action and reaction may be transactionally coupled. As active functionality is useful beyond active databases, we aim to provide configurable services to construct active object systems on behalf of a distributed object computing platform, CORBA in particular. While DAOS in general addresses many of the dimensions of active systems as classified in [50, 44, 14], in this paper we focus on X^2TS, an integration of notification (NOS) [37] and transaction service (OTS) [40] for CORBA. X^2TS realizes an event-action model which includes transactional behavior, flexible visibilities of events and rich coupling modes. The prototype leverages COTS publish/subscribe MOM to reliably distribute events. The prototype deals with the implications of asynchronous reactions to events. We explicitly consider that multiple subscribers with diverse requirements in terms of couplings need to be supported. The architecture also anticipates to plug in composite event detectors.

The rest of this paper is organized as follows. In the next section we will briefly discuss related work. In Section 3 we will introduce the concept of coupling modes and the dimensions for a configurable service. We discuss the application of active object concepts to event-based process enactment. In Section 4, we will present the architecture and implementation of the X^2TS prototype. Section 5 concludes the paper.

2 Related Work

Work has been done on integrating databases as active sources of events and unbundling ECA-like rule services [18, 27] for distributed and heterogeneous environments. Various publish/subscribe style notification services have been presented in the literature [35, 33, 10, 20, 47]. They focus on filtering and composing events somewhat similar to event algebras in active DBMSs [15, 41]. In [49], the authors identify the shortcomings of transactional enqueue/dequeue in common message oriented middleware and suggest message delivery and message processing transactions. The arguments can be applied to transactional publish/subscribe [12], as well. Besides the work in [49], to our knowledge the provision of publish/subscribe event services that are transaction aware has not been considered so far. A variety of process support and workflow management systems are event-driven - or based on a centralized active DBMS - and encompass execution models for reliable and recoverable enactment of tasks or specifying inter-task dependencies [53, 21, 25, 28, 13, 11]. However those systems are merely

closed and do not provide transactional event-services per se. The relationship of a generic workflow management facility to existing CORBA services and the reuse and the integration thereof needs further investigation [46,8].

3 Notifications & Transactions: Couplings

In a distributed object system, atomicity of computations can be achieved using a two-phase commit protocol as realized by the CORBA OTS for example (see section 4.1 for a brief introduction to OTS). All computations carried out on behalf of an OTS transaction are said to be in an atomicity sphere. Atomicity spheres may be nested, but still are restricted to OTS transaction boundaries. OTS transactions relief the software engineer to deal with all possible error situations that could arise to the various failure modes of participating entities: either all computations will have effect, or none of them will (all-or-nothing printciple). If isolation of access to shared data and resources is an issue, a concurrency control mechanism must be applied, typically two-phase locking as in the CORBA Concurrency Service. As a consequence, the isolation sphere corresponds to the transaction boundaries. Objects may be stateful and require durability. In that case full fledged ACID transactions are appropriate.

The design issue arises, where to draw the transaction boundaries. There are trade-offs between transaction granularity and failure tolerance: always rolling back to the beginning is not an option for long running computations. On the other hand, simply splitting the work into smaller transactional units re-introduces a multitude of failure modes that must be dealt with explicitly. In addition, there are trade-offs between transaction granularity and concurrent/cooperative access to shared resources and consistency. Long transactions reflect the application requirements as they isolate ongoing computations, hide intermediate results and preserve overall atomicity. However, common concurrency control techniques unnecessarily restrict cooperation and parallel executions. Short transactions allow higher throughput but might also commit tentative results.

3.1 Coupling Modes for Event-Based Interaction

When looking at event-based interactions, a variety of execution models are possible. If we take reliability concerns into account, the consumer's reaction to an event notification can not considered to be independent of the producer in all cases. For example, dependent on the need of the event consuming component, one could deliver events only if the triggering transaction has commited. In other cases, such a delay is inacceptable. The reaction could even influence the outcome of the triggering transaction or vice versa. Therefore the notion of a coupling mode was introduced in the HiPAC [15] project on active database management systems. Coupling modes determine the execution of triggered actions relative to the transaction in which the triggering event was published [7,5]. X^2TS provides a mechanism to reliably enforce flexible structures of atomicitiy spheres between producers and consumers through the provision of coupling modes.

We consider the following dimensions that define a coupling mode in X^2TS:

- *visibility*: the moment when the reacting object should be notified
- *context*: the transaction context in which the object should execute the triggered actions
- *dependency*: the commit(abort)-dependency between triggering transaction and triggered action
- *consumption*: when the event should be considered as delivered and consumed

Table 1 shows the configurable properties assuming a flat transaction model. We have also considered the possible coupling modes in case of nested transactions. Because of the arbitrary deep nesting of atomicity spheres, many more coupling modes are possible. Yet, not all of them are useful, as transitive and implicit dependencies must be considered. For sake of simplicity, they are not discussed here and will be presented elsewhere.

visibility	immediate, on commit, on abort, deferred
context	shared, separate top
forward dependency	commit, abort
backward dependency	vital, mark-rollback
consumption	on delivery, on return, explicit, atomic

Table 1: Coupling modes

- With *immediate* visibility, events are visible to the consumers as soon as they arrive at the consumer site and independent of the triggering transaction's outcome. In case of *on commit* (*on abort*) visibility, a consumer may only be notified of the event if the triggering transaction has committed (aborted). A *deferred* visibility notifies the consumer as soon as the event producer starts commit processing.
- A *commit* (*abort*) forward dependency specifies, that the triggered reaction commits only if the triggering transaction commits (aborts).
- A backward dependency constrains the commit of the triggering transaction. If the reaction is *vital* coupled, the triggering transaction may only commit if the triggered transaction has been executed and completed successfully.
  If the consumer is coupled in *mark-rollback* mode, the triggering transaction is independent of the triggered transaction commit/abort but the consumer may explicitly mark the producer's transaction as *rollback-only*.
  Both backward dependencies imply, that a failure of event delivery will cause the triggering transaction to abort.
- Once an event has been consumed, the notification message is considered as delivered and will not be replayed in case the consumer crashes and subsequently restarts. The event may be consumed simply by accepting the notification (*on delivery*) or when returning from the reaction (*on return*). Alternatively, consump-

tion may be bound to the commit of the consumer's atomicity sphere (*atomic*) or be *explicit*ly indicated at some point during reaction processing.

If the reaction is coupled in a *shared* mode, it will execute on behalf of the triggering transaction. Of course this implies a forward and backward dependency, which is just the semantic of a sphere of atomicity. Otherwise, the reaction is executed in its own atomicity sphere, i.e. *separate top* and the commit/abort dependencies to the triggering transaction can be established as described above.

3.2 Discussion

The most notably distinctions between coupling modes in X^2TS and in active databases as proposed in [7,5] are that

i) coupling modes may be specified per event type even across different transactions
ii) X^2TS supports parallel execution in the shared (same) transaction context
iii) publishing of events is non-blocking, i.e. blocking execution of triggered actions, is not supported

Backward dependencies even if implicit, as in ii) - raise the difficulty of how to synchronize asynchronous branches of the same atomicity sphere. This is why we introduce checked transaction behavior. A triggering transaction may not commit before the reactions that have backward dependencies are ready to commit (and vice versa). Publish/subscribe in principle encompasses an unknown number of different consumers. Therefore, we require that the consumers, that may have backward dependencies and which need to synchronize with the triggering action, be specified in a predefined group.

With the same arguments as presented in [43], event composition may span several triggering transactions and we support couplings with respect to multiple triggering transactions.

The visibility modes suggested, provide more flexibility than transactional messaging (and transactional publish/subscribe). Transactional messaging, as provided by popular COTS MOM, only focuses on message integrating transactions [49], that is messages (events) will be visible only after commit of the triggering transaction. This restricts the system to short transactions and therefore provides no flexibility to enforce adequate spheres of atomicity (and isolation). Also, flexible and modular exception handling - as supported by abort dependencies in our system - is not possible.

On the other hand, principally ignoring spheres of atomicity - using transaction unaware notifications - is not tolerable by all applications: with *immediate* visibility, computations are externalized (by publishing events) that might be revoked, later. With respect to serialization theory [2] publishing an event can be compared to a write operation, while reacting to it introduces a read-from relation. Therefore dirty reads - reactions to immediate visible events of not-yet-committed transactions - may lead to non recoverable reactions or cascading aborts. While in some cases it may be possible to revoke and compensate an already committed reaction, in other cases it might not be. The design of X^2TS considers the fact, that a single execution model will not be flexible enough to fulfil the diverse needs of different consumers. Therefore, coupling modes,

most important visibility, are configured on a per consumer basis. While one consumer may be restricted to react to events *on commit* of the triggering transaction, others may need to react as soon as possible and even take part in the triggering transaction e.g. to check and enforce integrity constraints. More loosely couplings may be useful, too. For example an active object that reacts immediately without a forward commit dependency but additionally registers an abort dependent compensating action, in case the triggering transaction eventually aborts.

An event-based architectural style is characterized to exhibit a *loosely coupled* style of interaction. We suggest that for the sake of reliable and recoverable executions in event-based systems, atomicity spheres and configurable coupling modes should be supported. Nevertheless, X^2TS provides the advantages of publish/subscribe many-to-many interactions with flexible subject based addressing in contrast to client-server request reply and location based addressing. The asynchronous nature of event-based interactions is sustained as far as possible and the differing demands of heterogeneous subscribers are taken care of.

3.3 Example: Event-Based Process Enactment

Separation of concerns is the basis for a flexible and extensible software architecture. Consequently, one of the principles of workflow and process management - separating control flow specification and enforcement from functionality specification and implementation - is a general design principle for component-oriented systems [1,3,29,45]. Flow independent objects, that implement business logic, are invoked by objects that reify the process abstraction, i.e. process flow knowledge. In the following, we will briefly characterize the principles of event-based process enactment and the application of transaction aware notifications.

A process model (workflow schema) is defined using some process description language to specify activities that represent steps in the business process. Activities may be simple or complex. A complex activity contains several subactivities. Activities must be linked to an implementation, i.e. the appropriate business object. Activities are connected with each other using flow control connectors like split and join. In addition event connectors should be provided to support inter-process dependencies and reactive behavior at the process model level [21]. An organizational model and agent selection criteria must be specified, as well.

The process enactment system at runtime instantiates for each activity a task and a task manager. The task manager initiates the agent assignments and business object invocations, keeps track of the task state, handles system and application level errors, verifies the pre- and postconditions and enforces control flow dependencies. In any case, a protocol must be established for the interaction between task manager and business objects as well as task manager and other process enactment components (i.e. other task managers, resource managers, worklist handlers).

In the case of an event-based architecture, a task manager subscribes and reacts to events originating from tasks (i.e. activity instantiations), other task managers, resource and worklist managers as well as from external systems. This requires the participating components to externalize behavior and publish events of interest or allow instrumentation for event detection. Task state changes are published as events, for example "task

creation", "task activation"; "task completion". The task manager reacts to error situations, task completions, etc. Different phases of task assignment to agents, like "assignment planned", "assignment requested", "assignment revoked", may be published to support situation aware worklist managers.

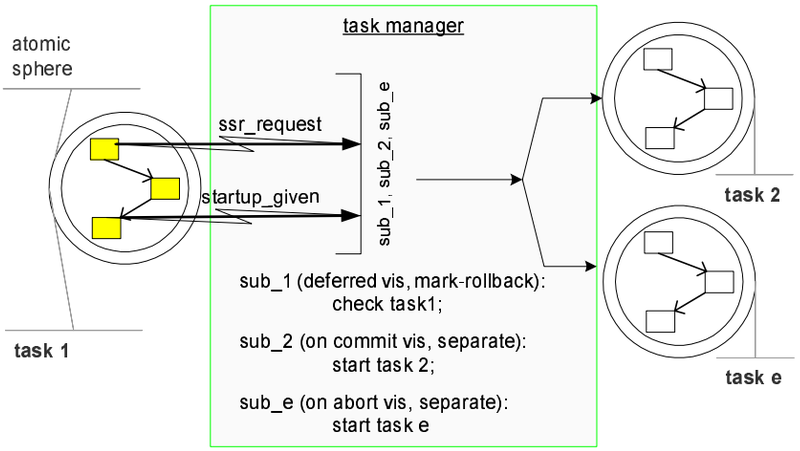


Figure 1 Event-based process enactment

Figure 1 illustrates the application of coupling modes. A task manager keeps track of the execution of business objects in order to check consistency of task executions (subscription `sub_1`); if consistency is violated, the triggering transaction will be marked rollback-only. Otherwise, the task manager chains the respective follow-up task (subscription `sub_2`), optionally using complex events to distinguish alternative execution paths depending on significant events which are external to the process. If task execution errors occur, restarts can be initiated or contingency steps may be invoked (`subscription sub_e`). As already mentioned, task managers may not only react to primitive events signalled by the controlled task but also to composed events signalled by other task managers of predecessor tasks, task managers of a different process or even external systems. Therefore X^2TS must support composition and couplings over transaction boundaries.

Events published by controlled tasks and task managers will not only be of interest in a local process enactment scope but also in a more global scope. In a recent study in cooperation with the German air traffic control authorities [30, 23], we found that the notification of regional and supra regional capacity planning systems and decision support systems about progress and critical situations in the flight departure (gate-to-runway) process is essential. In addition, worklist managers at the controller's workplace must be situation aware, that is track the status of different but inter-related process instances, and track planned, past, and current task assignments for several agents involved in inter-related processes, and timely notify the controller of significant situations.

Different event-subscribers will have different requirements with respect to visibility and coupling of atomicity spheres. For example, worklist managers will typically require immediate visible notifications and additionally react to the abort of actions or change in assignment status in separate transactions. Reliable task control flow can be implemented using *on commit/abort* visibilities whereas consistency checks typically are *deferred* and share the same transaction context. The open nature of the envisaged event-based architecture may provide the necessary flexibility to integrate next-generation air traffic control IT systems.

Activities are mapped to transactions defining atomicity spheres on the workflow schema level. In the simple case, an activity is mapped to a transaction and subactivities are mapped to (possibly open) subtransactions. Additionally, spheres of atomicity could span more than one activity. This approach shares many ideas with [13] and [1]. X^2TS does not implement multitransactions (i.e. multilevel transactions [52], nested SAGAS [19], DOM [6]) by itself, but supports the construction of multitransactions using X^2TS, as the necessary commit/abort dependencies can be established and for example compensations can be triggered by X^2TS. Recently, based on the work in [42], a proposal for *Additional Structuring of the OTS* [24] has been submitted to the OMG, which provides activity management, e.g. allows for the realization of SAGAs and compensations, on top of OTS and which we think could benefit from X^2TS. We note, that multitransactions alone do not solve the problem of workflow recovery, as multitransaction concepts all bear the problem of specifying compensations that preserve correctness in terms of serializability. Such correctness criteria typically are based on commutativity of operations [52, 26] and thus require application level knowledge. We think, that in a CORBA world it will be hard to define which operations commute and which do not. However, compensating activities at the workflow schema level seem to be a promising approach. In that case, not all local transactions of a task execution need to be compensated automatically.

4 X^2TS Prototype

First, we will summarize the key features of CORBA OTS and NOS as far as relevant to the X^2TS prototype and give a short overview of which part of the services are supported by our prototype and the interfaces that are essential for using it. The design and implementation of the X^2TS prototype will be discussed later on.

4.1 CORBA OTS

OTS can be thought of as a framework to manage transactional contexts and orchestrate the two-phase-commit processing (2PC) between potentially remote recoverable servers. A CORBA transaction context - identified by its `Control` object reference - thus represents a sphere of atomicity. OTS neither provides failure atomicity nor isolation per se but delegates the implementation of recovery and isolation to the participating recoverable servers. Isolation can be either implemented by the transactional object itself or by use of the Concurrency Service [39].

A CORBA recoverable server object must agree upon a convention of registering callback objects with the OTS *Coordinator*. The latter drives the 2PC through invocation of callback methods as depicted in Figure 2 (subtransaction interfaces are left out).

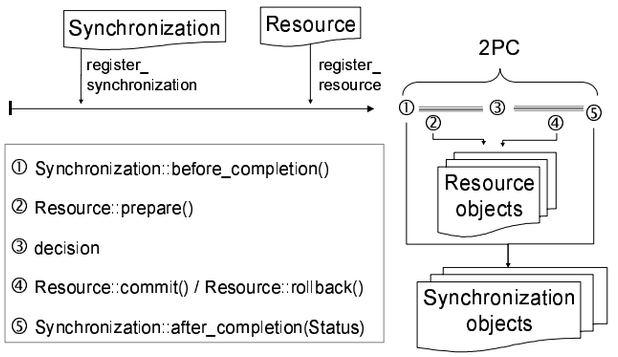


Figure 2 Callbacks in OTS

`Resource` objects participate in the voting phase of the commit processing and impact the outcome of the transaction. The `Synchronization` object is not essential for 2PC but for additional resource management, like the integration of XA compliant databases and queue managers. The OTS engine will call back the registered `Synchronization` objects before start of and after termination of the two phase commit processing. In X^2TS we make use of *Synchronization* objects to realize checked transaction behaviour and to add publishing of transaction state change events (also see section 4.4).

4.2 CORBA NOS

The CORBA Notification Service has been proposed as extension to the CORBA Event Service. The most notably improvements in NOS are the introduction of filters and the configuration of quality of service properties. NOS principally supports interfaces for push-based and pull-based suppliers and consumers. Events may be typed, untyped or structured and signalled to the consumer one-at-a-time or batch-processed. A `StructuredEvent` consists of a fixed event header, optional header fields, a filterable event body and additional payload.

An `EventChannel` is an abstraction of a message bus: all subscribers connected to a channel can in principle see all events published to that channel. Consumers may register conjunctions or disjunctions of filters, each of which matches events to a given constraint expression. Event composition is not supported by NOS - besides underspecified conjunction and disjunction operators. Quality of service (QoS) and filters can be configured at three levels providing grouping of consumers. NOS implementations may support specific QoS parameters, e.g. event ordering, reliability, persistence, lifetime etc. Configurations are programmed by setting the respective properties, represented as tagged value tuples.

4.3 Services Provided by the X^{2}TS Prototype

X^{2}TS integrates a push/push CORBA Notification Service and a CORBA Transaction Service. The two basic mechanisms provided by the OTS, context management/propagation and callback handling are a suitable basis for incorporating extended transaction coordination. X^{2}TS supports indirect context management, implicit context propagation and interposition. We have implemented an XA-adapter [48], mainly to integrate RDBMSs, currently there is support for accessing Informix IUS using embedded SQL in a CORBA recoverable server.

In our prototype, we only support the push-based interfaces with `StructuredEvent` one-at-a-time notifications. We assume that events are instances of some defined type and that subscription may refer to events of specific types and to patterns of events. Patterns may range from simple filters to complex event compositors. To this end, we do not specify the event model and specific types of events, but suppose the `StructuredEvent` to act as a container for whatever structured information an event of some type carries.

The overall architecture of the combined transaction and notification service as provided by X^{2}TS is shown in Figure 3.

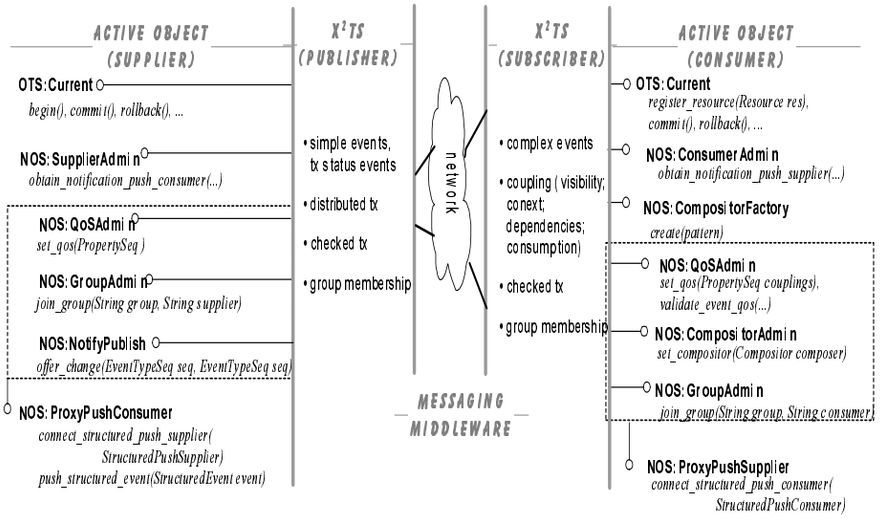


Figure 3 X^{2}TS architecture

An event supplier creates, commits or aborts transactions through the use of the `Current` pseudo object (indirect context management). Events are published on behalf of the current transaction context using the `ProxyPushConsumer` interface. The proxy is provided by the `SupplierAdmin` factory. An event supplier must register (advertise) the types of events to be published using the `NotifyPublish` interface. The supplier then publishes events by invoking the `push_structured_event` with the

appropriate parameter i.e. the event type is set in accordance with the previous type advertisements.

A consumer - representing an active object - subscribes to events or patterns of events by registering a `PushConsumer` callback object with a `ProxyPushSupplier` provided by the `ConsumerAdmin` factory. Subscription to patterns of events and transactional couplings are imposed by configuring the service proxy, i.e. setting the appropriate properties through `QoSAdmin` and `CompositorAdmin`. Detecting patterns of events is realized by pluggable event compositors that are created by a specific `CompositorFactory`. We do not support the standard NOS filters but added our own proprietary `Compositor` interface.

The `GroupAdmin` interface provides group services, which are essential to realize checked transactions and vital reactions. The semantic imposed by a *vital* reaction of subscriber in a separate transaction as well as reactions in a shared context require, that the subscriber is known to the publisher site X^{2}TS before start of commit processing. Otherwise, communication failures could hinder the vital (or shared context) subscriber to be registered as a participant in the commit processing, and falsify the commit decision.

X^{2}TS currently supports service configuration only at the `ProxyPushSupplier` (`ProxyPushConsumer`) level. Event patterns are coded as strings and set in service properties at the proxy level. An event pattern contains the event type declarations and the composite event expression. If any coupling modes are specified, the couplings must refer to event types of the pattern declaration. We propose that facades should be defined which simplify configuration by providing predefined coupling mode settings. For event composition to cope with the lack of global time and network delays we introduce (in)accuracy intervals for timestamping events and suggest to use supplier heartbeats. The details are out of the scope of this paper - basic ideas can be found in [31]. We are implementing a basic operator framework for building application specific compositors, incorporating some ideas of [55].

We remark that a*ctive objects* are rather a concept than a programming language entity. It compares to the virtual nature of a CORBA object. To refer to active objects declaratively, they must be part of the CORBA object model. We think, that the specification of an active object should take place at the component level. In fact, the upcoming CORBA Components model [38] includes a simple facet for containers to provide event services and components to advertise publishers and register subscribers. Different knowledge models for specifying reactions, e.g. rules, are possible. Active objects, once specified, could use the services offered by X^{2}TS.

4.4 X^{2}TS Prototype: Architecture and Implementation

In the next section, we will briefly describe the relevant features of the MOM used for transport of notifications and then present the architecture and some implementation details of the X^{2}TS prototype.

4.4.1 Pub/Sub Middleware

X^{2}TS is implemented on top of a multicast enabled messaging middleware, TIB/ObjectBus and TIB/Rendezvous [51]. TIB/Rendezvous is based upon the notion of the *In-*

formation Bus [36] (interchangeable with "message bus" in the following) and realizes the concept of *subject based addressing*. The subject name space is hierarchical and subscribers may register using subject name patterns. Three quality of service levels are supported by TIB/Rendezvous: reliable, certified and transactional. In all modes, messages are delivered in FIFO order with respect to a specific publisher. There is no total ordering in case of multiple publishers on the same subject. Reliable delivery uses receiver-side NACKs and a sender-side in-memory ledger. With certified delivery, a subscriber may register with the publisher for a *certified session* or the publisher preregisters dedicated subscribers. Atomic message delivery is not provided. The TIB/Rendezvous library uses a persistent ledger in order to provide certified delivery. Messages may be discarded from the persistent ledger as soon as all subscribers have explicitly acknowledged the receipt. Acknowledgements may be automatic or application controlled. In both variants, reliable and certified, the retransmission of messages is receiver-initiated by sending NACKs. Transactional publish/subscribe realizes message integrating transactions [49] and is therefore not suitable for the variety of couplings we aim to support.

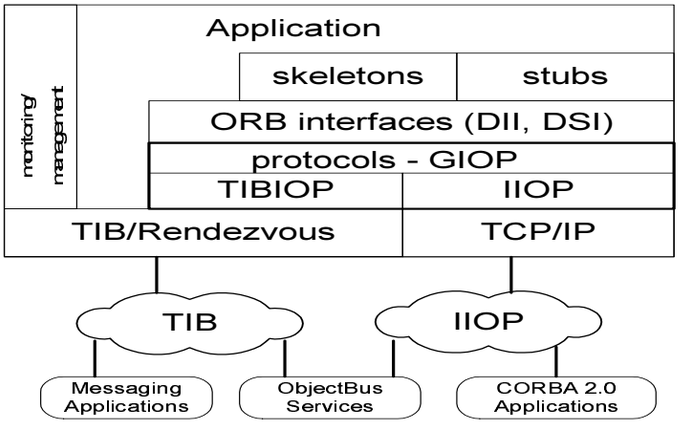


Figure 4 ObjectBus architecture

The diagram in Figure 4 depicts, how the multicast messaging middleware is introduced to CORBA in ObjectBus. The General Inter-ORB Protocol is implemented both by a standard IIOP layer and a TIBIOP layer. When using TIBIOP, the GIOP request messages are marshalled into TIB/Rendezvous messages and published on the *message bus*. In order to preserve interoperability, server objects may be registered with both, TIBIOP and IIOP profiles at the same time. Most important for the X^2TS prototype is the integration of the ORB event loop with TIB/Rendezvous, in order to be able to use the messaging API directly.

One of the design goals is to leverage the features of the underlying MOM as much as possible in order to provide asynchronous publication, reliable distribution of events and replay in case of failures. If a consumer crashes before the events have been con-

sumed or a network error occurs, the MOM layer should initiate replays of the missed events on recovery of the consumer.

We have to ensure exactly-once delivery guarantees, end-to-end with respect to CORBA supplier and consumer. Event composition requires the possibility to consume events individually. Once an event is consumed, it must not be replayed. Additionally, consumption of events may depend on the consumer's transaction to commit. Therefore, at the consumer side, X^{2}TS supports explicit acknowledgement of received and consumed events as well as transactional acknowledgement and consumption of events. One of the complexities when realizing the prototype was to map delivery guarantees at the X^{2}TS supplier and consumer level to acknowledgements at the reliable multicast layer provided by the TIB/Rendezvous MOM. We use certified delivery and explicit acknowledgements of messages at the MOM layer, i.e. between X^{2}TS publisher and subscriber components. The publisher's ledger will persistently log events as long as there are subscribers that have not been delivered the message. In order to be able to consume events individually, the `MOM Connector` at the consumer site must persistently log consumption of events. In case of transactional subscribe, consumption of events will be logged on behalf of the consuming transaction and the consumer's transaction outcome depends on the logging to be successful. Additionally, we must filter out duplicate events, in case the event was consumed at the X^{2}TS level but not acknowledged at the MOM level.

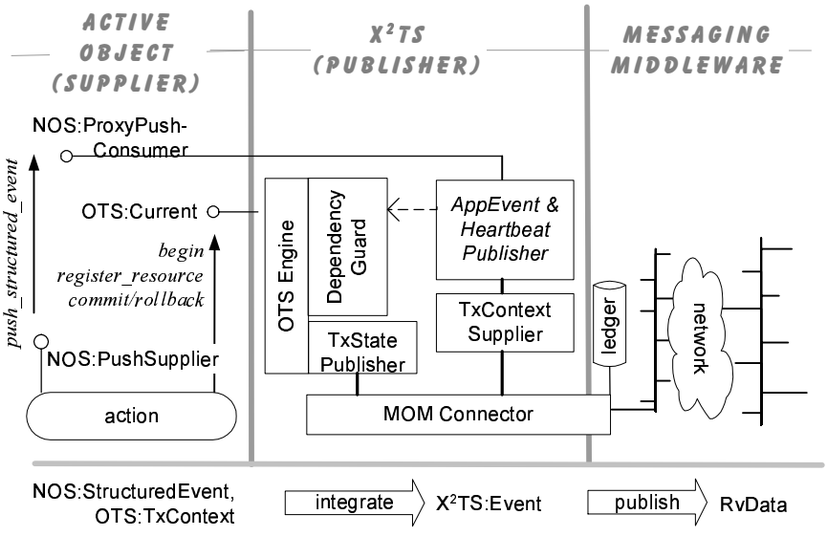


Figure 5 X^{2}TS publisher components

4.4.2 X2TS Publisher Components

Figure 5 depicts the components of X^{2}TS at the supplier site. The supplier has access to its transaction context through the `Current` interface and may push an event to the

`AppEvent Publisher`, which implements the *`StructuredProxyPushConsumer`* interface. In addition to the application events, which are signalled non-periodically, the event stream is enhanced with periodic heartbeat events. Thereby, event composition at the consumer site can correlate events based on occurrence order, instead of an unpredictable delivery order [31].

The `TxContext Supplier` is responsible for enclosing the current transaction context with the event. We use the *`OptionalHeaderFields`* of the event to piggy back transaction context information as well as event sequence numbers, and timestamps. The `MOM Connector` marshals events into a TIB/Rendezvous specific message (`RvData`) and publishes the events on the message bus.

The asynchronous nature of publishing events should be preserved, even if different consumers couple their reactions in different modes. For example, the *on commit* visibility of some consumer shall 1) not restrict other consumers to use *immediate* visibility and 2) not block the supplier in 2PC processing unnecessarily. Therefore, events are always published immediately on the message bus and visibilities and forward dependencies are resolved at the consumer site without requiring further interaction with the supplier in most cases. The design is based on the idea to publish transaction state change events asynchronously and treat visibilities and dependencies as a special case of event composition. In order to do so, the `TxState Publisher` publishes each state change of a transaction, such as 'operation-phase-terminated', 'tx-committed' and 'tx-aborted'. While such an approach would be straight forward in a centralized system, in a distributed environment with unpredictable message delays we cannot say at a consumer node, if for example a commit has not happened yet or the commit event has not been received yet. We will discuss in the section 4.6, how those situations are managed, following the principle of graceful degradation.

While forward dependencies can be resolved at the consumer site, backward dependencies impact the outcome of the triggering transaction and must be dealt with at the supplier site. The `Dependency Guard` couples the commit processing of the supplier's transaction to the outcome of reactions that are vital or that are to be executed in the same transaction context. In order to do so, for each backward coupled reaction a *Resource* object must be preregistered with the triggering transaction. Additionally, this requires to define the group of consumers which impose backward dependencies. A supplier must explicitly join this group and thereby allow the backward dependencies to be enforced.

4.4.3 X²TS Subscriber Components

The components at the consumer site are depicted in Figure 6 below. The `MOM Connector` maps a X²TS consumer's registration to appropriate MOM subscriptions. In the simple case, NOS event types and domain names are directly mapped to corresponding TIB/Rendezvous subjects. However, more complex mappings are possible and useful to leverage subject-based addressing for efficient filtering of events.

On incoming `RvData` messages, the MOM Connector unmarshals the event into an X²TS internal representation. The components run separate threads and are connected in a pipelined manner. Events are first pushed to the `Visibility Guard`, which enforces the various visibility QoS by buffering events until the necessary condition on

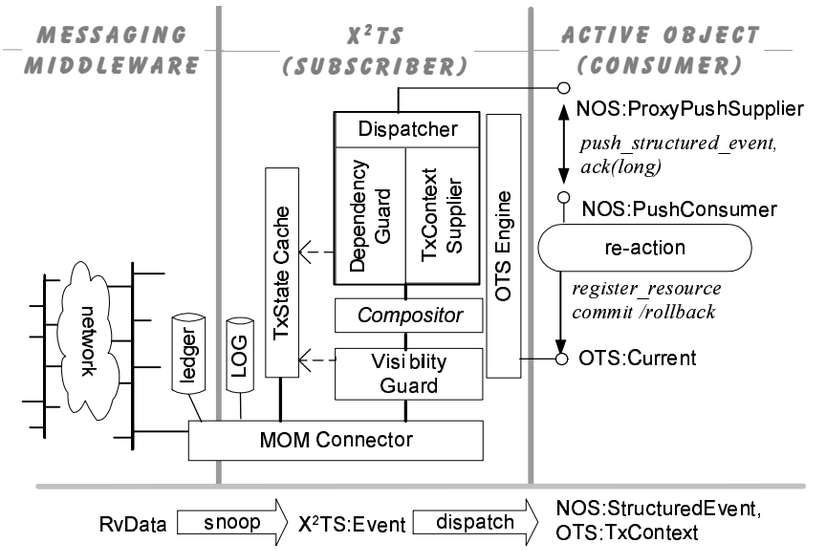


Figure 6 X²TS subscriber components

the triggering transaction outcome is fulfilled. If events becomes visible, they are pushed to the `Compositor` (which also may be a simple filter). When the `Compositor` detects matching events, it passes them to the `Dispatcher`, which establishes the appropriate transaction context for the reaction to run in and sets up the forward and backward dependencies, before finally pushing the event to the *PushConsumer* object.

4.5 Enforcing Visibilities

For each different visibility QoS which the consumer has configured, a `Visibility Guard` and its corresponding `RV Connector Pipe` are created. The `Visibility Guard` is running in its own thread and dequeues incoming events from the `MOM Connector` through the `RV Connector Pipe`. The `Visibility Guard` holds the events in a per transaction buffer until an appropriate Tx state change is signalled or a timeout occurs. Thereby, events (potentially of different types) are grouped by visibility. In addition, a transaction event listener snoops all transaction status events and forwards them to the interested `RV Connector Pipes` - and its associated `Visibility Guard` - and additionally to the `TxState Cache`.

Assume two transactions tx1 and tx2, both publishing event types e1 and e2. Further assume that the consumer subscribed *on commit* for an event pattern that contains e1 and e2 (whatever the pattern logic is).

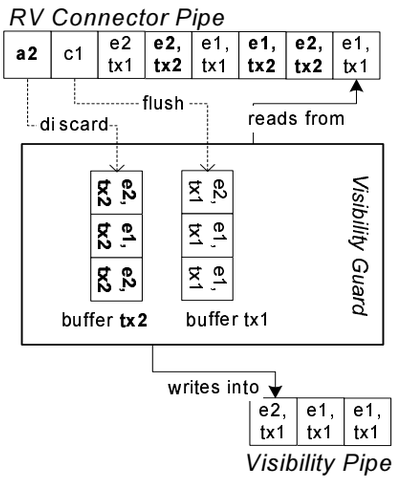


Figure 7 Visibility guard

Figure 7 depicts what the actions of the `Visibility Guard` would be in case some supplier(s) publish *e1;e1;e2* on behalf of tx1 and other supplier(s) publish *e2;e1;e2* on behalf of tx2. The buffer for tx1 would be flushed to the `Visibility Pipe` when the commit status event of tx1 arrives, the buffer for tx2 would be discarded when the abort status event of tx2 arrives. The `Compositor` - running in its own thread - would be signalled when events are pushed into the `Visibility Pipe`.

4.5.1 Transaction Status Propagation

`Visibility Guard` and `Dependency Guard` at the consumer site depend on the transaction status events to arrive on time. This is not necessarily so. Transaction status events are published in reliable mode, only. Therefore a commit/abort event could get lost because of communication failures. Other situations to be considered are long running triggering transactions, a supplier crash (and abort) as well as consumer crash and recovery. In all cases, the situation could arise

i) that events are buffered by the `Visibility Guard` but no transaction status event will ever trigger the application events to become visible or be discarded.

ii) completion of a triggered transaction cannot progress because of an unresolved forward dependency (see following section 4.6)

To cope with i), we additionally feed timeout events to the `Visibility Guard`. For each referenced triggering transaction, the `Visibility Guard` will then consult the local `TxState Cache` and request a current `TxStatus` event to be submitted to its `Rv Connector Pipe`. If the cache does not have a hit, it queries caches on near-by nodes. If there is no success either, then the *RecoveryCoordinator* at the supplier site will be contacted. The *RecoveryCoordinator* object is part of the OTS engine, although it is implemented as a per node daemon in a separate process. It either forwards a trans-

action status request to the *Coordinator*, which is colocal to the supplier. In case of a crash of the supplier process, the *RecoveryCoordinator* will inspect the transaction log for the transaction outcome. The principle of graceful degradation restricts synchronous callbacks to the supplier site to (presumably) rare occasions of major network failures. Still, eventual progress is guaranteed in spite of various failure situations.

4.6 Transaction Dependencies

The `Dependency Guard` components orchestrates the behavior of dependent distributed transactions. On the subscriber site the `Dependency Guard` enforces forward dependencies, such as commit and abort dependencies by registering a *Resource* with the transaction *Coordinator*. In that case, a lightweight approach is possible to synchronize a *separate top* triggered action to the end of the triggering transaction. We may benefit from the fact that transaction status changes are pushed to the consumer site. As noted above, we cannot know at a consumer site, how long it takes the triggering transaction to complete and the commit/abort event to be delivered. In case the triggered reaction is to be completed while the status of the triggering transaction is not yet known, we heuristically decide to throw an exception and signal potentially unchecked behavior.

On the publisher site, the synchronization of distributed branches of the same transaction context and the imperative consideration of all control delegates is required. We need to consider the implications of coupling modes in case of multiple asynchronous executing reactions. We have to provide checked transaction behavior, that is we need a mechanism for joining multiple threads[1] of control for commit processing in case of coupling modes that use a shared context or specify backward dependencies. The situation may arise that the triggering transactions is to be committed while there are still branches (reactions) active. We leverage the callback framework inherent to the OTS in order to place *bolts* for checked transaction behavior and influence transaction outcome. A bolt realizes a synchronization barrier, that is checked before commit processing may start. Backward dependencies such as reactions in a shared context are realized by setting bolts at the triggering transaction. The *Synchronization* callbacks at the consumer site is then used to remove the (remote) bolts and eventually allow the triggering transaction to start commit processing. We do not block a commit call in such cases but raise an exception that the transaction is unchecked. The object in control of the transaction may then decide to retry later or take other measures. In order to establish an appropriate bolt - for example at the triggering transaction site - it is required to anticipate the reacting active objects in advance, using a group service.

5 Conclusions

X^2TS provides services to realize concepts of *active object systems* in a CORBA environment. X^2TS integrates CORBA notifications and transactions, leveraging multicast enabled MOM for scalable and reliable event dissemination. Exactly-once notifications can be realized without the need for a supplier based transactional enqueue. Notifica-

1. we refer to a thread of control in the X/Open XA sense

tions may be consumed individually and in non FIFO order by the consumer, which enables complex event composition. Coupling modes can be configured as quality of service on a per consumer basis. Therefore, the event-based application can control the manner in which to react to events published on behalf of transactions and dependencies between spheres of atomicity can be dynamically established and will be enforced by X^2TS. We consider the asynchronous nature of "loose" coupling in event-based systems and let the application decide in how far to trade time independence and flexibility of asynchronous interactions against synchronization with respect to transaction boundaries. As far as possible, visibilities and dependencies are resolved at the consumer site without calling back to the triggering application.

We suggest, that in particular process enactment can benefit from the services provided by X^2TS and multitransactions can be built on top. Still, declarative means to introduce the concept of active objects into the CORBA object model are to be explored.

Once the prototype implementation [34] is enriched with more powerful event compositors and reaches a stable state, performance experiments need to be conducted. There is still conceptual work to be done with respect to recovery of events. We suppose, that a more flexible approach to recovery of events is required than simply replaying the event history.

References

[1] G. Alonso, C. Hagen, H. Schek, and M. Tresch. Distributed Processing over Stand-alone Systems and Applications. In *23rd Intl. Conf. on Very Large Databases*, pages 575–579. Morgan Kaufmann, August 1997.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[3] K. Bohrer, V. Johnson, A. Nilsson, and B. Rubin. Business Process Components for Distributed Object Applications. *Communications of the ACM*, 41(6), June 1998.

[4] C. Bornovd, M. Cilia, C. Liebig, and A. Buchmann. An Infrastructure for Meta-Auctions. In *2nd Intl. Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, pages 21–30. IEEE Computer Society, June 2000.

[5] H. Branding, A. Buchmann, T. Kurdass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In *Proceedings of the International Workshop on Rules in Database Systems (RIDS '93)*, pages 111–126, Edinburgh, Scotland, September 1993.

[6] A. Buchmann, M. Oszu, M. Hornick, D. Georgakopoulos, and F. Manola. A Transaction Model for Active Distributed Object Systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan Kaufmann, 1992.

[7] A.P. Buchmann. Active Object Systems. In A. Dogac, M.T. Szu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*. Springer Verlag, 1994.

[8] C. Bussler. OMG Workflow Roadmap. Technical Report Version 1.2 OMG Document bom/99-08-01, Object Management Group (OMG), January 1998.

[9] A. Carzaniga, E. Di Nitto, D. Rosenblum, and A. Wolf. Issues in Supporting event-based architectural Styles. In *Proceedings of the third international workshop on Software Architecture (ISAW98)*, pages 17–20, 1998.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[11] S. Ceri, P. Grefen, and G. Sanchez. WIDE: A Distributed Architecture for Workflow Management. In *Research Issues in Data Engineering*. IEEE Computer Society, 1997.

[12] A. Chan. Transactional Publish/Subscribe: The Proactive Multicast of Database Changes. In *Intl. Conf. on Management of Data (SIGMOD 98)*. ACM Press, June 1998.

[13] Q. Chen and U. Dayal. A Transactional Nested Process Management System. In *12th Intl. Conf. on Data Engineering*. IEEE Computer Society, March 1996.

[14] C. Collet, G. Vargas-Solar, and H. Ribeiro. Toward a semantic event service for database applications. In *9th International Conference DEXA 99*, volume 1460 of *LNCS*, pages 16–27, Vienna, Austria, August 1998.

[15] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A Rosenthal, S.K. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. In *SIGMOD Record*, volume 17 (1), March 1988.

[16] L.G. DeMichiel, L.U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans. Specification, public draft Version 2.0, Sun Microsystems, JavaSoftware, May 2000.

[17] F. Cummins. OMG Business Object Domain Task Force. White Paper bom/99-01-01, OMG, January 1999.

[18] H. Fritschi, S. Gatziu, and K. Dittrich. FRAMBOISE - an Approach to Framework-based Active Data Management Construction. In *Proceedings of the seventh on Information and Knowledge Management (CIKM 98)*, pages 364–370, Maryland, November 1998.

[19] H. Garcia-Molina, D. Gawlik, J. Klein, C. Kleissner, and K. Salem. Coordinating Multitransaction Activities with Nested Sagas. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 16, pages 466–481. Prentice Hall, 1998.

[20] R. Gruber, B. Krishnamurthy, and E. Panagos. High-Level Constructs in the READY Event Notification System. In *SIGOPS Euroean Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998. SIGOPS.

[21] C. Hagen and G. Alonso. Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems. In *Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 450–457. IEEE Computer Society, 1999.

[22] M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Specification Version 1.0.2, Sun Microsystems, JavaSoftware, November 1999.

[23] B. Hochberger and J. Zentgraf. Design of a workflow management system to support modelling and enactment of processes in air traffic control. Technical report, Darmstadt University of Technology, June 2000.

[24] IBM, IONA, VERTEL, and Alcatel. Additional Structuring Mechanisms for the OTS Specification. Submission orbos/2000-04-02, OMG, Famingham, MA, April 2000.

[25] G. Kappel, P. Lang, S. Rausch-Schott, and W. Retzschitzegger. Workflow Management Based on Objects, Rules, and Roles. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995.

[26] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases*, pages 95–106, Brisbane, Queensland, Australia, August 1990. Morgan Kaufmann.

[27] A. Koschel, S. Gatziu, G. von Bueltzingsloewen, and H. Fritschi. Unbundling Active Database Systems. In A. Dogac, T. Ozsu, and O. Ulusoy, editors, *Current Trends in Data Management Technology*, chapter 10, pages 177–195. IDEA Group Publishing, 1999.

[28] N. Krishnakumar and A.P. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-Wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[29] F. Leymann and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1), 1997.

[30] C. Liebig, B. Boesling, and A. Buchmann. A Notification Service for Next-Generation IT Systems in Air Traffic Control. In *GI-Workshop: Multicast - Protokolle und Anwendungen*, pages 55–68, Braunschweig, Germany, May 1999.

[31] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings 4th IFCIS Conference on Cooperative Information Systems (CoopIS'99)*, pages 70–78, Edinburgh, Scotland, September 1999. IEEE Computer Press.

[32] L. Liu, C. Pu, and W. Tang. Supporting Internet Applications Beyond Browsing: Trigger Processing and Change Notification. In *5th Intl. Computer Science Conference (ICSC'99)*. Springer Verlag, December 1999.

[33] C. Ma and J. Bacon. COBEA: A CORBA-based Event Architecture. In *Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 117–131, New Mexico, USA, April 1998. USENIX.

[34] M. Malva. Integrating CORBA Notification and Transaction Service. Master thesis, in preparation, Darmstadt University of Technology, August 2000.

[35] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, Department of Computing, Imperial College, London, UK, December 1995.

[36] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SIGOPS '93*, pages 58–68, December 1993.

[37] Object Management Group (OMG). Notification service specification. Technical Report OMG Document telecom/98-06-15, OMG, Famingham, MA, May 1998.

[38] Object Management Group (OMG). Corba components (final submission). Technical Report OMG Document orbos/99-02-05, OMG, Famingham, MA, May 1999.

[39] Object Management Group (OMG). Concurrency service v1.0. Technical Report OMG Document formal/2000-06-14, OMG, Famingham, MA, May 2000.

[40] Object Management Group (OMG). Transaction service v1.1. Technical Report OMG Document formal/2000-06-28, OMG, Famingham, MA, May 2000.

[41] N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag (New York), September 1998.

[42] F. Ranno, S.K. Shrivastava, and S.M. Wheater. A system for specifying and coordinating the execution of reliable distributed applictions. In *Intl. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, 1997.

[43] W. Retschitzegger. Composite Event Management in TriGS - Concepts and Implementation. In *9th Intl. Conf. on Database and Expert Systems Applications (DEXA'98)*, LNCS 1460. Springer, August 1998.

[44] D.R. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 344–360, Zurich, Switzerland, 1997.

[45] S. Schreyjak. Synergies in a Coupled Workflow and Component-Oriented System. In *Workshop on Component-based Information Systems Engineering (CBISE '98)*, Pisa, Italy, 1998

[46] W. Schulze. Fitting the Workflow Managment Facility into the Object Management Architecture. In *3rd Workshop on Business Object Design and Implementation, OOPSLA'97*, April 1997.

[47] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Australian Unix Users Group Annual Conferece (AUUG'97)*, July 1997.

[48] R. Stuetz. Design and Implementation of a XA Adapter for CORBA-OTS. Master's thesis, Darmstadt University of Technology, October 1999.

[49] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms and Open Distribued Processing (Middleware 2000)*, New York, USA, April 2000. Springer-Verlag.

[50] The Act-Net Consortium. The Active DBMS Manifesto: A Rulebase of ADBMS Features. *SIGMOD Record*, 25(3), September 1996.

[51] TIBCO Software Inc. TIB/ActiveEnterprise. www.tibco.com/products/enterprise.html, July 2000.

[52] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[53] S.M. Wheater, S.K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications. In *IFIP Intl. Conf. in Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[54] X/Open Company Ltd. Distributed Transaction Processing: The XA Specificiation Version 2 . Technical Report X/Open Document Number S423, X/Open, June 1994.

[55] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 392–399, Sydney, Australia, March 1999. IEEE Computer Society Press.

Advanced Transactions in Enterprise JavaBeans

Marek Prochazka

Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering,
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
prochazka@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz

Abstract. Enterprise JavaBeans (EJB) is a new technology that aims at supporting distributed transactional component-based applications written in Java. In recent years, a lot of new advanced software applications have arisen, which have new requirements for transaction processing. Since EJB is modern concept that deals with transactions, the paper discusses the support of EJB for those requirements for advanced transactions and identifies weaknesses of transactions in EJB. The paper also proposes an extension of the current transactional concepts in EJB, which can be a remedy for some of the weaknesses identified. The extension, called Bourgogne transactions, allows a transaction to delegate bean objects to other transactions, to share bean objects with other transactions, and to establish flow control dependencies between transactions. Implementation issues together with pitfalls of the proposed extension are discussed.[1]

1 Introduction

Enterprise JavaBeans (EJB) is an emerging standard for distributed component-based applications written in Java. One of the main goals of EJB is the support of electronic transactions on the Internet. In EJB, distributed flat transactions are supported with no means for supporting long-lived or cooperating transactions. The goal of the paper is to identify weaknesses of the transactions in EJB and to introduce Bourgogne transactions – an extension of the current EJB transactions. The purpose of the extension is not to introduce a new transaction model or to extend EJB with some of the existing transaction models. Instead, we would like to introduce an extension that brings new features to the EJB concept of transactions and allows developers to use a rich set of transaction models.

The paper is organized as follows. What are we missing in the concept of transactions in EJB is discussed in Section 2. An extension of EJB transactions, Bourgogne transactions, is introduced in Section 3 and details of its implementation are discussed in Section 4. An evaluation of the proposed extension is provided in Section 5, where goals

[1] This work is partially supported by the PEPiTA project (the Eureka project number 2033), the Grant Agency of the Czech Republic (project number 201/99/0244), and the High Education Development Fund (project number 1938/2000).

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 215-230, 2001.

achieved are discussed. The related work is discussed in Section 6 and our intentions to the future are described in Section 7. The paper concludes with Section 8.

2 What are We Missing in EJB Transactions

Transactions became a widely-used technique for assuring data stability, application reliability and recoverability. They are used extensively in database systems, where all data manipulation operations are demarcated by transaction boundaries. Current databases use the flat transaction model [9], some use the nested transactions, transactions with savepoints, or other simple transaction models. With expansion of object-oriented and component-based architectures, requirements for instruments guaranteeing data consistency and durability have changed. In this section, the focus is put on the identification of the most important of the new requirements, new applications' aspects, and concepts, and their relation to Enterprise JavaBeans. First, several weaknesses that are common for the majority of existing transactional systems are identified. Second, weaknesses that are specific for EJB are identified and discussed.

2.1 Weaknesses of EJB Transactions Adopted from other Transactional Systems

Although EJB is a relatively young standard, transactions in EJB suffer from many of the weaknesses of the existing transactional systems. In some cases, the reason comes from the fact that EJB have origin in classical transactional concepts, because EJB have to collaborate with legacy applications, databases, and transaction monitors.

Currently, the only supported transaction model in EJB is the distributed flat model. Many concurrent transactions can be executed and each of them is completely isolated from others. Transactions cannot cooperate on underlying database level or on the bean object level. The former is caused by the fact that today's databases do not support sharing resources, defining transaction-specific exceptions from operation conflict table, or other means. The latter is EJB-specific: beans cannot be shared between transactions, except for stateless session beans that do not have identity and thus are transaction-unaware. In EJB, there are no differences between read-only methods and methods that affect data stored in the underlying database. Allowing a transaction to invoke read-only methods on locked bean objects could enhance the application effectivity, since in current EJB all beans are locked until the transaction that had locked them commits or aborts. This, perhaps, comes from an assumption that the majority of enterprise beans will use a traditional database (connected by JDBC connections) as the storage of data.

Transactions are also completely isolated in terms of their lifecycle and flow control. A transaction is unable to change its behavior based on a state of another transaction. For example, it is not possible to mark a transaction abort-dependent on another transaction, such that if the second transaction aborts, the dependent transaction will also abort. If applications are mostly based on transactions, it is desirable to express bindings and dependencies between them. In several papers ([1], [7], [11], [18]), dependencies

between transactions are discussed. There is no reason for not supporting such an extension in EJB.

EJB has no support for long-living or open-ended transactions. A lot of bean objects can be involved in a long-lived transaction, which can reduce system effectivity and throughput, as there is no support for partial rollbacks, early-release locks, savepoints, Chained transactions, or other advanced transaction models, such as Sagas [8], where compensating actions take place. It should be possible to release bean objects during a long transaction execution, or to adopt less restrictive criteria for transaction isolation.

The component model, which is fundamental in EJB, is in fact a model with semantically rich operations. In special cases, EJB 2.0 draft [6] distinguishes between read and write methods: persistent fields of a bean with container-managed persistence are accessible via accessor methods – *setters* and *getters*. However, this concept is not employed for increasing the level of sharing bean objects; EJB container is able to realize if the bean object state has changed or not and thus to optimize storing bean object persistent fields. In our opinion, enterprise beans' methods should be treated as semantically rich operations. For each bean, a method-commutativity table that makes it possible to mark some methods as non-conflicting should be created. This would increase the application's knowledge about a bean, and, consequently, the potential for sharing a particular bean object.

2.2 EJB Specific Weaknesses

Beyond the features partially inherited from the current software applications supporting transactions, EJB have several new features that bring new challenges and are also one of the sources of EJB weaknesses.

There is no explicit locking in EJB. A transaction is not able to acquire locks on selected beans; instead, beans are implicitly locked when they are involved to a transaction.[2] Therefore, locking is provided on a coarse granularity level, and transactions cannot manage their locks according to the applications' requirements.

It would be helpful to allow associating selected methods with a user-defined transaction attribute. This can be beneficial in beans, where the transaction association of a particular method depends mostly on the requirements of the client transaction and beans do not care about the transaction attribute. If a client will, for example, read an account balance and the `Account.getBalance()` method is associated with the TX_REQUIRED transaction attribute, the `Account` bean object will be locked by the container after the `getBalance()` method invocation although it is not necessary. If the TX_NOT_SUPPORTED attribute is used, the client is not able to use the `getBalance()` method in the scope of a transaction, where an atomic execution of several methods takes place. Finally, if the `getBallance()` method is associated with the TX_SUPPORTS transaction attribute, the client is not able not to lock the `Account` bean if the `getBallance()` method is invoked in the scope of the client transaction.

[2] This is not true for stateless session beans, which have no identity, their instances are assigned to a transaction from a pool of instances, and thus their awareness about transactions makes no sense.

Using the current EJB, this can be solved by providing more `getBallanceXXX()` methods, each associated with a particular transaction attribute. A better solution is to allow an association of a method with a set of transaction attributes. The client could dynamically select an attribute appropriate for the actual transaction requirements. Moreover, the set of transaction attributes that can be used to associate with a particular method could be also dynamic. For example, a bean object originally associated with the TX_SUPPORTS transaction attribute could be switched to strictly transactional mode (e.g., because its state becomes a part of a reliable application) by allowing the TX_MANDATORY transaction attribute only.

The fact that the set of transaction attributes is fixed is very limiting. If more transaction models are supported, the set of transaction attributes has to be enlarged. If, for instance, nested transactions will be introduced to EJB in the future, one can imagine the TX_REQUIRES_NEW_NESTED transaction attribute with a similar semantics as the TX_REQUIRES_NEW attribute. If a method invoked in the scope of a client transaction is associated with that attribute, a newly created container-managed transaction is a nested transaction of the client transaction. The method of the transaction context propagation and use of container-managed transactions has to be specified more precisely.

In the EJB 1.1 specification, there are no means for distinguishing between methods changing a particular bean object state and between *read-only* methods. This is one of the reasons why the synchronization on a finer granularity level is not employed. The transaction isolation levels could be put to use for synchronization at the bean level, but specifying transaction isolation levels in the bean deployment descriptor was expunged from EJB 1.1. In the EJB 2.0 specification draft [6], bean accessor methods are denoted as getters or setters, which allows a container to recognize if a bean object state was changed or not and thus to optimize storing bean object states in a persistent store. The finest synchronization granularity can be obtained by considering beans as objects with semantically rich operations. In this case, the commutativity table would be provided for methods of particular beans. If two methods of a bean commute, they can be invoked on a single bean object in the scope of different transaction. If not, the synchronization policy should be applied.

In EJB 2.0, the Java Messaging Service (JMS, [10]) is integrated using special *message-driven beans*, different from session and entity beans. In JMS, sending a message can be a part of a transaction if the `XASession` interface is used, but the transaction context is never transferred to a message receiver. If a bean is specified as using the container-managed transaction demarcation, either the TX_REQUIRED or the TX_NOT_SUPPORTED transaction attributes have to be used. The specification argues that use of other transaction attributes is not meaningful for message-driven beans, because there can be no pre-existing transaction context and no client to handle exceptions. Nevertheless, it could be very useful to allow a client transaction sending transactional requests asynchronously, so the client does not have to wait for results and can obtain return values or potential exceptions later. In this case, the transaction context has to be transferred together with the message. Different JMS delivery modes and transacted sessions should be employed for enforcing transactional message passing. If

the transaction context is passed together with the message, all transaction attributes would be meaningful for message-driven beans.

The EJB specification is not clear regarding multithreaded transactions. The Java Transaction API specification (JTA, [3]) says: "The `UserTransaction.begin()` method starts a global transaction and associate the transaction with the calling thread. The transaction-to-thread association is managed transparently by the Transaction Manager. A thread's transaction context is either null or it refers to a specific global transaction. Multiple threads may concurrently be associated with the same global transaction." Another note says that "some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by JTA." Also, "Depending on the implementation of the application server, different Java threads may be involved with the same `XAResource` object. The resource context and the transaction context may be operated independent of thread context." It is not clear, however, how a newly created thread can be associated with a previously started transaction. Most current implementations do not associate a thread with a transaction in whose scope it is started. Since synchronization is provided implicitly at the level of transactions, it is also not clear how to synchronize threads in the scope of the same transaction.

Transactions distributed to EJB servers provided by different vendors are a natural requirement. Since each vendor provides its own implementation of classes from the `javax.ejb` and `javax.transaction` packages, a programmer has to employ a private classloader for each of the involved EJB servers to isolate the proprietary implementation classes. The transaction context cannot be propagated implicitly in this case and a client has to manage the transaction context propagation to beans deployed to containers of different vendors himself.

3 Bourgogne Transactions

To address some of the weaknesses indicated in the previous section, we propose an extension of today's EJB transaction concepts. The extension focuses on enriching EJB by new transaction primitives that allow using arbitrary transaction models.

This approach stems from ACTA [4], which provides a comprehensive formalism for specifying transaction models. We have adopted the ACTA basic idea that an arbitrary transaction model can be specified using the definition of transaction significant events (such as transaction creation, start, commit, and abort), establishing flow control dependencies between transactions, delegation of resources from a transaction to another transaction, and the definition of criteria for sharing resources between transactions.

The purpose of the paper is also to discuss implementation issues of each of the advanced primitives applied to EJB. Since the EJB specification is tightly related to the JTA specification, many of the newly proposed primitives also affect the mechanisms proposed in JTA. This paper crosses the boundary between EJB and JTA where necessary, but it is more focused on the concept of transactions in EJB and does not discuss the impact of the proposed changes to the JTA specification. The reason for this approach comes from the fact that the newly proposed API uses some primitives from

EJB (e.g., `BeanObject`), but JTA is a universal interface supporting transactions in Java generally. To design a general extension of the JTA specification could be a challenge for another paper.

The newly proposed extension to the EJB transactions is termed *Bourgogne transactions*[3].

3.1 Basic Transaction Primitives: Significant Events

In Bourgogne transactions, basic transaction primitives represent the *begin*, *commit*, and *abort* transaction significant events. In contrast to ACTA, new significant events cannot be defined. More exactly, basic transaction primitives comprise methods from the `javax.transaction.UserTransaction` interface. The semantics of each of the methods can be found in the JTA specification [3].

3.2 Advanced Transaction Primitives

Advanced transaction primitives extend EJB transactions by tools for delegating resources from a transaction to another transaction, sharing resources between transactions, and establishing dependencies between transactions. The advanced transaction primitives include:

Dependencies. A transaction is able to establish a dependency on another transaction. Dependencies are conditional bindings between significant events of the participating transactions. For example, if t_j is *abort-dependent* on t_i (t_j AD t_i), than if t_i aborts than t_j also aborts.

Resource sharing. A transaction can give another transaction permissions to access data that it owns. The permission can be for reading or writing, or a transaction can permit access to parts of its data objects, e.g., by enabling to invoke only some of the objects' operations.

Delegation. A transaction can move data objects associated with it to another transaction, so that the accepting transaction becomes responsible for commit or abort of operations executed before the delegation of the objects.

To simplify the usage of the new primitives, grouping transactions and beans is introduced in our proposal. We introduce the `TransactionSet` and `BeanSet` classes that extend the `AbstractSet` class. Transactions can be added to the `TransactionSet` and beans can be added to the `BeanSet`. These newly introduced

[3] This name is not based on an acronym and it does not have any special meaning. The name originates from the fact that one of the first discussions on such a concept was taken in Bourgogne, France.

classes are used in the definition of the `BourgogneTransaction` interface and make it more user-friendly.

The definition of the proposed Java interfaces can be found in [17]. The new transaction primitives and their implementation in EJB are discussed in detail in the following section.

4 Implementation Issues

The basic idea is to support advanced transaction primitives by providing a new interface inheriting from the `javax.transaction.UserTransaction` interface. The full definition of the newly proposed `BourgogneTransaction` interface is shown in [17]. A client controls the lifecycle of a transaction by means of the `BourgogneTransaction` interface in similar way as the `UserTransaction` interface. In addition, he can use the new transaction primitives thanks to the `createDependency()`, `addPermission()`, and `delegate()` methods of the `BourgogneTransaction` interface. If a bean acts as a client for other beans, it can employ the `BourgogneTransaction` interface for working with advanced bean-managed transactions.

For container-managed transactions, a developer cannot directly use the newly proposed `BourgogneTransaction` interface. Instead, he has to specify the way of use of the Bourgogne transactions advanced primitives in the bean deployment descriptor. Transaction attributes seem insufficient for this purpose, because they indicate only several special types of container-managed transactions. More details on container-managed transactions and transaction attributes are provided in Section 4.4.

4.1 On Dependencies

Establishing dependencies between transactions allows expressing flow control relations between two or more transactions. For example, in the nested transactions, each parent is commit-dependent on all its children and, conversely, all its children are abort-dependent on the parent. The `BourgogneTransaction` interface defines methods for establishing single transaction dependencies on another transaction or a set of transactions:

```
// Create a dependency on a transaction
void createDependency(BourgogneTransaction, depType);

// Create a dependency on a set of transactions
void createDependency(TransactionSet, depType);

// Remove a dependency on a transaction
void removeDependency(BourgogneTransaction, depType);

// Remove a dependency on a set of transactions
void removeDependency(TransactionSet, depType);
```

Similarly, the `BourgogneSet` interface (presented in [17]) defines methods for establishing dependencies of a set of transactions on a single transaction or another set of transactions.

A particular dependency is canceled after it is applied. All the ACTA dependencies are based on two types of dependencies. The first dependency type specifies that an execution of some significant event (i.e., begin, commit, or abort) will force an execution of another significant event. For instance, the abort dependency is of the first type. The second dependency specifies the execution order of two significant events. The commit dependency is an example of this dependency type. Let us name the first dependency type the *enforcing* dependency and the second type the *ordering* dependency.

In EJB, the ordering dependency can be implemented by freezing an execution of the method representing a significant event. On the other hand, the implementation of the enforcing dependency depends on the fact if the commit or abort operations are involved. Basically, abort of a transaction can be enforced immediately by marking the transaction as rollback-only, while a transaction commit cannot be enforced. Thus, the dependency enforcing commit of a transaction has to be ensured by the invalidation of the dependency condition. This is applied in the strong commit dependency: if the `tj` transaction is strong-commit-dependent on the `ti` transaction, the `tj.commit()` method is invoked, and `ti` is aborted, `tj` has to be marked as rollback-only and aborted finally by the transaction manager.[4] The force-commit-on-abort dependency cannot be implemented in some cases: if a transaction is aborted, then commit of the dependent transaction cannot be enforced. In the case that the dependent transaction is aborted, an exception is raised. Potential force-begin-on- dependencies form a new type of dependencies that would be able to start a transaction without the `begin()` method invocation. This is not possible in the current Bourgogne transactions, but we would like to allow this kind of dependencies in the future.

We use a set of constants that represent individual dependencies, but it is more desirable to allow employing a particular dependency by means of handling events occurred. In the future, we would like to specify dependencies by means of registering pre- and post- events of methods which represent transactions' significant events. Each dependency could be specified by the registration of methods reacting to such pre- or post- events. This model could be used for more advanced dependencies, such as dependencies on starting or finishing bean business methods, creating bean instances, or other means. This functionality could be implemented by using the Java Message Service [10].

Compensating transactions can be easily implemented by means of Bourgogne transactions in EJB. We can establish the begin-on-abort dependency between a transaction and a transaction compensating effects of the original transaction. The compensating transaction is started if the original transaction is aborted. Troubles occur if a compensating transaction aborts. This problem can be solved by establishing the force-begin-on-abort compensating transaction dependency on itself. This should be

[4] Note that when a method representing a significant event of a transaction is frozen by the transaction manager, the transaction can be still marked as rollback-only or a new dependency which affects this transaction can be established.

combined either with a policy for a transaction timeout or with a number of allowed attempts to finish the compensating transaction.

Dependencies with container-managed transactions can be hardly established at runtime. The dependency between a container-managed transaction and a client transaction is the only dependency that makes sense if a bean method is called in the scope of the client transaction. The dependency type could be specified in the bean deployment descriptor. More details on container-managed transactions are provided in Section 4.4.

Note that establishing dependencies works correctly with bean-managed transactions. A bean-managed transaction is treated as client transaction, where a bean acts as a client for other beans.

4.2 Sharing Bean Objects

In the `BourgogneTransaction` interface, we define the following methods for giving permissions to another transaction to access bean objects:

```
// Give a transaction permission to invoke a method of
// a bean object
void addPermission(BourgogneTransaction,
          javax.ejb.EnterpriseBean, java.lang.method);

// Give a transaction permission to invoke any method
// of a bean object
void addPermission(BourgogneTransaction,
          javax.ejb.EnterpriseBean);

// Give a transaction permission to invoke any method
// of a set of bean objects
void addPermission(BourgogneTransaction, BeanSet);

// Give a transaction permission to invoke any method
// of any locked bean object
void addPermission(BourgogneTransaction);
```

The `BourgogneTransaction` interface also defines methods for giving permissions to a set of transactions (these methods use the `TransactionSet` class). A transaction can give a permission to another transaction or a set of transactions to invoke a particular bean object method. In the original EJB concept, if a stateful session bean or an entity bean is used in a transaction, another transaction cannot access its methods. Our proposal allows making exceptions to this policy, so some methods of the locked bean objects can be invoked by selected transactions. Permissions can be canceled thanks to `removePermission()` methods (the full listing is in [17]).

A transaction gives a permission to another transaction to allow sharing of its bean objects. For example, if a client uses the `Account` bean object, he can give permissions to all concurrent transactions to invoke the `getBallance()` method. If the permission is given by the client transaction `t`, no conflict occurs if another transaction invokes the

`getBallance()` method. Since `t` still remains responsible for the commitment or abortment of the `Account` bean object, the code of `getBallance()` is effectively executed in the scope of `t`.

Container-managed transactions can give permissions to access bean object methods; this is allowed by extending the deployment descriptor. For example, if a method is associated with the TX_REQUIRES_NEW transaction attribute, it is possible to specify permissions that will be applied to all concurrent transactions. On the other hand, a transaction cannot explicitly give permission to a particular container-managed transaction. A container-managed transaction can obtain a permission from a client transaction in three ways. The client can invoke the `permit(bean, meth)` method, which gives the permission to all concurrent transactions to invoke the `meth()` method on the `bean` bean object. If the client invokes the `permit(bean)` method, he gives the permission to all concurrent transactions to invoke any method of the `bean` bean object. It is also possible to use the `permit(bean1, meth, bean2)` method, which gives the permission to container-managed transactions started during a method invocation of the `bean2` bean object to invoke the `meth()` method of the `bean1` bean object. `permit(bean1, meth1, bean2, meth2)` gives the permission to the container managed transaction started due to the `meth2()` method invocation. Note that giving permissions works correctly with bean-managed transactions.

If an X/Open XA-compliant database is employed as a persistent store through the JDBC connection, giving permission at the enterprise bean object level would lead to the write or read permission in the underlying database. However, this is not supported in today's commercial databases. Setting of the transaction isolation level is the only isolation feature that can be set up there.[5] Isolation levels cannot be employed for giving an explicit permission to access some object (or group of objects) by another transaction.

Note that our strategy for giving permissions has to be changed, if explicit locking primitives will be introduced to EJB. This could be more practical than the current implicit locking mechanism.

4.3 Delegation of Bean Objects

The `BourgogneTransaction` interface defines methods for bean object delegation as follows:

```
// Delegate a bean object to a transaction
void delegate(BourgogneTransaction,
         javax.ejb.EnterpriseBean);

// Delegate a set of bean objects to a transaction
void delegate(BourgogneTransaction, BeanSet);

// Delegate all bean objects to a transaction
void delegate(BourgogneTransaction);
```

[5] The read-only transaction can be considered as a transaction associated with a special isolation level. For instance, Oracle uses READ ONLY as one of its transaction isolation levels [16].

Delegation can be seen as rewriting the computation history: if a bean object is delegated from a transaction to another transaction, the transaction manager behaves like all the operations (methods) executed by the donor transaction were executed by the acceptor transaction. For each transaction, the EJB transaction manager keeps a list of involved bean objects. If a bean object is delegated from a transaction to another transaction, the transaction manager atomically removes the bean object from the list of the donor transaction and adds it to the list of the acceptor transaction. If a group of bean objects is delegated, the same procedure is applied for each bean object from the group.

The bean object delegation cannot be provided if X/Open XA-compliant databases are employed. However, some EJB servers do not support the two-phase commit. In these EJB servers, connection pools based on the `DataSource` class are usually employed. Delegation of bean object works correctly in these EJB servers – more details are provided in [17]. If JDBC 2.0-compliant database drivers that support X/Open XA resources are employed and the EJB server supports the two-phase commit protocol (i.e., the `XADataSource` class is used and database connections are obtained using the `XADataSource.getXAConnection()` method invocation), all local transactions in a the database that work in the scope of one global EJB transaction are associated with the same transaction XID identifier. Thanks to this, if more JDBC connections to the same database are opened in the scope of the same global transaction, they can access the same data items in the database without any conflict. Delegating of bean objects from one transaction to another transaction is just a transfer of the responsibility for the commitment or abortment of operations executed in the scope of the transaction. This leads to moving parts of the database transaction log to another database transaction, which is not possible in today's commercial databases.

A client transaction can delegate its beans to a container-managed transaction by means of the `delegate(bean, delegatedBean)` or `delegate(bean, delegatedBeanSet)` methods, which delegates the `delegatedBean` bean object or the `delegatedBeanSet` set of bean objects to a container-managed transaction started on the `bean` bean object. If there are started more container-managed transactions (this is possible due to eventual permissions given) or no one container-managed transaction is started, an exception is thrown. The client can invoke the `delegate(bean, meth, delegatedBean)` or `delegate(bean, meth, delegatedBeanSet)` methods to specify that the bean object or the set of bean objects are delegated to the container-managed transaction started due to the `meth()` method invocation. A container-managed transaction can delegate its bean objects to the client transaction in which scope it was created. This has to be specified in the bean deployment descriptor.

4.4 On Transaction Attributes

In today's EJB, the way of the transaction context propagation and the way of employing container-managed transactions are specified by transaction attributes. In fact, transaction attributes are very limiting. The set of transaction attributes is fixed, because semantics of the transaction context propagation is defined by setting a value of a bean method's

transaction attribute. If more transaction models will be supported, the set of transaction attributes has to be enlarged. The way of the transaction context propagation and use of container-managed transactions have to be specified more precisely. Generally, the following points have to be addressed:

Transaction context propagation: A method can be executed either with the client transaction context, or the client transaction is suspended. If the client transaction is suspended, a new (container-managed) transaction can be created or the method is not invoked in the scope of any transaction. It also has to be specified, if the client is allowed to invoke a particular method in the scope of a transaction or not. In other words, a set of allowed transactional invocation patterns has to be specified.

Relations between transactions: What is necessary for specifying new transaction attributes corresponding with newly introduced transaction models is to allow using the advanced transaction primitives with the container-managed transactions. For example, the TX_REQURES_NEW_NESTED transaction attribute specifies: if a method associated with this attribute is invoked, then 1) a new transaction is always started and there are no invocation patterns that imply an exception raising, 2) if the method was invoked in the scope of a client transaction, the newly created transaction is abort-dependent on the client (parent) transaction, which is commit-dependent on the newly created (sub)transaction, and 3), at the commit time, all bean objects associated with the (sub)transaction are delegated to the client transaction.

A rich set of transaction models can be used if advanced transaction primitives are applied to the container-managed transactions. Current EJB mixes the two points indicated above and gives names to selected combinations of transaction context propagation features. In our opinion, the deployment descriptor should allow to set the way of the transaction context propagation, raising potential exceptions, creating new container-managed transactions, and setting relations between transactions by means of dependencies, giving permissions, and delegation.

To enhance container-managed transactions by the Bourgogne transaction advanced transaction primitives, the deployment descriptor is extended as follows. In the `container-transaction` section, the `trans-attribute` value specifies the way of the transaction context propagation. Values allowed for transaction attributes remains the same as in the EJB 2.0. If the TX_REQUIRES_NEW transaction attribute is used, an additional information about relation between the client transaction and container-managed transaction can be specified in the deployment descriptor. For other transaction attributes, specifying additional information makes no sense. For bean-managed transactions, the Bourgogne transactions do not allow defining relations between a client transaction and a bean-managed transaction. It cannot be specified, for instance, that a bean-managed transaction, if created, is a child of the client transaction in which scope it is created. In other words, a bean-managed transaction is always independent on the transaction in which scope it is created. This approach prevents situations, where the bean provider employs a top-level bean-managed transaction, but

since the bean-managed transaction is executed in the scope of a client transaction, its semantics can be modified by the application assembler.

For the TX_REQUIRES_NEW attribute, the application assembler has to specify if there are some dependencies between the client transaction and the container-managed transaction, if container managed transaction delegates its bean objects to the client transaction, and which permissions are given by the container-managed transaction in the time of its initiation. The new `client-dependency` and `cmt-dependency` elements of the deployment descriptor specify the client transaction dependency on the container-managed transaction and vice versa. These dependencies are established in the time of the container-managed transaction initiation. The `delegate` element specifies whether the container-managed transaction will delegate all its bean objects to the client transaction in which scope it was created. Other alternatives for delegation by the container-managed transaction are not possible. The application assembler can give a permission to invoke the method associated with the TX_REQURIES_NEW transaction attribute by means of the `permission` element. The allowed values are `ALL`, which gives the permission to all concurrent transactions, and `CLIENT`, which gives the permission to the client transaction.

5 Evaluation

The proposed EJB transaction extension addresses some of the goals listed in Section 2. Concurrent transactions are not completely isolated; they can cooperate by means of sharing bean objects or establishing flow control dependencies between themselves. Sharing is not based on the classical read/write model; instead, it is based on the semantics of bean methods. The granularity of sharing is not set at the level of bean objects, but at the level of beans objects' methods. This is beneficial mainly for increasing the degree of sharing bean objects and thus for increasing the throughput of the EJB server. Establishing dependencies between transactions is a classical concept that allows to express the semantics of the application in terms of involved transactions.

The extension is designed so that long-running activities are supported by the introduced transactional concepts. Bean objects need not be locked during whole transactions. They can be shared with other transactions or they can be released before commit by means of delegation to another transaction, which can be committed or aborted afterward. Moreover, compensating transactions can be defined using the begin-on-abort dependency between the original transaction and the transaction that compensates effects of the original transaction.

The extension does not deal with an explicit locking mechanism. It allows to invoke methods on locked bean objects, but assumes that each bean object participating in a transaction is locked by the transaction manager. In our opinion, a transaction-aware locking service similar to the CORBA Concurrency Control Service [15] should be introduced.

The proposed transaction primitives are used for improved specification of the transaction attributes. The application assembler is allowed to specify the way of the transaction context propagation and also to specify relations between the client

transaction and a potentially created container-managed transaction by means of dependencies, giving permissions, and delegation. The set of transaction attributes is not fixed as in the current EJB; instead, an arbitrary transaction attribute can be specified. On the other hand, our extension does not allow defining "user-defined" or "dynamic" transaction attributes. The proposed extension also does not introduce a concept for asynchronous or queued transactions. This is one of our future intentions.

There is one important limitation of the proposed EJB extension: All the proposed concepts will work with the EJB platform, but some of them cannot be used if EJB applications employ traditional transactional software. Today's databases and transaction monitors support neither transferring resources from one transaction to another nor giving explicit permissions to access locked resources. Perhaps, in the future, the concepts proposed in this paper will be employed to this kind of transactional software.

6 Related Work

Chrysanthis and Ramamrithan in [4] introduce ACTA, a formal framework for specifying extended transaction models. ACTA allows intuitive and precise specification of extended transaction models by characterizing the semantics of interactions between transactions in terms of different dependencies between transactions, and in terms of transaction's effects on data objects. However, ACTA does not focus on the implementation of advanced transaction models in a programming language.

ASSET [1] provides a set of transaction primitives extending a programming language. Beyond the traditional transaction primitives (e.g., `begin`, `commit`, `abort`, `get_parent`) it introduces new primitives allowing creating dependencies between transactions, resource delegation, and giving permissions for an access to acquired resources. However, no implementation based on a real architecture is shown and interfaces of the proposed primitives are not defined precisely. ASSET does not use the object paradigm for the proposed primitives; it rather uses the procedural programming style.

PJama is a clone of the Java programming language that supports object persistency. In [5], the support for customizable transactions in PJama is introduced. PJama introduces the `TransactionShell` class that provides basic transaction primitives. Custom transaction models could be provided using inheritance from the `TransactionShell` class, overloading its methods, or introducing new methods representing more advanced transaction primitives. A developer of a new transaction model can use the `LockingCapability` class for the support of ignoring conflicts between transactions, delegation of responsibility for locks, and notification of a conflict detection. The way of creating an arbitrary transaction model, however, is not fairly clear. Note that PJama has no relation to the JTS or JTA specifications.

7 Future Intentions

For the future, we plan to design our idea on container-managed transactions more specifically. We would like to propose the way of specifying an arbitrary transaction attribute in the deployment descriptor. Also, "user-defined" and "dynamic" transaction attributes should be introduced. We would like to develop creating dependencies between transactions by means of registering pre- and post- events. We would like to introduce force-begin-on- dependencies that could start a transaction without the `begin()` method invocation. We also plan to employ asynchronous transactions, which should allow to transfer the context of a transaction by sending a message from a client to a bean object. For last but not least, we plan to develop a prototype of the transaction service supporting the proposed extended transactional functionality. For this purpose, we would like to take advantage of an open-source EJB server implementation.

8 Conclusion

In this paper, we introduce an extension to the concept of transactions in Enterprise JavaBeans. We identify several weaknesses of the current transactions in EJB. Our extension, called Bourgogne transactions, solves most of the weaknesses that we have identified. Bourgogne transactions allow to employ advanced transaction models to EJB. They introduce new transactional primitives allowing to establish flow control dependencies between transactions, to delegate bean objects from a transaction to another transaction, and to give permissions to access bean objects locked by a transaction. Implementation details of the proposed extension and the impact to the concept of EJB transactions are also discussed.

References

1. Biliris, A., Dar, S., Gehani, N. H., Jagadish, H. V., Ramamritham, K.: ASSET: A System for Supporting Extended Transactions. Proceedings of ACM SIGMOD International Conference on Management of Data (May, 1994)
2. Cheung, S.: Java Transaction Service 1.0 Specification. Sun Microsystems Inc. (December 1, 1999)
3. Cheung, S., Matena, V.: Java Transaction API 1.01 Specification. Sun Microsystems Inc. (April 29, 1999)
4. Chrysanthis, P. K.: ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis (September, 1991)
5. Daynès, L., Atkinson, M. P., Valduriez, P.: Customizable Concurrency Control for Persistent Java. In Advanced Transaction Models and Architectures, Editors: S. Jajodia, L. Kerschberg (August, 1997)
6. DeMichiel, L. G., Yalçinalp, L. Ü, Krishnan, S.: Enterprise JavaBeans Specification 2.0 Draft 2. Sun Microsystems Inc. (September 11, 2000)

7. Elmagarmid, A. K.: Database Transaction Models For Advanced Applications. Morgan Kaufmann (1992)
8. Garcia-Molina, H., Salem, K.: Sagas. In proceedings of the ACM SIGMOD Conference, 1987
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
10. Hapner, M., Burridge, R., Sharma, R.: Java Message Service API Specification 1.02, Sun Microsystems Inc. (November 9, 1999)
11. Jajodia, S., Kerchsberg, L.: Advanced Transaction Models and Architectures. Kluwer (1997)
12. Matena, V., Hapner, M.: Enterprise Java Beans Specification 1.0. Sun Microsystems Inc. (March, 1998)
13. Matena, V., Hapner, M.: Enterprise Java Beans Specification 1.1 Public Release. Sun Microsystems Inc. (August 10, 1999)
14. Object Management Group: Object Transaction Service (December, 1997)
15. Object Management Group: Concurrency Control Service (December, 1997)
16. Oracle Corporation: Oracle8 Concepts, Release 8.0, Part No. A58227-01 (December, 1997)
17. Prochazka, M.: Extending Transactions in Enterprise JavaBeans. Technical Report 3/2000, Department of Software Engineering, Charles University, Prague (May, 2000)
18. Wachter, H., Reuter, A.: The ConTract Model. In Ahmed K. Elmagarmid: Database Transaction Models for Advanced Applications (1991)
19. White, S., Hapner, M.: JDBC 2.1 API. Sun Microsystems Inc. (October 5, 1999)
20. White, S., Hapner, M.: JDBC 2.0 Standard Extension API. Sun Microsystems Inc. (December 7, 1998)
21. Yang, J., Kaiser, G. E.: JPernLite: Extensible Transaction Services for WWW. CUCS-009-98, Department of Computer Science, Columbia University (1998)
22. X/Open Distributed Transaction Processing: The XA Specification (1991)

Service Integration

Michael Goedicke

Specification of Software Systems, University of Essen, Germany
goedicke@informatik.uni-essen.de

1 Introduction

Service integration tries to tackle the problem to deploy new services using existing legacy components in a network centric environment. Network centric was considered here the Internet with the set of protocols supporting the World Wide Web. Usually the legacy components were developed with the assumption of a more closed non-distributed world. Especially they were originally not planned for use in the web.

The availability of new enabling technology like object oriented middleware, meta data approaches and their standardization by XML and component concepts provide a new base one can use and rethink the problem field of service integration. Basically this problem was already addressed quite often in different more limited fields – e.g. case tools integration, DBMS integration – to mention only a few. Given all this new technology service integration still raises a number of interesting problems while the new technology base allows new and easier solutions than before.

First of all one wants to create new services on the web not just only to bring an isolated (legacy) component to the web. Thus it is necessary to describe services and provide means to combine existing services to form new ones. Then there is the set of problems, which are created by the distributed nature of the system created by a particular integrated set of components. This means, services have to be made known and accessible by potential users. This is addressed by mechanisms to register and locate services. Last but not least is the problem to integrate different views on the data and control used by the different services of the (legacy) components.

In the session two complimentary papers were presented and discussed:

"Customizable Service Integration in Web-Enabled Environments" by Richard Gregory and Kostas Kontogiannis and the paper "Migrating and Specifying Services for Web Integration" by Ying Zou and Kostas Kontogiannis.

The two papers in this session provided a complimentary discussion of the aspects of service integration. In summary a lightweight approach to distributed service integration was proposed.

The general organization of the session was a presentation of the key aspects of the service integration problem by the authors of the papers in an interactive way which included questions and contributions of the audience.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 231–234, 2001.

Generally the issues discussed where centered around a number of questions regarding the particular approach the authors have developed in a co-operation between the group at University at Waterloo and the IBM Lab in Toronto.

2 What is Service Integration: Issues and Problems?

The particular aim of the presented approach was to provide a lightweight way to flexible service integration based on legacy components. Although in the examples and case studies addressed in the authors' work the source code of the legacy components was available potentially it was the (unstated) intention not to change those components too much. In broad terms the following primary areas were addressed:

- Legacy system migration
- Control integration
- Service/component registration
- Data integration

The general approach to deal with these issues can be organized in the following way:

Service description: It is necessary to specify a service offered either indirectly only to other components or by a (web-based) user interface. The proposed solution is based on an XML based language that specifies general properties. These properties are important for transparent service localization due to the distributed nature of the "new" services. In addition the service specification is used to specify the interface service properties. This is necessary to provide a transparent way of service invocation.

Service registration: In large distributed software systems there is the need to find a particular partially specified service. The necessary precondition to find a service is to make all available service "publicly known" at some place. In the proposed approach the solution to this problem is achieved in a standard way through a central repository.

Service localization: More difficult and thus interesting is the problem how to find a required service. In particular approaches are needed to find a service in the case that the required properties are only partially specified. In the proposed approach it is done through matching of general service properties stored in the repository with required properties of the services sought.
In addition to these questions two other questions regarding integration appear in this setting. Usually the legacy components considered for the integration have their own distinct model of data and control. In particular often the transaction concepts and database schemata have to be integrated such that a combined service can be realized.

Control integration: In the proposed solution an approach based on event condition action rules (ECA) was used. It was considered a main contribution to a flexible way to the control integration problem the ECA rules were easy

to customize in the setting of a combination of specific – possibly different – business transaction logics and workflows. Concepts like these are similar to those often found in active Databases.

Data integration: In addition to the problem to integrate the control flow models of the different components to be integration it is necessary to bridge the gap between the various data models of those components as well. Here the use of XML to normalize data used by different sources was realized successfully.

In the discussion during and after the presentations a number of problems were raised and discussed. Clearly a number of advantages were recognized and realized using the enabling technologies (CORBA, EJBs etc.) Also it was clearly marked that the entire work needs to be evaluated using case studies on a larger scale. However, important goals could be reached, such as an open architecture and flexible integration by the ECA-rules approach. In the implementation and also for the conceptual design in many places technology was used based on XML. One important result was that XML-based tools and approaches are a key to easy and open integration.

Some problems were also discussed. Mainly these were obviously due to the early stage of the work and the lack of large-scale case studies.

- Rule sets can be become very complex. Providing an additional integration layer on top of modularized components might add an opaque layer on top of the integrated components. As a consequence this layer might become large and not easy to comprehend in larger and complex cases. Thus tools are needed to debug, analyze, and modularize the rule-set.
- Also it was discussed whether the rule-based approach is a good choice as other approaches to component integration e.g. based on scripting (c.f. CORBA 3) are available as well.
- Service description language is not based on formal model. A formalization of the rule-language might be instrumental to achieve the desirable level of tool-based analysis of the rule set. It also might help to achieve deeper insight into problems like ovelap between rules, cyclic rule dependencies etc.
- Quality of Service (QoS) was particularly identified as a missing point in the rule language. Thus the service description language should take into account how QoS can be expressed and how at run-time this information can be monitored.
- In the control integration problem area a particular problem arises if the system has no control over transactions initiated outside the scope of the ECA rules. Such a situation is especially difficult to handle if the components cannot be changed to introduce new ways to supervise a (legacy) component.
- Based on an addition to the rule-language to allow to express the degree of QoS, new ways to select a service from a set of possible services can be bases on more elaborated approaches like alternative trading approaches.

In summary the work presented showed a nice and lightweight way to component and thus service integration. It made also impressively clear how the

XML-standard helps to deliver a flexible and very general solution to the problem. This in particular shows how XML might influence the work in distributed system design and software engineering/software architecture in general in the next few years.

Acknowledgments

This summary is based in part on the comments of the following workshop attendees, who contributed to the discussion during this session: Wolfgang Emmerich, Alfonso Fuggetta, Michael Goedicke, Richard Gregory, Kostas Kontogiannis, Cecilia Mascolo, Walter Schwarz and Stefan Tai.

Customizable Service Integration in Web-Enabled Environments*

Kostas Kontogiannis[1] and Richard Gregory[2]

[1] Department of Electrical and Computer Engineering,
University of Waterloo
[2] IBM Toronto Lab,
Toronto, Canada

Abstract. In recent years we have been experiencing a tremendous change in software development processes, where new systems are built by utilizing distributed, possibly heterogeneous, components. In this paper, we propose an infrastructure and a meta programming environment that allows for distributed components to be integrated, in a fully customizable manner, into Web-enabled environments. In particular, we propose an architecture that conforms to the event-condition-action paradigm. A set of event-condition-action rules combined with a rule enactment engine serves as a driver that determines the transaction logic by which remote services are invoked. A prototype system using the proposed architecture applied to the domain of e-commerce is also presented.

1 Introduction

Over the past decade, Web browser technology has revolutionized the way the Internet is utilized for gathering and presenting information to users. With the emergence of distributed object technologies and new programming languages, the Internet is now moving from a worldwide information pool towards a service providing facility.

The convergence of the Internet and distributed-object technologies extends this "information-based" Internet to a worldwide "services-based" Web. This evolution is referred to as the Internet's second-wave, where software services and content are distributed openly over the Internet, corporate intranets , and extranets [1].

As the availability of data and software components on the Web increases, services can be accessed through unique addresses and run as processes that are dynamically executed on a server at the request of arbitrary clients. Furthermore, existing legacy systems, as well as new software systems, are conforming to tighter requirements such as interoperability, flexibility, customizability, as business processes are continuously reengineered. Consequently, content (data) and

* This work was funded by the IBM Canada Ltd. Laboratory - Centre for Advanced Studies (Toronto), the Centre for Information Technology of Ontario (CITO), and the Institute for Robotics and Intelligent Systems (IRIS).

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 235–252, 2001.

software components, located virtually anywhere in the world, can be combined on an as-required basis, thus forming collaborative information systems.

We present a system architecture where a meta integration language, encoded in XML, determines the manner in which existing software applications interact. This language allows Event-Condition-Action (ECA) rules [2] to encode the transaction logic by which processes interact. As new services are added to the application system, or as business processes change, all that is required are changes to the rules that encode the specific transaction logic. Moreover, these changes require relatively little effort to implement, so the behavior of the system can be customized to meet new requirements. This is in contrast to existing technologies such as CORBA [3] or Enterprise JavaBeans™ [4] where considerable knowledge is necessary when the interaction of distributed components must be changed.

In this context, several issues must be addressed that relate not only to communication between processes but also to interactions that occur between formerly independent software applications. Also, since we cannot assume all enterprises will adopt one single architectural standard, open communication with other systems is a strong requirement. The work presented in this paper provides such an open architecture, and builds on previous work presented in [5].

This paper is structured as follows. In Section 2, we highlight related work. In Section 3 we describe the basic architecture and the system's components. Section 4 gives a detailed look at our proposed rule language including a simple example. Following that, in Section 5 we describe a simple prototype that we have implemented using Enterprise JavaBeans and WebSphere [6] by IBM. Finally, Section 6 outlines our future plans for the project and concludes the paper.

2 Related Work

Our system has its roots in CoopWARE [5], a generic data and control integration environment applied to the reverse engineering domain. In CoopWARE, program coordination is also facilitated by a set of rules and an event-driven rule execution mechanism. In this paper we extend the architecture and scope of CoopWARE with respect to a new generic architecture modeled for Web environments.

In [7] a generic architecture that also utilizes the Event-Condition-Action (ECA) framework for managing Web-based applications, is proposed. The major difference between the work presented in [7] and this paper is that we focus mostly on control integration aspects (process invocation and termination) whereas CoopWARE focuses mostly on data integration aspects, and integration of the semantic content provided by various components in a Web-based cooperative system. With our rule and task enactment engine, rule encoding and service invocation are also greatly simplified.

The ToolBus [8] is a system designed to control interactions between software components. Direct inter-tool interactions are not supported in the ToolBus and are instead controlled by a script based on process algebra that formalizes all

the desired interactions among tools. Although service interaction is abstracted by the process algebra, modifications must still be carried out by skilled programmers.

C3DS [9], Control and Coordination of Complex Distributed Services, is a project whose goal is "to exploit distributed object technology to create a framework for complex service provisioning." Like our system, C3DS aims to provide a framework where new services can be composed from existing ones, as well as to facilitate dynamic control over service interaction. Another goal it has in common with our own is to provide an integration environment that is suitable for non-programmers.

Finally, Jini™ [10] Connection Technology is an evolving standard under development by Sun Microsystems. Although it is concerned with device connection, many of the ideas, such as communication between heterogeneous systems and ease of connection also apply to the work presented in this paper.

3 System Architecture

In this section, we describe the overall architecture of the proposed system. We also discuss the way in which individual system components interact with each other and with their operating environment.

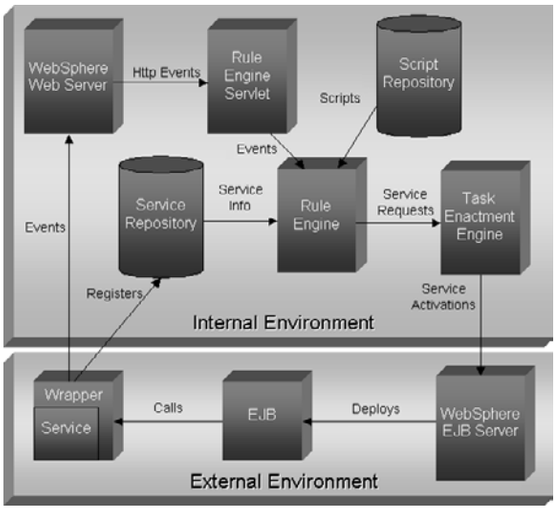


Fig. 1. The System Architecture

Figure 1 illustrates a high-level view of the proposed system architecture. Specifically, a Web server intercepts events (for example, those that are sent at the termination of a service) and forwards them to the rule engine. The rule engine enacts Event-Condition-Action (ECA) scripts to determine whether a

service can be invoked (i.e. the conditions in an ECA rule are satisfied). This component is contained within a servlet that is plugged into WebSphere [6] and is equivalent to a monitor process.

The rule engine forwards any service requests to the task enactment engine. This component determines how the service is to be invoked based on information provided by the service repository. The service repository relates names of services, interface descriptions, IP addresses, ports, and URLs. A container that is capable of deploying Enterprise JavaBeans (EJB) objects is used for invoking the service (the container may be part of the same Web server that captured the original event, but it would normally exist at another location). An EJB is used to call a service that we wish to execute.

The service itself is contained within a wrapper that allows it to be used within the proposed architecture. This wrapper accepts calls from an EJB and it knows where and how to return any events that are sent back by the service to the system. The external environment may belong to one or more instances of our system. That is, a single service may be be registered (recorded in a service repository) with any number of service integration systems.

3.1 Web Server and Events

Events are implemented as HTTP requests that are intercepted by the Web server and are initiated by services. Events are encoded as strings of XML-formatted data that contain event details such as its name, type (useful when a service is to be invoked as a result of any event of some particular class), sender, parameter values, and any other pertinent information. This approach was chosen since a Web server can be easily used to capture events as they are sent across a network. A servlet plugged into the Web server can then process these events. The implementation of the proposed prototype is built on top of the WebSphere server developed by IBM [6].

In addition to capturing incoming events, WebSphere also provides an infrastructure that simplifies the invocation of remote services since it is capable of deploying EJBs. These beans can dynamically load classes that contain or wrap the desired remote services. We rely on EJBs, rather than directly loading remote classes, since EJBs can execute remotely. These are then able to invoke other non-Java components, such as those obtained from legacy systems that can only run on certain platforms and operating environments. For each site containing a service, an instance of the Web server (or, at least, an EJB container) must exist, and for each service an EJB must be deployed.

Note that we could have used an implementation of CORBA [3] (in fact, WebSphere provides an implementation of CORBA) or even Java servlets. However, we found that the EJBs simplify many tasks and provide a higher level of abstraction than using an Object Request Broker (ORB). Java servlets, at the other extreme lack many features found in EJBs such as transaction management and persistence mechanisms.

3.2 ECA Rule Engine

At the heart of the system architecture is the component that allows for process transaction logic to be encoded and enacted by a collection of event-condition-action rules and a forward chaining inferencing engine. This engine is what allows us to achieve the desired level of customizability and flexibility.

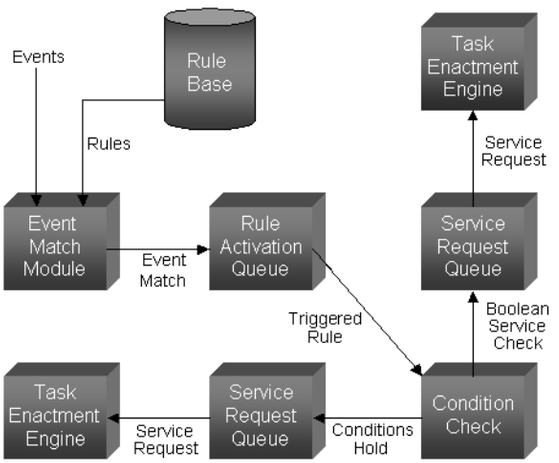


Fig. 2. The Rule Engine Module

The rule engine is implemented as a servlet that accepts incoming events by the Web server. These events may cause a number of rules to be activated. For each rule triggered, a condition clause is checked and, if it is satisfied, the respective rule actions (i.e. service names) are passed to the task enactment engine and invoked (see Figure 2).

As new services are added to the system, new rules can be written to control the invocation of these services and capture any new events they may send. Furthermore, any changes to business processes involving existing services can easily be reflected in the system by simply modifying or adding new rules.

The ECA rules are encoded using XML and are stored in the rule base along with the DTD that corresponds to the rule language grammar [11]. ECA rules are parsed by using IBM's XML4J parser [12]. This parser also provides an API to access the resulting DOM tree (an internal representation of the parsed XML document, analogous to a program's abstract syntax tree).

3.3 Services

The actions in an ECA rule typically involve invoking a service. A service is defined informally as a method or program that relates to a process and can run either locally or remotely. Upon completion, a service may return results to

the rule engine. The results come as parameters that are enclosed within a new event that is sent by the service (for example, see Figure 8).

In a basic case, when a service is invoked, it may simply return a value to the system. In more complex cases, a service may have a state, and subsequent calls to that service may require that the same instance of the service be invoked. As an example, consider an application where there is user interaction. The application might send an event that eventually triggers a rule that invokes one of the application's display methods (the display method would be a service). It is important that the same instance of that application be invoked as the display service and not some other instance (we want to results to come back to where some request originated). These are described in [13] as interactive services.

In general, a service can be invoked several times, where each invocation is dependent on earlier invocations. We use session identifiers to distinguish between events that may belong to different transactions. We will explain these identifiers further in Section 4.5.

A typical service is the functionality that is provided by some component. Components can often be obtained from legacy systems using reverse engineering techniques [14] or from software modules built specifically for a given application. Strategies and techniques to automate the decomposition and re-engineering of legacy systems into modules for migration to network-centric environments have been presented in [15] and [16].

Once services are identified, they can be encapsulated by a wrapper class and be used within the integration architecture. The finer the granularity of the components (i.e. the higher the cohesion), the more the flexibility that can be provided when the individual components are wrapped. For example, at one extreme, a legacy system can be wrapped as a whole, providing thus a monolithic, but inflexible, service. On the other extreme, the legacy system can be decomposed into small subsystems that correspond to implementations of Abstract Data Types, or highly cohesive components that deliver specific functionality. For maximum flexibility, components (especially the ones extracted from existing legacy systems) should provide a spectrum of related services as opposed to a single service, and have a well defined interface with the rest of their operating environment [17].

No matter how the services are obtained (by new development or by components extracted from legacy systems), an essential requirement is that these services are able to exist anywhere on a network or the Internet. Usually they run on the system where they reside and may be implemented in any programming language. This is in contrast to other network-centric paradigms whereby code is downloaded and executed on the thin-client side.

As we will see in Section 4, it is not necessary for a person composing the rules to be concerned with the details of where the service is located, or how it is to be invoked. It is up to the system to determine this. Also, new services may be added to the system at any time. We are working on simplifying and even automating the registration process. In particular, on-going work [18] involves the design of mobile agents that locate services that are available to register

with the system, based on the type and interface description of these services. A sample component, and its XML interface description, are illustrated in Figure 3 and Figure 4 respectively.

```
public Class BookItem {
  private:
    char* title;
    char* barcode;
  public:
    void  setTitle(char *title);
    char* getTitle();
    void  setBarCode(char* barcode);
    char* getBarCode();
    void setAuthor(char* author);
    char* getAuthor();
}
```

Fig. 3. Example Service Interfaces for the `BookItem`

Services are registered in the proposed system by adding an entry to the service repository. This repository is used by the system to determine, in a way that is transparent to the client processes, where services are located and how they are to be invoked. It is also used to assist in composing new ECA rules by providing information such as which events may appear and which services are available. An XML DTD can optionally be registered along with any service to assist in data integration [7] [19]. The repository is currently implemented as a static table, but work is in progress to implement it as a built-in service that will handle service registration events.

One advantage of the proposed architecture is that it simplifies the transaction logic between diverse systems that do not use the ECA approach. A service that integrates data represented as XML files and can be used to translate ECA events and parameters into a format that can be used by other systems is presented in [19], [20]. A similar service can also translate data arriving from other systems into ECA events. This addresses the problem of differing data formats between related services. Since most services correspond to components obtained from diverse legacy systems, only few will share a common data format even though one service may be required to provide data to another service. We believe that as standards for data interchange are developed [21], [22], these adapters will be standardized as well.

4 The ECA Scripting Language

This section presents an overview of the scripting language used for modeling the Event-Condition-Action (ECA) rules. The ECA rules are encoded in XML and are composed of three parts as described in the following sub-sections. Other

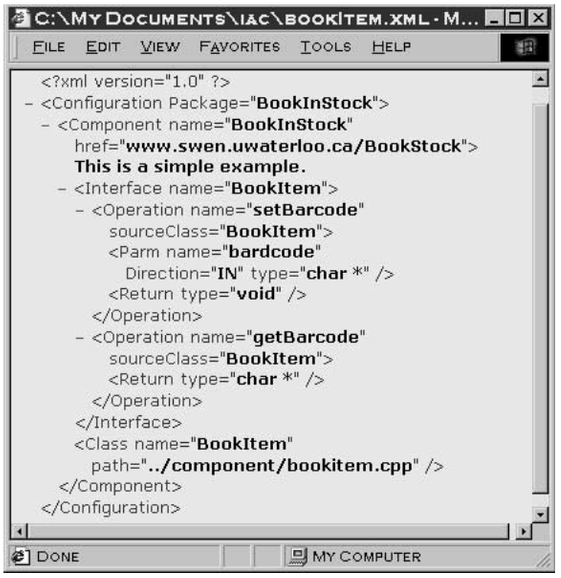


Fig. 4. A Sample Interface Description

language features are described at the end of this section. An example of a simplified ECA rule script is illustrated in Figure 5. This sample script is taken from an experiment that is described in Section 5.

4.1 Types and Declarations

An ECA rule is given a name attribute so that it can be distinguished from other rules. This allows for the removal or replacement of rules at runtime and is useful for debugging purposes. A rule begins with variable declarations consisting of an identifier and type. Values for these are set within an event clause and can be used in the condition and action components of the same rule. For example, in Figure 5, the variable `CustID` will have it's value set when a `CheckoutCar` event is received. This value is stored in the DOM tree and will be used when the `CheckAccount` service is invoked as part of the condition check. If the actions are executed, the value will be passed to the `PlaceOrder` service. Although not shown in the example, it is possible to compose complex data types to avoid repeated use of long parameter lists. Constants may also be defined as an added convenience.

Both complex types and constants may be defined in a global scope, in which case they may be used in any rule contained within the script. These types and constants may also be used by service definitions within the service repository. Conversely, types, constants, and the service names that are declared in service definitions may also be used within an ECA script. To disambiguate definitions,

```
<ECARule name="Checkout Cart">
  <Declarations> <Variable identifier = "CustID">
      <Type name = "Integer"/> </Variable>
    <Variable identifier = "CDs">
      <Type name="CDList"/></Variable></Declarations>
  <Events> <EventExpr>
    <Event name = "CheckoutCart">
      <SetVariables>
        <Identifier name = "CustID"/>
        <Identifier name = "CDs"/> </SetVariables>
    </Event> </EventExpr> </Events>
  <Conditions> <ConditionExpr> <Condition>
    <Service name = "CheckItemsInStock">
      <Class name = "IsItemsInStock"/>
        <UseVariable> <Identifier name = "CDs"/>
    </UseVariable> </Service> </Condition>
    <AND/> <Condition>
      <Service name = "CheckAccount">
        <Class name = "IsAccountInGoodStanding"/>
          <UseVariable> <Identifier name = "CustId"/>
          </UseVariable> </Service>
      </Condition> </ConditionExpr> </Conditions>
  <Actions> <Service name = "PlaceOrder">
    <Class name = "PlaceOrder"/> <UseVariable>
        <Identifier name = "CustId"/> </UseVariable>
      <UseVariable>
        <Identifier name = "CDs"/> </UseVariable>
    </Service> </Actions> </ECARule>
```

Fig. 5. A Sample ECA Rule

the use of packages are employed in a manner very similar to those used in the Java programming language.

Global variables may also be declared. These, however, may cause more problems than the convenience they provide, since their value may be non-deterministically set at any time by events received as part of other rules. To avoid this problem (to some extent), we have allowed one type of global variable that can only be set within one rule, but can still be used within other rules. There is still a possibility of it not being defined at other points of use, but an extra condition can be added to check this case. For in depth explanations and examples of variable and type declarations, as well as the use of packages and other language features, refer to [11] and [13].

4.2 Events

An *event* component appears in a rule following the declarations. This is an expression composed of named events that are separated by *and* or *or* connectives. An event expression must be satisfied for a rule to be triggered. In a

simple case, such as in Figure 5, the expression consists of a single event. As an example of a more complex case, if an event clause contained the expression $Event1 \vee (Event2 \wedge Event3)$, the rule would trigger (depending on the parameters received, as discussed next) when either $Event2$ and $Event3$ were received or when $Event1$ was received.

An event that is received may include parameters whose values will be bound to the respective identifiers. In the example in Figure 5, if a $CheckoutCart$ event is received, the first parameter value will be bound to $CustID$ and the second to CDs. Similar to function overloading, if the number and type of parameters received does not match one event in a rule, it may match another. An event may also have a type, which means that, if an event exists in a rule where a type is specified instead of a name, this could also cause the rule to trigger (assuming there is a parameter list match and any other needed events are received). This allows rules to be triggered whenever any of a general class of events is received.

Notice that the complex example above may present a problem similar to that of serializability in the database field. If several $Event2$ and $Event3$ events are received, the number of times the rule will trigger depends on the order in which those events are received. Consider the case where two $Event2$s are received, followed by two $Event3$s. The rule would trigger once and, if we discarded the second $Event2$, the rule would not trigger again until a third $Event2$ was received. Therefore, we keep a queue of events such that, in our example, when the second $Event3$ is received, the rule will trigger for a second time (due to an $Event2$ having been in the queue).

However, given that there may be different parameter values associated with each event, there may still be different effects depending on the order of event receipt. The combination of one of the $Event2$s with one of the $Event3$s that triggers the rule is generally non-deterministic. This must be kept in mind when composing ECA rules.

When a rule is triggered, it is important that all variables be bound. Therefore, where there is an *or* expression, any variables that appear on the right hand side must appear on the left hand side. Conversely, where there is an *and* expression, any variables that appear on the right hand side must not appear on the left. To allow otherwise would introduce ambiguity since variable binding would occur twice.

4.3 Conditions

The next component of an ECA rule is the condition clause, which is a Boolean expression using the logical connectives *and*, *or*, and *not*. A predicate corresponds to either a service that returns a Boolean value or a test on the values that were bound to the event parameters. In the example of Figure 5, the conditions would be satisfied if the $CheckItemsInStock$ and $CheckAccount$ services both returned true. The values that were received with the *CheckoutCart* event will be passed to these services. Of course, more rules may be necessary to handle the case where either of these services returns false.

Services that can be used as condition checks must be registered as such in the service repository. This is so that the rule engine knows that the return value belongs to the condition check and is not just another event. Boolean services are invoked, and their values returned, in the same manner as is described in the next section.

Another use of the condition component is to perform a check on the values of variables. We have defined various tags that allow operators, such as *equals* and *greater than*, to be placed between constants and variables within a condition expression. An in-depth example of a condition component is provided in both [11] and [13].

If the entire condition clause of a rule is satisfied, the action component of the rule is executed as described next.

4.4 Actions

The final ECA rule component is the action component and it encodes instructions for the invocation of a sequence of remote services. The services may all be invoked simultaneously or sequentially. Services that are named in a triggered rule are passed to the task enactment engine which manages actual service invocation.

If a service name is specified, as is the case in Figure 5, the system will look for that name in the service repository (it will look in the packages that are specified) and use the recorded information such as host name and port number to call the EJB that performs the service. Future extensions of our work would likely employ a robust directory and naming service, such as JNDI [23], to locate the services. If a service type is given instead of a specific name, the system will invoke any service that matches that type.

Each service specified in the actions component may also pass the values of variables as parameters to services. At runtime, any session identifier that was received with an event from an interactive service (as was described in Section 3.3, is also passed to the service, so that it can pass it back to let the system determine the context of the result. With our current system, an event component cannot contain an expression with events from two different interactive services, since there can only be one session identifier.

On-going work is focusing on developing a mechanism that allows for the localization and selection of remote services by the task enactment engine. This can be based on criteria related to the current load of the server on which the service is located, the communication latency, and the performance rating of the service in terms of speed, accuracy, and cost [18].

4.5 Concurrency Issues

Each event that is sent to the system comes through WebSphere and this results in a new thread of the rule engine being created. This means that more than one rule can be active at any given time as part of different sessions. Consequently,

more than one service may be instantiated at any given time. Therefore to deal with concurrency problems related to multiple instantiations of any given service, we have included the concept of sessions within the ECA rule language. For example, if service S is invoked, and we are interested in the return value from S in another rule, we need to know whether the return value we received is the same one we were expecting. After all, we may have received the value from some other invocation of S.

Session identifiers can be used to address this problem. When an application sends an event that begins the chain of rule invocations (such as an interactive service), the rule engine includes a unique identifier with that event's parameters and passes it to each service that is invoked. This is not seen in Figure 5, since it is strictly a runtime parameter. Each service that receives a session identifier passes it back in addition to the parameters as seen in the example. Note that these identifiers are not written into the ECA scripts, they are added by the system at runtime.

Other concurrency issues, resulting from many services accessing the same data resources, are effectively handled by the implementation offered by EJBs or by the back-end services themselves (i.e. database management systems such as UDB/DB2 Data Base Management System). For other services that do not offer concurrency control mechanisms, such algorithms can be implemented separately from the integration architecture proposed in this paper [24]. In particular, we would like to maintain atomicity in all transactions that occur as a result of service invocations. For the prototype discussed here, the atomicity, consistency, isolation, and durability issues for the operations that result from the invocation of remote services are addressed by the back-end services themselves, while the message delivery guarantees are handled by the EJBs and the CORBA protocol which provides at-most-once-semantics.

Fig. 6. The Home Page

```
<ECARule name="CDArtistQuery">
  <Declarations>
    <Variable identifier = "Artist">
      <Type name = "String"/>
    </Variable>
  </Declarations>
  <Events>
    <EventExpr>
      <Event name = "ArtistCDListRequest"
         sessionId = "CDArtistQuery:10000">
        <SetVariables>
          <Identifier name = "Artist"
                      value = "Rush"/>
        </SetVariables>
  </Event> </EventExpr> </Events>
  <Conditions> </Conditions>
  <Actions>
    <Service name = "RequestArtistList">
      <Class name = "SQLService"
             sessionID = "CDArtistQuery:10000"/>
      <UseVariable>
        <Identifier name = "Artist"
                    value = "Rush"/>
      </UseVariable>
    </Service>
  </Actions>
</ECARule>
```

Fig. 7. A Query Rule at Runtime

5 An E-Commerce Application Scenario

To demonstrate the applicability of the proposed system, we present its use in deploying an e-commerce on-line system. The system implements the functionality offered by a virtual CD store where, users are able to order CDs using a Web browser. This section presents how the proposed architecture is used.

On the surface, the system is similar to many existing on-line ordering systems. A typical session may run as follows: a user connects to the CD Online web page, enters the name of a recording artist, obtains a list of available CDs from that artist, then selects one CD from the list. Any number of CDs may be added to a shopping cart, and the customer can fill out an information form, specify a payment method, and have the CDs shipped to a given address. Also similar to many existing systems, our system is connected to a database and other systems such as inventory and accounting.

Where our system begins to differ from familiar on-line systems is the underlying architecture, its rule engine, and the customizability it offers. For example, it took only a few hours to customize the environment from another e-commerce

```
<Event name = "ReturnedRequestArtistList"
       sessionId = "CDArtistQuery:10000">
  <Param name = "Results"
      type = "XMLString"
      value="<Artist>Rush</Artist>
        <Albums>
          <Album>Moving Pictures</Album>
          <Year>1981</Year>
          <Album>Permanent Waves</Album>
          <Year>1980</Year>
          <Album>Hemispheres</Album>
          <Year>1978"</Year>"
       </Albums>"/>
</Event>
```

Fig. 8. An Event Resulting from the Completion of the `RequestArtistList` Service

application for selecting and purchasing auto-parts instead of CDs [25]. Pricing information, availability of goods, and special offers are all handled by the back-end service which in our case was IBM's Universal Database UDB/DB2.

In all cases, incoming HTTP requests are intercepted by a servlet that is implemented also as a service. This service transforms the request into an event and sends it to the rule engine. In the sample scenario, actions are programs that encapsulate SQL queries. The parameter values that will be used (i.e. artist name, song title) are those that are filled by the client as he or she fills the on-line forms on the client machine. Note that the Web browser is an interactive service where a new session identifier is created every time a form is submitted.

A snapshot trace of the system in operation proceeds as follows. The user visits the virtual store Web page and selects the artist as illustrated in Figure 6. What is sent back to the Web server when the form is submitted is an HTTP request containing XML formatted text that is passed as an event to the rule engine. This matches the event clause illustrated in Figure 7, which illustrates the rule as it would appear at runtime.

As a consequence, the rule in Figure 7 is activated and its corresponding action is sent to the rule engine. The action requests a service named `RequestArtistList`. The `sessionID` field value guarantees that the specific request will be carried out on behalf of client that initiated the request. The service request is passed from the rule engine to the task enactment engine which invokes the service through the appropriate EJB.

In this example, the service `RequestArtistList` expects a parameter that will be formed into an SQL query. In this case, it has the value "Rush". The service repository will provide the server IP number and the port at which the EJB for this service is available. Upon its completion, the service will return a new event as shown in Figure 8. This will allow the rule engine process to

Fig. 9. Choosing Album Titles

continue with the next rule. The `sessionId` value that was passed to the service (`CDArtistQuery:10000`) will also be passed back from the service.

This new event matches the event premise of the rule illustrated in Figure 10. In this rule, the `sessionId` value is assigned appropriately and the parameter value is bound to the variable `Artist`. The action to be carried out in this case is the `ConvertToHMTL` service which is a program that converts XML, using XSL style sheets, to HTML. Finally, this service will send a new event (not shown due to space limitations), that triggers a rule that will send the results back to the web server so they may be passed on to the client's Web browser. At this point the session identifier is used to match these particular results with the correct client (other clients may have invoked the same sequence of rules and the rule engine may return different sets of results back to the Web server). The resulting HTML data is displayed as shown in Figure 9.

Similar rules handle the case where more information is requested for a particular CD, or when a CD is added to a shopping cart. When an order is placed for a set of CDs, the inventory system is notified. Should there be insufficient stock of any ordered item, an event is sent from the inventory system. A rule exists that causes the shopping cart service to modify the contents of the order so that there will be an indication (when the order is sent to the invoicing/shipping system) that an item was out of stock. The out-of-stock event also triggers another rule and causes an order for the item to be placed with a supplier.

Customizable transaction business logic, purchase policies, and special offers can all be encoded as ECA rules. Using the meta-language and the proposed environment provides a means by which services can be invoked in a fully customizable way.

6 Conclusion and Future Work

In this paper, we have presented a generic architecture that allows for the customizable integration of services in Web-enabled environments. In particular, we presented a technique for remote services to be represented and integrated using a meta-language based on the ECA paradigm. Moreover, we presented a task enactment engine that utilizes the ECA rules and around which the system architecture is built. Finally, we discussed a prototype e-commerce application which has been built using the ECA approach and the proposed architecture.

The prototype is currently being extended as a research prototype at IBM Toronto Lab, Center for Advanced Studies. This includes the dynamic registration of services, the automatic generation of wrappers given service interface descriptions and finally, the run-time selection of services when there are replicated services offered by the system.

Acknowledgments

This project could not have progressed as it did without the help of several other individuals. In particular, we would like to thank Kelvin Cheung of the University of Waterloo for incorporating the rule engine into the existing implementation. We would also like to thank Evan Mamas, also of the University Waterloo, and Jianguo Lu of the University of Toronto, for their suggestions and feedback. We are also indebted to Teo Loo See, Daniel Tan and Daniel Wee of Nanyang Polytechnic, Singapore, for an earlier demonstration prototype they have built using IBM's electronic business application framework, components of which were adapted for the demonstration prototype presented in this paper. Finally we would like to thank David Lauzon, Bill O'Farrell and Weidong Kou of the IBM Toronto Lab for their support and guidance for our project.

References

1. P. Dreyfus, "The Second Wave: Netscape on Usability in the Services-Based Internet", IEEE Internet Computing, March/April 1998. 235
2. J. Widom, S. Geri: editors "Active Data Base Systems: Triggers and Rules for Advanced Database Processing", Morgan Kaufmann, 1996. 236
3. Object Management Group, http://www. corba.org. 236, 238
4. Sun Microsystems, Enterprise JavaBeans™ Specifications, http://java.sun.com/products/ejb/docs.html, December 1999. 236
5. J. Mylopoulos, A. Gal, K. Kontogiannis, M. Stanley, "A Generic Integration Architecture for Cooperative Information Systems", Proceedings COOPIS '96, July 1996. 236
6. B. Nusbaum, et al, WebSphere Application Servers: Standard and Advanced Editions, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245460.pdf, July 1999. 236, 238
7. A. Gal, J. Mylopoulos "Towards Web-Based Application Management Systems" in IEEE Transactions on Knowledge and Data Engineering, 2000 (to appear). 236, 241

8. J. A.Bergstra, P. Klint, The Discrete Time ToolBus.
http://adam.wins.uva.nl/~/olivierp/toolbus/index.html, February 1995 236
9. Control and Coordination of Complex Distributed Services,
http://www.newcastle.research.ec.org/c3ds 237
10. Sun Microsystems, Jini Connection Technology,
http://www.sun.com/jini/overview/index.html, Feb 2000. 237
11. DTD for an ECA Scripting Language
http://www.swen.uwaterloo.ca/~rwgregor/thesis/ECADTD.html 239, 243, 245
12. IBM XML Parser, http://www.alphaworks.ibm.com/formula/xml. 239
13. R. Gregory, "A Customizable and Extendable Distributed Service Integration Environment", Master's Thesis, University of Waterloo, Department of Electrical and Computer Engineering, October, 2000. 240, 243, 245
14. L. Etzkorn, C. Davis, "Automatically Identifying Reusable OO Legacy Code", Computer, IEEE, October, 1997. 240
15. K. Sartipi, K. Kontogiannis, F. Mavaddat, "Architectural Design Recovery Using Data Mining Techniques",In Proceedings of IEEE Conference on Software Maintenance and Reengineering (IEEE-CSMR'00). 240
16. K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems", In Proceedings of IEEE Conference on Software Technology and Engineering Practice (IEEE-STEP'99). 240
17. H. Sneed, "Generation of Stateless Components from Procedural Programs for Reuse in a Distributed Systems", In Proceedings of IEEE Conference on Software Maintenance and Reengineering, Zurich, March 2000, pp.183-188. 240
18. Y. Zou, K. Kontogiannis, "Migration and Web-Based Integration of Legacy Services" to appear in Proceedings of CASCON 2000, Toronto, Ontario, November 2000. 240, 245
19. J. Lu, J. Mylopoulos, J. Ho, "Towards Extensible Information Brokers Based on XML", to appear in CAiSE*00, 12th Conference on Advanced Information Systems Engineering, Stockholm. 241
20. A. Gal, S. Kerr, J. Mylopoulos "Information Services for the Web: Building and Maintaining Domain Models", International Journal of Cooperative Information Systems, 8(4):227-254, 1999. 241
21. MicroSoft Corp. "BizTalk: Overview"
http://www.microsoft.com/industry/biztalk/business/highlights.stm 241
22. J. Held, C. A. T. Susch, A. Golshhan, "What Does the Future Hold for Distributed Object Computing", StarandView Vol. 6, No.1, March 1998. 241
23. Sun Microsystems, "Java Naming and Directory Interface, Application Programming Interface", http://java.sun.com/products/jndi/ 245
24. G. Koulouris et.al "Distributed Systems: Concepts and Design", Addison-Wesley, Second Edition, 1996. 246
25. W. Ku et. al, "End-to-End E-commerce Application Development Based on XML Tools", in IEEE Data Engineering, Vol. 23, No. 1, pp. 29-36. 248

```
<ECARule name="CDArtistQueryResults"
  <Declarations>
    <Variable identifier = "Results">
      <Type name = "XMLString"/>
  </Variable> </Declarations>
  <Events> <EventExpr>
    <Event name ="ReturnedRequestArtistList"
           sessionId = "CDArtistQuery:10000">
      <SetVariables>
        <Identifier name = "Results"
          value="<Artist>Rush</Artist>
            <Albums>
             <Album>Moving Pictures</Album>
             <Year>1981</Year>
             <Album>Permanent Waves</Album>
             <Year>1980</Year>
             <Album>Hemispheres</Album>
             <Year>1978"</Year>
            </Albums>"/>
      </SetVariables>
  </Event> </EventExpr> </Events>
  <Conditions> </Conditions>
  <Actions>
    <Service name = "ConvertToHTML"
      <Class name = "XMLtoHTML"
             sessionId = "CDArtistQuery:10000"/>
      <UseVariable>
        <Identifier name = "Artist"
                    value ="<Artist>Rush</Artist>
              <Albums>
                <Album>Moving Pictures</Album>
                <Year>1981</Year>
                <Album>Permanent Waves</Album>
                <Year>1980</Year>
                <Album>Hemispheres</Album>
                <Year>1978</Year>
              </Albums>"/>
      </UseVariable>
    </Service>
  </Actions>
</ECARule>
```

Fig. 10. A Query Result Rule at Runtime

Migrating and Specifying Services for Web Integration

Ying Zou, Kostas Kontogiannis

Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada

{yzou, kostas}@uwaterloo.ca

Abstract. With the explosive growth of the Internet, businesses of all sizes aim on applying e-business solutions to their IT infrastructures, migrating their legacy business processes into Web-based environments, and establishing their own on-line services. To facilitate process and service integration, a complete and information rich service description language, is essential for server processes to be specified and for client processes to be able to locate services that are available in Web-enabled remote servers.

Within the context of emerging technologies, such as XML, the Internet, and Network-Centric Computing, we propose an architecture that allows for Web-based integration of distributed components and services. The architecture is based on component wrapping, a service description language that allows for the specification of services, and on techniques that support service registration and dynamic service localization.

1 Introduction

Tremendous changes are taking place in the business world today due to the frequent introduction of new technologies. As these technologies become the mainstream, the focus of e-commerce activities is shifting from customer-to-business transactions, to an e-business to business (B2B) model [8], which integrates business services and business process models, across corporate Intranets or the Internet. Towards this objective, multi-tier architectures, networking, and distributed object technologies have made possible for organizations to deploy complex software applications over the Internet.

Modern software systems must conform to requirements, such as flexibility, adaptability, time to market, and ability to withstand continued business process reengineering. Driven by these requirements, the migration and integration of legacy systems towards new platforms and operating environments provide an effective strategy for organizations to maintain their competitive edge [1]. In this context, many consulting firms such as the Gartner Group are predicting that organizations that integrate new development with the existing legacy systems will have a higher success rate, at optimal cost, in the implementation of client/server applications.

In this paper we present an architecture that allows for the migration and integration of existing stand-alone services into distributed environments.

W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, pp. 253-270, 2001.

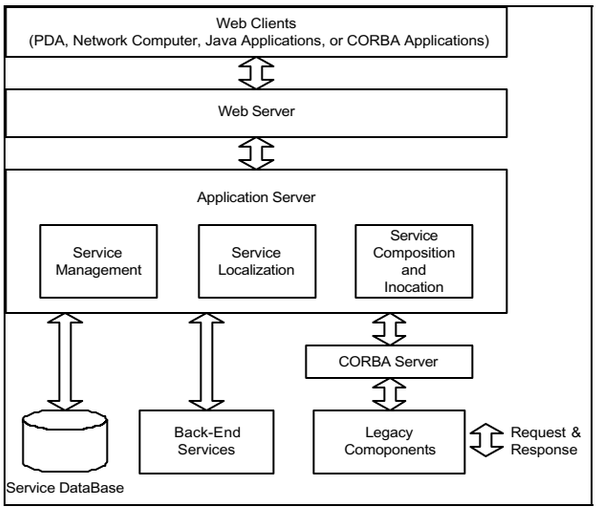


Fig. 1. Overall Architecture

In the core of the system lies a service description language that provides standard, enriched, and well-understood information about the interfaces and the functionality of the offered services. A service registration tool allows for services to be easily registered with the environment. Finally, a search engine can effectively locate services according to specific search criteria, allowing thus for service location transparency.

This paper is organized as follows. Section 2 introduces the architecture for Web-based Integration. The issues that arise in Web-based service integration are addressed in section 3. The issues pertaining to the migration of software components as legacy services are presented in section 4. Section 5 discusses the major components of a service description language. The service registration and localization modules are presented in section 6 and 7 respectively. An application scenario is illustrated in section 8. Finally, section 9 provides a summary and the conclusion of the paper.

2 Architecture for Web-based Service Integration

The Web-based service integration architecture focuses on the use of Web as an open infrastructure where e-business related services and tasks can be defined, composed, and enacted in a fully customizable way.

As e-commerce services can be scattered virtually everywhere on the Web, we need an architecture that allows for the separation of business application logic from the client-side presentation logic. The three-tier architecture is instrumental for the deployment of the distributed objects on a Web-enabled environment.

We present an open, multi-tier infrastructure, where a service can publish itself, and easily be integrated with other legacy components and services. The proposed

architecture is depicted in Fig. 1. The first layer (top) consists of a wide range of Web clients, including Web browsers for handheld and embedded systems, or Java/Pure and CORBA based applications running on fully loaded desktops. The second layer relates to services provided by the Web server and application server. The Web server captures the requests from Web clients and directs the requests to the Application server. The application server has been widely adopted as the runtime environment of choice for integrating heterogeneous applications. The core part of the architecture is the underlying services that are added to the application server, including Service Management, Service Localization, and Service Composition and Invocation.

The Service Management module maintains a database of the descriptions of the available services. It enables the deployed services to dynamically register their information in a repository. The Service Management module provides a repository for the client processes to use in order to locate available services and compose them for the completion of elaborate business tasks. A service description language provides a customizable way to represent distributed services with enriched information.

The Service Localization module is responsible for selecting the required services among many available ones, according to the criteria set by the client process. The service localization enables the clients to search the service by functionality, signatures, performance, and customizability.

Finally, the Service Composition and Invocation module provides a framework and a scripting language for dynamically enacting and composing remote services. This module serves as an integrator that allows back-end services and legacy systems to be composed seamlessly.

In order to enable the integration of legacy applications in a Web-based environment, we adopt the CORBA standard. The standard allows for legacy applications to be encapsulated in remote objects using wrapper classes and behave as distributed components. This wrapping technology allows clients in virtually any software or hardware platform to invoke remote legacy components in their native operating environments. The legacy application is resident on the CORBA server, which acts as the gateway for the integration. Such a framework provides system support for the invocation and integration of legacy back-end services from any clients.

3 Web-based Service Integration Mechanism

The generic three-tier architecture can be further separated into four layers [11]: *a)* presentation layer (Web client); *b)* content layer (Web server); *c)* application layer (application server); and *d)* back-end data and services layer, as shown in Fig. 2. The Web server is responsible to accept HTTP requests from Web clients, and deliver them to the application server, while the application server is in charge of locating the services and returning responses back to the Web server. On the other hand, a thin client in the presentation layer has little or no application logic.

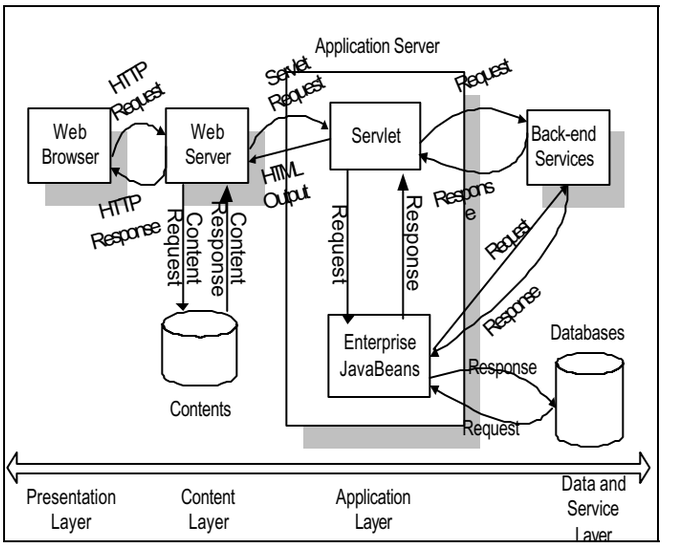


Fig. 2. Control Flows in Three-Tire Architecture

The Web server can maintain a content repository, or a file system, where the information-based resources are stored and serve as static HTML pages. Upon receiving a request from the client, the Web server retrieves the requested document from the content repository and sends it to the client. In this case, the client entirely relies on the Web server. Programming languages, such as Java, and scripting languages, like CGI, can be used to access the database.

To provide the dynamic information generated by software services, the Web server needs to constantly interact with the application server. A servlet provides the dynamic HTML content to clients. When the Web server receives the request for a servlet, it re-directs the client's request along with the parameters to the application server, which loads the servlet and runs it. Servlets not only have all the features of Java like automatic memory management, advanced networking, multithreading, and so forth but also, allow for enterprise-wide connectivity in the form of JNI (Java Native Interface), JDBC, EJBs, RMI, and CORBA. Servlets can make calls to back-end services, other servlets, or to the Enterprise JavaBeans [12].

Once the servlets are deployed on an application server, they can be accessed from any other Web server. This can be achieved provided that the client's request contains the URL of the servlet with the correct name, type, parameters, and initial values. The combination of EJBs and Servlets, CORBA objects and Servlets, and RMI objects and Servlets, can be used to invoke back-end services accessed by the Web clients via HTTP connections. However, CORBA and Enterprise Java Beans are not a panacea for all problems that may arise when integrating services in a distributed environment, but they provide the building blocks for distributing applications over a diverse range of platforms and operating environments.

4 Example Service Migration to a CORBA Environment

In order to integrate existing systems that encapsulate valuable business logic, the first step is to re-engineer these systems so that they can be used in a distributed environment. In the approach discussed in this paper, we utilize reverse engineering and design recovery techniques to identify specific components that encapsulate valuable business logic for a specific application. These techniques have been investigated as part of another project with IBM and are presented in [13, 14]. Once specific legacy components are identified through the use of program analysis, their behavior must be specified in terms of well-defined interfaces. In order to integrate the identified components to a heavily heterogeneous Web-enabled distributed environment, we must define an appropriate middleware. The CORBA specification provides a suitable infrastructure for integration, due to its platform, language, and vendor independence.

The component interface hides the implementation details inside the component and allows only signatures of services to be published to its clients. Moreover, it defines a set of properties and behaviors that represent a component's API. Properties are represented in terms of attributes, which can be accessed by accessors and mutators. Similarly, method parameters and return types can be represented by IDL interfaces.

In a related software migration case study we have used reverse engineering techniques in order to analyze and migrate the AVL GNU tree libraries from C procedural code to a new C++ object oriented implementation [13, 14]. The new migrant object oriented AVL tree library can be considered as a component, consisting of several classes. In this section we present how such a collection of C++ classes from the GNU AVL tree library cab be migrated in a CORBA environment.

In a nutshell, the interface for the new AVL tree component consists of several stub interfaces that correspond to wrapper classes. To migrate the standalone identified components into a distributed computing environment, the object wrapping approach can be adopted. The wrappers implement message passing between the calling and the called objects, and redirect method invocations to the actual component services. The concrete process to accomplish wrapping is implemented in terms of three major steps.

The first step focuses on the specification of components in CORBA IDL as shown in Fig. 3.

The second step deals with the CORBA IDL compiler to translate the given IDL specification into a language specific (e.g. C++), client-side stub classes and server-side skeleton classes. Client stub classes and server skeleton classes are generated automatically from the corresponding IDL specification. The client stub classes are proxies that allow a request invocation to be made via a normal local method call. Server-side skeleton classes allow a request invocation received by the server to be dispatched to the appropriate server-side object. The operations registered in the interface become pure virtual functions in the skeleton class.

The third step focuses on wrapper classes that are generated and implemented as CORBA objects, directly inheriting from the skeleton classes. The wrapper classes encapsulate the standalone C++ object by reference, and incarnate the virtual functions by redirecting them to the encapsulated C++ class methods. The new

```
module AVL{

interface corba_ubi_btRoot;
interface corba_ubi_btNode;
interface corba_SampleRec;

typedef char corba_ubi_trBool;

interface corba_SampleRec{
        void putName(in string val);
        string getName();
        void putNode(in corba_ubi_btNode val);
        corba_ubi_btNode getNode();
        long getDataCount();
        void putDataCount(in long aval);
};

interface corba_ubi_btNode {
        void putBalance(in char val);
        char getBalance();
        long Validate();
        //......
};

interface corba_ubi_btRoot{
        corba_ubi_trBool ubi_avlInsert(
                in corba_ubi_btNode  NewNode,
                in corba_SampleRec  ItemPtr,
                in corba_ubi_btNode OldNode );
        // ......
};
};
```

Fig. 3. AVL Component Interface Definition

functionality of the legacy object can be added in the wrapper class as long as the method name is registered in the interface.

For example, the SampleRec class is one of the classes identified within the AVL tree component. The wrapper_SampleRec inherits from the skeleton class sk_AVL::_sk_corba_SampleRec, which is generated from the CORBA IDL to C++ compiler. The wrapper class, wrapper_SampleRec, encapsulates a reference of SampleRec class as shown in Fig. 4.

When a client invokes a method through CORBA, it passes the CORBA data type parameters. The wrapper classes need to translate the CORBA specific data types from the client calls to the data types used by encapsulated C++ classes. Fig. 5 illustrates the transformation from the CORBA specific type such as corba_SampleRec_ptr to the SampleRec used in the C++ function. In the same way, the wrapper classes convert the returned values from the C++ class to the CORBA specific data type.

```
class wrapper_SampleRec : public _sk_AVL::_sk_corba_SampleRec
{
private:
        SampleRec& _ref;
        char *_obj_name;
public:
        wrapper_SampleRec(
                SampleRec& _t,
                const char *object_name = NULL):
                _ref(_t),
                _sk_AVL::_sk_corba_SampleRec(object_name);
        SampleRec* transIDLToObj(
                AVL::corba_SampleRec_ptr obj);
        void putNode(
                AVL::corba_ubi_btNode_ptr val);
        AVL::corba_ubi_btNode_ptr getNode();
        ~wrapper_SampleRec(){
                delete &_ref;
                free (_obj_name);};
        //.....
};
```

Fig. 4. An Example Wrapper Class

```
SampleRec* wrapper_SampleRec::transIDLToObj(
        AVL::corba_SampleRec_ptr obj)
{
if (CORBA::is_nil(obj)) return NULL;

// set up the data members of _ref object
 _ref.putName(obj->getName());
 _ref.putDataCount(obj->getDataCount());

//translate the ubi_btNode to corba_ubi_btNode_ptr by wrapper
//class NodeWrap
ubi_btNode *NodeImp = new ubi_btNode();
wrapper_ubi_btNode NodeWrap(*NodeImp, _obj_name);

//translate corba_ubi_btNode_ptr type returned from
//obj->getNode() to ubi_btNode * by transIDLToObj() in
//wrapper object NodeWrap.
_ref.putNode(NodeWrap.transIDLToObj(obj->getNode()));
 return &_ref;
}
```

Fig. 5. Example for Object Type Translation

Since IDL does not support overloading and polymorphism, each method and data field within the interface should have a unique identifier, in order to disambiguate references to programming entities that correspond to different languages. For example, C++ supports overloading, but C does not. If the polymorphism and overloading methods occur in one class, it is necessary to rename these methods by adding the prefix or suffix to the original name when they are registered in the interface, avoiding changing the identified objects. This “naming” technique allows unique naming conventions throughout the system, without violating code style stand-

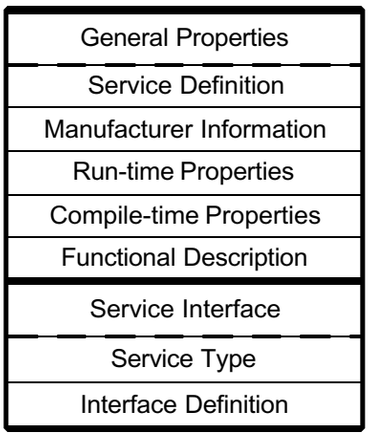


Fig. 6. Key Element of Service Specification

ards. The wrapper classes are responsible to direct the renamed overloaded and polymorphic methods to the corresponding client code.

If the polymorphism and overloading methods occur in the inheritance relationship, we can take advantage of C++ upcast feature, only register the sub-class in the component interface, and upcast the sub-class to its super class when the polymorphic or overloading methods in super class are invoked.

5 Service Description Language

In this section, we present the prototype of a service description language that provides a standard format to represent, register, and store information related to back-end services. To facilitate the integration of back-end services, a meta level description language is essential to effectively locate registered services. The meta-level description for the software services can be published at the same time as the distributed objects are deployed onto the application servers, or some time later when the enterprise would like to make their software services available.

5.1 Structure of Service Description Language

Generally, a service can be represented in a multi-faceted way, by specifying, for example, vendor, run time specifications, compile time requirements, method signatures, as well as, pre- and post-conditions. Each of these aspects is denoted as a Service Description Fact. Different Description Facts specify different properties of the services.

In a nutshell, the specification of the software services is divided into two layers namely *General* properties and *Service Interface* properties, as illustrated in Fig. 6.

```
<?xml verstion="1.0"?>
<SDL>
<GeneralInfo>
 <ServiceDef>
    <ServiceName> <!-- Specify service ID and name -- >
    </ServiceName>
    <ServiceCatalog><!--Specify service category-->
    </ServiceCatalog>
    <URL > <!-- Specify service URL linke -->
    </URL>
    <VersionNumber /> <!--Specify version number-->
 </ServiceDef>
 <Manufacturer> <!-- Specify vendor information-->
 </Manufacturer>
 <RunTimeEnv> <!-- Specify run-time environments-->
    <OSs>
          <OS name="" version=""/>
    </OSs>
 </RunTimeEnv>
 <CompileTimeEnv> <!-- Specify compile time environments-->
 </CompileTimeEnv>
 <Funcationality>
          <!-- Specify abstract and detailed information -->
          <!-- about service funcationality-->
  </Functionality>
</GeneralInfo>
<ServiceInterface>
  <Types>
  <!--Lists the Types of components inside the service interface. -->
  </Types>
  <ServletML>
  <!--Lists the servlet interface -->
     <Parameters>
          <Parameter>
              <Name /><Type /><Value />
          </Parameter>
    </Parameters>
  </ServletML>
  <EJBML> <!--Specify the EJBs interface -->
  </EJBML>
  <CORBAML> <!--Specify the CORBA interface -->
  </CORBAML>
</ServiceInterface>
</SDL>
```

Fig. 7. Overall Structure of Service Description Language

Each layer contains specific information at different levels of abstraction. The structure of a service description document is illustrated in Fig. 7.

To enable a service binder to locate the requested correct service with high precision and recall levels, the General properties should contain facts that relate to such aspects as general service definition, manufacture information, run-time and compile-time properties, signatures, version numbers, implementation language, and functional descriptions. For example, for a CORBA wrapped service object, it is important to specify the ORB agent address, which is responsible for invoking the requested CORBA object by the name and URL address of the object.

For the purpose of Web-based service integration, it is important to disclose the interface of the distributed components to client processes. Similarly, the Service

```
<?xml version="1.0"?>
<!ELEMENT newTags (newTag)+>
<!ELEMENT newTag (startingPoint, tagDef)>
<!ELEMENT startingPoint (#PCDATA)>
<!ELEMENT tagDef
    (tagName, attList*, containedTags*)>
<!ELEMENT tagName (#PCDATA, tagContent*)>
<!ELEMENT attrList (attr)+>
<!ELEMENT tagContent (#PCDATA)>
<!ELEMENT containedTags (tagDef+, group)>
<!ELEMENT group (group* | tagName*)>
<!ATTLIST group groupName CDATA #REQUIRED>
<!ATTLIST group groupType (SEQ|OR) #IMPLIED >
<!ATTLIST group groupOccurs
          (once|optonal|requried) #IMPLIED>
<!ATTLIST tagDef occurs
          (once|optional|required) #REQUIRED>
<!ATTLIST attr attrName CDATA #REQUIRED>
<!ATTLIST attr attrType CDATA #REQUIRED>
<!ATTLIST attr attrValue CDATA #IMPLIED>
```

Fig. 8. DTD for Adding New Fact and Content

Interface layer specifies the APIs of the registered components. As stated earlier, there are different technologies to make the back-end services available to remote clents. These technologies include servlets, EJBs, and CORBA. Each type of back-end services is registered by its own specific interface description.

For servlets, the inputs are embedded in HTML forms, which contain the HTML types of inputs, the names of parameters and the allowable values. For the EJBs, the back-end services can be composed of several beans (session beans, or entity beans) in one jar file. Each bean has its own home interface and remote interface. When a service is implemented by the CORBA standard, it may include several CORBA IDL interfaces as encapsulated in the CORBA IDL "module" name scope. For the interface within CORBA and EJBs components, it is necessary to declare the available methods, parameters and the types of method parameters and return values. To reduce the complexity in definition of service description language, we inherit the interface from EJBs and CORBA IDL by inserting them under the <EJBML> tag and <CORBAML> tag respectively.

5.2 Extensibility of the Service Description Language

The extensibility of the service description language is crucial for representing services in distributed Web-enabled environments. For example, new service categories can be added or existing service descriptions can be extended.

The DTD for the Service Description Fact specifies its syntax and allows for such Service Descriptions to be proven syntactically valid. The DTD for the Service Description language is illustrated in Fig. 8. New Service Description facts can be added by using the *newTag* element contained in the *newTags* element.

```
<?xml version="1.0"?>
<!DOCTYPE newTags SYSTEM "patSpec.dtd">
<newTags>
  <newTag>
    <startingPoint>SDL.GeneralInfo</startingPoint>
      <tagDef occurs="once">
        <tagName>RunTimeEnv</tagName>
        <containedTags>
            <tagDef occurs="required">
              <tagName>OSs</tagName>
              <containedTags>
                <tagDef occurs="required">
                  <tagName> OS </tagName>
                  <attList>
                    <att attrName="name" attrType="CDATA" />
                   <att attrName="version" attrType="CDATA" />
                  </attList>
                </tagDef>
            </containedTags>
          </tagDef>
          </containedTags>
      </tagDef>
  </newTag>
</newTags>
```

Fig. 9. Run Time Environment Fact Definition

With the fact specification DTD, the addition of new facts is uniquely identified and inserted in a way that maintains the syntactic validity of the description. In Fig. 9, for example, the addition of the Run-time environment fact is illustrated. In this example, the new fact is inserted under the *GeneralInfo* element with the tag name of *RunTimeEnv*. *RunTimeEnv* element can occur once under *GeneralInfo* element. It contains an *OS* element, which specifies the operating systems to run the service. The *OS* element may occur one or more times, and can have attributes that denote its name and version number.

Meanwhile, new content can be easily introduced into the existing service description under the tag <ServiceCatalog> category (Fig. 7).

5.3 Structure of Database

Service Descriptions and fact specifications (DTDs) require a database for the persistent storage of the XML encoded component and service interface description.

To keep the database management simple and achieve flexibility in the service description, we use one table to map the service ID and the external XML filename for each service description. Each fact consisting of a service description is stored in a separate table. The primary key of these tables is the service ID generated during registration. In the same way, another table is created to store the file name of the DTD for each fact.

The Service Management module (shown in the Fig. 1) is responsible to maintain the service database. It can insert a new service description, delete, and modify the existing one. For this task, we utilize the IBM DB2 XML extender to map XML

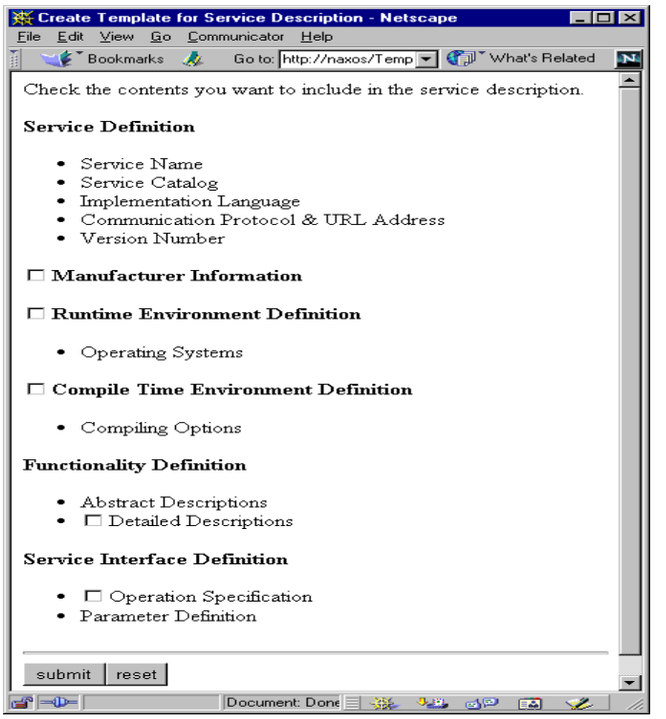


Fig. 10. Service Specification Fact List

Service Descriptions to DB2 tables. In general, the service manager retrieves from the database table the description filename, and then extracts the whole XML document by using traditional SQL queries. When a service is registered, the service manager can check for duplicate definitions, generate the service ID, and insert the description into the database. The Service Management Module is implemented by Enterprise Java Bean, which provides the support for transactions.

6 Service Registration

For the service registration, we have designed a Web based interface to serve as a service registration authoring tool, which allows for the user to specify the service description by filling in forms in a Web interface, as shown in Fig. 10. Then the service description is generated automatically from the information provided.

As mentioned earlier, in order to provide maximum customizability, the service description language is separated into independent facts. Moreover, the environment allows for new facts to be added at anytime. This interface allows the user to select the required facts by filling predefined forms. Some facts are indispensable, such as Service Definition. After submission, the Web Interface will create an HTML form as

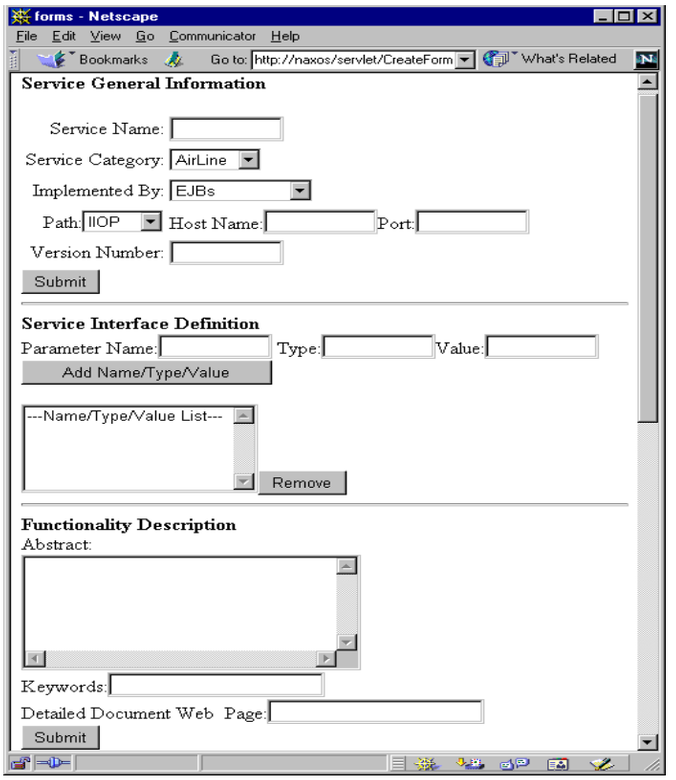


Fig. 11. Service Description Interface

shown in Fig. 11, where the user can add more information about the newly registered service.

For example, once the legacy components are wrapped as distributed objects, as discussed in section 4, their information can be described through the Web interface and registered into the service repository.

7 Service Localization

A prototype service localization mechanism allows for distributed software services to be located much like a search engine locates content (data) in Web pages.

The system can provide two ways for clients to submit search queries, via the Web HTML Interface, or by an XML formatted document that may be part of a client's request for a service.

The design of the query language aims to allow users to provide as many features as possible in order to specify the services being sought. The service description facts

```
<?xml version="1.0" ?>
<!ELEMENT searchSpec (ID*, location*, category*, implBy*, platforms*,
funcDsc*, vendor*, version*, timeLimit*)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT category ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT implBy ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT platforms ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT funcDsc ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT vendor (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT timeLimit (#PCDATA)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT AND (keyword | AND | OR | NOT)*>
<!ELEMENT OR (keyword | AND | OR | NOT)*>
<!ELEMENT NOT (keyword | AND | OR | NOT)*>
```

Fig. 12. DTD for Service Localization Query

as shown in Fig. 7, allow for the user to search for services according to specific criteria such as service categories, functionality, implementation techniques, and operating platforms. The grammar for the query language is defined in terms of a DTD as shown in Fig. 12. The root element for this DTD is the *searchSpec* element, which can include zero or more children, such as service ID, service location, service category, etc. The query doesn't need to include every element under the *searchSpec* element. Some elements, such as *category* are composed of the several *keyword* elements, *AND*, *OR* and *NOT* elements. The *keyword* element represents the keyword for the search criteria. The keywords can be conjuncted by *AND*, *OR*, and *NOT* elements. For example, *AND* elements can have children as *keyword* elements, *AND* elements, *OR* elements, and *NOT* elements. When new Description Facts are added into the service description language, the DTD of the query will be edited correspondingly in order to reflect the changes in the new service descriptions.

The Web interface of the service search engine is currently designed for the HTTP users, as shown in Fig. 13. After submission, the search criteria composed in the XML format are sent to the Service Localization module as illustrated in Fig. 1.

By extracting the requirements from the query specification, the service localization module looks up the service database. If multiple results meet the search criteria, the available services can be listed and ranked according to their registered features, such as performance, or estimated response time from the server. Since service description facts are encoded in XML, the search locator has been implemented using the XML DOM (Document Object Model) API and incorporating search logistics, such as exact search, sub-string search, precedence search, and stemming.

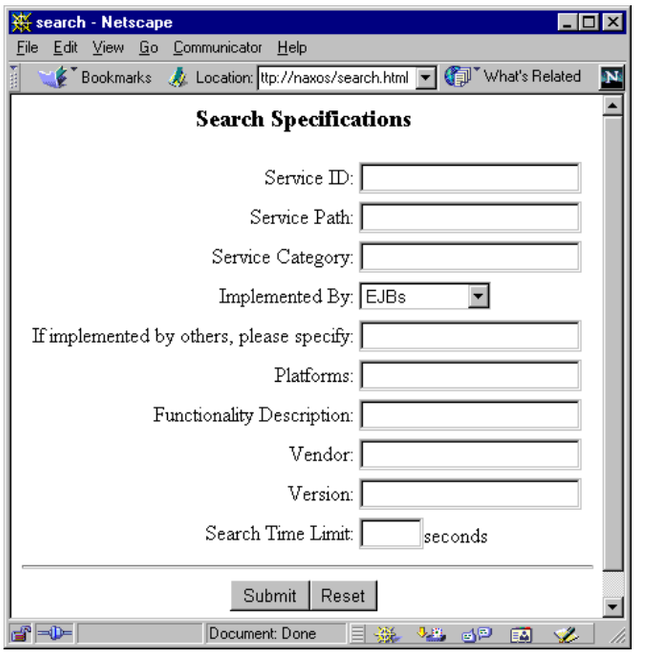


Fig. 13. Web Interface for Search Query

8 Application Scenario

As an example, consider an application scenario for the Web-based service integration architecture presented in this paper, whereby a global infrastructure enables distributed components that have been developed independently or migrated from legacy systems, to be integrated with each other and to facilitate complex business tasks.

In this way, distributed components located virtually everywhere in the world, can be combined on as required basis, forming thus collaborative systems. This integration can happen dynamically by allowing general service properties, functionality and, signatures of components that are specified in the service description language to be registered in the service database. Client processes can search for available distributed components in a same manner as a search engine would be used to locate information resources on the Internet. After meeting the search requirements, the client process can invoke the identified services without necessarily downloading all the components to the local client machine. On-going work, focuses on the invocation of services that is based on scripts encoded in XML and is enacted using the Event-Condition-Action (ECA) paradigm [15]. The overall proposed architecture is under development in collaboration with IBM Canada, Center for Advanced Studies, and is illustrated in Fig. 14. The core of the system is

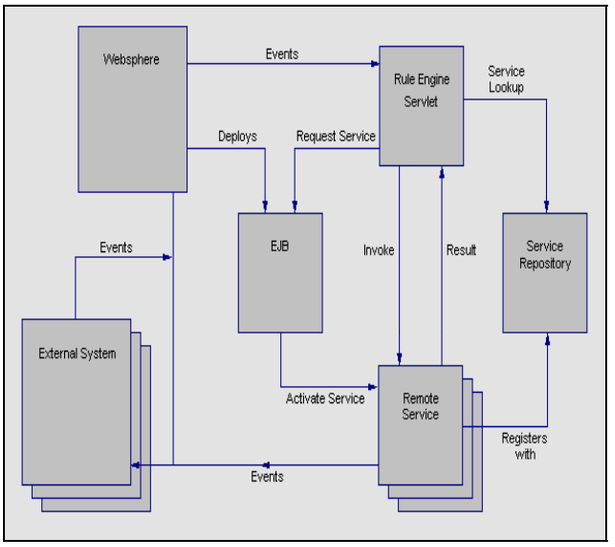


Fig. 14. Overall Service Integration Architecture

the Rule Engine Servlet, which accepts triggering events from the Web server. Once the premises (events and conditions) of specific ECA rules are satisfied, the requested service (action) by the rule is localized and invoked. Upon completion, services (actions) produce new events that may trigger new ECA rules. Deadlocks and loops are detected by building a rule dependency graph for a given script [16].

In its current form, the Internet and the Web provide reliable connectivity between client and server processes by using pre-defined, hard-coded transaction scenarios. These pre-defined transaction scenarios are currently implemented in terms of hard-coded URL links, CGI scripts and, Java applets. In this context, we address the issue of customizing the integration of services between client and server processes in distributed e-commerce and e-business environments. Services that are represented as remote components are encapsulated in wrapper objects. Such software components are obtained either as modules of legacy systems that encode mission critical business logic or, as modules of new applications developed with specific functional requirements in mind. In either case, these software components implement specific tasks that can be thought of, as building blocks of more complex interaction scenarios between client and server processes.

The customization of the transaction and integration logic required by various processes to complete complex tasks, opens new opportunities in Web-enabled e-Commerce and e-Business environments. In this sense, business partners can customize their business transaction models to fit specific needs or, specific contract requirements. This customization is transparent to third parties and, provides means to complete business transactions accurately and on-time. Organizations can enter the e-business arena by building and deploying extensible and customizable services over the Internet using software components that are readily available as services over the

Internet. Moreover, virtual agencies and portals that provide a wide range of services can be formed by integrating existing functionality and content over the Web. For example, a virtual travel agency can be formed, by composing in a customized manner, services that are readily available in various travel related Internet Web sites. Client processes may post requests to the virtual agency. The agency can enact its transaction logic (scripts) in order to integrate and compose data and services from a wide spectrum of sites. In this scenario, data about pricing, availability and, travel related special offers, can be fetched by various sites, processed by the agency and presented to the client in a customized and competitive for the agency way.

The prototype system under development at the Center for Advanced Studies is focusing on building virtual malls where different virtual stores (agencies) provide goods and services and, compete for the pricing, and the range of services offered.

9 Conclusion

In this paper, we present the issues of migrating monolithic services into distributed environments, and propose a service description language to specify back-end services. With the aid of a service description language, service registration and a service localization mechanism, component integration can be realized and service location transparency can be achieved.

In this context, we are especially interested in Web-based platforms because the Web is becoming the common denominator for accessing and presenting information over the Internet, Intranets and, Extranets. Moreover, the Web provides the deployment platform for many new enabling technologies such as CORBA, RMI and, EJBs.

As a result, this Web-based service integration infrastructure allows for the reuse of the existing software components, shortens the time to architect new applications, and eases the enterprise integration of business operations.

10 Acknowledgements

The authors would like to thank Bill O'Farrel and Steven Perelgut of IBM CAS and Evan Mamas and Richard Gregory of the University of Waterloo, for their valuable suggestions and insights.

References

1. Umar, Amjad, "Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies", Prentice Hall PTR, 1997.
2. RamPrabhu, Robert Abarbanel, "Enterprise Computing: The Java Factor", Computer, P115, June 1997 IEEE.

3. Walter Brenner, Rüdiger Zarnekow, and Harmut Wittig, "Intelligent Software Agents: Foundations and Applications", Springer-Verlag Berlin Heidelberg 1998.
4. Alan R. Williamson, "Java Servlets By Example", Manning Publications Co., 1999.
5. Victor Lesser, et al. "Resource-Bounded Searches in an Information Marketplace", IEEE Internet Computing, March/April 2000.
6. Tuomas Sandholm and Qianbo Huai, "Nomad: Mobile Agent System for an Internet-Based Auction House", IEEE Internet Computing, March/April 2000.
7. Ying Zou, Kostas Kontogiannis, "Localizing and Using Services in Web-Enabled Environments", 2nd International Workshop for Web Site Evolution, Switzerland, 2000.
8. "Business-to-Business e-Commerce with Open Buying on the Internet", http://www.ibm.com/iac/papers/obi-paper/intro.html .
9. "Gaining Competitvie Advantage in the Supply Chain: IBM Solution for Business Integration", http://www-4.ibm.com/software/info/ti/issues/scm.html.
10. Ronald Bourret, "XML and Databases", http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xml/XMLAndDatabases.html.
11. Paul Dreyfus, "The Second Wave: Netscape on Usability in the Services-Based Internet", IEEE Internet Computing, March/April 1998.
12. Joqquin Picon, et al, "Enterprise JavaBeans Development Using VisualAge for Java", http://www.redbooks.ibm.com
13. Prashant Patil, Ying Zou, Kostas Kontogiannis and John Mylopoulos, "Migration of Procedural Systems to Network-Centric Platforms", CASCON'99, Toronto, 1999.
14. Kostas Kontogiannis, Prashant Patil, "Evidence Driven Object Identification in Procedural Code", STEP'99, Pittsburgh, Pennsylvania, 1999.
15. Richard Gregory, Kostas Kontogiannis, "Requirements for a Distributed Tool Integration System", http://www.swen.uwaterloo.ca/~rwgregor.
16. George Koulouris et.al "Distributed Systems: Concepts and Design", Addison-Wesley, Second Edition, 1996.

Author Index