Lecture Notes in Computer Science 2952
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Nicolas Guelfi Egidio Astesiano
Gianna Reggio (Eds.)

Scientific Engineering of Distributed Java Applications

Third International Workshop, FIDJI 2003
Luxembourg-Kirchberg, Luxembourg, November 27-28, 2003
Revised Papers

Springer

Volume Editors

Nicolas Guelfi
University of Luxembourg
Faculty of Science, Technology and Communication
6, rue Richard Coudenhove-Kalergi, 1359 Luxembourg, Luxembourg
E-mail: nicolas.guelfi@ist.lu

Egidio Astesiano
Gianna Reggio
Università di Genova, DISI
Via Dodecaneso 35, 16146 Genova,Italy
E-mail:{astes,reggio}@disi.unige.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.2, H.4, H.3, H.5.3-4, C.2.4, D.1.3

ISSN 0302-9743
ISBN 3-540-21091-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10983829 06/3142 5 4 3 2 1 0

Preface

FIDJI 2003 was an international forum for researchers and practitioners interested in the advances in, and applications of, software engineering for distributed application development. Concerning the technologies, the workshop focused on "Java-related" technologies. It was an opportunity to present and observe the latest research, results, and ideas in these areas.

All papers submitted to this workshop were reviewed by at least two members of the International Program Committee. Acceptance was based primarily on originality and contribution. We selected, for these post-workshop proceedings, 14 papers, amongst 29 submitted, two tutorials, and one keynote talk.

FIDJI 2003 aimed at promoting a scientific approach to software engineering. The scope of the workshop included the following topics:

- design of distributed Java applications
- Java-related technologies
- software and system architecture engineering and development methodologies
- development methodologies for UML
- development methodologies for reliable distributed systems
- component-based development methodologies
- management of evolutions/iterations in the analysis, design, implementation, and test phases
- dependability support during system life-cycles
- managing inconsistencies during application development
- atomicity and exception handling in system development
- software architectures, frameworks, and design patterns for developing distributed systems
- integration of formal techniques in the development process
- formal analysis and grounding of modeling notation and techniques (e.g., UML, metamodeling)
- supporting the security requirements of distributed applications in the development process
- refactoring methods
- industrial and academic case studies
- development and analysis tools

The organization of such a workshop represents an important amount of work. We would like to acknowledge all the program committee members, all the additional referees, all the organization committee members, the University of Luxembourg, the Faculty of Science, Technology and Communication administrative, scientific, and technical staff, and the Henri-Tudor public research center.

FIDJI 2003 was mainly supported by the “Ministère de la Culture, de l’Enseignement supérieur et de la Recherche” and by the “Fond National pour la Recherche au Luxembourg.”

November 2003 Nicolas Guelfi, Egidio Astesiano, Gianna Reggio

Organization

FIDJI 2003 was organized by the University of Luxembourg, Faculty of Science, Technology and Communication.

Program Chairs

Guelfi, Nicolas	University of Luxembourg, Luxembourg
Astesiano, Egidio	DISI Genoa, Italy
Reggio, Gianna	DISI Genoa, Italy

International Program Committee

Astesiano, Egidio	DISI Genoa, Italy
Biberstein, Olivier	HTA, Bienne, Switzerland
Bouvry, Pascal	University of Luxembourg, Luxembourg
Di Marzo Serugendo, Giovanna	CUI, Geneva, Switzerland
Dubois, Eric	CRP Henri-Tudor, Luxembourg
Fourdrinier, Frédéric	Hewlett-Packard, France
Guelfi, Nicolas	University of Luxembourg, Luxembourg
Guerraoui, Rachid	EPFL, Lausanne, Switzerland
Huzar, Zbigniew	Wroclaw University of Technology, Wroclaw, Poland
Keller, Rudolph	Zühlke Engineering, Schlieren, Switzerland
Karsenty, Alain	JSI, Marseille, France
Koskimies, Kai	University of Helsinki, Finland
Kozaczynski, Wojtek	Microsoft Corporation, USA
Molli, Pascal	LORIA, Nancy, France
Parnas, David	University of Limerick, Ireland
Petitpierre, Claude	EPFL, Lausanne, Switzerland
Reggio, Gianna	DISI, Genova, Italy
Romanovsky, Sacha	DCS, Newcastle, UK
Rothkugel, Steffen	University of Luxembourg, Luxembourg
Rottier, Geert	Hewlett-Packard, Belgium
Sendall, Shane	EPFL, Lausanne, Switzerland
Souquières, Jeanine	LORIA, Nancy, France
Vachon, Julie	DIRO, Montreal, Canada
Warmer, Jos	De Nederlandsche Bank, Netherlands

Organizing Committee

Amza, Catalin	University of Luxembourg, Luxembourg
Chabert, Sylvain	University of Luxembourg, Luxembourg
Guelfi, Nicolas	University of Luxembourg, Luxembourg
Kies, Mireille	University of Luxembourg, Luxembourg
Perrouin, Gilles	University of Luxembourg, Luxembourg
Pruski, Cédric	University of Luxembourg, Luxembourg
Ries, Benoît	University of Luxembourg, Luxembourg
Sterges, Paul	University of Luxembourg, Luxembourg

Additional Referees

Chabert, Sylvain
Damien, Nicolas
Foukia, Noria
Kechicheb, Belkacem
Oriol, Manuel
Pawlak, Michel
Perrouin, Gilles
Pruski, Cédric
Ramel, Sophie
Razafimahefa, Chrislain
Razavi, Reza
Renault, Simon
Ries, Benoît
Sterges, Paul

Sponsoring Institutions

This workshop was supported by the University of Luxembourg, the Ministry for Culture, Higher Education and Research, and the National Research Fund.

Table of Contents

A Framework for Resolution of Deployment Dependencies in Java-Enabled Service Gateways 1
Jose L. Ruiz, Jose L. Arciniegas, Rodrigo Cerón, Jesús Bermejo, Juan C. Dueñas

A Java Package for Class and Mixin Mobility in a Distributed Setting 12
Lorenzo Bettini

Streaming Services: Specification and Implementation Based on XML and JMF 23
Björn Althun, Martin Zimmermann

Hard Real-Time Implementation of Embedded Software in JAVA 33
Jean-Pierre Talpin, Abdoulaye Gamatié, David Berner, Bruno Le Dez, Paul Le Guernic

Experiment on Embedding Interception Service into Java RMI 48
Jessica Chen, Kun Wang

BANip: Enabling Remote Healthcare Monitoring with Body Area Networks 62
Nikolay Dokovsky, Aart van Halteren, Ing Widya

Structural Testing of Mobile Agents 73
Márcio Delamaro, Auri Marcelo Rizzo Vincenzi

A Model of Error Management for Financial Systems 86
Hans Ewetz, Niloufar Sharif

Software Model Engineering and Reuse with the Evolution and Validation Environment 96
Jörn Guy Süß, Andreas Leicher, Fadi Chabarek

Distributed Composite Objects: A New Object Model for Cooperative Applications 106
Guray Yilmaz, Nadia Erdogan

A Java-Based Uniform Workbench for Simulating and Executing Distributed Mobile Applications 116
Hannes Frey, Daniel Görgen, Johannes K. Lehnert, Peter Sturm

Seamless UML Support for Service-Based Software Architectures 128
Matthias Tichy, Holger Giese

Model Generation for Distributed Java Programs 139
Rabéa Boulifa, Eric Madelaine

Keynote Talks

Software Inspections We Can Trust 153
David Parnas

Tutorials

J2EE and .NET: Interoperability with Webservices 155
Alain Leroy

Author Index 157

A Framework for Resolution of Deployment Dependencies in Java-Enabled Service Gateways

Jose L. Ruiz, Jose L. Arciniegas, Rodrigo Cerón, Jesús Bermejo[2], and Juan C. Dueñas[1]

Department of Engineering of Telematic Systems,
Universidad Politécnica de Madrid
ETSI Telecomunicación, Ciudad Universitaria s/n, E-28040 Madrid, Spain
{jlruiz,jlarci,ceron,jcduenas}@dit.upm.es,
jesus.bermejo@telvent.abengoa.com

Abstract. Software deployment refers to the set of activities regarding the movement of the software from the development environment to the final delivery environment. These activities cover the release, installation, activation, adaptation, deactivation, update, removal and retirement of software components in a set of hosts. The deployment of services in distributed systems is still an open problem, despite the fact of the several research efforts and standards that have covered this activity in the lifecycle of systems. The substitution of programs by assemblies of components, which is typical of service oriented architectures, has made this situation even worse. In this article, we describe a framework for the resolution of dependencies between services implementations and its application to the Java-based service model for services gateways (OSGi). This framework performs the recursive identification and resolution of dependencies between services in OSGi platforms, including the resolution of native dependencies. The framework requirements and its architecture are presented. It is being applied to an industrial case study about the distributed deployment of services on heterogeneous platforms.

Keywords: Java related technologies, component based development, frameworks and design patterns for developing distributed systems, case studies.

1 The work presented here has been developed in the projects OSMOSE and FAMILIES (Eureka 2023, ITEA ip02009 and ip00004), partially supported by the Spanish company Telvent and by the Spanish Ministry of Science and Technology, under reference TIC2002-10373-E. José L. Arciniegas and Rodrigo Cerón are visiting professors from Universidad del Cauca, Popayán, Cauca, Colombia. Rodrigo Cerón is sponsored by COLCIENCIAS - Colombia and AECI – Spain, and José L. Arciniegas work is partially supported by the Ministry of Science and Technology, under reference TIC2002-04123-C03-01.

2 Jesús Bermejo works for Telvent, Spain.

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 1–11, 2004.

1 Introduction

Software deployment refers to the set of activities regarding the movement of the software from the development environment to the delivery environment. These activities cover the release, installation and activation of the software in its final place.

As can be seen, software deployment is mainly a practical discipline, and as such, many practical solutions have been tried successfully in the past. For the development based on static linked programming languages, the application is created as a monolithic executable file, and therefore the deployment is as easy as to put the file in a certain place of the secondary storage of the computer, once it has been checked that the platform (composed by the hardware plus the operating system) is compatible.

A second stage can be identified with the usage of dynamically linked libraries (DLL): sets of libraries available in the computer that can be loaded in the memory image of programs at the execution phase. Their main advantage is the reduction of the executable files size and promotion the reuse of these libraries by sharing them among several programs. The deployment of software using DLLs increases in complexity: in order to install a program, it is necessary to check that all the libraries it needs are available, and install them otherwise. Here, the concept of dependency appears: the "dependency" relation means that one piece of software needs of another one to work properly. In fact, the management of dependencies – especially in presence of versions and evolution – is one of the main problems in deployment nowadays.

Component based development [1] brings the concept of partition of the system and the identification of dependencies to the core development of software systems. The main (predominant) component models are based on Java (EJB [2]), or they are language independent (.NET [3], CORBA Component Model-CCM [4]). Following this approach, and in the same way that DLL's are shared at execution time, components are shared at development time [9]. The current approach to the development of distributed systems called "service oriented architectures" (SOA) also influences deployment. The single system is replaced by a collection of services defined by interfaces (generically, contracts about the functions to be provided). These services are dynamically linked and therefore reusable, and can be deployed and removed from the system platform during its operation. The same concept of dependency again appears, but now widening its application: dependencies between service interfaces, services implementations, virtual machines, system dynamically linked libraries, kernel modules, drivers…

Thus, the overall scenario for deployment has increased in complexity. Other issues that contribute to this increased complication are the need to handle different versions and variants of services at once, the need to incorporate accounting models for the deployment and use of services, push-pop models, etc.

In our research, we have focused on the management of dependencies between elements of a service-oriented architecture; specifically, in the provision of a mechanism to perform the deployment of services with a large degree of variation in the elements that support them, in a distributed and heterogeneous environment.

The organization of the rest of the article is as follows: in section 2, deployment in current models is introduced. In section 3, a sketch about the Open Service Gateway Initiative, its service model and its deployment elements is given. The section 4 explains the requirements for dependencies resolution at deployment on OSGi model, and an architectural design is provided in section 5. The last section describes our current work in its application to an industrial case study; its status, conclusions obtained so far and further work.

2 Configuration and Deployment Activities (Domain Analysis)

Following several authors definitions, software deployment refers to all the activities that make software systems available for use, comprehending the process and activities related to the release, installation, activation, adaptation, deactivation, update, removal and retirement of software components in a set of hosts [5]. For them, a software system is a coherent collection of artifacts, such as executable files, source code, data files and documentation.

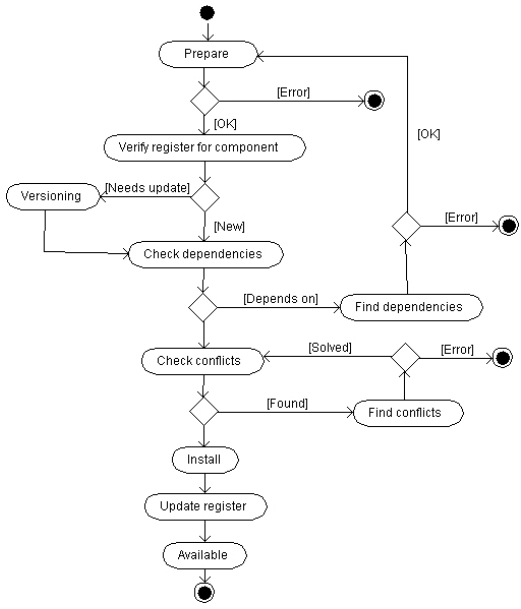


Fig. 1. The delivery process.

We have extended the model of installation as the key concept of deployment. The general process is shown in Fig. 1 and it has been divided into the several subprocesses:

- Preparation: is the activity responsible for the assessment of all expected non-functional requirements or quality features, for example checking if available free space in a hard disk is enough, memory constrains, performance, etc. If any condition is not fulfilled, the component is not installed; and in this case it may be necessary to choose other.
- Verify register for component, this sub-process aim is to check whether there is an old version installed and then this component goes to versioning process for its update.
- Check dependencies is in charge of analysing the component dependencies. A list of required components is obtained.
- Find dependencies: it takes the list of components obtained before and locates them into a repository. In case of finding it, the delivery process is done recursively. If the required component is not available, delivery finishes.
- Check conflicts detects possible conflicts of components. A component can generate problems in execution when other competing components have been installed; for example, a component could block other components when accessing to the same resource or overloads the platform capabilities. As a result of this sub-process a list of possible conflicts is obtained.
- Find conflicts, once found the possible conflicts, this sub-process would warn this situation and would suggest some solutions (for example, uninstall incompatible component). However, for security reasons the delivery process might be aborted.
- Installation, this process implies unpacking the components and locating them adequately in the system and possibly setting some environment variables and executing some special tasks, if required.
- Update register, when a new component is installed it is essential to save it in a database; whose information is relevant for future installations.

The artifacts in deployment may change its state during their lifecycle. The set of states that seem to be useful to describe this evolution are: non-registered, available, active, disabled and retired. Summing up, the delivery process execution begins with a component in a “Non registered” state, follows the chain of states, and ends with the component in the “Available” state. In order to carry out this process it is necessary that the component has available meta-data to solve dependencies and conflicts.

A rough classification of the tools currently used for deployment divides them into installation tools (that generate a set of executable and artifact files that must be provided to the customer site); package managers that handle software packages described by meta-data in order to install and configure the software; and centralized application management systems, able to handle deployment in large or medium organizations.

After the definition of the activities in the deployment and before presenting the requirements for a heterogeneous and distributed deployment engine, let us focus our attention in one of the successful service oriented architectures and their needs and capabilities for deployment.

3 Service Architectures and Deployment (Specific Domain)

The Open Services Gateway initiative (OSGi) [6] was launched, in 1998, by more than fifty companies in the area of embedded systems, with the aim to develop a series of open specifications for a Java based service platform, able to act as a gateway between Internet and the local area networks that can be found at home, at a car, and other types of restricted environments.

The third release of this specification defines a service platform (an instantiation of a Java virtual machine, an OSGi framework and a set of running services) that includes a minimal component model and a small framework for managing components, including a packaging and delivery format (see Fig. 2). In OSGi, a service is defined by its service interface (a specification of the service public methods) and implemented by, at least, a service object; the service must be registered in the service registry, also part of the platform. This model is considered complementary with the EJB model [7].

Component models as proposed in OSGi or .NET delay some configuration and deployment activities to run time. As a consequence of the loosely coupled approach followed, relationships do not need to be resolved at design time. These late bindings improve reusability and eases incremental development and maintenance of applications. Furthermore, products can be reconfigured without rewriting code or changing assets. On the other hand, technologies as CCM or EJB carry out configuration and deployment activities mostly at design time, which is less flexible than the former, but nevertheless is obviously safer.

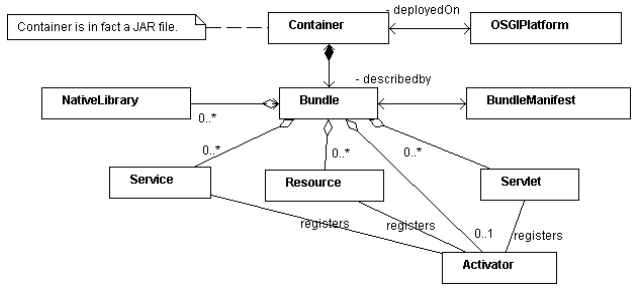


Fig. 2. Bundles and services in the OSGi framework.

In OSGi, services are deployed by means of bundles. A bundle is a Java archive file (JAR), containing the classes implementations, native code libraries, the resources needed to implemented services and a meta-data manifest file. This file contains key information: the Java packages the bundle provides, the packages the bundle requires and the name of the class that activates it.

The bundle life cycle has the next states: installed, resolved, started, active, stopped and uninstalled. There is a configuration service that allows setting configuration information of deployed bundles. Dependencies are described as Java packages, and over them OSGi carries out an automatic verification, but not an auto-

ages, and over them OSGi carries out an automatic verification, but not an automatic resolution. The provision of this capability (finding recursively the services implementations required by other) is the key issue that guided our work. While other approaches, such as Beanome [7] address the problem of finding the implementation of a given service at run-time already deployed, ours try to cover the problem of service deployment.

There are also services defined by OSGi for the remote platform management including bundle life-cycle management (installing, starting, updating, stopping and uninstalling), security, configuration, fault, accounting and performance management.

4 Resolution of Dependencies in Deployment (Requirements)

In service-oriented architectures, especially those focused on gateway functions or geared towards limited systems, that must handle many different variant elements, it is necessary to keep bundles as small as possible and to avoid loading service implementations and native libraries more than once. We propose a mechanism where the attempt to deploy a bundle launches the search for other bundles that satisfy the "requires" relationship in a recursive manner; after creating this set of bundles (or native libraries), it is analyzed for incompatibilities or for the selection of the best choice, and then the actual deployment takes place. This mechanism can be made automatic, thus easing the management of deployment configurations.

After this basic function of the framework, there are other aspects that must be taken into account in the design of the deployment mechanism:

- Two-phase protocol: the deployment may fail due to different reasons (lack of available memory, incompatibilities found, etc). Given the difficulty to roll back a deployment process once any element has already been deployed, we propose to use a two-phase protocol. In its first phase, the dependency closure (composed by all the elements that are reached by a dependency relationship) is obtained and thus the list of elements to be deployed, while the second one is for the actual delivery.
- Atomicity/preemptability: since the changes the engine performs on the platform are permanent, and an eventual mistake may lead to an unstable system, the engine should perform atomically. The engine should lead the platform in a stable situation, either by the complete installation of the required package, either by keeping the platform as it was before. However, the full execution of the deployment cycle may take a long time (while other applications may execute).
- Distribution: since the deployment process may be launched both by the owner of the platform as well as a remote control center, a certain degree of distribution in the system must be handled. Another reason is that some platforms will present reduced memory or disk size, and most of the phases that can be remotely performed must be done so. The solution must ensure that either local (in the platform) or remote (through the network) resolution must take place. Different scenarios for deployment are envisaged, where in some cases the local platform is an embedded system without the resources enough to execute the resolution, that

must then be delegated to the control center. This introduces a complete set of problems derivate from the need of network, the deferred execution of the resolution under disconnected states, the atomicity of the resolution process, etc.

- Heterogeneity: the dependencies go from OSGi service to service implementation; from service to native libraries, and from native libraries (packages or programs) to other ones. This means that the resolution process must not only handle the dependencies and installation process at the OSGi platform, but also the native part. For this purpose, the best solution found is to create a bridge to the native resolver mechanism provided by the operating system supporting the OSGi software platform; in our case it is Linux-Debian.
- Register-based. A key element in the resolution process is the capability to know if a certain bundle or library has already been deployed on the platform, and its current version (and other configuration data). The resolution process must access this information, either by querying a central configuration repository (that can also been located on the platform or in another location), or by checking the actual deployment of the library. It is clear that these two options have a strong impact on the dependability of the solution, the usage of resources (memory and network), and the security of the approach.
- Include information on the platform (physical and logical). The resolution process must be able to run on a large number of different platform configurations, obtained by the different usage of operating system kernels, modules, drivers, virtual machines, etc. Even more, some services exhibit dependencies to certain devices (a camera or a TV card, for example); we foresee the need to incorporate the description of these elements and the inclusion of the dependencies that affect them.
- Cost-aware: in service architectures such as OSGi, oriented towards the final user, cost models for the installation and usage of services are available. Some of these costs must be known at the deployment phase, and they must guide the user in the process. The resolver must be able to extract the cost model information and handle this information in order to perform decisions on the bundles to be deployed.
- Secure and security-aware: the execution of the resolver by unauthorized users may lead to broken platforms. Thus, the mechanism must be executed under specific security assumptions (roles and access control lists). A second issue on security is that the user may take the decision of the desired configuration based on the level of confidence the user may have got on the sources of bundles.
- Distributed repository, artefacts will be available through the network from different nodes. The metadata that describe the application should be downloadable independently of the actual packages, in order to allow a previous verification before proceeding to the download and installation of the artefacts themselves.
- Dependencies: different types of dependencies will lead to different activities within the resolution process. Each artefact, source of dependencies, might theoretically show relationships of any type and number with other artefacts, endpoints of the dependencies. Providing a classification of the dependencies and a syntax to describe them are basic requirements in order to automate the resolution process. Using a terminology analogous to the one used in digital electronics, dependencies will be classified as: AND (the endpoint is mandatory for the source, if

not available in the deployment target, it will have to be deployed), NOT (a conflict exists between the source and the endpoint, if the endpoint is installed in the platform a decision will have to be taken, as both can not coexist in the deployment target), OR (one or more artifacts can be selected in a resolution of an OR dependency) and XOR (analogous to the previous one, but in this case only one artefact may be selected).

5 Resolution of Dependencies in Deployment (Architectural Design)

The main elements of the architecture for the framework (see) are now explained.

Native Resolver (apt-dpkg): is the element in charge of connecting to the local-native resolver. In our case, the deployment will be a mix between OSGi services and Debian-Linux libraries and modules. The native resolver is thus a wrapper to the Debian resolver (apt-dpkg), able to activate several of the commands offered by the native resolver, such as asking for the installation of a certain package, checking if a package is already installed in the node, or requiring to remove a certain package (provided this will not harm others). Following the design patterns terminology, this part of the design corresponds to the adapter pattern.

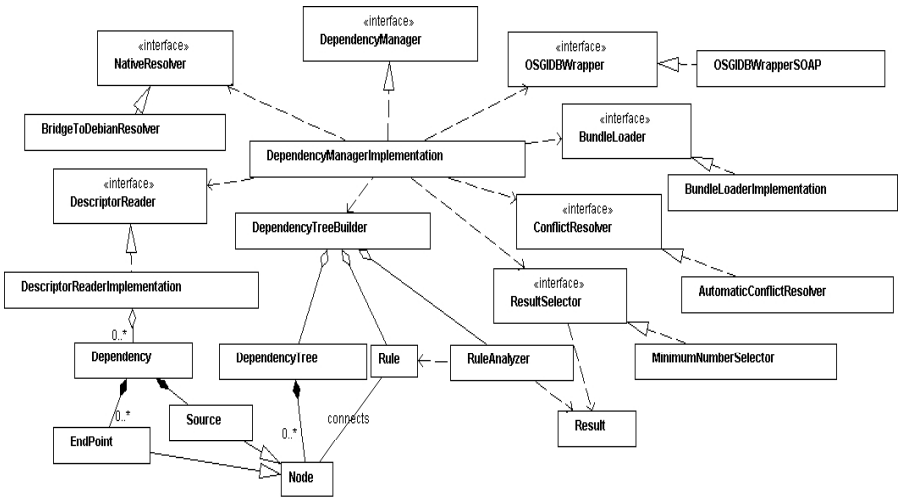


Fig. 3. The architecture of the deployment framework.

Deployment descriptor: is a piece of XML [8] description that contains the full description of the OSGi service, with especial focus on the relationships of this service implementation to others. The set of relations we are handling allows facing the situations in which a package is: mandatory for other one, can coexist, is incompatible, or a subset of a set of packages must be installed. Let us remark that these relations have traditionally been used in Debian packages deployment successfully for

years. Other important information in the deployment descriptor deals with accounting model for the service, policies for security, etc. The deployment descriptor can be obtained from a service deployment server or a services portal, or from the local configuration.

Dependency tree: once the set of deployment descriptors are traversed, the whole transitive closure of elements (both native and OSGi bundles) is found. In fact, it is a set of possibilities that can be installed; if this set is empty means that the configuration is not compatible (this is, not realizable) and the service cannot be deployed without changes to the current configuration (for example, removing an already available package), and therefore a conflict policy and policy handler are required. If several of the possibilities are found, a deployment selection policy must be incorporated in order to select the final set of elements (bundles and binary packages) to be installed, depending on the number of packages, the disk usage, the numbers of versions, etc.

Deployment selection policy: if any of the bundles or packages contain an “or” or “xor” dependency, and the dependencies relationship does not solve the option, the set of possible configurations to be downloaded and installed in the deployment process contains more than one possible solution. The deployment selection policy is in charge of performing the selection (or at least the ordering) in this set, following a certain criteria. The policy is described in the architecture by an interface (conformant to the strategy design pattern); several classes can implement it. Examples of possible selection policies are: that with the lowest number of new elements, or with the highest number, or with the highest priority, etc.

Conflict resolver: under certain circumstances, it is possible that the user requests the deployment of a certain service in the platform, the dependencies of one service implementation are analyzed and the result is that the service is incompatible with any of the bundles-libraries already installed in the system. In the simplest case, there will be not memory enough to hold the new service implementation, but there are also other situations in which the transitive nature of dependencies, as well as the incompatibility relation between service implementations, may lead to this conflict. The conflict resolver implements the "strategy" pattern in order to take a decision in case of conflict: it might be to identify the minimal set of incompatible elements in order to remove them or to reject the deployment. This last case must take into account that no degradation of the system is allowed. Another strategy we are using is providing a priority for the service implementations.

Loader: is the piece of the architecture in charge of downloading the specific elements to be deployed in the platform. This receives the detailed list of elements to be installed, and its work is to get the files, unpack them, and put the final set of files in their corresponding places. In order to get the files, several network protocols and accesses to file systems are envisaged (ftp, http, https, dvd, cdrom, disk, memory stick). The loader must be aware of the local file system in order to place the unpacked files. It is also in charge to update the centralized configuration data store.

DBwrapper: the role that in other systems play a centralized repository of changes to the basic configuration (the Windows registry) is played in this situation by the

wrapper to a lightweight database management system that is able to store information about the services and libraries installed in the system, as well as (this is for the near future) the elements that compose the physical infrastructure of the node, ranging from the processor type to the amount of available physical memory, data cards, device drivers, and so on. For some minimal configurations this element can not be placed in the local system and the registry is located and kept in the service provisioning centre.

6 Conclusion

New configuration and deployment challenges (service oriented architectures, a large degree of variation in the elements that support them, distributed and heterogeneous environments), have brought new needs to the main actors in the software value chain. This paper contains the requirements for the deployment and configuration in a mix OSGi-native scenario. The architecture and the basis of an automatic dependencies resolver in this context has been proposed.

The work presented here is still in progress. We are currently using the formerly described mechanism for the deployment of multimedia services on top of Java OSGi-enabled embedded platforms, under the control of a centralized provisioning server.

The work scenario is composed by a control centre (acting as the service provisioning server) plus several home platforms enabled with some OSGi implementations, and with different owners. On top of them, the user would like to get a new service, for example, for intrusions detection by means of several small cameras. The user browses the service portal and selects the service implementation to put on the OSGi platform, but there are also other pieces of services (alarm detection and signalling to the intrusion detection server and the service for programming alarms activations) and native libraries (as certain parts in the Java Media Framework implementation, for example) that must be deployed at the same time. Without our framework, the service provisioning server would provide a single bundle, but considering that the OSGi service implementations may vary depending on the OSGi platform version, the Java profile, the Java virtual machine; and the native libraries would depend on the operating system kernel, the kernel modules, the physical devices and their drivers, the number of bundles covering all combinations grows exponentially. Our framework makes it possible to adapt the deployment to each of the variant configurations.

Among the topics not solved yet for its industrial application, we find the support for versioning of service implementations on the same Java virtual machine, because no definitive and small-intrusion mechanism has been found yet. Other points where the current work will be extended are the native support mechanisms (that will include bridges to .NET platform), and providing a library of policies, specifically those that allow for the selection of a configurations offering a certain quality of services.

As seen, several open issues yet, but with the final goal of adding a function to the middleware layer that simplifies the service creation, deployment and maintenance.

References

1. Szyperski, C.: Component software, beyond object-oriented programming. 2nd edn. ACM Press Addison Wesley, London (2002).
2. Sun Microsystems, Inc. Enterprise JavaBeans™ Specification, Version 2.1. Proposed Final Draft (2002).
3. Microsoft, .NET, http://www.microsoft.com/net/.
4. OMG CORBA Components, formal document 02-06-65, version 3.0. June 2002.
5. A. Carzaniga, A. Fuggetta, R.S. Hall, A. Van der Hoek, D. Heimbigner, A.L. Wolf. A characterization framework for software deployment technologies. Technical report, CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.
6. Open Services Gateway Initiative. "OSGi Service Platform," Specification Release 3.0, March (2003).
7. Cervantes, H., Hall R.S. Beanome: a Component Model for the OSGi Framework.Software Infrastructures for Component-Based Applications on Consumer Devices, Lausanne, September 2002.
8. XML, Extensible Markup Language, http://www.w3.org/XML/
9. "Deployment and Configuration of Component-based Distributed Applications Specification".OMG Draft Adopted Specification ptc/03-07-02. June 2003.

A Java Package for Class and Mixin Mobility in a Distributed Setting*

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
bettini@dsi.unifi.it

Abstract. Mixins, i.e., classes parameterized over the superclass, can be the right mechanism for dynamically assembling class hierarchies with mobile code downloaded from different sources. In this paper we present the Java package `momi` that implements the concepts of the language MoMi, which is a calculus for exchanging mobile object-oriented code structured through mixin inheritance. This package can be thought of as an "assembly" language that should be the target of a compiler for a mobile code object-oriented language. In order to show an usage of the package, we illustrate how it is used by the compiler of X-Klaim, a programming language for mobile code where distributed processes can exchange classes and mixins through distributed tuple spaces.

1 Introduction

Mixins [11,17] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to standard class inheritance. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding or overriding specific sets of methods. The same mixin can be used to produce a variety of classes with the same functionality and behavior, since they all have the same sets of methods added and/or redefined. The superclass definition is not needed at the time of writing the mixin definition. This minimizes the dependences between superclass and its subclasses, as well as between class implementors and end-users, thus improving modularity.

Due to their dynamic nature, mixin inheritance can be fruitfully used in a *mobile code* setting [25,14], where distributed applications exchange code through the network to be dynamically integrated in the running processes. In [5], we introduced MoMi (Mobile Mixins), a coordination language for mobile processes that exchange object-oriented code. The underlying idea motivating MoMi is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not appear to scale well to a distributed context with mobility. MoMi's approach consists in structuring mobile object-oriented code by using mixin-based inheritance, and this is shown to fit into the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that has to access some files, can be completed with a

* This work has been partially supported by EU within the FET – Global Computing initiative project *MIKADO* IST-2001-32222, and by MIUR project *NAPOLI*. The funding bodies are not responsible for any use that might be made of the results presented here.

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 12–22, 2004.

base class in order to provide access methods that are specific of the local file system. Conversely, critical operations of a mobile agent, enclosed in a downloaded class, can be redefined by applying a local mixin to it (e.g., in order to restrict the access to sensible resources, as in a *sand-box*).

MoMi highly relies on typing. The most important feature of MoMi's typing is the *subtyping* relation that guarantees safe, yet flexible, code communication. We assume that the code that is sent around has been successfully compiled in its own site (independently from the other sites), and it travels together with its static type. When the code is received on a site (whose code has been successfully compiled too) it is accepted only if its type is compliant with respect to the one expected, where compliance is based on subtyping. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors such as "message-not-understood") without requiring any further type checking of the whole code. Thus, dynamic type checking is performed only at communication time. This is a crucial matter for mobility, since mobile code and in particular mobile agents are expected to be autonomous: once the communication successfully occurred, transmitted code behaves remotely in a (type) safe way (no failure communication to the sender will ever be required). This makes the code exchange an *atomic* action.

In this paper we present the implementation of MoMi in Java that consists in a package `momi`. This package provides the run-time system, or the virtual machine, for classes, mixins and objects that can be downloaded from the network and dynamically composed (via the *mixin application* operation). It thus provides functionalities for checking subtyping among classes and among mixins and for building at run-time new subclasses. Since MoMi abstracts from the specific communication and mobility features, this package does not provide means for code mobility and network communication, so that MoMi can be smoothly integrated into existing Java mobility frameworks. A concrete example of how to integrate `momi` within a real mobile code language will be shown in Section 4 by illustrating how the package `momi` is used within X-Klaim [7, 8,2], a tuplespace based programming language for mobile code.

We would like to stress that this package should be thought of as an "assembly" language that is the target of a compiler for a high level language (in our case the language is X-Klaim). If `momi`, as it is, was used for directly writing MoMi expressions, the programmer would be left with the burden of writing methods containing Java statements dealing with `momi` objects, classes and mixins, and to check manually that they are well typed according to MoMi types. Basically these are the same difficulties a programmer has to face when using an assembly language directly, instead of a high level language. We could say that `momi` enhances the functionalities of the Java virtual machine: while the latter already provides useful mechanisms for dynamically loading new classes into a running application, the former supplies means for dynamically building class hierarchies (based on mixins) and for inserting new subclasses into existing hierarchies (which is not possible in Java).

At the best of our knowledge, this is the first approach that employs mixins in a mobile code environment for flexible and safe code exchange. This is the distinguishing feature w.r.t. to other mixin-based languages and implementations, such as, e.g., [11, 17,1]. Other works, such as, e.g., [16,24,21,18] are concerned in merging concurrency and object orientation and do not deal explicitly with mobile distributed code, while

other OO mobility languages such as [13,12] do not treat dynamic inheritance. Due to lack of space we will not provide the formal description of MoMi (we refer the interested reader to [5,3]) and we give only a brief description of the `momi` package; its full presentation can be found in [2]. All the software presented here is freely available at `http://music.dsi.unifi.it`.

2 Main Features of the Package

Since `momi` is thought of as an assembly language that is target of a compiler for a high level programming language, in order to describe the package we will use a very simple object-oriented programming language (based on the syntax of X-Klaim). We will not give the formal syntax of this language, since it should be quite intuitive. Let us describe informally mixin usage through a tutorial example (for simplicity, instance variables are considered *private*, methods are considered *public* and no method overloading is employed):

```
mixin M
  expect n() : int;
  redef m2(a : int) : str ...
  def m1(b : bool) : int
end

class C
  n() : int ...
  m2(a : int) : str ...
end

(new (M <> C)).m1()
```

Each mixin consists of three parts:

1. methods *defined* in the mixins, like m_1;
2. *expected* methods, like n, that must be provided by the superclass;
3. *redefined* methods, like m_2, where *next* can be used to access the implementation of m_2 in the superclass.

The mixin application $M <> C$ constructs a class, which is a subclass of C. This operation is type correct only if the class C provides all the methods requested by the mixin (i.e., n and m_2). The type checker will also guarantee that there is no conflict among methods defined by the mixin and methods defined by the class. Mixin types encode the information about the types of mixin methods that are defined, redefined and expected.

In a mixin application class and mixin expressions can also be class and mixin variables that will be replaced at run-time with actual classes and mixins downloaded from the network. In this crucial case, it is safe to replace these variables with classes and mixins that have a subtype w.r.t. the corresponding types of variables. A class C' is a *subtype* of a class C if the type of C' has *at least* the methods of C. A mixin M' is a subtype of a mixin M if the type of M' has *at least* the methods *defined* by M, *at most* the methods *expected* by M and *the same* methods *redefined* by M. For objects, the subtype relation is the same as for classes. For motivations about these subtype relations we refer to [5]. Notice that we consider only *width* subtyping, i.e., the methods with the same name must have the same type, not a subtype; for a formal treatment of *depth* subtyping in MoMi we refer to [6]. Finally, due to lack of space, we do not consider here possible run-time name clashes among methods *defined* by the mixin and also by the

class in a mixin application (for such methods we use static binding instead of dynamic binding as illustrated in [2]).

The package `momi` provides means for creating new classes by applying a mixin to an existing class. This is similar to the task performed by a compiler for an object-oriented language when a class derives from a base class, but the main difference is that this operation will take place at run-time. The main aim of the package is that of providing this mechanism in a transparent way: once a class is generated after a mixin application it can be used to generate objects, but it can also take part in other mixin applications as a base class. The objects created through class instantiation are fully fledged and, so, ready for method invocations.

The crucial feature of MoMi is that classes and mixins are themselves mobile code, i.e., code that can be dynamically downloaded from a remote source. This implies that, at mixin application time, the actual implementation of mixins and classes may differ from their expected (static) interface, provided that the former is subtyping-compliant with the latter in order to guarantee the absence of run-time errors. This highlights the main difference between our approach and other mixin based languages in the literature (see, e.g., [17,1]): in a mixin-based language, when the mixin application takes place in a specific part of a program, both the code of the mixin and the one of the classes are available for the compiler. This does not hold in MoMi since mixin application can act also on mixin and class (polymorphic) variables.

So the task of generating a new class by applying a mixin to a class must be done in MoMi at run-time, when the class and mixin variables have been actually instantiated dynamically with effective code (possibly downloaded from the network). Basically this is the same difference that distinguishes a language with dynamic class loading, such as Java, and one with static loading, such as C++. This is similar to what happens in calculi where classes and mixins are "first-class citizens", i.e., they can be passed as parameters to functions (see, for example, [10]). It is important to notice, though, that in such calculi the matching between actual and formal parameters is always based on equality (at least at the best of our knowledge), because the burden of checking extra constraints at parameter-passing time, to avoid inheritance/subtyping conflicts, is not worthwhile in a sequential scenario. In a mobile distributed scenario, instead, where flexibility is a must because the nature of the downloaded code cannot always be estimated a priori, the use of such extra subtyping constraints at communication time is a small price to pay, especially since this allows to combine local and foreign code without any recompilation.

Dynamic type checking is minimized (i.e., only during the exchange of code); thus, in implementing the package `momi`, a particular importance was given to this matter: the actual content (the methods) of classes and mixins is only examined during mixin application in order to make any method call on objects completely untyped. This reflects, as we will remark during the description of objects and methods, in the fact that no further type-checking is performed when invoking a method on an object (and when performing casts in the Java-code corresponding to MoMi method bodies), and it is also consistent with the idea of having a virtual machine, that basically executes untyped code as in an assembly language. Of course this relies on the assumption that the compiler generating code that uses `momi` is correct. How to have a formal proof of such a soundness property may be the subject of future studies.

3 The Main Classes of the Package

The package `momi` contains the classes that abstract the concepts of the MoMi language: objects, mixins, classes, types and methods. Indeed also methods are low level structures in the package: they are manipulated in order to assemble new classes at run-time and to build object method tables. Given a class C (resp. a mixin M) in the high level language, the compiler is supposed to generate a subclass of `MoMiClass` (resp. `Mixin`), a subclass of `MoMiObject` and a subclass of `MoMiMethod` for each methods defined in the class C (resp. *defined*, *redefined* and *expected* in the mixin M). All the functionalities provided by the package that concern manipulation of these generated classes (i.e., object instantiation and mixin application) are based on the assumption that the compiler has already successfully type-checked the source language.

The class `MoMiMethod` represents a method defined in a MoMi class or mixin and it is the base class from which all the generated methods have to derive. Every `MoMiMethod` owns a private stack where parameters are pushed before calling the method, popped from within the method body, and where the return value is pushed from within the method. Classes derived from `MoMiMethod` have to implement the Java method `invoke`. This method can throw a `NoSuchMethodException` in case of a "message-not-understood" error; however, this exception should never be raised as long as the compiler produces correct code. Consistently with standard object-oriented language implementations (see, e.g., [23,22]) the pointer to the current target object `self` (or `this`) is passed to the method at invocation time.

Thus, the steps for calling a method consist in pushing the parameters on the stack of the method (`pass_argument`), invoking the method passing the object on which the method is called and, in case, retrieving the result from the stack (`get_result`). These basic instructions are to be generated by the compiler, that also has the responsibility of pushing the parameters on the stack in the right order and of trying to retrieve the result only if the method returns a value (in this case it has also to cast the returned value to the right type).

Example 1. Let us assume that we have to call a method that takes as argument an integer and a string (passing the integer `100` and the string `"foo"`) and returns a boolean. The code that the compiler should generate, if the reference to the method is stored in `meth` (how to obtain such a reference and how to pass the "right" `self` pointer will be shown in the following), is similar to the following one:

```
meth.pass_argument(new Integer( 100 ));
meth.pass_argument(new String( "foo" ));
meth.invoke(_pass_self);
b = (Boolean) meth.get_result() ;
```

The body of `invoke` has to be generated accordingly: it must pop the arguments from the stack and assign them to the parameters, and cast the pointer `self` to the actual type. A return statement will be translated to a push of the value in the stack followed by an exit statement for leaving the method (i.e., a standard Java `return`).

Example 2. Going back to the previous example, let's assume that the body of the method we are calling is as follows

```
m(i : int, s : str) : bool
begin
 if (i < 10) then
  return i > my_field # my_field is an instance variable
 endif;
 return i >= 0
end
```

The compiler should produce code similar to the following one:

```
public void invoke(MoMiObject _self) throws NoSuchMethodException {
   MyObject self = (MyObject) _self;
   String s = (String) stack.pop();
   Integer i = (Integer) stack.pop();

   if ( i.intValue() < 10 ) {
      stack.push( new Boolean( i.intValue() > self.my_field.intValue() ) );
      return ;
   }
   stack.push( new Boolean( i.intValue() >= 0 ) );
   return ;
}
```

Indeed the `self` pointer is casted to the actual (expected) type (say `MyObject`), and the arguments are popped from the stack and assigned to the formal parameters (once again after casting). Notice how the `my_field` instance variable is explicitly prefixed with the object `self` in the generated code.

If the compiler for our language correctly type-checked the source program, the type casts in the generated code will be type-safe.

The base class for all MoMi objects is `MoMiObject`. The derived classes only have to add the fields (instance variables) declared in the class this object is instantiated from, and this is basically the only task the compiler has to perform when generating code for MoMi objects. The class `MoMiObject` relies on the base class `WithMethods` that includes a table of `MoMiMethods`, `methods`, i.e., an `Hashtable`, where the key is the method name (the untyped nature of the package is proved by methods being searched only by name and not by their types). Such table corresponds to the table used for method dynamic binding in C++ and Java. `WithMethods` is also the base class for `Mixin` and `MoMiClass` representing, respectively, MoMi mixins and classes (shown later).

An object contains also the structure of objects of superclasses. In languages such as C++, this boils down to memory offset careful management: the instance variables of an object of a derived class start, in the memory layout of the object, where the instance variables of an object of the parent class end. This enables objects of derived classes to be used in place of objects of superclasses. Once again, in such languages, this management can be performed statically by the compiler since the structure of the subclass and of the superclass are available at compile time.

This cannot be done in MoMi since the structures of objects are known only at run-time. Thus the relation between the instance variables of the mixin (subclass) and those of the superclass has to be established dynamically through a reference; this reference is represented by the field `next` in the `MoMiObject` class. This is also consistent with the "compositional" nature of mixins: mixin-based inheritance is more similar to object

composition, than to object extension. Thus, when an object of the derived class has to be passed to a method inherited by the superclass (i.e., a method expected by a mixin, or the inherited implementation of a redefined method), the object pointed to by `next` has to be passed, instead of `self`. This is a crucial matter, since a method can access the fields of the `self` object and indeed it expects an object of the class where the method is defined (notice the cast to `MyObject` in Example 2).

When the `self` pointer has to be explicitly passed at method invocation time, the "right" `self` has to be passed, i.e., the one that is instance of the class the called method is defined in. Since dynamic binding is used for method invocation, the "right" `self` depends on the specific method version that is called. That is why we need to use, apart from the method table, also a *self table* indexed by method names. Thus, first the reference to the method to be called has to be retrieved by using `get_method` and then the pointer to the right `self` has to be retrieved by using `get_self`. The compiler has not to take care of building the `self_table`: this task is performed during object creation by the package `momi`.

The technique shown above allows to implement dynamic binding for method invocation: in particular, if the method that is being called is an expected method, then the implementation provided by the base class will be called, passing the self inherited from the base class. If a method of the base class is calling a method that has been redefined by the mixin, then the new implementation is called, passing as self the object of the derived class.

Example 3. If the method `m` of the Example 1 is invoked on the object `obj` the complete invocation sentence is as follows:

```
meth = obj.get_method("m");
_pass_self = obj.get_self("m");
meth.pass_argument(new Integer( 100 ));
meth.pass_argument(new String( "foo" ));
meth.invoke(_pass_self);
b = (Boolean) meth.get_result() ;
```

`_pass_self` may be different from `obj`: if `m` is an expected method, then `_pass_self` will be a `self` contained in `obj` (since such method is inherited from the superclass); alternatively, if `m` is a redefined method and it is being invoked from a method of a superclass, since dynamic binding is employed, `_pass_self` will be an object actually containing `obj`.

The class `Mixin` of the package `momi` is the most complex one since it provides means for performing the mixin application operation and for performing object instantiation. Notice that this class will also be the base class for `MoMiClass`; this makes sense from the design point of view, since a class can be seen as a mixin where there are no expected methods and no redefinitions.

The crucial part of `Mixin` is the (static) method `apply` that applies a mixin to a class and returns a new derived class. This method basically copies in the new class' method table all the methods defined and redefined by the mixin and all the methods expected by the mixin taking them from the superclass. The method `apply` only creates a new `MoMiClass` object, with the most specialized version of each method, but it does not perform any operation concerning dynamic and static bindings: indeed, this will be

performed at object instantiation time, carried on by the method `new_instance`. First of all this method calls a *factory method* [19], `_new_instance`, that classes deriving from `Mixin` and `MoMiClass` have to redefine in order to return an instance of the corresponding class deriving from `MoMiObject`. Finally, the `self` tables of the new created object is initialized.

A crucial feature of MoMi is that object-oriented values (i.e., objects, classes and mixins) are explicitly typed when they are exchanged among distributed sites. The compiler has to statically decorate these values when they are sent to a remote site. This annotation will consist in inserting the statically built type in a message containing an object-oriented value. The package `momi` supplies the classes representing the types for the following items: methods, objects, classes and mixins, apart from basic types (that depends on the run-time systems). The interface for MoMi types is `MoMiType`, containing methods `equals` and `subtype`.

The class `RecordType` stores method types in an hashtable, and performs comparison on record types, not considering the order of method types. `ObjectType` and `ClassType` basically rely on this class for performing comparisons. The class for mixin types, `MixinType` keeps a different record type for defined, expected and redefined methods. The comparison is consistent with the subtyping rule presented at the beginning of Section 2: for redefined methods the equality is required, while for expected methods the subtyping relation is inverted. Let us observe that in the previously described classes all these classes for types are never used. This shows the untyped nature of the run-time environment. Types will be used only during the communication.

4 Object-Oriented Mobility in X-Klaim and Klava

X-Klaim [7,8,2] is a mobile code programming language where communication takes place through multiple tuple spaces (as in Linda [20]) distributed on sites; sites are accessible through their localities (e.g., IP addresses). The reserved locality `self` can be used to access the local execution site (in order to avoid confusion with the object-oriented `self`, we use `this` in object-oriented expressions instead of `self`). A tuple t can be inserted in a tuple space located at locality ℓ with the operation **out**$(t)@\ell$ and removed (resp. read) with **in** (resp. **read**). Pattern matching is used for selecting tuples. X-Klaim programs are compiled into Java programs that use the package Klava [9] that provides the run-time system for X-Klaim operations. Both X-Klaim and Klava have been extended in [2] in order to use `momi` for object-oriented code mobility. In particular, if x is a class variable, the operation **in**$(!x)@\ell$ retrieves a tuple from ℓ containing a class with a subtype of x. X-Klaim is based on the kernel language Klaim [15]; the formal extension of the Klaim model with the MoMi features is presented in [4].

The package Klava already provided all the primitives for network communication, through distributed tuple spaces, and, in particular, for code mobility, not supplied by `momi`. Thus the package has been modified in order to be able to exchange code that relies on `momi`, and for performing subtyping on `momi` elements during pattern matching by relying on the `MoMiType` classes and the associated subtyping. On the other hand, the X-Klaim compiler generates code that uses both the Klava package and `momi`. Obviously, before generating code, it also performs type checking according to the typing system defined by MoMi. Let us now consider a simple mixin definition:

```
public class MyMixin extends Mixin {
  public MyMixin() {
    set_method("myinit", new MyMixin_myinit());
    set_method("print_fields", new MyMixin_print_fields());
    set_method("init", new MoMiMethod("init", MoMiMethod.EXPECTED));
    set_method("get_i", new MoMiMethod("get_i", MoMiMethod.EXPECTED));
  }

  protected MoMiObject _new_instance() { return new MyMixinObject(); }

  public static MixinType create_type() {
    MixinType new_type = new MixinType();
    new_type.addMethod(MyMixin_myinit.create_type(), MoMiMethod.DEFINED);
    new_type.addMethod(MyMixin_print_fields.create_type(), MoMiMethod.REDEFINED);
    new_type.addMethod(MyMixin_init.create_type(), MoMiMethod.EXPECTED);
    new_type.addMethod(MyMixin_get_i.create_type(), MoMiMethod.EXPECTED);
    return new_type;
  }
}
```

Listing 1. The Java class generated by the compiler for the mixin `MyMixin`.

```
mixin MyMixin
  expect init(a : int, b : str);
  expect get_i() : int;
  redef print_fields() begin ... end;
  def myinit(x : int, y : str) begin ... end
end
```

that expects from the super class two methods, redefines one, and defines a new method. The Java class generated by the X-KLAIM compiler for this mixin definition is shown in Listing 1. For each method of the mixin, a subclass of `MoMiMethod` is generated by the compiler. The generated constructor of `MyMixin` builds the method table. Notice that, for each expected method, objects of the base class `MoMiMethod` are inserted. Instead, the generated classes for expected methods are only useful for retrieving the types of these methods. The compiler generates an appropriate `create_type` method that allows to retrieve the static type (e.g., for decorating mobile code that has to be sent to a remote site). For instance, concerning `MoMiMethod`s, such a type will contain the entire method signature, while for mixins it will contain three record types. Classes generated for `MyMixinObject` and the other methods are similar.

4.1 A Mobile Printer Agent in X-KLAIM

The programming example shown in this section involves mixin code mobility, and implements "dynamic inheritance" since the received mixin is applied to a local parent class at run-time. We assume that a site provides printing facilities for local and mobile agents. Access to the printer requires a driver that the site itself has to provide to those that want to print, since it highly depends on the system and on the printer. Thus, the agent that wants to print is designed as a mixin, that expects a method for actually printing, `print_doc`, and defines a method `start_agent` through which the site can start its execution. The actual instance of the printing agent is instantiated from a class dynamically generated by applying such mixin to a local superclass that provides the

```
mixin MyPrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str
 begin
  return
   this.print_doc
    (this.preprocess("my document"))
 end;
 def preprocess(doc : str) : str
 begin
  return "preprocessed(" + doc +")"
 end
end

rec SendPrinterAgent[server : loc]
 declare
  var response : str
 begin
  out(MyPrinterAgent)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
mixin PrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str;
end

class LocalPrinter
 print_doc(doc : str) : str
 begin
  # real printing code omitted :−)
  return "printed " + doc
 end;
 init()
 begin
  nil # foo init
 end
end

rec ReceivePrinterAgent[]
 declare
  var rec_mixin : mixin PrinterAgent;
  var result : str
 begin
  in(!rec_mixin)@self;
  result :=
   (new rec_mixin <> LocalPrinter).start_agent();
  out(result)@self
 end
```

Listing 2. The printer agent example.

method `print_doc` acting as a wrapper for the printer driver. However the system is willing to accept any agent that has a compatible interface, i.e., any mixin that is a subtype of the one used for describing the printing agent. Thus any client wishing to print on this site can send a mixin that is subtyping compliant to the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 2, where **rec** is the X-KLAIM keyword for defining a process, presents a possible implementation of the printing client node (on the left) and of the printer server node (on the right). The printer client sends to the server a mixin `MyPrinterAgent` that complies with (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent`. In particular `MyPrintedAgent` mixin will print a document on the printer of the server after preprocessing it (method `preprocess`). On the server, once the mixin is received, it is applied to the local (super)class `LocalPrinter`, and an object (the agent) is instantiated from the resulting class, and started so that it can actually print its document. The result of the printing task is then retrieved and sent back to the client.

We observe that the sender does not actually know the mixin name `PrinterAgent`: it only has to be aware of the mixin type expected by the server. Furthermore, the sent mixin can also define more methods than those specified in the receiving site, thanks to the mixin subtyping relation. This adds a great flexibility to such a system, while hiding these additional methods to the receiving site (since they are not specified in the receiving interface they are actually unknown statically to the compiler).

Acknowledgments. I would like to thank the two co-authors of MOMI, Viviana Bono and Betti Venneri. The anonymous referees provided helpful suggestions for clearing up some aspects in the paper.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP 2000*, number 1850 in LNCS, pages 145–178, 2000.
2. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.
3. L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Manuscript.
4. L. Bettini, V. Bono, and B. Venneri. O'KLAIM: a coordination language with mobile mixins. In *Proc. of Coordination 2004*. To appear in LNCS.
5. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, number 2315 in LNCS, pages 56–71. Springer, 2002.
6. L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *FOOL 10*, 2003.
7. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th IEEE WETICE*, pages 110–115. IEEE Computer Society Press, 1998.
8. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
9. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.
10. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.
11. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
12. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
13. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
14. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. ICSE '97*, pages 22–33. ACM Press, 1997.
15. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
16. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *CONCUR '96*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996.
17. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
18. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join Calculus. In *FSTTCS 2000*, volume 1974 of *LNCS*, pages 397–408. Springer, 2000.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
20. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
21. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Proc. HLCL '98*, volume 16.3 of *ENTCS*. Elsevier, 1998.
22. T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
23. S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.
24. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Proc. TPPP 94*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
25. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.

Streaming Services: Specification and Implementation Based on XML and JMF

Björn Althun and Martin Zimmermann

University of Applied Sciences, Offenburg, Germany
m.zimmermann@fh-offenburg.de

Abstract. The central purpose of this paper is to present a novel framework supporting the specification and the implementation of media streaming services using XML and Java Media Framework (JMF). It provides an integrated service development environment comprising of a streaming service model, a service specification language and several implementation and retrieval tools. Our approach is based on a clear separation of a streaming service specification, and its implementation by a distributed JMF application and can be used for different streaming paradigms, e.g. push and pull services.

1 Introduction

The last few years have witnessed an explosive growth in the development and deployment of networked applications that transmit and receive audio and video content over the Internet. New multimedia networking applications, also referred to as continuous streaming media applications – entertainment video, IP telephony, Internet radio, distance learning, and much more – seem to be announced daily.

Streaming applications can be characterized by the following features:

- **Dynamic changes**: a distributed streaming application may be reconfigured during its lifetime due to evolutionary and operational changes, e.g. creation or deletion of additional encoders components.
- **Different service modes**: Streams may be delivered via push and / or pull services.
- **Bandwidth requirements:** the correct execution of streaming services is often critically dependent on the available bandwidth between a media server and client

Fig. 1 shows a realistic example of a streaming service: a stream composed of a video and a audio live source (e.g. a speaker) and a image file (e.g. a background image) delivered via a hierarchy of media servers to the users. The temporal relationships define, that the video channel, the audio channel, and the image are delivered in parallel. The configuration of such a streaming service is a very complex process: selection, creation, and configuration of components (e.g. an encoder component), design and implementation of a distribution policy, etc..

The idea of our approach is to enable a simple specification of a streaming service, and to generate the required distributed software infrastructure, i.e. required software components, and their configuration by a service manager.

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 23–32, 2004.

Our approach is based on a clear separation of a streaming service specification - and its implementation by a distributed Java Media Framework (JMF) application and can be used for different streaming paradigms, e.g. push and pull services. The proposed framework is part of a multimedia project, financed by the state government of Baden-Württemberg in Germany.

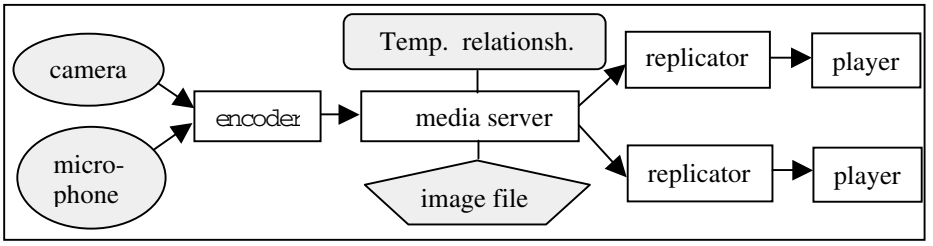


Fig. 1. Example of a media streaming service

The following figure (Fig. 2) illustrates our application development environment: A streaming service is specified in terms of media objects, layout and temporal relationships between media objects, quality of service requirements, and distribution policies. Such a service specification is then analyzed by a service manager. Driven by a JMF component library which contains existing streaming software components, such as encoders, and different types of media servers, the service manager creates a tailor-made distributed streaming application configuration.

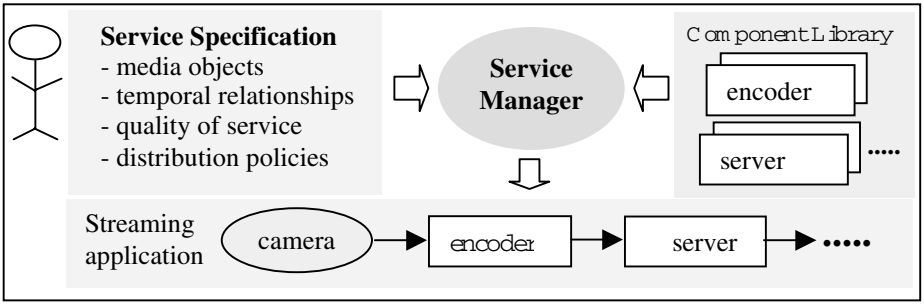


Fig. 2. Configuration of a media streaming service

In the following sections the streaming service specification language, the component library, the configuration rules, and the mapping onto a JMF application infrastructure are described in more detail.

2 Streaming Service Specification

A media stream can be identified by its media objects, related locations, and the quality of service used for access. Moreover, media streams can be categorized according to how the data is delivered:

- Pull: data transfer is initiated and controlled from the client side.
- Push: the server initiates data transfer and controls the flow of data.

The degree of control that a client program can extend to the user depends on the type of data source being presented. For example, an MPEG file can be repositioned and a client program could allow the user to replay the video clip or seek to a new position in the video. In contrast, broadcast media is under server control and cannot be repositioned.

In the following, we illustrate our specification technique. A streaming service specification, which is based on XML, is composed of the sections `<MediaObjects>`, `<QoS>`, and `<DistributionPolicies>`.

2.1 Media Objects

In the part `<mediaObjects>`, all media objects are listed, which are part of the streaming service.

We use the language SMIL [7] extended with some properties. In addition to SMIL, we can specify the type of a media object (live / file). Moreover, in case of live streams, the node where the live stream comes from can be defined.

Recommended by the World Wide Web Consortium (W3C), SMIL is designed to be the standard markup language for timing and controlling streaming media objects [2]. SMIL is a collection of XML elements and attributes that we can use to describe the temporal and spatial coordination of one or more media objects. A SMIL presentation is a structured composition of autonomous media objects, e.g. <video> and <audio> objects.

```
<StreamingService>
<MediaObjects>
<par>
        <image id="i1" src="bgLogo.gif" />
        <video id="v1" type="live" node="c1.fh-offenburg.de" />
        <audio id="a1" type="live"  node="c1.fh-offenburg.de" />
</par>
</MediaObjects>
```

Fig. 3. Specification of media objects based on SMIL

The <par> tag is used to specify that all objects are all rendered in parallel. In our example, the background image and the live stream (audio and video) is played in parallel (Fig. 3).

2.2 Quality of Service

In the part <QoS> the used service models are specified (Fig. 4). A streaming service can be made available to the users as a push and / or pull service. In case of the push

mode, we can additionally define when the media objects are send on the net (date and time), and the available bit rates. Moreover, the name of the transport protocol and the codec to be used can be specified.

Date and time attributes are also used in case of pull services. However, in case of a pull service these attributes define, when a media service is available, i.e. can be requested by a client. Finally, one can specify the maximum bandwidth which can be used ("consumed") by the related service.

```
<StreamingService>
   <MediaObjects> …. </MediaObjects>
   <QoS>
   <ServiceMode>
       <push codec="MPEG" protocol="RTP" date="2003-08-22" time= … >
                   <bitrate>100</bitrate>
                   <bitrate>200</bitrate>
    </push>
    <pull codec="MPEG" protocol="RTP" startDate=… endDate=… >
                   <bitrate>200</bitrate>…
       </pull>
    </ServiceMode>
    <MaxBandwidth>1000</MaxBandwidth>
   </QoS>
```

Fig. 4. Specification of QoS

2.3 Distribution Policy

Fig. 5 illustrates the specification of distribution policies. Each media service must be available on a *primary server*. As an optional part a set of *replicators* can be introduced. Replicators enable us to deliver streams to multiple media servers. At its most basic level, replication is a technology that enables one component to deliver a live stream to another component. In such a server-to-server replication, a primary server transmits a live stream to one or more additional servers. This allows to distribute the same stream to multiple media servers across a network.

A distribution strategy is related to a media service specification. In the conventional, or "push" form of server- to-server replication, the transmitter initiates the connection to the receiver. When a media player requests the broadcast, the receiver is ready to deliver the stream.

```
<StreamingService>
  <MediaObjects> …. </MediaObjects>
  <DistributionPolicy>
<primaryServer node="s1.fh-offenburg.de"/>
   <replicator node="s2.fh-offenburg.de" strategy="pull" creationOn="serviceRequest"/>
      <replicator node="s3.fh-offenburg.de"/>
      <replicator node="s4.fh-offenburg.de"/>
   </DistributionPolicy>
```

Fig. 5. Specification of a distribution policy

In case of pull replication, the initial media server (primary) does not deliver the stream to the receiver until the first media player makes a request. There is a slight delay as the receiver requests, receives, and delivers the stream. After that, the stream is live on the receiver, and subsequent player requests do not involve the session setup delay.

The attribute `creationOn` in Fig. 5 defines when a component (e.g. a replicator) should be created. In case of value "serviceRequest", the component is created when the service must be provided (e.g. .in case of push services depending on the value of time and date).

3 Component Library

Available JMF streaming components are stored in a separate component library. The properties of existing components (e.g. a specific encoder component) are described by related XML specifications.

Component Property	Encoder
Name	encoder (avi2mpeg)
Source (Reference to JMF code)	…
Processor Speed Req.	2 Ghz
Operating System	…
Input Formats	avi
Output Formats	mpeg
Bitrates	12k,…,1.5M
Sure Stream	Yes
Service Mode	Push/Pull

Fig. 6. Component Properties

Our current development environment uses the following component categories (Fig. 6 illustrates the properties of a encoder component):

- **Encoder:** During encoding, a source media object is transformed into streaming media using "codecs" (compression/decompression algorithms). Encoder receives

the source media as a file or live audio/video and uses a codec to compress the media source's data into an output format. SureStream enables to create a single stream recorded for multiple target audiences. For example, a video can be delivered for 56 kbps, and 112 kbps audiences. A player will then automatically use the correct stream based on the user's connection speed.

- **Media Server**: stores media objects. In case of live streams an encoder must be connected with a media server. Replicators are media servers which store replicas of media objects.
- **Buffer**: is a component which allows to store streaming data. Such a component is required, when the available bandwidth is smaller then the required application bitrate.

4 Implementation

In the following, we describe the management of services, components, and configurations as well as the mapping of service specification onto JMF.

4.1 Management of Services, Components, and Configurations

Multimedia services as well as properties of software components are stored as XML documents in a XML database.

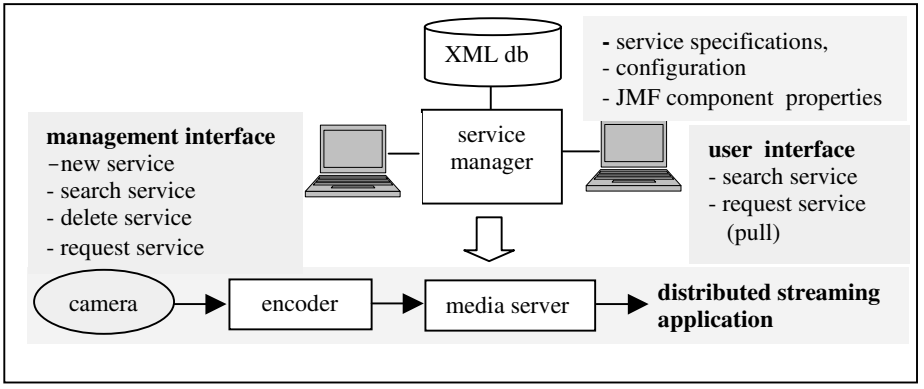


Fig. 7. Service Mananger

The services are managed by a service manager (see Fig. 7), which provides in the current implementation a simple interface to search, and start (request) services. Moreover, a management interface enables creation of new services, and deletion of existing services. Push services are started automatically according to the specified date and time values. Starting a service means to create the required JMF components, e.g. a encoder and a media server component as well as to establish the communication relationships by construction of appropriate media locators.

Before a new service is accepted and stored in the service manager, a consistency checker validates the availability of nodes, and media objects according to the service specification. Then a service specification is analyzed by the service manager, which will select and configure appropriate JMF components.

We use Apache's Xindice [11], which is a database designed from the ground up to store XML data or what is more commonly referred to as a native XML database. XPath is a (W3C-standard) language used to address a section of an XML document. It uses a compact, non-XML syntax to facilitate use of XPath within URIs. A single string of XPath can address any section of an XML document. According to our service specification language, properties of media objects and service modes can be used as part of a retrieval (see Table 1).

Table 1. Search criteria

Search Criteria	**Examples**
Media Objects: Existence	existence of elements, e.g. <audio>, <video>
Properties	attribute values, e.g. type, size, format
Service Modes: Existence	existence of services, e.g. pull services
Properties	attribute values, e.g. date, time, bitrates

Example:
"Select all live services which are available in push mode on 1 September 2003 and delivered by at least 100 kbit/s."

XML documents can be represented as a tree view of nodes. XPath uses a pattern expression to identify nodes in an XML document. By using square brackets in an XPath expression we can specify an element further. In XPath all attributes are specified by the @ prefix. In the following, a search expression for our example is illustrated: The search expression specifies two required properties of media objects: all push serviceMode elements which have an attribute named date with a value of '2003-09-01' and an attribute named bitrate with an value of at least '100':

```
/StreamingService/QoS/ServiceMode[push[@date='2003-09-01'
and @bitrate>100]]
```

4.2 Implementation of Components Using JMF

Java Media Framework (JMF) [8] provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF uses a simple streaming processing model (see Fig. 8). A *data source* encapsulates the media stream much like a video tape and a *player* provides processing and control mechanisms similar to a VCR.

Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors. Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media.

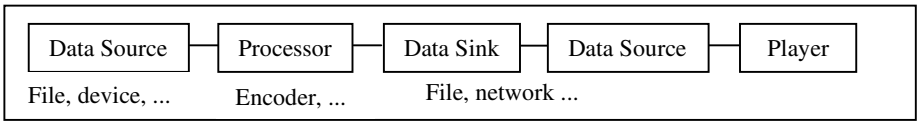


Fig. 8. JMF streaming model

A *processor* component provides control over what processing is performed on the input media stream. In addition to rendering media data to presentation devices, a Processor can output media data through a DataSource so that it can be presented by another Player or Processor, further manipulated by another Processor, or delivered to some other destination, such as a file.

The JMF API consists mainly of interfaces that define the behavior and interaction of objects used to capture, process, and present time-based media. By using intermediary objects called *manager*s, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes.

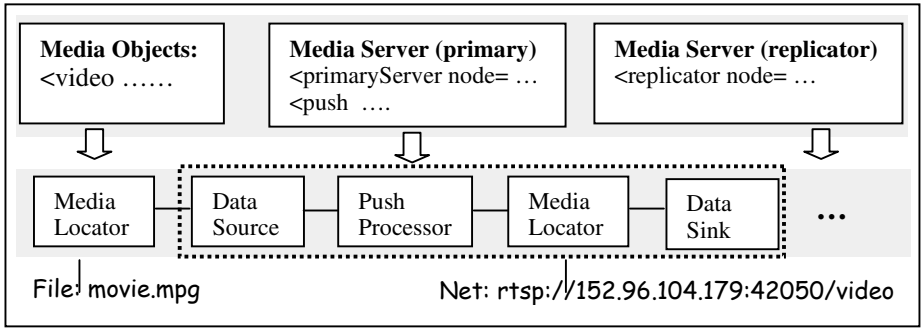


Fig. 9. Mapping onto a JMF architecture

A DataSource encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the source cannot be reused to deliver other media. A DataSource is identified by either a JMF MediaLocator or a URL. The following implementation rules are used to map a service specification onto a set of JMF components (see Fig. 9) :

- Media objects in XML are implemented by Media Locators in JMF
- Server components are implemented by a JMF "pipeline" configuration: DataSource, Processor, MediaLocator, and DataSink objects.
- The functionality of a component (e.g. encoding, splitting) is implemented by an appropriate JMF processor object (e.g. a push processor object)
- Communication relationships between server components (e.g. a primary server and a replicator) are implemented by DataSource and DataSink objects.

Fig. 10 shows some a Java code fragment, which implements the initial configuration of the main objects as part of a media server. A MediaLocator is similar to a URL and can be constructed from a URL.

First, a MediaLocator object for a movie object (movie.avi) is created. The result of calling the function createProcessor is a new Processor object, which is connected with the previously created MediaLocator.
Finally a MediaLocator object bis created and connected with the network.

```
//Creation of a media locator for a video file
mMovieLocator = new MediaLocator("movie.avi");

//Creation of a processor that knows where the movie is.
mProcessor=Manager.createProcessor(mMovieLocator);

//Get the output data source of the processor
mDataSource = mProcessor.getDataOutput();

// Creates a RTP transmit data sink, and start transmission
// rtp://Client_IP_Number:port/media object
MediaLocator mClientLocator =
  new MediaLocator("rtp://"+"152.96.104.179"+":"+42050+"/video");
mDataSink = Manager.createDataSink(mDataSource, mClientLocator);
mDataSink.open();  mDataSink.start();  .....
```

Fig. 10. Example JMF code

5 Related Work

The Session Description Protocol (SDP) [1] allows to specify sessions in terms of session identifiers, protocols, media objects. However, it does not support distribution policies and QoS specifications.
In [6] a component-based approach is described, which reflects the devices being used by the application. In the approach a explicit configuration is used, based on component types encoder, filter, network connector, and display.
In [5] stream interface descriptions based on compatibility rules are introduced to determine if two or more devices (their interfaces) are capable to interoperate (bind to each other). If interfaces are incompatible (unable to bind) media processing (filters) are inserted which transcodes between the incompatible interfaces. In [4] a concept for association management is introduced which enables passing of streamed movies with synchronization & control.
Infopipes [9] provide a high-level interface tailored to information flows and flexibility in controlling concurrency and pipeline setup. However it does not provide a service specification technique.
In [10] a mobile streaming media CDN (Content Delivery Network) architecture is presented in which content segmentation, request routing, prefetch scheduling, and session handoff are controlled by SMIL (Synchronized Multimedia Integrated Language) modification. The approach concentrates on the segmentation aspect, for mobile users.

6 Conclusion

A new approach for specification and implementation of streaming services has been introduced. The major contributions and extensions aim to provide a high level specification of streaming services in terms of media objects, quality of service, and distribution policies. Based on such a service specification, a set of tools support automatic creation of a distributed streaming application based on a component library containing JMF streaming components.

The clear separation of specification and implementation has several advantages. First, it supports modularity, reusability and extensibility. Additionally, our approach supports more flexibility in combining different aspects taking into account the specific application requirements.

The prototype has shown, that the object model behind JMF is well suited for automatic generation of implementations. At implementation level, a general object-oriented architecture supports modularity and reuse of software.

We developed a set of tools for the mapping of application specifications onto an object-oriented implementation model. This includes tools for consistency check of a service specification as well as generation of application configurations based on a JMF. We use the tools to enable a tailormade configuration of distance learning streaming applications. Future work will concentrate on the extension of the service model to support also personalized multimedia streaming.

References

1. H. Schulzrinne, J. Rosenberg: The Session Initiation Protocol: Internet-Centric Signaling, IEEE Communications Magazine, October 2000
2. SMIL: www.w3.org/TR/smil20/
3. M. Zimmermann: Implementing Configuration Management Policies for Distributed Applications*, in Distributed Systems Engineering Journal , 1996*
4. V. Kahmann, L. Wolf: A Proxy Architecture for Collaborative Media Streaming Workshop on Multimedia Middleware, October 5th, 2001, Ottawa, Canada, ACM Press, New York
5. H.O. Rafaelsen: Towards support for ad-hoc multimedia bindings, Workshop on Multimedia Middleware, October, 2001, Ottawa, ACM Press, New York
6. H. Naguib, G. Coulouris: Towards Automatically configurable multimedia applications Workshop on Multimedia Middleware, October, 2001, Ottawa, ACM Press, New York
7. J. Ayers et al., *Synchronized Multimedia Integration Language (SMI*L) 2.0, World Wide Web Consortium Recommendation, Aug. 2001, www.TR/2001/REC-smil20-20010807/.
8. http://java.sun.com/products/java-media/
9. A.P. Black: An abstraction for multimedia streaming, Multimedia Systems 8(2002)
10. Takeshi Yoshimura: Mobile Streaming Media CDN Enabled by Dynamic SMIL, WWW2002, May 7–11, 2002, Honolulu, Hawaii, USA, ACM 1-58113-449-5/02/0005
11. www.apache.org

Hard Real-Time Implementation of Embedded Software in JAVA***

Jean-Pierre Talpin, Abdoulaye Gamatié, David Berner, Bruno Le Dez, and Paul Le Guernic

INRIA-IRISA, Campus de Beaulieu, 35042 Rennes, France

Abstract. The popular slogan "*write once, run anywhere*" effectively renders the expressive capabilities of the JAVA programming framework for developing, deploying, and reusing target-independent applets. Its generality and simplicity has driven most attention of the compiler technology community to developing just-in-time and runtime compilation techniques, local and compositional optimization algorithms. When it comes to real-time JAVA and to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems) where conformance to real-time specifications is critical. We show that synchronous design tools, and particularly the design workbench POLYCHRONY, allow for a complete modeling of embedded software written in a high-level and general purpose programming language such as JAVA. The synchronous approach provides a formal engineering model and methodology, using global transformation and optimization techniques, that allow for a JAVA program written once to be mapped on any distributed target architecture. We present a technique to import a resource constrained, multi-threaded, RT JAVA program, together with its runtime system API, into POLYCHRONY. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given JAVA program on either a single-threaded target or a distributed architecture. This technique allows to generate stand-alone (JVM-less) executables and to remap threads onto a given distributed architecture or a prescribed target real-time operating system. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design (in JAVA) and its actual, real-time and resource constrained implementation.

1 Introduction

The well-known slogan "*write once, run anywhere*" effectively renders the expressive capabilities of the JAVA programming framework for developing, deploying, and reusing target-independent applets. Its generality and simplicity has driven

* JAVA is a registered trademark of SUN Microsystems

** Work funded by the RNTL project EXPRESSO

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 33–47, 2004.

most attention of the compiler technology community to developing just-in-time and runtime compilation techniques, local and compositional optimization algorithms. When it comes to real-time Java (Rt Java) and to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems) where conformance to real-time specifications is critical.

We show that synchronous design tools, and particularly the design workbench Polychrony[1], allow for a complete modeling of embedded software written in a high-level and general purpose programming language such as Java. The synchronous approach provides a formal engineering model and methodology, using global transformation and optimization techniques, that allow for a Java program written once to be mapped on any distributed target architecture.

We present a technique to import a resource constrained, multi-threaded, Rt Java program, together with its runtime system Api, into Polychrony. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given Java program on either a single-threaded target or a distributed architecture. This technique allows to generate stand-alone (Jvm-less) executables and to remap threads onto a given distributed architecture or a prescribed target real-time operating system. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design (in Java) and its actual, real-time and resource constrained implementation.

We illustrate this technique by considering a simple yet representative case study: an even-parity checker (Epc). We show how correctness by construction is enforced from the description of its functional architecture (in Rt Java) to its mapping onto a given target architecture. This case study demonstrates the ability of the Polychrony workbench to automatically perform otherwise difficult engineering tasks such as thread remapping and target-specific deployment, starting from a given, high-level, Rt Java design.

Overview. In section 2 we present the considered functional profile of Rt Java. After a brief introduction to our modeling platform Polychrony in section 3, section 4 describes the architecture of our modeling tool. In section 5 we finally describe how to take advantage of the formal techniques of Polychrony in order to address critical issues in hard real-time system design.

2 A Hard Rt Java Profile

The Rt Java profile (i.e. the Api of the Jvm extension) considered in the present article has been implemented in the context of the Expresso project[2] and inspired by the Ravenscar-Java "high-integrity profile" (Hip) for Rt Java [12]

[1] `http://www.irisa.fr/espresso/Polychrony`
[2] `http://www.irisa.fr/rntl-expresso`

(there are other specifications for Rt Java such as [4]). It consists of a subset of the Rt Java specification aimed at meeting non-functional requirements of airborne systems. Table 1 details the most significant features of this profile (some contructs like memory management are not considered in this article). The Expresso packages provide extensions to thread management classes allowing for the simulation of real-time threads and event handlers. The class AsyncEvent and its sub-classes define data-structures to encapsulate hardware interrupts and events to be recycled within the real-time Jvm by appropriate handlers (class AsyncEventHandler and sub-classes) according to the pace of the real-time kernel. Processing in nominal mode within a real-time application is performed using real-time and periodic threads (classes RealTimeThread and PeriodicThread). These allow to schedule the execution of a sequential piece of code according to real-time constraints (period, deadline, duration, priority) and can be set using shared data-structures defined in the SchedulingParameters class.

Table 1. Excerpt of the Expresso package class hierarchy

. . . /. . .	
class SchedulingParameters	class RealtimeThread
class AsyncEvent	(implements Schedulable)
class SporadicEvent	class Initializer
class SporadicInterrupt	class NoHeapRealtimeThread
class AsyncEventHandler	(implements Schedulable)
(implements Schedulable)	class PeriodicThread
class BoundAsyncEventHandler	class MonitorControl
	. . . /. . .

Airborne Software Requirements. The Hip profile for Rt Java further describes programming guidelines for the use of this profile to meet structural and functional requirements imposed by certification authorities. The syntactic simplicity of these requirements make them at the same time easy to translate into programming guidelines by users, easy to implement as syntactic checks with the help of a modeling tool, and easy to analyze using rate-monotonic analysis techniques.

- A program consists of a fixed number of threads.
- Thread and memory allocation is performed during service startup.
- No dynamic memory management during operational service.
- Threads have access to the scope of the program.
- Threads are either periodic or sporadic.
- Threads use synchronization to avoid priority inversion.

In the present study towards modeling Rt Java within the synchronous multi-clocked design workbench Polychrony, we center our focus on this very subset of Java and demonstrate that its functionalities fit within our model of computation.

An Intermediate Format. To allow for the translation of a RT JAVA program into POLYCHRONY starting from either its source code or its bytecode, we use the tool SOOT[3] as JAVA compilation front-end to pre-process classes and obtain an optimized JIMPLE intermediate representation. JIMPLE is a very handy format to process JAVA classes. It consists of explicitly typed, stack-less, 3-address statements (grammars *stm* and *rtn*) that manipulate either immediates or references (grammars i and v).

(immediate)	i	$::= l \mid c$	(register or constant)
(operator)	$\star$	$::= + \mid - \mid \ldots$	(native plus, minus, etc)
(variable)	v	$::= i[i] \mid i.[x] \mid x \mid l$	(array , class field or local)
(reference)	r	$::=$ caughtexception	(current exception)
		$\mid$ parameter c	(method parameter)
		$\mid$ this	(self)
(declaration)	*dec*	$::= v = i$ instanceof t	(instantiation)
		$\mid v =$ new $t[i]$	(memory allocation)
		$\mid t\,x$	(type declaration)
(program)	*run*	$::=$ *blk* $\mid$ *run*; *run*	(sequence of blocks)

Fig. 1. A grammar of JIMPLE (programs)

In the JIMPLE grammar (Figure 1), a local variable is noted l, a constant c, a type t, a program label L, and class field name x. The *run* sequence of a thread consists of a sequence of blocks *blk* that consist of a label L, a sequence of operations *stm* and a return statement *rtn*. A declaration (grammar *dec*) may not occur in the run method of a thread or in an event handler as it might dynamically allocate memory. Hence, all declarations are assumed to be present in the main initialization class of the program, or in the init method of thread classes (executed once at system start).

(block)	*blk*	$::= L :$ *stm**; *rtn*	(sequence of *stm*s)
(statement)	*stm*	$::= v = i \star i$	(native operation)
		$\mid v =$ invoke $i\,(i^*)$	(method invocation)
		$\mid l := [v \mid @r]$	(local variable)
(return)	*rtn*	$::=$ [enter $\mid$ exit]monitor i	(locks)
		$\mid$ goto L	(goto)
		$\mid$ if i then L	(test)
		$\mid$ return	(return)
		$\mid$ throw i	(throw exception)
		$\mid$ catch t from L to L using L	(catch exception)

Fig. 2. A grammar of JIMPLE (statements)

As a result, the SOOT toolbox provides an appropriate front-end to resolve high-level object-oriented features and perform specific optimizations, allowing

[3] http://www.sable.mcgill.ca/soot

us to focus on the translation of the JIMPLE imperative notation into the multi-clocked data-flow design language SIGNAL, presented next. JIMPLE produces explicitly typed and initialized declarations as well as explicit locks in the presence of exceptions (monitors are released before exceptions are raised).

3 POLYCHRONY for Embedded System Design

The POLYCHRONY workbench implements a multi-clocked synchronous model of computation (the polychronous MoC [9]) to model control-intensive embedded software using the SIGNAL language and to support a formal, refinement-based, design methodology [13] with companion decision procedures to validate key design steps using precise formal design properties (e.g. controllability of a component by its environment or invariance of a design refinement under flow-equivalence) and/or automatic design transformations and refinement algorithms (control hierarchization, protocol synthesis).

3.1 An Introduction to SIGNAL

SIGNAL belongs to the family of synchronous languages such as ESTEREL and LUSTRE. A SIGNAL *process* P consists of simultaneous equations over *signals*. A signal $x \in \mathcal{X}$ describes an infinite flow of values $v \in \mathcal{V}$. We write $\boldsymbol{x}$ for a sequence or tuple $(x_1, \ldots, x_n)$ of signals. An equation $\boldsymbol{x} := f\boldsymbol{y}$ denotes a relation between a sequence of input signals $\boldsymbol{y}$ and a sequence of output signals $\boldsymbol{x}$ by a function or combinator f. SIGNAL requires three primitive operators: the equation $x := y\$1\,\mathsf{init}\,v$ initially defines x by v and then by the previous value of y in time, the equation $x := y\ \mathsf{when}\ z$ defines x by y when z is true and the equation $x := y\ \mathsf{default}\ z$ defines x by y when y is present and by z otherwise. The synchronous composition $P \parallel Q$ of two processes P and Q consists of the simultaneous solution of the system of equations P and Q.

$$P ::= \boldsymbol{x} := f\,\boldsymbol{y} \mid (P \parallel Q) \mid P\,\mathsf{where}\,x$$

A First Example. As an illustration we consider the definition of a simple counting process Count , below.It accepts an input signal rst and delivers the integer output signal val. A local variable cnt, initialized to 0, stores the previous value of val (equation cnt := val$1 init 0). When the event rst occurs, val is reset to 0 (i.e. 0 when rst). Otherwise, cnt is incremented (i.e. (cnt + 1)). The activity of Count is controlled by the clock of its output val which differs from that of its input rst. We write process Count = (? event rst ! integer val) for the declaration of a process named Count of input rstand of output val.

```
process Count = (? event rst ! integer val)
  (| cnt := val$1 init 0
   | val := (0 when rst) default (cnt + 1)
   |) where integer cnt end;
```

time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
rst		true					true			
val	1	0	1	2	3	4	0	1	2	3
cnt	0	1	0	1	2	3	4	0	1	2

Clocks and Causality Relations. In SIGNAL, sequential code generation (to, e.g., ANSI C, JAVA, VHDL) is a design stage performed on a given model (e.g. the modulo 3 counter, figure 3) subsequently to an analysis of synchronization relations (e.g. signals i, s and o are synchronous, written $o \hat{=} s \hat{=} i$, figure 3) and scheduling relations (e.g. c and o cannot happen before s when i is present, written $s \rightarrow^i c$ and $s \rightarrow^i o$, figure 3) are inferred. This relational information is used to construct a global and canonical control flow graph (described next). This transformation allows, for instance, to synthesize a single automaton from the model of multiple threads. Conversely, it allow to synthesize synchronization protocols to correctly map these threads on a distributed architecture.

model	clocks	schedule	code
s:=o\$1 init 2	$s \hat{=} o$	$\emptyset$	if i then { c = (s == 2);
c:=truewhen ($s = 2$)	$c \hat{<} s$	$s \rightarrow^i c$	if c then o = 0
o:=(0 when c) default $s + (1$ when $i)$	$o \hat{=} s \hat{=} i$	$s \rightarrow^i o$	else o = s + 1;
			s = o; }

Fig. 3. From synchronous multi-clocked equations to sequential C code

The hierarchization of the control flow graph (Figure 4, from [3]) is the key transformation performed by POLYCHRONY on a given SIGNAL specification. Given an inferred clock relation (e.g. $h_3 = h_1 op\ h_2$, below), it allows to optimally place clocks (e.g. h_3, below) in the control-flow tree by determining their least upper bound (e.g. h, s.t. $h > h_1$ and $h > h_2$, below). Each clock (e.g. c for $(s == 2)$, above) is the trigger of a set of actions that are scheduled in sequence by respecting the inferred scheduling relations (e.g. $s \rightarrow^i o$, above).

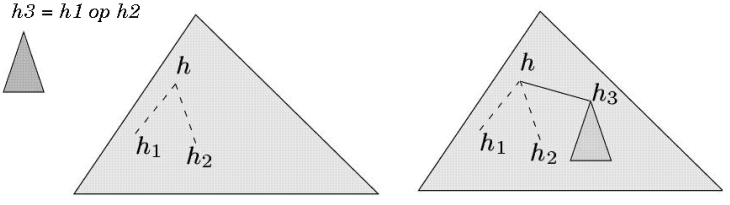


Fig. 4. Hierarchization of a control-flow graph for scheduler generation.

3.2 An Introduction to the APEX-ARINC RTOS Model of POLYCHRONY

The APEX[4] interface, defined in the avionics standard ARINC [2], provides avionics application software with basic services needed to access operating system resources. Its definition relies on the Integrated Modular Avionics approach (IMA [1]). A main feature in an IMA architecture is that several avionics applications (possibly with different levels of criticality) can be hosted on a single, shared computer system. Of course, an essential issue is to ensure safe allocation of shared computer resources in order to prevent fault propagation from

[4] APEX (**Ap**plication **Ex**ecutive) is a real-time operating system standard API for avionics applications

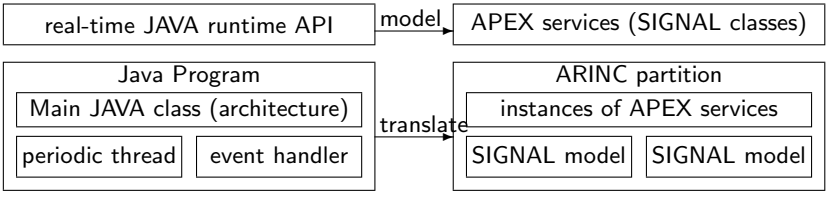
one hosted application to another. This is addressed through functional partitioning of the applications w.r.t. available time and memory resources. The allocation unit that results from this decomposition is the *partition* (see Figure 7 in the EPC example). A partition is composed of *processes* which represent the executive units (an ARINC partition/process is akin to a UNIX process/task). When a partition is activated, its owned processes run concurrently to perform the functions associated with the partition. The process scheduling policy is priority preemptive. Each partition is allocated to a processor for a fixed time window within a major time frame maintained by the operating system. Suitable mechanisms and devices are needed for communication and synchronization between processes (e.g. *buffer*, *event*, *semaphore*) and partitions (e.g. *ports* and *channels*). The APEX interface includes both services to achieve communication, synchronization, and services for the management of processes and partitions. Such services have been modeled in POLYCHRONY [6] and its commercial implementation (RT-builder, by TNI-VALIOSYS, `http://www.tni-valiosys.com`) is used at Hispano-Suiza and Airbus Industries. We use these services to model and implement hard RT JAVA applications.

4 A Hard RT JAVA Plugin for POLYCHRONY

We introduce our method to translate and model multi-threaded real-time JAVA programs in POLYCHRONY. Starting from a given, multi-threaded, RT JAVA design, POLYCHRONY allows to remap the functional thread architecture and to retarget the design onto a different target real-time operating system. Furthermore it can generate stand-alone (JVM-less) executables. As a result, it allows for a complete separation between the virtual threading architecture of the functional-level system design and its actual, real-time implementation.

4.1 Modeling the RT JAVA Virtual Machine Using POLYCHRONY

Modeling a RT JAVA application in POLYCHRONY requires to distinguish between the architecture of the application and its periodic threads and event handlers. The architecture (i.e. shared data-structures and thread parameters) can be obtained by scanning the main (initialization) class of the JAVA program as well as the init methods of thread classes.



The periodic threads and event handlers are described in the run methods of the thread classes. The init methods make use of operating system support (the RT JAVA API) that is modeled using the APEX services library of POLYCHRONY.

The architecture of the entire framework is formed by the APEX services of POLYCHRONY that model the RT JAVA API and correspond to the virtual machine. The structure of the actual JAVA program, that describes the functional threading and memory architecture, is translated to instances of APEX services and ARINC partitions. JAVA threads and event handlers are translated into SIGNAL processes.

4.2 Modeling RT JAVA Threads Using POLYCHRONY

To model RT JAVA threads we consider the JIMPLE intermediate format. In this format the model of a JAVA class is defined by the translation function $[\![run]\!]_E = \langle p \rangle$ and $[\![blk]\!]_E^C = \langle p \rangle$ which takes the code of an input method *run* as input, along with the activation clock of the target SIGNAL process, denoted by C. The *run* method is given an environment E which associates: method references r to local variables l (i.e. $\mathcal{R}_E(l) = r$); block and section labels L to activation clocks C (i.e. $\mathcal{C}_E(L) = C$); exceptions t raised at a label L to the corresponding exception flow (i.e. $\mathcal{X}_E(t, L) = (L_1, L_2)$, of target L_1 and handler L_2); a control-flow graph $\mathcal{G}_E(L)$, i.e. main$(\mathcal{G}_E)$ gives the main entry label of *run* and pred$^*(\mathcal{G}_E)(L)$ all predecessors of label L in the graph (Figure 7).

A real-time periodic thread (or event handler) is viewed as a sequence of critical sections that receive control from the partition-level scheduler via a clock tick, which triggers the execution of the thread, and a variable next_block, which directs control to the very block to execute in the thread. The type of the next_block variable is the enumeration of the block label names L, which are referenced to as $\#L$ in SIGNAL.

A *section* or block *blk* consists of a sequence of elementary statements *stm* delimited by a label L and a return statement *rtn*. A section label defines the entry point for a given transition. Hence, it is the symbolic value of the global state variable next_block of use in the current APEX partition. A block label is denoted by an event: it is present iff the corresponding block is active during the current transition. Every statement of the block (computation *stm* or control *rtn*) is conditioned by that clock. A *statement stm* takes three forms:

The definition $l := r$ of a local variable l. The use of local variables in JIMPLE code facilitates data-flow analysis. In the case of a location, it guarantees that the reference r is read and written once within a given block or section. The reference is translated to the previous value of the corresponding signal and the local variable is translated by a local (volatile) signal. In the case of a method, the reference is associated to the local l in the environment E of the translator.

The call to an external method $v =$ invoke $i\,(i^*)$, that means a method whose byte-code is not available to SOOT. All methods from available classes are inlined in the JIMPLE code of the thread, in order to globally optimize its control flow.

A native operation $v = i \star i$ on immediate values i and j is directly translatable by the corresponding equation scheduled at the context clock C.

Lock monitoring is modeled by inlined SIGNAL processes (e.g. entermonitor and exitmonitor). *Control*, via goto or if, consists in the activation of the clock that corresponds to the target block label. The handling of exceptions does not depart

from this scheme: an *exceptional* control-flow branch is produced by SOOT to handle a throw in a way similar as a goto, by associating the corresponding catch to a label. In the particular case of the return statement, translation is performed by installing the corresponding pattern of the APEX protocol at partition level (see Figure 7). Variables v and references r are translated as is ($i.[x]$ as $i.x$ (a datum) or $i.x$ (a method), parameter c stays as is, this is removed, etc.).

$$
\begin{aligned}
[\![blk; run]\!]_E &= \langle p \,|\, q \rangle \text{ where } [\![blk]\!]_E = \langle p \rangle \text{ and } [\![run]\!]_E = \langle q \rangle \\
[\![L : stm_{1\ldots n}; rtn]\!]_E^C &= \langle \mathcal{C}_E(L) ::= \text{when next\_block\$1} = \#L \text{ when tick} \,|\, p_1 \,|\, \ldots \,|\, p_n \,|\, p \rangle \\
&\quad \text{where for } 0 < i \leq n, [\![stm_i]\!]_E^C = \langle p_i \rangle \text{ and } [\![rtn]\!]_E^C = \langle p \rangle \\
[\![l := v]\!]_E^C &= \langle l ::= v\$1 \text{ when } C \rangle \\
[\![l := @r]\!]_E^C &= \langle l ::= r \text{ when } C \rangle \\
[\![v = \text{invoke } i\,(i^*)]\!]_E^C &= \langle v ::= \mathcal{R}_E(i)((i^*) \text{ when } C) \rangle \\
[\![v = i \star j]\!]_E^C &= \langle v ::= (i \star j) \text{ when } C \rangle \\
[\![\text{entermonitor } i]\!]_E^C &= \langle \text{entermonitor}\{S, \mathcal{R}_E(i)\}(\text{when } C) \rangle \\
[\![\text{exitmonitor } i]\!]_E^C &= \langle \text{exitmonitor}\{S, \mathcal{R}_E(i)\}(\text{when } C) \rangle \\
[\![\text{goto } L]\!]_E^C &= \text{if } (L \notin \text{pred}^*(\mathcal{G}_E)(L)) \text{ then } \langle \mathcal{C}_E(L) ::= \text{when } C \rangle \\
&\qquad \text{else } \langle \text{next\_block} ::= \#L \text{ when } C \rangle \\
[\![\text{if } i \text{ then } L]\!]_E^C &= \text{if } (L \notin \text{pred}^*(\mathcal{G}_E)(L)) \text{ then } \langle \mathcal{C}_E(L) ::= \text{when } i \text{ when } C \rangle \\
&\qquad \text{else } \langle \text{next\_block} ::= \#L \text{ when } i \text{ when } C \rangle \\
[\![\text{throw } i]\!]_E^C &= \langle \mathcal{C}_E(L_2) := \text{when } C \,|\, \mathcal{C}_E(L_1) ::= \mathcal{C}_E(L_2) \text{ when } C \rangle \\
&\quad \text{where } C = \mathcal{C}_E(L) \text{ and } \mathcal{X}_E(\mathcal{R}_E(i), L) = (L_1, L_2) \\
[\![\text{return}]\!]_E^C &= \langle \text{next\_block} ::= \text{main}(\mathcal{G}_E) \text{ when } C \rangle
\end{aligned}
$$

Fig. 5. Import of a RT JAVA thread from JIMPLE to SIGNAL

4.3 A Case Study

To illustrate our modeling principles and outline its application to the implementation of a formal refinement-based design methodology, we study the refinement of the functional architecture of an even-parity checker (EPC) towards its distributed implementation. This case study - even if it is small compared to what could be handled by the tool - demonstrates the capability of the POLYCHRONY workbench to automatically perform otherwise challenging engineering tasks such as thread remapping, generation of stand-alone executables, or target-specific deployment, starting from a given, high-level, RT JAVA design.

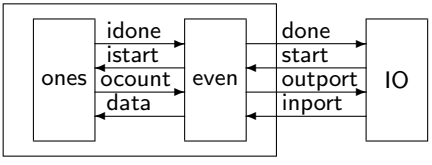


Fig. 6. Model of an even-parity checker in POLYCHRONY

The EPC consists of three functional units shown in Figure 6: an IO interface process, a master even parity check process, and a slave ones bit-counting process. The IO process will give a start signal to the even process and will then wait for the signal done to read the result. On start, even will read the input data, pass it

to the process ones and notify it with the istart signal. Ones counts the number of bits of the input data that are true and notifies even about the completion. Even finally checks if the result is an even number and notifies IO.

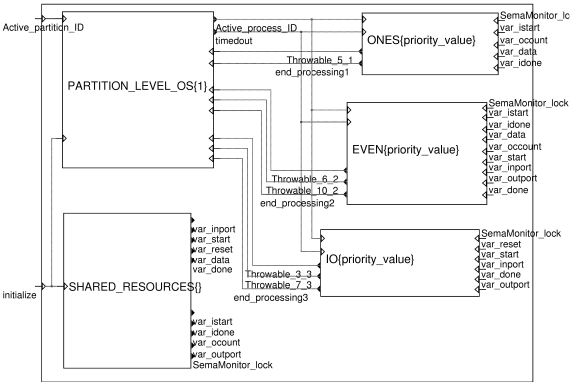


Fig. 7. Architecture of the ARINC partition for the EPC.

Example (thread ones). Having grasped the structure of the whole system and the architecture (Figure 7) of our case study, we have a closer look at the translation of one specific thread, the ones thread. The concurrency model of RT JAVA starts at a design level where implicit architecture choices are already made: the system consists of a set of threads that interact via shared variables and locks. The thread ones (Figure 8, left) determines the parity of an input data. Upon receipt of the start notification, ones shifts data until it becomes 0 and the internal count is assigned to the output ocount and done notified. The thread even notifies ones to start processing data and waits until done is notified to read the final ocount and checks whether it is an even number.

The SIGNAL model of thread ones (figure 8) consists of one critical section, delimited by a pair of monitor statements. The process is activated when it obtains the lock on istart. Then, at its own rate (now conditioned by the clock c1), it determines the result. When it is finished, it sends the notification. Native operators, e.g. the unsigned operators >> and &, or separately compiled functions can be imported into the model as external functions with a behavioral type specification (the spec declaration).

```
function rshift = ( ?i1 !i2 ) spec (| i1 ^= i2 | i1→i2 |)
                  pragmas JAVA_CODE "i2 = i1 >> 1" end pragmas;
function xand = ( ?i1, i2 !i3 ) spec (| i1 ^= i2 ^= i3 | i1→i3 | i2→i3 |)
                pragmas JAVA_CODE "i3 = i1 & i2" end pragmas;
```

Example (Architecture of the EPC Main). To illustrate how a RT JAVA architecture description (as specified in the main method of its top-level class) is analyzed and used to generate a SIGNAL instance of APEX services that models it, we consider the case of the EPC class parity (Figure 9). The processing of the

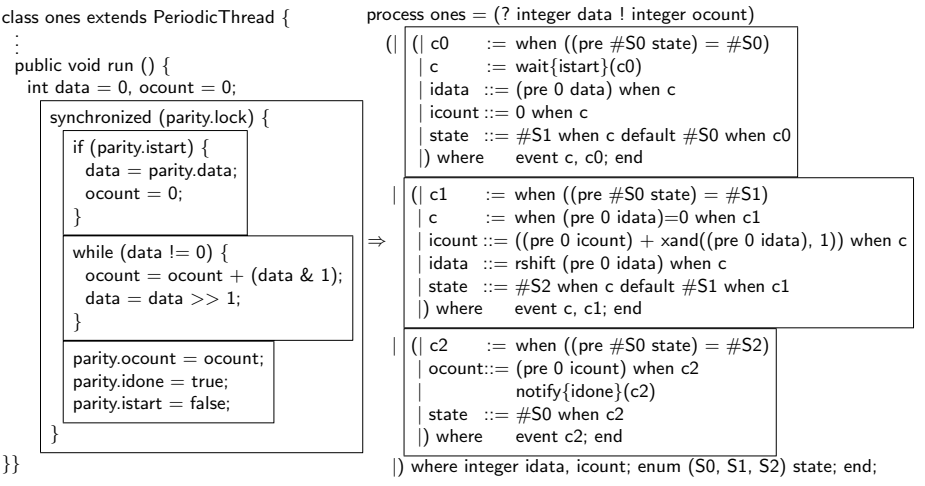


Fig. 8. Translation of the ones threads in POLYCHRONY

main method of the parity class starts with a linear analysis of its declarations and *dec* statements which produces a tree structure where each node consists of a SIGNAL data structure that renders the category of each of the items initialized in this method (shared data-structure, periodic or sporadic thread, event handler) together with its initialization parameters (size, real-time parameters, trigger and handler). Once the main method is scanned, the translation of real-time threads and event handlers starts, in order to determine the remaining architecture parameters from the init method of each class and the number of critical sections from the run method of each class.

These data are used to instantiate the generic APEX service models of POLYCHRONY (figure 9) and finalize the model of the application architecture. This

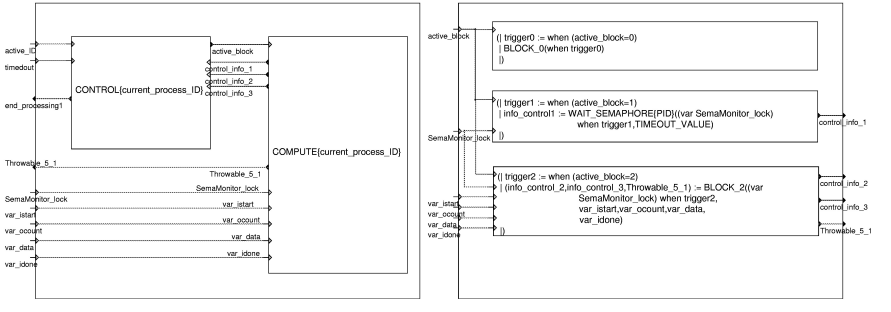


Fig. 9. Control and computation nodes of the ARINC model for thread ones.

yields the structure depicted in Figure 9 for the ones thread: a control process is connected to the partition-level scheduler and a computation process.

5 Checking the Correctness of System Design Refinements

In this section, we demonstrate the use of our modeling tool to check the refinement of the even-parity checker (Epc) from its initial specification towards its distributed implementation. This is done by the insertion of communication protocols and by the merge of secondary threads on a given architecture. Our goal is to demonstrate how our polychronous design tools and methodologies allow us to consider high-level system component descriptions and to refine them in a semantic-preserving manner into a Gals implementations (globally asynchronous locally synchronous, aka. Kahn network).

Architecture Design Refinement. To model the physical distribution of the threads ones and even and in order to allow them to communicate asynchronously via a channel structure, we install a double handshake protocol between them. The installation of this channel incurs a desynchronization between the two processes, hence a potential change of behavior. The model of the send and recv methods in Signal is obtained from a message sequence specification (Figure 10, left). The ready and ack flags stand for state variables declared in the lexical scope of send and recv in the module that defines the protocol; eReady and eAck stand for events. Sender and receiver use a simplified wait/notify mechanism similar to that of asynchronous event handlers. In [13], we show that the validation of a protocol insertion such as the Epc model upgrade with a double-handshake protocol amounts to checking that the initial and upgraded designs have equivalent flows, which is amenable to model checking by proving that both design outputs always return the same sequence of values.

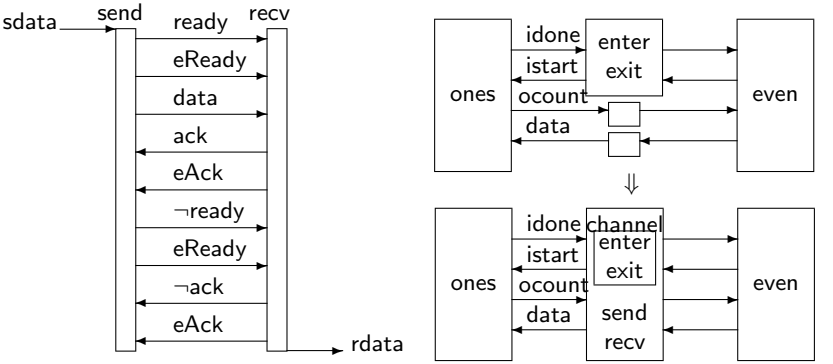


Fig. 10. Refinement of the Epc architecture with the double-handshake protocol

Remapping Threads onto a Target. The encoding of the even-parity checker demonstrates the capability of Signal to provide multi-clocked models of Java

components for verification and optimization purposes. Polychrony allows for a better decoupling of the specification of the system under design from early architecture mapping choices. For instance, the SIGNAL compiler can be used to merge the behaviors IO and even and combine their control-flow using the clock hierarchization algorithm.

Then, checking the merge of the threads IO and even correct w. r. t. the initial specification amounts to checking that it is deterministic. As demonstrated in [9], this is done by checking that the clock of its output can be determined from that of its input data and from the master simulation tick and that checking that the frequency of the output is lower than that of both inputs data and tick.

Table 2. Execution times of multi-threaded applications [ms]

Case study	JCC	TURBOJ	POLYCHRONY
I	145	139	50
II	1428	1375	450
III	14312	13677	4540

In [5], experiments made in the context of the RNTL-EXPRESSO project are reported, showing that for several case-studies (multi-threaded RT JAVA programs consisting of five to ten threads), the speedup obtained by automatically remapping all threads into a stand-alone (JVM-less) and serialized executable was in average 300% compared to a commercial JAVA compiler implementing the RT JAVA specification (TURBOJ) and to the standard JAVA compiler (JCC).

Note that the construction of a single automaton from multiple JAVA threads may not be always possible: the corresponding SIGNAL model may not necessarily exhibit a single control-flow tree after hierarchization (figure 4). However, it may still be possible to repartition the model into a smaller set of threads (at most equal to the initial one) having a hierarchizable control-flow tree.

Proving Global Design Invariants. In addition to methodology-specific verification issues addressed in the present article, the model checker of SIGNAL allows to prove more general properties of specification requirements: reachability, attractivity, and invariance (see [10] for details). Example applications are, for instance, to check that the refinement of a model with a finite FIFO buffer satisfies requirements such as: "*one never reads an empty* FIFO *queue*" or "*one never writes to a full* FIFO *queue*".

Such requirements have previously been studied in [7], in the context of the modeling of APEX avionics application in SIGNAL and the companion design methodologies. Another common requirement is non-interference: e.g. a lock is never requested from two concurrently active threads. In SIGNAL, this problem reduces to a satisfaction problem. Suppose two requests lock $::= b_1$ when c_1 and lock $::= b_2$ when c_2 to a lock. Checking non-interference amounts to proving that $c_1 \wedge c_2 = 0$ w. r. t. clock constraints. These constraints are inferred by the SIGNAL compiler for a specific hard real-time implementation.

6 Related Work and Conclusions

The present work is based on previous results in the POLYCHRONY project. It uses the polychronous model of computation [9], the model of the APEX real-time operating system standard [6], and a formal refinement-based design methodology [13]. PTOLEMY [11] and ROSETTA [8] are approaches that partly aim at a similar goal. They examine system implementations using different computation models. The synchronous model, which we consider, is only one of these, but it allows the treatment of more general, multi-clocked systems. We presented a new engineering technique allowing for the integrated modeling, optimization, verification, and simulation of embedded systems in a functional subset of the RT JAVA specification, which is compliant with certifiable software engineering requirements in avionics. This platform-based design using the multi-clocked synchronous framework POLYCHRONY allows to perform precise and aggressive optimizations and transformations, that would be hard, yet impossible to achieve using common techniques.

References

1. AIRLINES ELECTRONIC ENGINEERING COMMITTEE. "Design Guidance for Integrated Modular Avionics". ARINC Report 651-1, November 1997.
2. AIRLINES ELECTRONIC ENGINEERING COMMITTEE. "Avionics Application Software Standard Interface". ARINC Specification 653, January 1997.
3. AMAGBEGNON, T. P., BESNARD, L., LE GUERNIC, P. "Implementation of the data-flow synchronous language SIGNAL". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.
4. BOLLELA, G. ET AL "The real-time specification for JAVA". Addison-Wesley, 2000.
5. D. COSTARD. Evaluation d'une chaîne de compilation JAVA temps-réel. *Rapport de stage de fin d'étude*, ESIEE, Juin 2003.
6. GAMATIÉ, A., GAUTIER, T. Modeling of modular avionics architectures using the synchronous language. In *proceedings of the 14th. Euromicro Conference on Real-Time Systems, work-in-progress session*. IEEE Press, 2002. Available as INRIA research report n. 4678, December 2002.
7. GAMATIÉ, A., GAUTIER, T. The SIGNAL approach to the design of system architectures. In *10th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE Press, April 2003.
8. Perry, A., Kong, C.. Rosetta: Semantic support for model-centered systems-level design. In IEEE Computer, 34(11):6470, November 2001.
9. LE GUERNIC, P., TALPIN, J.-P., LE LANN, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design, R. Gupta, S. Gupta, S. K. Shukla Eds.* World Scientific, 2002. Available as INRIA research report n. 4715, December 2002.
10. MARCHAND, H., RUTTEN, E., LE BORGNE, M., SAMAAN, M. Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.
11. J.T. BUCK, S. HA, E.A. LEE AND D.G. MESSERSCHMITT. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, v. 4, pp. 155–182. Ablex, April 1994.

12. Kwon, J., Wellings, A.J., King, S. A High Integrity Profile for Real-time Java. *Proceedings of the Joint ACM Java Grande Conference*. Acm press, 2002.
13. Talpin, J.-P., Le Guernic, P., Shukla, S. K., Gupta, R., Doucet, F.. Polychrony for formal refinement-checking in a system-level design methodology. In *Application of Concurrency to System Design*. IEEE Press, June 2003.

Experiment on Embedding Interception Service into Java RMI

Jessica Chen and Kun Wang

School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
{xjchen,wang133}@cs.uwindsor.ca

Abstract. Middleware interceptors are a well-recognized powerful technique that enables the extensions of the middleware functionality in a way transparent to both the existing middleware implementations and the distributed applications built on top of them. In recent CORBA specifications, OMG has provided the standard specification of portable interceptors. This specification determined the interception service that any CORBA implementation should provide. As a well-known middleware however, Java RMI did not offer the interception mechanism. Based on our experiment, we address in this paper, the architectural and design issues on embedding interception service into the existing implementation of Java RMI from SUN Microsystems. Our interception service follows the style and the functionality of CORBA interception service to support *request portable interceptors*.

Keywords: Interceptors, Java RMI, CORBA, Dynamic Proxy, Reflection.

1 Introduction

Middleware interceptors and their associated interception service are a well-recognized and powerful technique designed to allow people to enhance the basic middleware functionality without any modification to the middleware implementation itself. This is realized by requesting the middleware implementation to provide an interception service, i.e. to automatically invoke the execution of the dynamically registered interceptors during its remote object invocations. The implementation of the interceptors encodes the additional work to do and thus, when exploited properly, can enhance the basic functionality of the existing middleware. Examples of such enhancements include test control, software monitoring, debugging, authentication and encryption, fault-tolerance, load balancing, etc. Consider a group of servers that provide an identical service. We would like to keep the load on each server well balanced. Load-balancing service is helpful when we have large volume of computations from clients or when a large number of clients are requesting service from a same server causing bottleneck. Such a service can be achieved by implementing suitable interceptors who are able to check the load on each server and whenever necessary, redirect the client's requests to other servers with lower load.

N. Guelfi, E. Astesiano, G. Reggio(Eds.): FIDJI 2003, LNCS 2952, pp. 48–61, 2004.

There are many design issues in providing the interception service, such as when the interceptors should be called, what kind of internal information should be made available to the interceptors, what kind of behavior is allowed or disallowed in the interceptor implementations, etc. These design issues directly affect the restrictions on and the potential powerfulness of the interceptors. In the recent CORBA specifications [11], OMG has provided the standard definition of *portable interceptors*. It includes the interfaces to be provided by the interception service, the predefined interception points, the characteristics of the intrusion to the ORB execution, the way to raise exceptions, the information from ORB accessible to the interceptors, the registration of interceptors, etc.

Like CORBA ORB, Java Remote Method Invocation (RMI) [8] is also a distributed remote object model that facilitates the communications among distributed processes by means of remote method invocations. It allows programmers to develop distributed Java applications with the same syntax and semantics as those that are used for non-distributed ones. With the widespread use of Java and the increasing needs of distributed systems, RMI has gained its own popularity in the past few years. The interception service is however, not offered in the RMI implementation. This limits the flexibility of RMI in term of extending its basic functionality.

In this paper, we address the architectural and design issues on embedding interception service into the existing implementation of Java RMI from SUN Microsystems. Our interception service follows the style and the functionality of CORBA interception service. There are typically two kinds of portable interceptors defined in CORBA: portable IOR (Interoperable Object Reference) interceptors and portable request interceptors. Here we only consider portable request interceptors.

Based on different modifications to different RMI architectural layers, we show three possible strategies on modifying existing RMI implementation in order to embed the interception service. More precisely, we consider

- Use dynamic proxy technique to wrap the remote objects when they are dynamically *looked up*;
- Automatically generate the modified class definitions of stubs and skeletons of the remote objects;
- Catch each remote object invocation at the RMI remote reference layer.

All of these approaches are generic in the sense that with the modified RMI implementation, (i) the interceptors can be implemented independently and transparent to the RMI implementation; (ii) the interceptors are transparent to both the client and the server program: they do not need to be modified or recompiled. For each approach, we discuss its potentials, its limitations and its comparison to the corresponding CORBA interception service. The discussion is based on our experiment part of which has been adopted into our environment for reproducible testing of distributed Java applications [2,5,14].

The paper is organized as follows. We first give a short review of CORBA portable interceptors related to our discussion (Section 2). Then, for each RMI

modification strategy, we explain its implementation details, limitations, and compare it with CORBA portable interceptors (Section 3). Related work will be discussed in Section 4, followed by conclusion and final remark (Section 5).

2 CORBA Portable Interceptors

CORBA portable interceptors are hooks into the ORB through the interception service in order to intercept the normal flow of the ORB executions. The implementation of the portable interceptors is independent of and transparent to the CORBA implementation, and the use of portable interceptors is transparent to both the client and server implementation.

There are typically two kinds of portable interceptors in CORBA: portable IOR (Interoperable Object Reference) interceptors and portable request interceptors. The former allows us to modify a CORBA object reference represented by IOR in a way transparent to the application. The latter is designed to intercept the flow of a call request or a call reply sequence through the ORB at some specific points. Here we are interested in portable request interceptors, simply called request interceptors below. A request interceptor can, for example, make object invocations itself before allowing the current request to be executed, or alter the request outcome by raising a system exception. By exploiting request interceptors, we can enhance in many ways the basic functionality provided by the middleware. We can realize, for example, test control, software monitoring, debugging, fault-tolerance, load balancing, etc., without any modification of the existing CORBA implementation. There are two kinds of request interceptors: the client request interceptors (also called client interceptors below) and the server request interceptors (also called server interceptors below).

As we mentioned, there are many design issues in providing the interception service, which directly affect the restrictions on and the potential powerfulness of the interceptors. In CORBA specifications on request interceptors, much of the details in this regard are provided. For example,

- it defined all the interception points;
- it defined all the internal information accessible and/or modifiable by the interceptors;
- it defined what interceptions are allowed and disallowed, as well as how to raise system exceptions;
- it also defined the way to register the interceptors.

The interception points for request interceptors include *send_request*, *send_poll*, *receive_reply*, *receive_exception*, *receive_other*, *receive_request_service_context*, *receive_request*, *send_reply*, *send_exception*, *send_other*. The first five belong to the client-side interception points, and the last five belong to the server-side ones. Among them, we are mainly interested in *send_request*, *receive_reply*, *receive_request* and *send_reply*.

- *send_request* is the interception point that allows an interceptor to query about the request information, to modify the service context, or to raise a system exception, before the request is sent to the server.
- *receive_reply* is the interception point that allows an interceptor to query about the information on a reply or to raise a system exception after the reply is returned from the server and before control is returned to the client.
- *receive_request* is invoked just before the request is passed to the object implementation, and allows an interceptor to query about the request information or raise an exception.
- *send_reply* is invoked right after the object has generated the reply on the server side. It allows an interceptor to query about the information on the call reply or to raise an exception.

An interceptor that implements at least one of the above methods can be registered into ORB via an associated ORBInitializer object that implements the ORBInitializer interface. We can register more than one interceptor. When an ORB is initialized, it will call each registered ORBInitializer object to register its interceptor. If several interceptors are registered, the ORB may invoke them in an arbitrary order.

The internal information accessible and/or modifiable by the interceptors is defined in interface *RequestInfo*. As such information differs from client side to the server side, we have interfaces *ClientRequestInfo* and *ServerRequestInfo* inherited from it. For each remote method invocation, a *ClientRequestInfo* object is passed to the client-side interception points and a *ServerRequestInfo* object is passed to the server-side interception point. *request_id*, *arguments* and *result* are all essential attributes of *RequestInfo*. *request_id* uniquely identifies an active request/reply sequence. *arguments* contains the arguments on the operation being invoked (not available for Java-CORBA implementation), and *result* contains the result of the invocation. Apart from the *arguments*, *result*, etc. that are directly obtained from the request and reply respectively, there is some other information related to a particular invocation that is kept in a *ServiceContext* object of *RequestInfo*. The access to a *ServiceContext* object is realized via methods *get_request_service_context* on the client side and *get_reply_service_context* on the server side. To modify the service context, we have method *add_request_service_context* and *add_reply_service_context*.

A rich and powerful functionality has been defined in the specification of CORBA request interceptors. For example, a request interceptor can delay a call reply, can generate its own request, can block a request/reply by raising an exception, and can redirect a request via *ForwardRequest* exception. Note that if an implementation of a CORBA request interceptor redirects the request, this interceptor will also be invoked for the redirected request when it is reissued.

Due to the portability concerns, there are also some restrictions on CORBA request interceptors. For example, a request interceptor cannot generate its own reply to intercepted request, cannot alter the arguments and return value of the request/reply, and cannot modify the service context except for adding its own content.

On one hand, the specification of CORBA portable interceptors defines the way how people can use the interception service, and on the other hand, it also determines the characteristics of the interception service that any CORBA implementation should provide. In the following, we discuss some possible solutions in providing Java interception service according to such characteristics.

3 Interception into RMI

The RMI implementation is originally built from three abstract layers: the Stub/Skeleton Layer, the Remote Reference Layer, and the Transport Layer. The Stub/Skeleton Layer is the top layer in the RMI architecture. It intercepts the remote method calls and redirects them to the corresponding remote object implementation. On the Remote Reference Layer, we can interpret and manage the references to the remote objects. The Transport Layer is based on TCP/IP connections among different machines in the network. It provides the basic connectivity and some firewall penetration strategies.

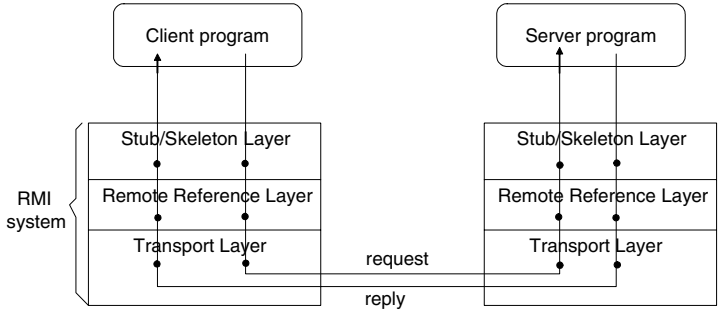


Fig. 1. Layered architecture in RMI with interception service

Theoretically, the interception service can be embedded into any of these three layers, as shown in Figure 1 (the black dots denote the possible interception points):

- We can embed the interception service into the Stub/Skeleton Layer by altering the way that stubs and skeletons are generated. We will call it below *stub/skeleton interception.*
- We can embed the interception service into the Remote Reference Layer by modifying the Java class libraries for the runtime environment. We will call it below *remote call interception.*
- We can implement the interception service in the Transport Layer by modifying the existing communication protocols defined within RMI.

Due to the fact that the related documents on Remote Reference Layer and Transport Layer are not publicly available, we have worked directly on the source code in the Java runtime libraries, and we found it difficult to go with the third choice. Below we will only consider the first two approaches.

Another possibility is to exploit dynamic proxy technique to embed the proxies of the remote objects through the Java Naming Services. We will also include the discussion with this approach, called *registration interception*, based on our experience.

Common to all approaches is the extension of the Java Runtime Environment API. The extension consists of a set of interfaces and classes to be packaged into the SDK class library. The essential part of this set includes the definitions of the interfaces *ClientRequestInfo* and *ServerRequestInfo*, classes *ClientInterceptor* and *ServerInterceptor* (similar to those in CORBA Portable Interceptors). Figure 2 shows the UML class diagram for this set of interfaces and classes.

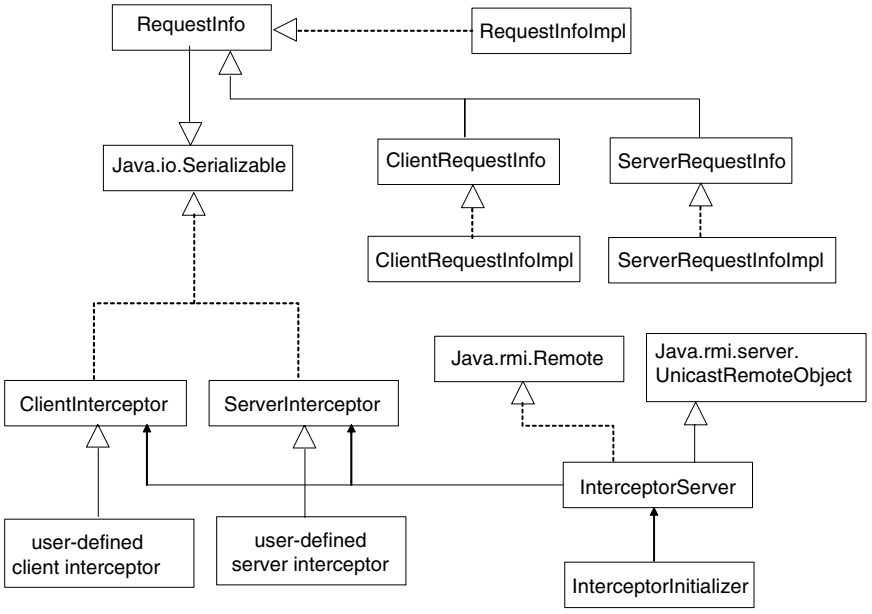


Fig. 2. UML class diagram of the extended interfaces/classes to the Java core API

As we mentioned, the CORBA *RequestInfo* provides the internal information about the remote invocation that is accessible to the interceptors. There is a lot of information introduced in the *RequestInfo* in CORBA Portable Interceptors. Now what kind of internal information can be accessed by the RMI interceptors in our setting?

According to our experiment, the available information we have at the interception points is the remote object and the remote method being invoked. So the attributes of our *RequestInfo* accessible to the interceptors may consist of any information retrievable from the remote object and the invoked method via Java Reflection.

Java Reflection is a built-in API that reflects the classes, interfaces, and objects in the current Java Virtual Machine. It plays an important role in providing runtime information of Java objects, their running environment and their interactions, and is thus, a powerful tool for the development of debuggers, class browsers, GUI builders, etc. With Java Reflection, we can, for instance, dynamically determine the class of an object, and further retrieve information of the methods, interfaces, etc. defined in that class. We can create an instance of a class whose name is statically unknown, and we can also invoke a method on an object while the method name cannot be known until runtime.

In our setting, we perform reflection on the remote object when a method is invoked on it. We can retrieve, for example, the signature of the called method. We can also obtain the name of the interface of the called method. This can be realized by comparing the signature of the method being called with those of the methods reflected from the remote object. Of course, this is under the assumption that in the class definition of the remote object, there are no two methods with identical signatures. Among information retrievable via Java Reflection, we have chosen the followings:

- the name of the interface of the invoked method;
- the name of the invoked method;
- the list of parameters contained in the invoked method;
- the return value of the invoked method.

The above information is encapsulated into a *RequestInfo* object, which is then passed as a parameter to methods *send_request*, *receive_request*, *send_reply* and *receive_reply*.

ClientInterceptor and *ServerInterceptor* are the classes to be used by the client-side interceptors and server-side interceptors respectively. The methods in these classes define the interception points. As we mentioned previously, we are interested only in the four interception points i.e. *send_request*, *receive_request*, *send_reply* and *receive_reply*. Thus, class *ClientInterceptor* defines methods *send_request(aClientRequestInfo)* and *receive_reply(aClientRequestInfo)*, while class *ServerInterceptor* defines methods *send_reply(aServerRequestInfo)* and *receive_request(aServerRequestInfo)*.

An *InterceptorServer* provides the registration and the lookup service of the interceptors. The user-defined interceptors, which are instances of *ClientInterceptor* or *ServerInterceptor*, must be registered into the *InterceptorServer* in either of the following two ways:

- by registering associated RMI interceptor initializer, which implements the *InterceptorInitializer* interface;

– by dynamically obtaining the reference of the *InterceptorServer* followed by invoking its method *addClientInterceptors* or *addServerInterceptors*.

In the following, we discuss the three possible approaches that we mentioned before on embedding interception service: the registration interception, the stub/skeleton interception and the remote call interception.

3.1 Registration Interception

When the server program performs the object registration to export the remote object, it actually registers the serialized stub object of the remote object. The basic idea of this approach is to modify the library of the RMI Naming service in order to wrap the remote object stub so that the client will receive a *proxy stub* rather than the real remote object stub. We provide our own implementation of the RMI registry based on the original one so that

– for client transparency, the proxy stub looks to the client just the same as the stub itself.
– the proxy stub can intercept all the operations on this stub object. Whenever a method invocation occurs on a remote object, it will arrive at the proxy stub's hand which will in turn provide an interception service to call all currently registered interceptors. These interceptors, as implemented independently from the middleware itself, make their own decision on what to do with the interception.

As the proxy stub should appear to the client just like the original remote object, it is required to implement all the interfaces that the original remote object does. Since the remote object and its stub are unknown a priori, the proxy stub object should be able to implement arbitrary interfaces given at runtime. This is realized via the dynamic proxy technique introduced in the Java 2 platform and J2SE 1.3. A Java dynamic proxy is a class created on-the-fly that implements a list of interfaces that are statically unknown.

The current implementation of RMI does not allow us to export a dynamic proxy object to the registry. This is because the Java language specification does not have stub class definition for a class that implements *java.lang.reflect.InvocationHandler*, which means that a proxy instance this invocation handler associated with cannot be a remote object, and thus cannot be transmitted to a remote process or host. As an unfortunate consequence, we cannot inject the interception service on the server side by wrapping the remote object stub from the server side: The only place we can inject the interception service is when the client looks up for the remote object from the registry. This remains as one of the major drawbacks of the present approach.

Figure 3 illustrates how the interception service is embedded into the RMI Naming Service using the dynamic proxy technique. When the client program performs *lookup* on the modified RMI registry, the stub is always wrapped into an interception service object before being returned to the caller. As

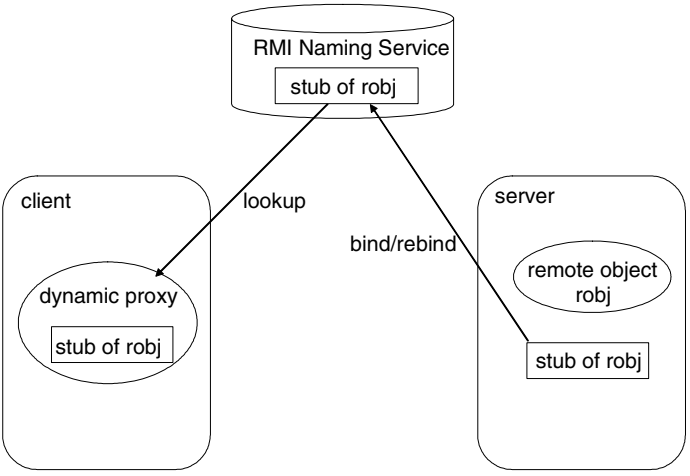


Fig. 3. Using dynamic proxy to embed the interception service

the wrapper is put only when the remote stub is *downloaded* from the registry to the client program, the interception points are *send_request* and *receive_reply* on the client side. Further extension of our current experiment to include *send_poll*, *receive_exception* on the client side is possible, but it is impossible to make interception points *receive_request_service_context*, *receive_request*, *send_reply*, *send_exception*, and *send_other* as in CORBA.

Another drawback of this approach is that we can only catch the remote invocations on registered remote objects. As we know, remote references can be obtained in two ways in Java RMI: (i) via object registration; (ii) being passed as parameters or return values from remote calls. Obviously, this dynamic proxy-based technique only applies to those remote objects whose remote references are obtained via object registration.

3.2 Stub/Skeleton Interception

The basic idea of this approach is to modify the stub and skeleton of each remote object upon the generation of stub and skeleton classes. We need to modify the RMI compiler in order to automatically generate the modified stubs and skeletons.

Again, in our experiment, we have introduced four interception points. It is possible to further extend it to include *send_poll*, *receive_exception* on the client side, and *send_exception* on server side. Note that in Java 2 SDK, an additional stub protocol has been introduced which actually eliminated the need of skeletons in Java 2 platform-only (and JDK 1.1 compatible) environment.

So with this technique, the server-side interception is restricted only to earlier versions of RMI.

In order to add the interception service into the Stub/Skeleton Layer, we need to modify the existing Java tool library apart from the modification of the Java core API that we mentioned before. In the Java tool library, we provide our modified *RMIGenerator*, which is an RMI compiler that generates the modified Java source code of the stub/skeleton class definitions.

3.3 Remote Call Interception

To embed the interception service into the Remote Reference Layer, we modify in Java core API, the class definitions of *UnicastRef* and *UnicastServerRef*. A *UnicastRef* represents the handler for a remote object and will be passed to the client program together with a stub. A stub uses the *UnicastRef* object to carry out a remote method invocation to a remote object. A *UnicastServerRef* represents a server-side handler for a remote object and implements the behavior of a remote object on the Remote Reference Layer.

With modified *UnicastRef*, a *UnicastRef* instance, once passed from the server to the client, will call the interception points *send_request* and *receive_reply* right before and right after the invocation of *invoke*, which makes the remote method invocation to the corresponding *UnicastServerRef* object on the server side. Before the interception point *send_request*, an instance of *ClientRequestInfo* is created and encapsulated with the information retrieved by using Java Reflection on the remote object and the remote method that we obtain at the Remote Reference Layer. This *ClientRequestInfo* is then passed to the interceptors with both *send_request* and *receive_reply*.

An instance of the *UnicastServerRef* is created whenever a remote object is exported, either implicitly by extending *UnicastRemoteObject* or explicitly through the *exportObject* method defined in *UnicastRemoteObject*. A remote object will not be ready to receive requests until it is exported. In class *UnicastServerRef*, method *dispatch* is the place where remote method invocation is actually forwarded to the remote object implementation on the server side. The interception points *receive_request* and *send_reply* are inserted right before and right after this method invocation, so that the corresponding methods of all registered *ServerInterceptor* objects will be called. Similar to the client side, a *ServerRequestInfo* object is created and passed to the implementation of the interceptors.

3.4 Interceptor's Capability

We have summarized before what a CORBA request interceptor can do, and what its limitations are. Now comparing to CORBA request interceptors, we list the capability and limitations of the RMI interceptors with our registration interception, stub/skeleton interception and remote call interception approaches respectively. See Table 1.

Table 1. Comparison of RMI interceptors and CORBA portable interceptors

	RMI interceptors (registration interception service)	RMI interceptors (stub/skeleton interception service)	RMI interceptors (remote call interception service)	CORBA portable interceptors
redirect a call	Yes	No	Yes	Yes
alter arguments	Yes	Yes	Yes	No
generate its own reply	Yes	Yes	Yes	No
make object invocations	Yes	Yes	Yes	Yes
delay a request/reply	Yes	Yes	Yes	Yes
piggyback information	No	No	No	Yes

A CORBA request interceptor can redirect a request to another target by throwing a *ForwardRequest* exception. We can also realize this for the RMI interceptors with the registration interception approach and the remote reference interception approach. With the stub/skeleton interception approach, however, this turns out to be difficult because the modifications in regard of embedding the interceptions happens inside the stubs and skeletons, and to skip the normal flow of execution there requires too much effort.

For portability concerns, a CORBA request interceptor is not allowed to alter any argument in the method invocation nor allowed to generate its own reply to the call request. This is not an issue for RMI interceptors: we can actually allow an interceptor to alter any of the arguments in the method invocation, or to generate its own reply to be returned to the call. Of course, such a feature is Java-specific, and technically, it can be achieved by Java Reflection. However, this functionality is restricted to alter the arguments or generate reply only: the interceptors are not allowed to change the type of an argument or the type of the result, nor allowed to put additional information to the call request itself. In the registration interception approach, the modification of the arguments and the generation of different reply can only happen on the client side as the server-side interception is not possible.

A CORBA request interceptor can make object invocation itself before allowing the current request to be executed. Consequently, an interceptor can also be used to delay a request or a reply for a certain amount of time. Without difficulty, we can also make the RMI interception service to provide such functionality. Again, with the registration interception approach, to make an object invocation or to delay a request/reply can only be allowed on the client side.

CORBA interception service allows interceptors to manipulate request service context such as to piggyback additional information onto a message. We cannot accommodate this feature in any of the three approaches we consider here for RMI. The difficulty mainly comes from the fact that we do not want

to change the actual request and reply of the method invocation at the RMI Transport Layer.

4 Related Work

An implementation of the interception service on the RMI Remote Reference Layer as described in this paper has been adopted and integrated into our environment of reproducible testing for distributed Java applications [2,5,14]. Reproducible testing (see e.g. [2,5,13]) is a specification-based testing technique that aims at *reproducing* the observations of the tests on nondeterministic programs. This is realized by controlling the execution of the *application under test* upon the order of some important events that may have impact on the internal nondeterministic choice of the *application under test*. Synchronization events [3,4], remote object invocation events and input events are all among such important events to be controlled. With the present technique on remote call interception, we have developed the *testing interceptors* that realize the control over the orders of the remote method invocations in the application under test [14]. Although we have adopted only the remote call interception mechanism into our reproducible testing environment, we have presented all the three possible ways that we have experienced for embedding the interception service into RMI, because we believe they will provide some valuable guidelines for people who may want to implement certain kinds of RMI interception service for various purposes.

In the past, people have discussed the instrumentation technique and related monitoring technique for Jini applications [7,12] based on the existing Jini implementation. The technique they used to realize a transparent monitoring system is similar to the *registration interception* technique that we discussed here: the Jini registration service is modified so that a dynamic proxy replaces the real servant.

Apart from the work on Jini instrumentation, much of the work on middleware instrumentation is CORBA-based. We have the discussion on the instrumentation technique on existing CORBA ORB implementation [15] for the management issues. The basic mechanisms implementable by CORBA request interceptors are listed in [1] where people have also discussed some limitations of CORBA request interceptors, as well as a proxy-based technique to overcome these limitations. In [6], it is shown how to use CORBA portable interceptors to enhance the client-side ORB functionality for various purposes such as caching, load-balance, flow control, and fault-tolerance. To the best knowledge of the authors, the RMI instrumentation techniques have not been discussed.

The concept of interception points corresponds to a special case of *join points* in the Aspect-Oriented Programming (AOP) [10], when we consider the points of the beginning and the end of remote method calls. In fact, as an aspect-oriented Java extension, AspectJ (see e.g. [9]) also allows users to write additional code to be executed at the *join points* of a Java program. However, the ultimate goals of AOP and CORBA interceptors are quite different. The former aims at improving the programming language itself on the application level, and the additional

code are application-specific. In the latter case, an interceptor implementation is injected into the middleware and can become part of the middleware providing additional *service*. In doing so, the interception service has to be powerful enough: not just to allow execution of additional code, but also to allow redirecting a remote call, piggybacking additional information, etc.

5 Conclusion and Final Remark

To embed an interception service into an existing middleware implementation is obviously a lot more difficult than taking an interception service into account while developing the middleware implementation itself. Thus, it is hard, if not impossible, and it is also beyond our scope, to put into existing RMI implementation an interception service that *fully* realize the CORBA specification on portable interceptors. Instead, we have shown three possible ways to embed into existing RMI implementation an interception service that realizes its *basic* features. These basic features of the interception service enable us to implement some useful interceptors, for example, to trace the process interactions and further control their orders.

We did not discuss all the potentials that an RMI interception service can provide. For example, we did not consider the exception handling for the moment. For future work, we are interested in extending the current experiment and discussion on other features that are addressed in CORBA specification and that an RMI interception service can possibly provide.

Acknowledgements. This work is supported by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References

1. R. Baldoni, C. Marchetti, and L. Verde. CORBA request portable interceptors: Analysis and applications. *Concurrency and Computation: Practice and Experience*, 15:551–579, 2003.
2. X. Cai and J. Chen. Control of nondeterminism in testing distributed multithreaded programs. In *Proc. of the 1st Asia-Pacific Conference on Quality Software (APAQS 2000)*, pages 29–38, 2000.
3. R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 66–74, Mar. 1991.
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.
5. J. Chen. On using static analysis in distributed system testing. In *Proc. of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000). Lecture Notes in Computer Science Vol. 1999*, pages 145–162. Springer-Verlag, 2000.
6. R. Friedman and E. Hadad. Client-side enhancements using portable interceptors. *Journal of Computer System Science and Engineering*, 17(2):3–9, 2002.

7. P. Hasselmeyer and M. Vob. Monitoring component interactions in Jini federations. In *SPIE Proceedings 4521*, pages 34–41, 2001.
8. S. M. Inc. Java remote method invocation specification v1.4, 2002. ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of European Conference on Object-Oriented Programming (ECOOP 2001). Lecture Notes in Computer Science, Vol. 2072*, pages 327–355, 2001.
10. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP 97). Lecture Notes in Computer Science, Vol. 1241*, pages 220–242. Springer-Verlag, 1997.
11. O. M. G. (OMG). Common Object Request Broker Architecture: Core specification v3.0, 2002. http://cgi.omg.org/docs/formal/02-11-01.pdf.
12. D. Reilly and A. Taleb-Bendiab. Dynamic instrumentation for Jini applications. In *Proc. of the 3rd International Workshop on Software Engineering and Middleware (SEM 2002). Lecture Notes in Computer Science Vol. 2596*, pages 157–173. Springer-Verlag, 2003.
13. 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.
14. K. Wang. Constructing a reproducible testing environment for distributed java applications, 2003. Thesis in preparation. School of Computer Science, University of Windsor.
15. M. Wegdam, D. Plas, A. van Halteren, and B. Nieuwenhuis. ORB instrumentation for management of CORBA. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, 2000.

BANip: Enabling Remote Healthcare Monitoring with Body Area Networks[1]

Nikolay Dokovsky, Aart van Halteren, and Ing Widya

University of Twente
PO Box 217
7500 AE Enschede, The Netherlands
{dokovsky,halteren,widya}@cs.utwente.nl

Abstract. This paper presents a Java service platform for mobile healthcare that enables remote health monitoring using 2.5/3G public wireless networks. The platform complies with today's healthcare delivery models, in particular it incorporates some functionality of a healthcare call center, a healthportal for patients that aggregates healthcare services from collaborative care centers. Our service platform simplifies the development of mobile healthcare applications, because application developers are shielded from the complexity of communicating with mobile devices over 2.5/3G networks. In this paper we focus on the BAN interconnect protocol (BANip) which is our solution to extend the local services offered by a Body Area Network (BAN) to remote healthcare center.

1 Introduction

Information and Communication Technology (ICT) plays an important role in cost reduction and efficiency improvement in healthcare delivery processes. ICT in emergency services for example enables trauma teams at hospitals to diagnose remotely injured patients at accident scenes and therefore improves the time to treatment [1]. The introduction of 2.5/3G wireless technology even takes healthcare further to mobile healthcare (m-health) which enables ambulatory patients to lead a normal daily life. For example, the pregnant women m-health trial in [2] investigates mobile and remote monitoring of the maternal and fetal vital signs whilst the pregnant women can proceed with their daily life activities rather than be hospitalised for monitoring. ICT also plays an important role in the introduction of new healthcare stakeholders in healthcare delivery such as healthcare call centres, healthportals which provide health services around the clock to patients located everywhere [3]. These healthcare call centres typically collaborate closely with secondary care centres, e.g. hospitals, laboratories and clinics. In the delivery of healthcare, they may be viewed as intermediaries between patients (e.g. at home) and healthcare practitioners (e.g. at hospitals).

[1] This work has been sponsored by the European Commission within the IST project MobiHealth (IST-2001-36006) [2].

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 62–72, 2004.

This paper presents a Java service platform for m-health. The platform can accommodate call centres, which may reside in hospitals as a front-office or are located remotely from a secondary care centre. In the latter case, call centres act as an added value stakeholder in healthcare delivery that aggregates healthcare services from collaborative care centres. In this context, the platform provides end-to-end m-health services to the end-users (i.e. patients and healthcare practitioners) using health intermediaries which are transparent to these users. In particular, the implemented platform provides remote monitoring services of measured vital signs of patients (e.g. ECGs, oxygen saturation in patients blood) in a store-and-forward as well as in a (near) real-time mode [4].

Besides the healthcare delivery stakeholders, the platform spans over the domains of several connectivity stakeholders, in particular ISPs and 2.5/3G wireless network operators. The development of end-to-end healthcare services spanning over multi stakeholders' domains is also a challenge addressed in this paper.

Moreover, the embedded physiological monitoring system in the platform is based on wearable Body Area Networks (BANs) [5], [6], which nodes are vital sign sensors and a gateway to communicate with nodes in the administrative domains of the healthcare stakeholders using 2.5/3G wireless infrastructures.

In this paper, we present the platform design using Java, Jini and Jini Surrogate technologies. We address how we apply these technologies to provide end-to-end m-health services in a wireless environment with intermediaries, while these intermediaries are in a way transparent for the end-users. We also present how these technologies have been used to discover, authenticate and register BANs which can be switched on anytime and anywhere in a 2.5/3G wireless network coverage area.

2 MobiHealth Service Platform

The MobiHealth service platform enables remote monitoring of patients using 2.5/3G public wireless infrastructures. Patient data is collected using a Body Area Network (BAN). A healthcare practitioner can view and analyse the patient data from a remote location. In this setting, the BAN acts as a provider of patient data and the healthcare practitioner acts as a user of that data.

This section presents the BAN, the service platform and identifies the technical challenges that have been addressed during its development.

2.1 Body Area Network

The healthcare BAN consists of sensors, actuators, communication and processing facilities. Depending on what type of patient data must be collected, different medical sensors can be integrated into the BAN. For example, to measure pulse rate and oxygen saturation, an oximeter sensor is attached to the patients' finger. In the case of an ECG measurement, electrodes are attached on the patient's arms and chest.

Communication between entities within a BAN is called intra-BAN communication. Our current prototype uses Bluetooth for intra-BAN communication. To use the BAN for remote monitoring external communication is required which is called extra-BAN communication. The gateway that facilitates extra-BAN

communication is called the Mobile Base Unit (MBU). Our current prototype employs an HP iPAQ H3870, that runs Familiar Linux and a J2ME [7] compliant Java virtual machine, as an MBU. Other mobile devices, such as a SmartPhone or a J2ME enabled PDA could also act as an MBU.

Fig. 1 shows the architecture of a BAN. Sensors and actuators establish an ad-hoc network and use the MBU to communicate outside the BAN.

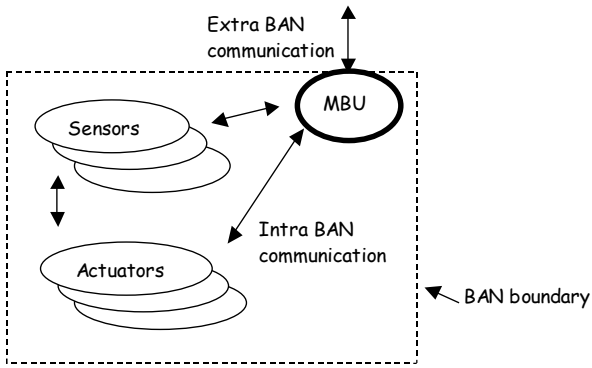


Fig. 1. BAN architecture

2.2 Platform Functional Architecture

Fig. 2 shows the functional architecture of the service platform. The dotted square boxes indicate the physical location where parts of the service platform will be executing. The rounded boxes represent the functional entities of the architecture, while the black oval shaped dots represent interaction points between the functional entities. The M-health service platform consists of sensor and actuator services, intra-BAN and extra-BAN communication providers and an M-health service layer. The intra-BAN and extra-BAN communication providers represent the communication services offered by intra-BAN communication networks (e.g. Bluetooth) and extra-BAN communication networks (e.g. UMTS), respectively. The M-health service layer integrates and adds value to the intra-BAN and extra-BAN communication providers. The M-health service layer masks applications from specific characteristics of the underlying communication providers.

The service platform supports two types of applications. The first type are targeted at direct interaction with m-health service users and can range from simple viewer applications that provide a graphical display of the BAN data, to complicated applications that analyze the data. They can be deployed on the MBU (for on-site use e.g. by a visiting nurse) or on the servers or workstations of the healthcare provider (for use e.g. by a healthcare practitioner). The second type of applications add functionality to the core m-health service, such as permanent data storage or on-demand real-time data streaming. These applications are deployed at the healthcare servers providing offline or online (with respect to the MBU status) data processing.

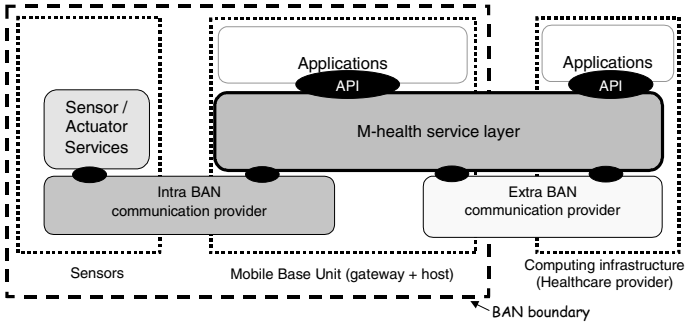


Fig. 2. Service platform functional architecture

The API interaction points in Fig. 2 represent the two possible locations where application components can interact with the m-health service platform.

2.3 Technical Challenges

In this paper, we focus on the platform parts related to the extra-BAN communications. The description of the MobiHealth service platform in the previous section exposes the following technical issues which we address in the next sections. In particular these issues are imposed by the characteristics of m-health delivery, e.g. ambulatory patients, privacy sensitive and trustability of medical data and the multi stakeholder involvement in today's healthcare delivery

- How does the platform discover and register BANs of ambulatory end-users?

A BAN and in particular the MBU will typically be switched on/off anytime at the convenience or the necessity of the end-user (e.g. patient) located anywhere in the coverage area of a 2.5/3G wireless network. A switch-off needs to be identified by the platform to release the service and its resources. On the other hand, the platform needs to discover a BAN which is switched on to enable a peer end-user at a healthcare centre to find the BAN data. This process of mobile BAN discovery is further complicated by the policy of the 2.5/3G wireless network operators, which typically assign an IP address to a terminal (i.e. MBU) dynamically from the private address space [8] at the establishment of the wireless connection (e.g. GPRS). Therefore, a BAN/MBU discovery and a release mechanism has to be developed for the MobiHealth service platform.

- How to deal with the limitation of the resources of BANs used for monitoring services?

In the MobiHealth service platform, the supplier of data for remote monitoring is the BAN, which should be wireless and wearable. Traditionally, providers of data (such as web servers) are deployed on a computing infrastructure with sufficient network and processing capacity. Consumers of data (such as web browsers) assume that providers are available most of the time (except for maintenance) and with sufficient

bandwidth to serve a reasonable amount of consumers. For the service platform, the producer and consumer roles are inverted because the provider of data is deployed on a mobile device (i.e. the MBU) while the consumer of data is deployed on a fixed host with sufficient processing and communication capacity. It is a technical challenge to deal with this inverted-producer-consumer problem in the design of the platform.

- How to provide reliable and secure end-to-end healthcare services in multi healthcare delivery stakeholders domains?

It is generally required that for privacy reasons medical data of patients has to be transferred in a secure way. The transfer of the data needs also to be reliable to enable correct interpretation by healthcare practitioners. It is a technical challenge to develop an architecture and the associated mechanisms for an m-health service platform involving multi healthcare delivery stakeholders that highlight the added value of the involved stakeholders. For example, the MobiHealth call centre which addresses the reliability and security issues of m-health delivery in such a way that it releases other stakeholders from the burden of addressing these issues in their full complexity themselves. The scarce bandwidth resources of GPRS offered today by the operators and their policy to prioritize voice above data complicate the data transfer mechanisms further. Application protocols designed for use in local area networks, such as RMI and IIOP, are in this context not applicable for the GPRS links.

- How to provide a scalable end-to-end m-health service?

The Mobihealth service platform must support a many-to-many relation between the BAN and the MobiHealth secondary care centre. Potentially more than 100.000 simultaneously operating BANs should be supported. The medical data produced by those BANs could be of interest to various healthcare practitioners. Therefore the BAN must be able to produce medical data from as many locations as possible (thus supporting full patient mobility) and the medical data must be accessible from as many as healthcare centres. In summary [9],

– the service platform must scale to a size that it can support 100.000+ BANs (numerical scalability)
– the platform must support multiple (secondary) healthcare centres (i.e. multiple organizational domains) (organizational scalability)
– the platform must scale to a large geographical area (i.e. European or world scale) (geographical scalability).

3 JINI and MobiHealth

The internals of the MobiHealth service platform depend on JINI technology [10], [11]. This section summarizes the key features of JINI and identifies how these features contribute to the services offered by our platform.

3.1 JINI

One of the problems that the MobiHealth service platform solves is the dynamic discovery of the services offered by a BAN. A BAN is configured to the needs of end-users and can typically be switched on/off anytime at the convenience or the necessity of the end-user (e.g. patient). A key design decision is to represent the services offered by a BAN as a JINI service. JINI provides mechanisms whereby the service lifetime, service location and service implementation details become irrelevant to the service user.

A JINI service provider can dynamically attach or detach itself from a given JINI network community, a so-called djinn. A special core service, called LookUp Service (LUS), supports the registration of services. A service provider (e.g. a BAN) registers a Service Proxy into LUS together with a set of predefined service attributes. Before a service provider can register a service it must contact the LUS, which is the discovery process (i.e. interaction 1 and 3 in Fig. 3). The figure shows the interactions between the service provider, service user and the LUS, the first 4 interactions belong to the set-up phase and are preliminary to the service execution phase, i.e. the data transfer phase in monitoring applications. An essential element of the discovery and registration process is the exchange of a ServiceProxy. This proxy encapsulates the client part of the service logic and communication protocols needed for service execution.

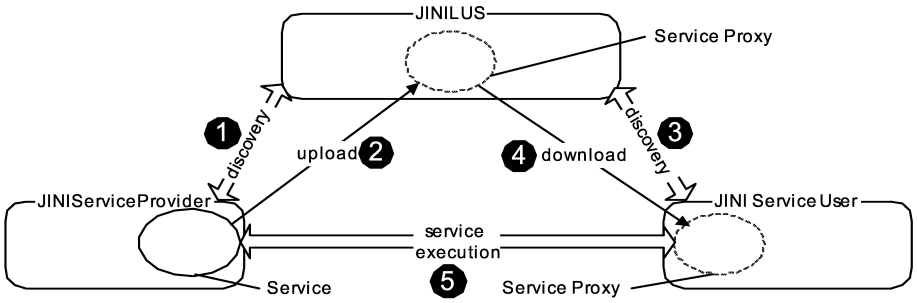


Fig. 3. Service discovery and registration

A service user only needs to know the interface to the service. In some cases the service proxy includes the whole service implementation, moving the service execution entirely into the service users virtual machine (VM). In other cases the service proxy contains networking functionality, which will allow the service user to connect to the remote service provider. The Mobihealth platform uses the latter scheme, which allows for adaptation of the protocol used for service execution, without changes in the service user (e.g. the peer client of the monitoring system at a healthcare center).

When a service user discovers a LUS, it can either browse it for a particular service or it can register itself for service availability notifications based on event messages.

3.2 Leasing

Service resources in a JINI network are acquired for a certain period based on a service lease. A service user must renew a lease within certain time period. If that lease cannot be renewed, for whatever reason (network failure, system crash, etc.), the client can conclude that the service is not available anymore. Similarly, the service provider can free specific resources just because a service user has not renewed a granted lease.

3.3 JINI Surrogate Architecture

Although service execution can be based on any protocol, JINI requires that RMI is available for LUS discovery. Consequently a service provider must execute in a VM that supports RMI. Limited resources inhibit the use of RMI on the MBU, therefore making the MBU unfit as a host for a JINI service provider. We employ the JINI Surrogate Architecture [12], [13] (Fig. 4) to enable a BAN to still offer services in a JINI network.

According to the Surrogate Architecture, a device which initially cannot join a JINI network because it does not meet the connectivity and functional requirements, can still join if it is represented by a surrogate object. The surrogate object acts as a service provider on behalf of the device and shields service users from the specific means to communicate with the device. At application level, this yields that a healthcare practitioner at a healthcare centre transparently retrieve BAN data from the surrogate host as if he retrieves the data from the BAN. In a multi stakeholder m-health delivery model, surrogate hosts typically reside at the domains of the call centers.

Furthermore, surrogates rely on a Surrogate Host for life cycle management and a runtime environment. Device specific communication between the surrogate and the device is called the Interconnect Protocol.

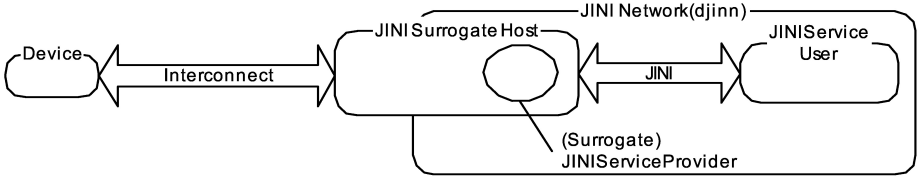


Fig. 4. Elements of Surrogate Architecture

The surrogate architecture specification requires the interconnect protocol to fulfill at least three mechanisms: discovery, surrogate upload and keep-alive.

The purpose of the discovery mechanism is to make a surrogate host aware of the device existence and vice versa. Implementation depends on the device communication capabilities.

Once a device and the surrogate host discovered each other, the device can join the JINI network. To do so, the surrogate host must be provided with the surrogate object

that will act in the JINI network on the behalf of the device. The device itself can upload the surrogate object or it can send to the surrogate host the location point from where the surrogate can be downloaded.

After a surrogate has been instantiated and activated by the surrogate host, the device must guarantee that it is able to perform the exported service. Consequently, the interconnect protocol must implement a keep-alive mechanism to inform the surrogate host if the device is still active and connected. As soon as the device cannot confirm its online status, the surrogate host can deactivate the JINI service and its correspondent surrogate object.

4 Design and Implementation of the BANip

The MobiHealth service platform uses a surrogate object to act on behalf of the MBU and thus allows a BAN to offer its services in a JINI network despite its resource limitations. The BAN interconnect protocol (BANip) is our protocol for interaction between the MBU surrogate and the MBU device. The BANip fulfils the three basic requirements of the surrogate architecture (i.e. discovery, surrogate upload and keep-alive).

4.1 Internal Structure

Fig. 5 shows a refined view of the M-health service layer in the data transfer phase. The arrows in the figure show the flow of the BAN data. The BANip entity is a protocol entity for the BAN interconnect protocol. Peer entities can be found on the MBU and near the surrogate component. The BANip entities communicate through a proxy, that has authenticated and authorized the BAN in the set-up phase. The surrogate component acts as a regular JINI service provider which in our case uses Remote Method Invocation (RMI) for service execution.

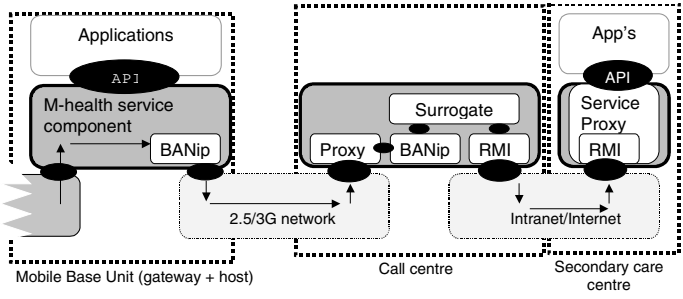


Fig. 5. Refined view of the service platform in de data transfer phase

Applications, e.g., situated in a secondary care center, requiring access to the MBU services download a service proxy in the set-up phase and use that proxy to transparently communicate with the surrogate in the data transfer phase.

4.2 Interconnect Protocol Design

The BANip defines a set of messages plus a specific programming model, following closely, in that respect, the implementation of the IP interconnect [14].

Since the MBU device can only support a Java 2 ME virtual machine (CDC or CLDC) and since the communication is over wireless networks, we chose to encapsulate the BANip into HTTP. HTTP client support is mandatory for all J2ME (both CDC and CLDC configurations) virtual machines, which makes our platform portable across various mobile devices. An additional benefit of using HTTP is that we can use standard HTTP proxy mechanisms for BAN authorization and we can use HTTPS for secure delivery of BAN data.

As identified earlier, the MBU obtains its IP address from a telecom operator network therefore it is often not possible to initiate a connection from the Internet to the MBU. This problem is solved when the communication between the MBU device and the MBU surrogate is encapsulated in HTTP requests that are initiated from the MBU device.

The BANip has three message categories: surrogate lifecycle messages, private interconnect messages and externally initiated messages. Fig. 6 shows an example for each of these message categories, labeled 1,2 and 3 respectively.

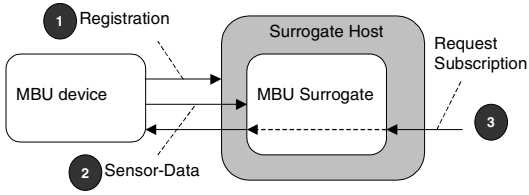


Fig. 6. Examples of BANip messages

The `Registration` message is a surrogate lifecycle message that an MBU device sends to the surrogate host. The message includes initialization data for the surrogate and a URL that points to the location of the JAR file that contains the MBU surrogate implementation. This is sufficient information for the surrogate host to obtain and instantiate the appropriate surrogate object.

Once activated the surrogate host expects at regular intervals a “keep-alive” message, which is another surrogate lifecycle message. If such is not received, the surrogate can be deactivated and its JINI service offer removed from the LUS.

The `Sensor-Data` message is a private interconnect message, pushed from the MBU to the surrogate. The purpose of this message is to deliver sensor data to the surrogate object. We chose for a push mechanism as sensor data arrives at the MBU at regular intervals and we want to cache this data at the surrogate.

The `RequestSubscription` message is an externally initiated message, since it is initiated at the service proxy and propagates through the surrogate to the MBU device. However, the surrogate has no means to actively post HTTP requests to the MBU device because this device cannot act as an HTTP server. Therefore, we

encapsulate the `RequestSubscription` message into the HTTP response of the keep-alive messages that arrive regularly at the surrogate.

Because the low bandwidth and high latency are very common characteristic of a GPRS link, we focused our attention in eliminating the overhead of the HTTP interconnect as much as possible. For example, instead of sending each BAN measurement in a single HTTP post request, a number of "sensor data" messages is first collected and then sent together. Additional improvement was achieved with HTTP chunking, where BANip messages are conveyed as chunks of one long-term HTTP request. The third optimisation is the implementation of a "deflate" compression algorithm.

5 Conclusions

The MobiHealth service platform resolves a number of technical challenges when communicating with mobile devices over a 2.5/3G public wireless infrastructure.

A key design decision for the platform is to represent the services offered by a BAN as a JINI service. Consequently, the discovery and registration problem of BANs has been solved. We solved the inverted producer-consumer problem using the JINI Surrogate Architecture and have implemented an HTTP based interconnect protocol called BANip. Our platform simplifies the development of remote healthcare applications, deployed in a call-center or in a secondary care center. The platform shields these organizations from the complexity of securely communicating with wireless mobile devices over 2.5/3G networks. We apply webserver technology for our BANip that can be extended with well-know techniques (e.g. load-balancing) to support at least 100.000 BANs. Our choice for HTTP provides improved flexibility for example for porting the BANip to other mobile devices. For this reason we believe our platform to be scalable in the numerical, organizational and geographical dimensions.

The platform will be deployed in nine healthcare field trials in four European countries to evaluate its functionality and to assess the feasibility of 2.5/3G wireless technologies in combination with Java and Internet technologies for m-health service delivery.

Acknowledgement. The authors would like to thank George Koprinkov for his diligent contribution to the implementation of the BANip and the MobiHealth service platform.

References

1. Gagliano, D.M. and Xiao, Y., "Mobile Telemedicine Testbed" in Proc. 1997 American Medical Informatics association (AMIA) Annual Fall Symposium, pp.383–387, National Library of Medicine Project N0-1-LM-6-3541, 1997;

2. MobiHealth, “MobiHealth project IST-2001-36006”, EC programme IST, 2002, http://www.mobihealth.org;
3. N. Maglaveras, et al, “Home care delivery through the mobile telecommunications platform: the Citizen Health System (CHS) perspective”, International Journal of Medical Informatics 68 (2002), pp. 99–111.
4. Hung, K. and Zhang, Y-T., “Implementation of a WAP-Based Telemedicine System for Patient Monitoring”, in IEEE transactions on Information Technology in Biomedicine, vol. 7, no. 2, June 2003, pp. 101–107;
5. Val Jones, Richard Bults, Dimitri Konstantas, Pieter AM Vierhout, Healthcare PANs: Personal Area Networks for trauma care and home care, Proceedings Fourth International Symposium on Wireless Personal Multimedia Communications (WPMC), Sept. 9–12, 2001, Aalborg, Denmark;
6. E. Jovanov, et al, “Stress Monitoring Using a Distributed Wireless Intelligent Sensor System”, in IEEE Engineering in Medicine and Biology Magazine, May/June 2003
7. Sun Microsystems, “CDC: An Application Framework for Personal Mobile Devices”, June 2003, http://java.sun.com/j2me
8. Y. Rekhter et al., “Address Allocation for Private Internets”, RFC 1918, February 1996.
9. B. Clifford Neuman, Scale in Distributed Systems, Readings in Distributed Computing Systems, IEEE Computer Society Press, ISBN 0818630329, 1994
10. Sun Microsystems, “Jini Architecture Specification”, http://www.sun.com/software/jini/specs/jini1_2.pdf
11. Sun Microsystems, “Jini Technology Core Platform Specification”, http://www.sun.com/software/jini/specs/core1_2.pdf
12. Sun Microsystems, “Jini Technology Surrogate Architecture Specification”, July 2001, http://surrogate.jini.org/sa.pdf
13. Sun Microsystems, “Jini Technology Surrogate Architecture Overview”, http://surrogate.jini.org/overview.pdf
14. Sun Microsystems, “Jini Technology IP Interconnect Specification”, http://ipsurrogate.jini.org/sa-ip.pdf

Structural Testing of Mobile Agents

Márcio Delamaro[1] and Auri Marcelo Rizzo Vincenzi[1,2]

[1] Centro Universitário Eurípides de Marília (UNIVEM)
Av Hygino Muzzi Filho, 529, 17525901, Marília - SP, Brazil
Phone: +55(14) 421-0836,
delamaro@fundanet.br
[2] Instituto de Ciências Matemáticas e Computação (ICMC-USP)
Av. Trabalhador Sancarlensse, 400, 13560-970, São Carlos - SP, Brazil
Phone: +55(16) 273-0996,
auri@icmc.usp.br

Abstract. Programs based on code mobility, in particular on Mobile Agents, are an alternative approach to conventional distributed systems. Many aspects of Mobile Agent development have been addressed. In special, languages and environments to support code mobility and Mobile Agents implementation have been proposed. The testing activity though, has practically not been addressed so far. In this paper we present an approach and a tool named JaBUTi/MA to aid the test of Mobile Agents based on structural testing criteria. The structural test of Mobile Agents introduces a series of new difficulties, in particular the problem of how to collect execution data on such a distributed environment.

Keywords: Development and analysis tools; Mobile Agent; Structural software testing; JaBUTi/MA

1 Introduction

Code mobility is defined as "the capability to reconfigure dynamically, at runtime the binding between the software components of the application and their physical location within a computer network"[1]. This is not a new concept and some "primitive" form of code migration has been around for a while [2]. Currently, several systems exist to provide enhanced forms of code mobility, aiming at exploring the characteristics that make this kind of system preferable to the conventional distributed systems.

Karnik and Tripathi [3], for instance, identify as advantages for mobile code: 1) reduction of network use; 2) increased asynchrony between clients and servers; 3) capacity to add client-specific functionality to servers; and 4) capacity to introduce concurrency. They cite the case of search and filtering applications which need to download a large amount of data to produce a relatively small quantity of result data. Using mobile agents – which execute in the server host – a bandwidth reduction is achieved, since only the result (in addition to the agent itself) has to transit through the network. Picco [4] also highlights a series

N. Guelfi, E. Astesiano, G. Reggio(Eds.): FIDJI 2003, LNCS 2952, pp. 73–85, 2004.

of advantages of using mobile code, such as: reduced bandwidth consumption and enhanced flexibility.

Much of the research on code mobility has been concentrated on its technological aspects. In particular, several languages and environments to support the development of code mobility applications have appeared in the last few years. Karnik and Tripathi [3] and Fuggetta *et al.* [2] have identified a dozen of them.

Testing is an essential activity in the software lifecycle. A lot of effort has been spent on the development of testing techniques and tools. Several issues are still open in this area and the adoption of new technologies introduces new ones. Distributed systems, in particular in the form of Mobile Agents, bring new challenges to the testing research area but very little has been done to face them.

This paper proposes the use of structural testing on Mobile Agents. It discusses the difficulties of doing so and presents a tool, named JaBUTi/MA, that supports the application of control-flow and data-flow adequacy criteria to Mobile Agents. In the next section, a review on Mobile Agents and μCODE – an API to support code mobility in Java – is presented. In Section 3 a tool for structural testing of (ordinary) Java programs named JaBUTi is presented. In Section 4 the problems of using structural testing on Mobile Agent based programs are discussed and the solutions adopted in an extension of JaBUTi are described. In Section 5 the final remarks are presented. In Appendix A a μCODE toy-agent used to exemplify the concepts proposed in the paper is shown.

2 Mobile Agents and μCode

Different paradigms can be used to develop an application with some sort of code mobility; for instance, Remote Evaluation, Code on Demand or Mobile Agent approaches [1]. In this paper we are particulary interested in the Mobile Agent paradigm which, according to [3], can be defined as "a program that represents a user in a computer network and can migrate autonomously from node to node to perform some computation on behalf of the user".

μCODE [5] is a lightweight and flexible Java API for code mobility that places a lot of emphasis on modularity and minimality. μCODE is not (just) a mobile agent system. It supplies the programmer with a set of primitive operations for code mobility. On top of them the programmer can build his/her own abstractions.

The use of μCODE revolves around three fundamental concepts: groups, group handlers, and class spaces. *Groups* are the units of mobility, and provide a container that can be filled with arbitrary classes and objects (including `Thread` objects) and shipped to a destination.

The destination of a group is a μServer, an abstraction of the run-time support. In the destination μServer, the mix of classes and objects must be extracted from the group and used in some coherent way, possibly to generate new threads. This is the task of the *group handler*, an object that is instantiated and accessed in the destination μServer and whose class is specified by the programmer at

group creation time. Any object can be a group handler. Programmers can define their own specialized group handlers and, in doing so, effectively define their own mobility primitives.

During group reconstruction the system needs to locate classes and make them available to the group handler. The classes extracted from the group must be placed into a separate name space, to avoid name clashes with classes reconstructed from other groups. This capability is provided by the *class space*. Classes shipped in the same group are placed together in a private class space, associated with that group.

μCODE also provides a few higher-level abstractions built on top of the core concepts described above. These abstractions include primitives to remotely clone and spawn threads, ship and fetch classes to and from a remote μServer, and a full-fledged implementation of the mobile agent concept. An abstract class *mucode.abstractions.MuAgent* is provided to implement the mobile agent abstraction. It extends the *java.lang.Thread* class. The programmer should extend the *MuAgent* class and implement its *run* method, that defines the agent's behavior and is executed in a new thread when the agent is started. The method *MuAgent.go(String host)* interrupts the execution of the agent thread and ships the *MuAgent* to the destination μServer, where it will be restarted. Since μCODE provides only weak mobility [2], the agent in the destination always starts executing from the beginning of the *run* method again. However, the data state of the agent, i.e., the values of its object fields, is preserved on migration.

The tool named JaBUTi/MA supports the collection of execution data of a mobile agent. We consider a mobile agent built on top of μCODE, although the approach can be adapted to work with other platforms. From this point on, when the term "mobile agent" (or simply "agent") is used it refers to a μCODE mobile agent, i.e., an instantiation of a subclass of *mucode.abstractions.MuAgent*. Appendix A shows the "Tripper", an example of such a mobile agent.

3 Structural Testing and JaBUTi

In spite of all the research and improvement on software development methods, techniques and tools, software products are still released with faults. In this scenario, the testing activity is extremely important and represents the last revision of the product characteristics and the last opportunity to identify sources of errors before releasing. The quality of software testing is closely bound to the quality of the **test set** used. **Adequacy criteria** determine requirements to be fulfilled by good test sets. Structural testing, specially control- and data-flow criteria [6] have been widely studied and explored in the last decades. Control-flow criteria define requirements such as coverage of nodes (statements) or edges (branches) from a program representation know as **Control Flow Graph** (CFG). Data-flow criteria associate information about variable assignment and usage to the CFG elements and then require the execution of paths that exercise associations between these assignments and uses (def-use associations).

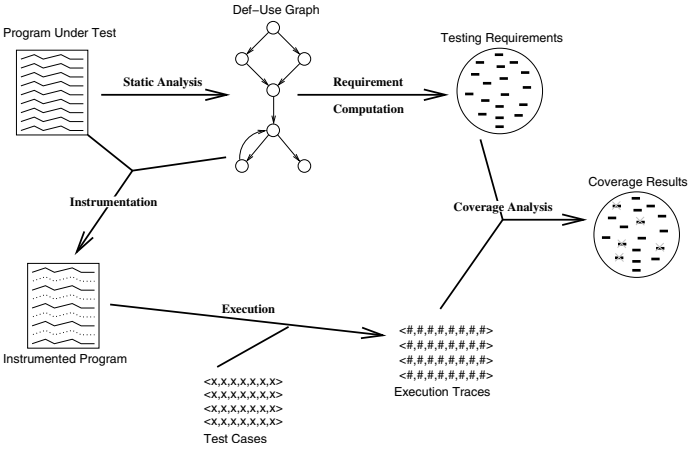


Fig. 1. Summary of a structural testing tool operation.

The use of structural testing requires a series of tasks such as program analysis for requirement computation and coverage assessment which are not trivial and clearly cannot be done manually by the tester. Testing tools to support such criteria are essential to assure quality in testing activity. Figure 1 summarizes how such a tool operates.

JaBUTi (Java Bytecode Understanding and Testing)[7] is a tool to support structural testing of Java programs. It is similar to other such tools and its implementation follows the diagram in Figure 1. What distinguishes JaBUTi from other tools is that its static analysis and instrumentation processes are not done on the Java source code but on the object code (Java bytecode). In fact, the tester does not even need the source code to test a program. However, because the Java bytecode retains information about the source code, if the source is available, the tool can map the data on test requirements and coverage back to the source level. For the test of Mobile Agents it is very important that the analysis is performed at bytecode level, as discussed in Section 4.2.

We can identify two main parts in JaBUTi. The first makes all the analysis, requirement computation and test case evaluation, and corresponds to the top part of Figure 1. The second (bottom part of Figure 1) is a test driver that instruments the classes, loads and executes the instrumented program.

The test driver has an appropriate class loader that instruments the original classes by placing at the beginning of each CFG node a call to a static method, *probe.DefaultProber.probe*, that stores the information about the piece of code (the node) about to be executed. When the program ends, another method, *probe.DefaultProber.dump*, is called and all the data collected by the calls to *probe.DefaultProber.probe* are written to a trace file. Those data constitute a new test case.

The analysis tool constantly polls the trace file. When a new test case is inserted in the trace file, the tool updates the test information, recomputing the requirement coverage. The analysis part also has several facilities to perform code visualization, CFG visualization, report generation and test case manipulation. A program test is organized in a test project that can be saved and reloaded by the tester.

The way JaBUTi instruments and collects execution data is limited and inappropriate for programs with code mobility. In the next section we discuss the problems of the implementation above and present an alternative solution which allows the collection of execution data from a Mobile Agent program travelling in a network.

Our approach does not introduce innovations in the field of structural testing technique or criteria. Our contribution is to provide a way to collect trace data from a Mobile Agent executing in a distributed environment. Its clear that such data can be used for code coverage assessment, as we have done here. However, whether or how that information could be used to deal with particular aspects of distributed systems such as concurrency, replicability or temporal location of the agent, is not our subject here.

4 Structural Test of Mobile Agents and JaBUTi/MA

As explained before, we are calling a Mobile Agent a thread that extends *mucode.abstraction.MuAgent* and that, with the aid of the μCODE API, can autonomously migrate from one host to another. In the destination the thread is received and restarted by a μServer. Most structural testing tools – including JaBUTi– would fail on the test of such an application, mostly because the mechanisms to collect execution data do not take into account the distributed execution of the application. Instrumenting the program to write traces to a trace file, for example, is unfeasible and useless. Suppose an instrumented agent starts at host **A** writing trace data to a trace file. When the agent moves to host **B** it looses the reference to the trace file. In host **B** the trace file does not exist, the filesystem may not be accessible to the agent and even if the agent could write trace data to another file, it would be useless because the file would not be accessible to the analysis tool. In addition, other mechanisms for remote communication between the agent and the analysis tool such as database connections, sockets, etc cannot be used as well because it is not guaranteed that such mechanisms are available in every host.

In summary, the agent must be instrumented in such a way that the trace data is communicated to a "central" point; it can rely only on the communication mechanisms guaranteed to be present in every host where it executes. In the case of Mobile Agents, the only communication mechanism known to be present is the own mobile agent structure, i.e., μCODE. Thus, the idea is to instrument the Mobile Agent under teste (MAUT) in such a way that the trace data is collected and eventually sent in a Test Agent (TA) to a Test μServer (TμS). The

TμS takes care of delivering the trace data to JaBUTi for analysis. Figure 2(a) depicts the main elements in this scenario.

In our implementation, called JaBUTi/MA there are three ways to test and evaluate the coverage of a Mobile Agent: 1) instrumenting the agent before it starts, i.e., at the client host (Figure 2(b)); 2) instrumenting the agent when it arrives at a given μServer and then forwarding the instrumented code to the next hosts(Figure 2(c)); and 3) instrumenting the agent when it arrives at a given μServer and then forwarding the original (non instrumented) code to the next hosts (Figure 2(d)).

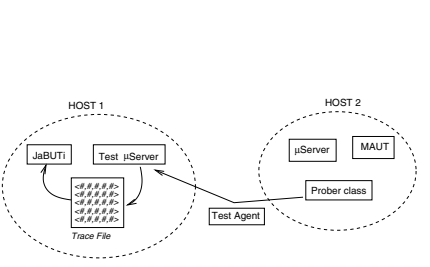
These three options intend to provide the tester with a higher degree of flexibility, allowing him/her to choose when and where to instrument the agent and to start collecting trace data. The first is the "traditional" way of instrumenting, which permits the tester to collect trace data from all the path of network nodes the agent executes on. The second is useful if one wants to restrict trace data collection to a set of hosts. It is the case, for instance, of an agent that from a given point in its trip through the nodes presents an unexpected behavior. With the server-based instrumentation the tester may focus his/her attention to the agent execution only from that point on. The third restricts even more the source of trace data collection. The program is instrumented and trace data is collected in a single host. It is useful, for instance, for testing the agent in a particular situation, in a given problematic host. In addition, in a single run the agent can find several of these single host instrumenters, allowing the tester to collect trace data from each one of them. Next we discuss each of these types of instrumentation.

4.1 Client-Based Instrumentation

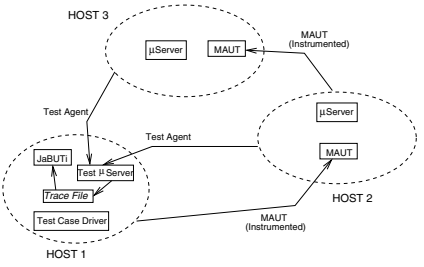
The first configuration of JaBUTi/MA to test Mobile Agents is shown in Figure 2(b). In this case the following elements are present in the testing environment:

- JaBUTi itself, i.e., the piece of the tool that allows the tester to create projects, analyze coverage, create reports, etc;
- The test server (TμS). It is a specialization of the μServer and is used to receive the test agents (TA) which bring data on the MAUT execution;
- The test case driver, which is used to instrument and start the MAUT execution.

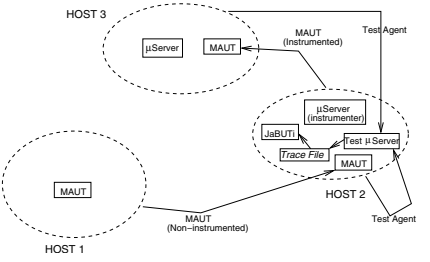
The first two must be in the same host, while the last may be in a different one. The JaBUTi/MA tool does the same work as the non-MA version, i.e., keeps polling a trace file and updating the coverage information. The TμS is attached to a port in the same host and receives the TAs with information about a MAUT in a given foreign host. At startup, the server must be configured to: 1) listen to a given port; 2) accept TAs from a given list of project names; and 3) write to a given list of trace files. Items 2) and 3) are used to bind a given name to a trace file. The TμS can serve several TAs, coming from several places and several



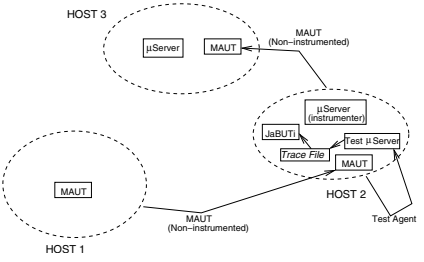
(a) Elements to test a Mobile Agent.



(b) Client instrumentation.



(c) Server instrumentation (forwarding instrumented code).



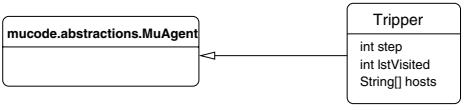
(d) Server instrumentation (forwarding non-instrumented code.)

Fig. 2. The possible configurations of JaBUTi/MA.

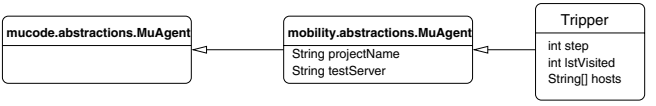
MAUTs, so the pair ⟨project name, trace file⟩ indicates to which trace file the server should write data when a given TA arrives. If a TA is not identified by the server, it is simply discarded. The name of the project is determined by the test driver when the MAUT is instrumented, as described below.

The test driver, as explained previously, is responsible for instrumenting and starting the program under test. It inserts in the code of the MAUT calls to a static method which stores information about the node of the CFG being executed. For JaBUTi/MA the static class taking care of trace collection and delivery to the analysis tool is not the same *probe.Prober*. It is the class *mobility.HostProber* (from this point on, referred to as "the prober class").

In addition to collecting trace data, the prober class must deliver it to the analysis tool. Two static fields are present in this class: the name of the project associated to the MAUT testing and the address of the destination TμS. When the MAUT is first run (already instrumented) at its original host, it has the support of the test driver that loads the prober class, initializes those two static fields and then starts the execution of the agent. When the agent finishes ex-



(a) Original hierarchy



(b) Hierarchy of instrumented code

Fig. 3. Instrumentation in the agent class.

ecuting, the prober class has the information necessary to create and send the TA. This works for the original host but not when the MAUT moves to another host. The code of all the classes and objects it depends on are migrated with it, including the prober class. Unfortunately, the values of static fields are lost; thus, in the next host the prober class would not have the correct values of the project and the destination TμS.

The solution we use is to instrument the agent class in such a way that that information is also inserted in the MAUT. This is done as illustrated in Figure 3 for the agent in Appendix A. The class of the agent (*Tripper*) gains two fields by interposing another class between it and the original *mucode.abstractions.MuAgent* in the class hierarchy. When the agent starts, if the fields in the agent are not set it means that the agent has been recently instantiated (did not migrate yet) and the information is read from the static fields in the prober class. Otherwise, the MAUT is being restarted in a new host and the information in the agent fields is used to initialize the fields in the prober class. In summary, the MAUT carries the information that cannot travel in the prober class.

In addition to inserting the fields in the agent, it is necessary to instrument the MAUT such that when it starts running, the project name and destination TμS fields are read from/written to the prober class and, when it stops, the trace data is sent to the TμS. As mentioned before, a Mobile Agent in μCODE is a thread that is always started (or restarted) in its *run* method. Thus, what is done is to instrument this method "wrapping" it in something similar to a `try/finally` statement, as shown in the code of Figure 4.

The instrumentation is done directly in the bytecode but reflects exactly the behavior of the code in Figure 4. When the thread (the MAUT) starts running, the static fields are read from (in the case the agent is being started for the first

```
1   public void run() {
2       if ( the project name and destination TμS  are set in the MAUT)
3            write them to the static fields in the class  HostProber
4       else
5            read them from the class  HostProber
6       try {
7           Here comes the code of the original method
8       }
9       finally {
10          Calls method  HostProber.dump that finishes trace data
11          collection and send it to the destination TμS  within a TA
12      }
13  }
```

Fig. 4. Representation of the method *run* in the agent after being instrumented.

time) or written to (in the case the agent is being restarted after migration) the prober class.

In addition, the instrumentation of the *run* method allows the agent to notify the prober class that it is time to create and send the TA to the TμS. The code inside the `finally` block takes care of it. If the agent ends "normally", i.e., finishes without moving to another server, the `finally` block is executed and the agent simply ends by falling off its code. If the agent moves to another host, it finishes by calling method *MuAgent.go*. That method builds a *Group* object with the thread object and needed code and sends it to the destination. Then the agent thread is stopped by raising a *java.lang.ThreadDeath* exception that goes up to the *run* method where the `finally` block is executed, the exception is rethrown and the thread ends.

Figure 5(a) shows the evaluation of the *Tripper* agent, instrumented in the client and executed in three different hosts. Each time the agent is started or restarted is characterized as a different test case. In this case, we have six test cases that correspond to two executions in each host. The first five correspond to calling twice method *perform* and then moving to the next host. The last one corresponds to testing the counter *step* and finishing the execution in the third host, so the code coverage is lower (15%) than in the other test cases.

4.2 Server-Based Instrumentation (Multiple Hosts)

To make the JaBUTi/MA environment more flexible, we also allow the instrumentation to be done in the server side, i.e., when the MAUT arrives at a given μServer. The scenario is depicted in Figure 2(c). The MAUT is not instrumented before it is launched. It is started normally – without the need of a test driver – and travels until it reaches a host with a μServer that performs the instrumentation. That is why it is so important to perform instrumentation at the bytecode. When arriving at a given server, only the bytecode is available, not the source code. The instrumentation is exactly the same described in the previous section. The instrumenter μServer, when started, must be configured to instrument a given set of classes and to set the project name and destination TμS.

When a MAUT arrives, the server unpacks it, checks which classes should be instrumented, does the instrumentation, sets the static fields in the prober

(a) Client instrumentation (b) Single server instrumentation

Fig. 5. Test case visualization of the *Tripper* agent.

class and then restarts the agent. From that point on, everything works as if the agent were instrumented in the client, i.e., if it moves to other hosts it will do in its instrumented form and the trace information will continue to be collected.

It is important to note that the instrumenter is a special version of the μServer, differently of those shown in Figure 2(b). We had to change μCode code in order to introduce the instrumentation step when the agent is unpacked. This is different from extending the μServer, as has been done, for instance, with the TμS. In TμS, the server is subclassed and new functionality is introduced, as writing data brought by the TA to the trace file. In the case of instrumentation though, the process of unpacking and starting the agent cannot be altered by simply sub-classing the server and a new version of μCode was created. The other servers in the path of the agent that do not perform instrumentation can be regular μServers.

The evaluation of the *Tripper* agent being instrumented in the first server after it leaves its original host would produce a result similar to that shown in Figure 5(a). There would be only five test cases though (in contrast to the six in the client-based instrumentation) because when it first executes in the original host it is not instrumented yet.

4.3 Server-Based Instrumentation (Single Host)

The last variation in the testing environment described above is to restrict the trace data collection to a single host. In this case, shown in Figure 2(d), it is still the server that does the instrumentation. The instrumented code is executed in that host but is not propagated to the next hosts.

The instrumenter μServer is configured at startup not to forward the instrumented code. The only difference in the server behavior is when the code arrives. Ordinarily, the instrumenter μServer unpacks the classes to reconstruct

the agent, instruments determined classes according to its configuration, loads them into a private class space (see Section 2), stores the instrumented classes' bytecodes and starts the agent. If the μServer is set to not forward the instrumented code, it loads the instrumented classes in the class space but stores the non-instrumented bytecode. Doing so, when the agent moves to another server, the original code is sent, not the instrumented one.

Figure 5(b) shows the evaluation of the *Tripper* being instrumented in the first server after it leaves its original host, but only there. As expected, only two test cases are present. They correspond to the execution of the MAUT, the two times it passes through that host.

5 Conclusion

Mobile code is an alternative to conventional distributed systems which according to the literature may present advantages for some classes of applications. Much of the effort in this area has been spent in technological aspects such as the development of environment and languages to support code mobility This paper presented an approach and a tool to support the application of structural testing criteria to applications based on the Mobile Agent paradigm. Code mobility introduces new challenges to the test activity. In particular, the use of structural criteria faces the serious problem of execution data collection, as the agent moves through the nodes of a network.

Most structural tools use program instrumentation to register and store such data. Code inserted in determined points in the program registers the execution flow in that point and usually stores that information in a trace file that is analyzed to evaluate code coverage. When Mobile Agents are used, this approach is unfeasible and another way to collect the trace data is necessary. In this work we still use program instrumentation to deal with Mobile Agents. However, we developed a more sophisticated instrumentation model, based on two important features: 1) the instrumentation is done directly at the object program (Java bytecode); and 2) the instrumented code uses Mobile Agents themselves to communicate the trace data to a centralized point where such data can be analyzed.

As far as we are aware, this is a innovative work and no other environment to aid test case evaluation for Mobile Agents have been developed so far. The example used in this paper is just a toy-agent whose executions are the same on every host it passes through. We are currently working on some "real" examples, trying to evaluate the advantages and problems of using the approach described herein. In addition, we believe the approach presented here to collected execution data in a distributed environment can be useful not only for the testing activities but also in other tasks that may require or be improved by code coverage information, for instance, reliability assessment, debugging, and dynamic metrics computation.

Acknowledgments. This work has been supported by CNPq project number 474890/01-5 and by FAPESP. Thanks to Prof. Mario Jino from FEE-UNICAMP and to Prof. José Carlos Maldonado from ICMC-USP for their work on reviewing this paper.

References

1. Carzaniga, A., Picco, G., Vigna, G.: Designing Distributed Applications with Mobile Code Paradigms. In: Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston - MA (USA), ACM Press (1997) 22–32
2. Fuggetta, A., Picco, G., Vigna, G.: Understanding Code Mobility. IEEE Transactions on Software Engineering **24** (1998) 342–361
3. Karnik, N.M., Tripathi, A.R.: Design Issues in Mobile-Agent Programming Systems. IEEE Concurrency **6** (1998) 52–61
4. Picco, G.: Mobile Agents: An Introduction. Journal of Microprocessors and Microsystems **25** (2001) 65–74
5. Picco, G.: μCODE: A Lightweight and Flexible Mobile Code Toolkit. In: Proc. of Mobile Agents: 2^{nd} Int. Workshop MA'98. LNCS 1477, Springer (1998) 160–171
6. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for program test data selection. In: Proceedings of the 6th International Conference on Software Engineering, Tokio, Japan (1982) 272–278
7. Vincenzi, A.M.R., Delamaro, M.E., Maldonado, J.C., Wong, W.E.: Java Bytecode Static Analysis: Deriving Structural Testing Requirements. In: 2nd UK Software Testing Workshop – UK-Softest'2003, University of York, York, England, (2003)

A A μCode Mobile Agent Example

The code below exemplifies how a Mobile Agent in the μCODE is. The class stores the list of hosts (μServers in the format *address:port*) on which the agent will circulate. It also has an step counter (variable *step*) and a variable *lastVisited* to indicate which is the next server to go to. In the text this agent has been used as an example. It is executed using three different host. The agent executes steps (0,1) in its original host A, moves to a host B where it executes steps (2,3), moves to a host C where it executes (4,5) and returns to A and B where steps (6,7) and (8,9), respectively, are executed. It still migrates to host C where the variable *step* has value 10 and the agent ends.

```
1  import mucode.*;
2  import mucode.abstractions.*;
3
4  // This class implements the agent. It prints a message and increments
5  // the "step" variable twice then moves to the next host
6  public class Tripper extends MuAgent {
7  int step = 0, lastVisited = -1;
8  String[] hosts;
9
10     public Tripper(String[] hosts, MuServer aServer) {
11         super(aServer);
12         hosts = h;
13     }
14
15     // Executes "perform" twice and then go to the next server
16     public void run() {
17         for ( ; step < 10 ;) {
18             try {
19                 for (int i = 0; i < 2; i++, step++) {
20                     perform();
21                     Thread.sleep(500);
22                     Thread.yield();
23                 }
24                 try {
25                     lastVisited++;
26                     lastVisited = lastVisited % hosts.length;
27                     go(hosts[lastVisited]);
28                 }
29                 catch (java.io.IOException e)    {e.printStackTrace(); }
30                 catch (ClassNotFoundException e) {e.printStackTrace(); }
31             }
32             catch (InterruptedException e) { e.printStackTrace(); }
33         }
34     }
35
36     // Prints the step counter
37     public void perform() {
38         System.out.println("Tripper: step " + step);
39     }
40 }
```

A Model of Error Management for Financial Systems

Hans Ewetz and Niloufar Sharif

Clearstream Services, Architecture Section - Luxembourg
{Hewetz.cs, Nsharif.cs}@clearstream.com

Abstract. This paper presents an overview of the lessons learned about error management during implementation of a large financial system at Clearstream International, part of Deutsche Börse Group. Proper error management across the software comprising a financial system is a crucial ingredient in ensuring integrity of assets stored as data in a persistent media. As an experience report this paper addresses issues related to error management encountered during implementation of a large financial system that involved more than a hundred developers. As part of this paper we present a model of errors, notifications and rule failures.

Keywords: Atomicity, Error, Exception, Business.

1 Introduction

Definition of what constitutes an error is often not formally defined in software projects. It is common that the focus of error management is put on *what* to do in case of an error. However, little effort is usually put on defining *what* constitutes an error. As a consequence, management of conditions that may or may not represent errors evolves in an ad-hoc manner during development. We have seen that failure to have a clear understanding of what constitutes an error and how errors should be managed before development starts, leads to a range of problems. At best, unnecessary and duplicate error management logic is implemented across the software system. At worst, transaction management is compromised with disastrous effects on data stored in persistent media. Since modern financial systems represent assets as data stored in persistent media, it is imperative that error management is performed in a consistent and well understood way in order to ensure data integrity.

In this paper we present a simple model of errors, failures and notifications in the context of implementation of a mission critical financial system. The model is implemented in a large financial system that is currently running in production at Clearstream International in Luxembourg. As an integral part of Deutsche Börse, Clearstream offers settlement and custody services to more than 2500 customers world-wide, covering over 150,000 domestic and internationally traded bonds and equities. Clearstream's core business ensures that cash and securities are promptly and effectively delivered between parties, and that customers are always notified of the rights and obligations attached to the securities they keep under our custody. Error management in the software is of utmost importance since the amounts processed in

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 86–95, 2004.

the settlement system reaches over 80 billion dollars per 24 hours. Failure to ensure integrity in financial data will have catastrophic consequences with high probability of global impact on the world's financial markets.

This paper is structured as follows: in the next section we present the motivation of our work. Then we give an overview of the error, failure and notification model we have developed. Finally, we identify some issues and lessons learned from our work.

2 The Business Case

As mentioned in the introduction, a modern financial system stores assets as data in a persistent media. Therefore, a fundamental requirement on all software that accesses persistent data is to ensure that the software is written in a manner so that data integrity is preserved. Typically this is done as a combination of two techniques. First, a number of business activities that must be executed as part of a business service are controlled through a business process. The business process ensures that all activities are executed in the correct order. Second, one or more activities are grouped into a technical transaction. The transaction either has a permanent effect on all data that has been processed or has no effect at all. Here we will look at error management in the context of technical transactions. In this section we will raise a number of issues that will be clarified and resolved in the next section.

2.1 Transactions and Data Integrity

When, at runtime, a condition occurs that invalidates the work currently in progress, the software must back out all changes done before terminating or before continuing the next task. Backing out changes done to data in persistent storage is done with the help of a transaction. At the code level, transaction management is done as follows:

```
Start transaction

Execute business logic. Create notification if a
condition that invalidates current work occurs

Commit transaction if work valid, else rollback
transaction
```

One of the problems that must be solved at the code level is the implementation of notifications and propagation of notifications. Another problem is related to determining what constitutes a condition that invalidates work that has been done.

2.2 Notifications at the Code Level

A notification is an abstract concept that can be simulated at the language level in multiple ways. Four different ways are commonly used:

1. Error return codes from functions
2. Flagging through global variables
3. Use of an exception mechanism
4. Jumps (long jumps) over the call stack to specific marked location in the code

For reasons familiar to most developers the choice of using exceptions is usually the preferred alternative. As we will demonstrate in the next section, a fifth alternative to propagation of notifications is to terminate execution of the software. In fact, we will later demonstrate that this is sometimes the *only* viable option.

2.3 Conditions That Invalidate Work

There are two types of conditions that invalidate work that has been started within a transaction. First, technical errors over which we have no control must generate a notification that invalidates work currently being done. Second, conditions that are not caused by the technical environment, but rather by design decisions, can also invalidate work in progress. We call such conditions *rule failures*.

Consider a case where part of the business logic is implemented using a rule failure. When one account sells securities to a second account that pays for the securities with cash, the following logic must be executed within one transaction:

```
Debit securities from account 1

Credit securities to account 2

Debit cash from account 2

Credit cash to account 1
```

Now assume there is a business rule stating that an account cannot go negative. We can perform the logic above by blindly executing the debits and credits. We are then relying on that the business rule will generate a notification if an account does not cover an amount debited. This type of notification is not an error since we are only using the notification mechanism to implement part of a business rule. It is clear that such a notification can be avoided by first locking all affected data in the persist media and checking that involved accounts have the necessary funds before executing debits and credits.

An interesting observation that emphasizes the importance of correct notification management appears when analyzing what happens if only part of the transaction is performed before committing to the media. Consider the case when the transaction

commits even though the software fails in debiting account 2. If we translate the scenario to the corresponding real world, we see that we have in fact created money. Account 1 has received the cash payment but account 2 still holds the cash. Essentially we have cloned cash from account 2 and injected the newly created cash into the financial system through account 1. Clearly, in a production system, if such inconsistencies occur, they will be caught at checkpoints where business invariance is verified.

In the next section we will describe precisely what constitutes an error and what constitutes a rule failure.

3 A Model of Errors, Failures, and Notifications

In this section we formalize and resolve issues that were raised in the previous section. We then describe a model that takes those issues into account. Following this we show how the model is mapped into software.

Our terminology is currently not aligned with terminology from the dependability domain. Specifically, the notion of *failure* carries a different meaning in the domain of dependability than it does in our model.

3.1 Technical Errors

When designing and coding software we make a number of assumptions about the environment in which the software will execute. Assumptions are made about the hardware, the data that is processed, third party products and connectivity to databases, file systems and networks. If at runtime one or more of the assumptions no longer hold, the software is executing in an environment for which it was not designed to execute in. An assumption that does not hold at runtime is called an *error*. When an error occurs, the work in progress must be invalidated and the application terminated.

3.2 Managed and Unmanaged Errors

A less rigid view of errors separates them into two groups.

The first group contains unmanaged errors as described in the previous section. That is, an unmanaged error is the detection at runtime of a condition that invalidates one or more assumptions that we made about the execution environment.

The second group contains managed errors that correspond to problems in the execution environment that we predicted at design time. For example, the loss of a database connection is a common problem in many execution environments. If, at design time we design the software in such a way that it will try to reconnect in case of a lost database connection there is no need to terminate the application in case of

connection loss. Instead, the code can be designed to reconnect and continue execution. Notice that this would not be possible if, at design time, the assumption was made that the database connection was always active.

3.3 Rule Failures

As mentioned in the first section, a rule failure is a condition that invalidates the work that has been done in a transaction but is not caused by an error in the execution environment.

There are two types of rule failures. First, *business rule failures* are conditions that can occur when executing a business rule. A business rule failure is part of the normal execution of a business rule. In practice however, a rule failure should be a condition whose occurrence is exceptional. In the first section we showed an example where a business rule was implemented with the help of a business rule failure.

Second, *technical rule failures* are conditions that are technically related. An example of a technical rule failure is *optimistic locking failure*. Optimistic locking is a technique used to manage contention between multiple users of a resource. In the context of objects stored in a relational database, a counter is associated with a specific set of objects. When a user of the objects loads them into memory it also reads the current value of the counter. At a later time, when the objects are about to be updated in the database, the user reads the value of the counter, this time taking out a hard lock on the counter. If the initial counter value is equal to the last read counter value, the user increments the counter and updates the objects in the database and commits the transaction. If however the counters differ, it means that another user has updated the counter, and as a consequence also the objects while the user was processing the objects. In this case the user generates an optimistic lock failure which triggers a rollback of the current transaction. Typically a *user* is a software process, a thread or some other unit of execution.

3.4 Business Errors – What Are They?

When we first started to analyze error management in the context of financial software we saw that many software implementations contained the concept off *business error*. As we described earlier, a technical error is a problem in the execution environment. Is a business error then a problem in the business? Our current understanding of what constitutes a business error is this: a business error is a bug at the level of the business. For example, inconsistencies between two or more business rules constitute a business error. A business error cannot in general be detected at runtime since it is not possible to distinguish between technical errors and bugs in the requirements i.e. the business that the software is based on. Our conclusion is that business errors are not part of the software domain. They are however part of the business domain and should be handled by business analysts.

3.5 Type of Notifications

We classify different types of notifications based on the type of conditions that generate them. Since business errors are not part of the software domain, only four types of conditions will create notifications at the software level:

1. Unmanaged errors
2. Managed errors
3. Business rule failures
4. Technical rule failures

Only the first type of condition, an unmanaged error, invalidates the execution environment. The only possible action that can be taken when an unmanaged error occurs is to immediately terminate the application. It is important to acknowledge that this is true whether the application is an online application or a batch application. To terminate an online application is a hard pill to swallow, but the alternative is to continue to execute in an environment for which the software was never designed to execute in. We call a notification caused by an unmanaged error a *fatal notification*.

The last three types of conditions do not invalidate the execution environment. The only outcome that must be guaranteed by a notification caused by one of the three conditions is that the work done in the current transaction is invalidated. We will describe different ways of mapping such notifications to language constructs later in this paper. The notification generated by a managed error is called an *error notification*. Notifications generated by business and technical rule failures are called *business notification* and *technical notification*.

3.6 Putting It All Together

Here we summarize the discussion above in the form of two conceptual models showing relationships between errors, rule failures and notifications. The first diagram shows the structural relationship between different types of errors and the dynamic relationship between errors and notifications. Notice that business errors do not generate notifications:

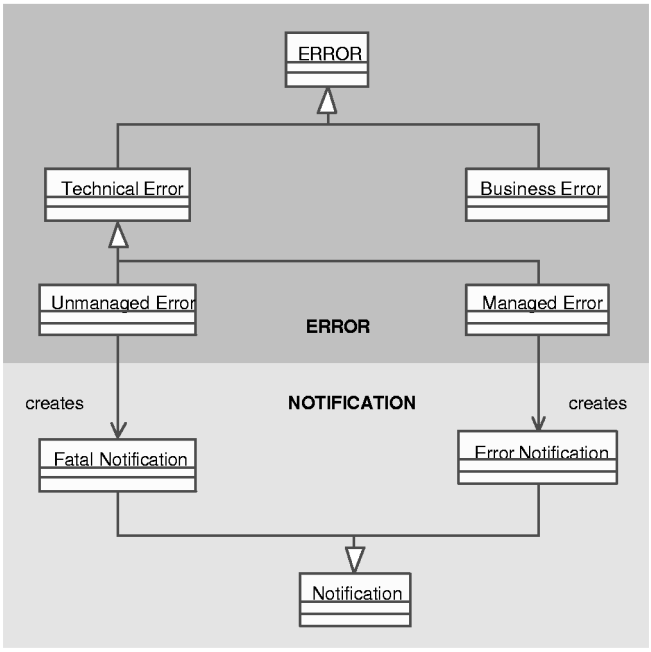


Fig. 1. Relationship between errors and notifications

The second diagram shows the structural relationship between rule failures together with the dynamic relationship between rule failures and notifications:

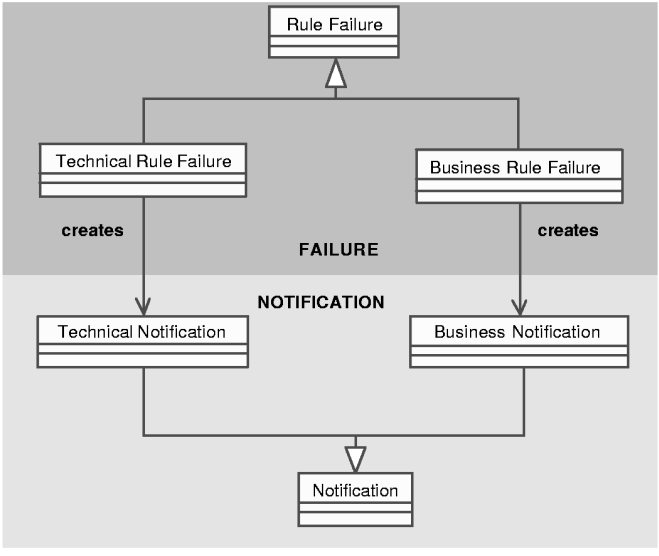


Fig. 2. Relationship between rule failures and notifications

3.7 Mapping of Model to Implementation

The target language in our development environment is C++ executing under UNIX. With C++ as the target language it is natural to map all notifications to C++ exceptions excluding *fatal notifications that* map to some mechanism that immediately terminates the applications. The C++ exceptions were implemented as part of a core technical infrastructure. The infrastructure also implements a framework that rolls back the transaction if and only if an exception is caught. In order to enforce rollback of the current transaction in the case of a notification, a coding rule has been put in place: code outside the technical infrastructure can only catch an exception if the caught exception is re-thrown. The coding rule together with the infrastructure support ensures that in the case of a non-fatal notification the current transaction is rolled back.

In the case of a fatal notification the choice of implementation is simple: the application must be terminated immediately. This is the only safe way to handle an unmanaged error since the occurrence of such an error indicates that the execution environment is corrupt. Termination of the application together with the database configured to rollback a transaction when the database connection is lost prevents corruption of data stored in the database.

4 Lessons Learned

In this section we discuss the experience we have had as members of the architecture team in a large-scale financial software project.

4.1 A Simple Lesson

The conclusions we have drawn from our experience can be summarized in two points:

1. A project as a whole must have a consistent view of what constitutes errors and management of errors
2. Error management and the management of the transaction boundaries must be encapsulated as much as possible in a framework

4.2 Consistent View of Errors and Management of Errors

One of the first lessons learned was the importance of uniformity of error management. Two important factors were identified as pivotal issues in order to ensure consistency in handling of errors:

1. Consistent view of what constitutes an error
2. Consistency in the actions taken when an error is detected

We experienced first hand that failure in ensuring the validity of these two points across all development teams and among developers within a team, resulted in inconsistent generation of errors and actions taken in case of an error. As a consequence, conditions that did not warrant rollback of the active transaction did in fact do exactly this in various parts of the system. The opposite was also true; conditions that should rollback a transaction did not always do so.

In order to ensure a consistent view of errors and management of errors a model must be developed and disseminated to all participants in a project. Here the model we described in the previous section provides support. The model is simple, intuitive and does not require a high degree of technical knowledge.

In the scope of consistent management of errors, it is important to realize that developers should not have to be creative when taking decisions about issues related to management of errors. In fact, it is harmful for the project as a whole to allow individual developers to make decisions about how to manage errors. Once a model of errors and management of errors has been put in place, there should be no room for creative solutions to problems related to error management.

4.3 Error Management and Frameworks

Roughly half of our code is in one way or another involved in management of errors. Therefore, it is crucial to support as much as possible any constructs and logic related to error management through an infrastructure that can be used by all developers in a project. The project, in which we have been involved, includes an infrastructure known as the *Technical Infrastructure*. The infrastructure supports error management as described below.

The infrastructure controls the transaction boundaries by wrapping business logic coded by developers in a *try-catch* block. In the try-catch block the transaction is rolled back in case of an exception and committed otherwise. Thus, a developer that codes business logic does not have direct control over *if* a transaction is committed. The infrastructure is used together with the following coding rule: an exception can be caught if and only if the same or a different exception is re-thrown. Unfortunately, because of limitations in the C++ language, this rule cannot be encapsulated in an infrastructure or a framework and must therefore remain as part of our coding standard.

The technical infrastructure supplies a set of exceptions that are available to developers. The constructor of an exception automatically generates a stack trace that is logged to a low level logging service. Currently, as a legacy requirement business exceptions intended to be used in conjunction with business errors is part of the suite of exceptions. However, except for reasons related to legacy, a business exception must not be used in code implementing business logic. As discussed in the previous section a business error is a bug at the level of the business and is not relevant at the software level.

4.4 Application Behavior and Project Standards

In our project we have seen that by encoding error management facilities in an infrastructure together with coding standards that are based on a model of errors and error management, it is possible to ensure that applications display certain characteristics that are desirable in our project. For example, we want all applications to ensure that a transaction is rolled back in case of a technical error. A developer can ensure this by following three simple rules:

1. Do not catch exceptions unless the exception is re-thrown
2. Code business logic and let the infrastructure manage the transaction
3. Throw an exception or exit the application if an error is detected. An error is defined in accordance with a model of errors and error management

5 Conclusion

In order to protect assets stored as data in a persist media a model of errors and management of errors should be developed. Doing so helps to ensure consistent and correct error management throughout the software comprising a financial system.

Our experience indicates that both the conceptual model of errors and management of errors together with the infrastructure supporting error management *must* be developed before business logic is implemented. We have seen that it is unlikely that developers will go back and retrofit their code to match a new or a modified support infrastructure.

In our environment we have gained tremendous benefits from enforcing certain project wide properties through an infrastructure together with coding standards. Here both the infrastructure and the coding standards are based on a model of errors and error management. The result is high confidence in correct handling of errors in the context of transaction management.

References

[K01] D.Kalev, The ANSI/ISO C++ Professional Programmer's Handbook, Indiana 2001.
[R01] A.Romanovsky, Exception Handling in Component-Based System Development.
[R01] A. Romanovsky, J. Kienzle, Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems, (2000).
[A00] A. Avizienis, J.C: Laprie, B. Randell, Fundamental Concepts of Dependability.
[J01] M.Jackson, Problem Frames, Analyzing and structuring software development problems, USA 2001.

Software Model Engineering and Reuse with the Evolution and Validation Environment

Jörn Guy Süß, Andreas Leicher, and Fadi Chabarek

Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
{jgsuess|aleicher|fadi}@cs.tu-berlin.de

1 Introduction

Reuse in software construction has been a core aspect of software engineering since the term was coined in late 70's [2]. Strategies for reuse like high level languages, structuring and encapsulation have made valuable contributions. Today, the software industry produces components as reusable parts, which deliver advantages like faster time-to-market, cost reduction, better maintainability, configurability etc. While the reuse of code and components improves, one important contributing factor of the engineering process remains out of scope: Although design reuse is more effective and beneficial than code reuse [1] models of software systems still are not assets of the software process. As a result design reuse does not impact the actual software engineering process. The impediments and successes of bringing models to the heart of the software process are reflected in the history of the Unified Modeling Language (UML). We will retell this story below to pick up the motivation for our own project, the Evolution and Validation Environment (EVE), which has the potential to globally reuse and share know-how expressed in models and model services, independent of and complementary to existing modeling tools. We give an overview of EVE's architecture and present an example of its application. Finally, we delve into the details of an EVE core component, the OCL profile validator service (MOCL), which is used to check UML models for compliance with profiles.

Scientific UML workbenches and environments have evolved for a long time ([4,8,12,11]), but all these approaches where meant for local operation and are based on propietary interchange and repository formats. The same problem exists for OCL tools, where the lack of a defined interface still exists and leads to low distribution of well-architected solutions like [3] and [5].

2 History of Modelling Challenges

Yesterday: Models as a Language for Engineers. In the late 80's a large number of different methodologies and notations existed, which fragmented the model user community and pitted the factions against each other in the 'object wars'. Industry convergence, brought on by the three main contenders, led from many similar modeling languages with minor differences in their expressive power to

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 96–105, 2004.

UML as a broadly-supported modeling language intended for general purpose use. Created as a collection of diagrammatic representation techniques and a set of axiomatic model elements (use case, class, constraint, component etc.) semantically contented by precise English text, UML was later supplemented by a static object-oriented meta-model and declarative constraints. Since its introduction in 1999, UML has arguably reached the first of its primary design goals, which is '*to provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.*' . Consequently, the current popularity of UML is mainly based on the positive properties of models in human communication, namely:

Abstraction. A suitable model abstracts from aspects the modeler does not currently think about or wishes to communicate. Therefore, good design is equivalent to understandable concepts and leads to an improved system's overview and insight. Models comprise constraints, behavior, structure, as well as non-functional properties on an abstract level. Developers do not need to deal with certain details of underlying technologies.

Comprehension. Formal or semi-formal models, like those formulated in UML, follow a common language definition, which is comparatively easy to understand. Therefore, people from different fields can understand design concepts in a common way and are able to discuss alternatives and consequences.

Today: Models as a Format for Tools and Basic Services. Meanwhile, UML has taken hold in many specialized areas of appilication, ranging from modelling the interaction of bioactive substances in cells over workflow design and management to the design of transaction-intensive business systems based on distribution platforms like CORBA and Java EJB. All these different application domains can mutually profit from each others models, but also need tools specific to their domain to efficiently work with their models. Both aspects hinge on the exchangeability of the model as data.

At first UML did not provide a standardized storage format, which led to a plethora of proprietary formats. Models could not be exchanged between different tools and therefore clients were tied to a specific vendor. This has created a barrier for reuse because models and services for the same application domain have to be reinvented in a costly and time-consuming way for different tools. Thus, the community could not reach a critical mass. This lack has principically been overcome through the appearance of XML, and the subsequent definition of the eXtensible Metainformation Interchange standard (XMI), which from a software practitioner's point of view appears as a file format for models expressed in UML.

Currently, we see tools move in this direction of cooperation. The tool market shows a small but ever increasing number of tools capable of working with XMI. To accelerate this movement, it is important to provide efficient export and import plugs for as many tools as possible. Consequently, one part of our architecture is dedicated to this end.

Another problem of today's modeling tools is the lack of basic functionalities. There are a number of functionalities that should be generally available regardless of the tool used:

Constraint Monitoring. Using checks against a formal semantic specification a modeller can discover design errors early during modeling. She can check types for consistency and validate assertions about behavior.

Model Evolution. The UseCase-to-Interface Method by Jansson [10] which we describe as part of our system walk-through on page 101 is an example of Model Evolution. In object-oriented modeling, methods often encompass dependency rules, which describe modification steps. Usually the existence of certain parts of the model in an earlier phase implies the existences of dependent parts in a later phase.

Code Synchronization. Models can be kept synchronized with code through roundtrip engineering. Consequently, the implementation stays consistent with the design and the model retains its value.

Because of their inherent complexity current gui-based modeling tools usually include only one or two of these functionalities, often in a limited form such as code generation instead of code synchronization. A few expensive tools provide the spectrum, but the implementations regularly lack quality. Because these functionalities are implemented against the propietary models of the GUI tools, they cannot be externalized and reused as independent services. Also, the regular use of models within the process is hampered, because merely viewing a model requires a cumbersome fat client.

The Future: Modeling Tools as Services and Thin Clients. Fortunately current technology enables us to overcome these technical problems. Fault tolerant and light-weight distribution platforms provide the deployment of services in arbitrary locations. Standardized storage formats provide a basis of model exchange and allow processing of models created by arbitrary tools. Ubiquitous browser technology and open graphics standards allow universal presentation.

Using such technology, the effort involved in handling repetitive processes on models and sharing models between people can be minimized. But what about the effort of *creating* a model for a specific domain? UML was created to be universal and therefore its internal constraints had to be lax to avoid the exclusion of any specific domain. But to use models as effective input for a domain process, closely-fitting domain constraints and documentation must be formulated.

Even if constraints and documentation were available in manner to be just copied-and-pasted into a model, this would still involve an enormous amount of manual work. Also, the constraints expressed do not really belong into the model proper. They clutter the model with background knowledge which spans all models and projects in a domain. What is needed is a mechanism which allows modelers to 'factor out' domain knowledge to communicate it seperately and to 'factor in' domain knowledge to apply it effeciently. UML Profiles solve this

problem by creating a set of templates for the domain, which can be added to the model and connected to its elements by tagging them with domain stereotypes.

In this way Profiles, which are part of the UML standard, enable the modeling community to exchange knowledge on domain semantics and methods. They tailor the UML to the domain and also offer a fixed document template, which contains three parts: A meta-model with stereotypes, icons and tagged values, a precise English language description of the semantics, and constraints which lay down the well-formedness rules in the Object Constraint Language (OCL).

Because previously no tool or environment could provide us with the capabilities outlined above, we began the development of EVE in July 2002 (figure 1). We designed EVE detached from modeling tools to enable a modelling community to collaborate, to promote good modeling methods and services into industrial practice and to gather valuable feedback for the research community. EVE features:

Model Services. Services are moved out of the fat-client tool and embedded in a sharing mechanism, so the whole community can benefit from a local development. Services developed in the scientific community come into the scope of industrial use, as practitioners can locate a domain expert in science and then apply her experimental domain services.

Model Viewers. Models and service feedback can be displayed in a browser, so the peruse of model and process data increases. 'Looking it up in the model' becomes as common as looking at source code within a Sourceforge CVS branch.

Domain-level Reuse. Profiles can be added to a model and act like a construction template. They at once introduce to the model the constructs and constrains of the domain in a form which people can understand and services can evaluate.

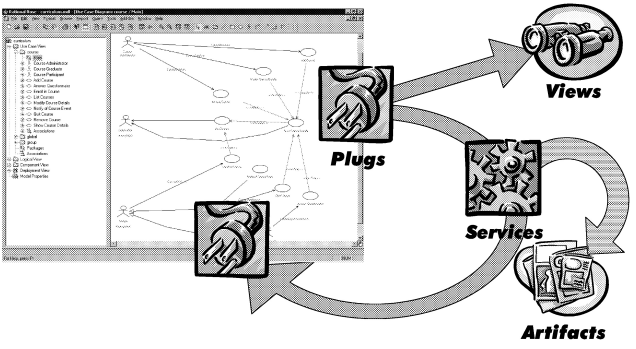


Fig. 1. EVE's Concept: Proprietary plugs give access to the model, external services work on it and may generate artifacts.

3 Architecture of EVE

EVE's system design is based on 'separation of concerns': It separates models and services from the modeling tool itself. Figure 1 shows the basic concepts of the framework. Models residing in arbitrary (UML) modeling tools are extracted and transformed into correct XMI representations. Subsequently, these models can be processed further (by services) or displayed through individual views. The framework consists of the following basic components:

Models. EVE is able to handle all model instances that are based on a MOF compliant metamodel, because it is based on the Netbeans MDR metadata repository.

Artifacts. The term artifact refers to all data that does *not* represent a (UML) model instance. Services may produce or consume arbitrary artifacts such as source code, reports etc. However, artifacts cannot be passed around in the primary framework. If services intend to cooperate on the content of the artifact, than the artifact must be *modeled*, i.e. expressed as a model instance and embedded into the model that is passed around. Otherwise the services must define a private, proprietary communication channel.

Plugs. Modeling tools handle and store model instances in different proprietary formats. In order to use these model instances, they have to be translated into proper XMI[1]. Plugs perform this translation and therefore provide the connection between modeling tools and the framework.

Services. Services define functionality that can be applied to model instances. Typical examples of model services are validation, evolution and artifact synchronization.

Views. Views are components for immediate visual presentation of models and system or service feedback. A View can be a HTML page, a RDF feedback, an error report, or a full-fledged GUI. Views are also used to monitor the status of the EVE network, as users must be able to determine which services are available.

We have based the EVE framework on two architectural styles: We used the strict 'layered system' style to encapsulate different access requirements of increasing complexity within different technologies: local access as JavaBeans components, LAN access as Jini services, Internet access as WebServices and JSP pages. To provide sequential transactional operation of services, we used the 'pipe-and-filter' style. This combination leads to the following basic components of the framework:

Frame. The `Frame` component encapsulates EVE services and provides the service integration into the framework. The `Frame` implements a coordinator for sequential operation (chaining) of services, allows service instantiation

[1] Most modeling tools support an XMI dialect that more or less complies with the XMI Specification. Thus, model instances have to be cleansed in order to become usable.

through dynamic class loading and provides transactional behavior to allow service composition. This component as well as the other basic framework components is realized as a JavaBean.

Share. To offer EVE services on a LAN, the `Share` component realizes a Jini wrapper: It encapsulates the `Frame` component and shares it on a Jini network as a service. A `Share` registers with a Jini lookup service and can be accessed by `NetFrame`s, as well as any other Jini client.

NetFrame. The `NetFrame` component executes an EVE service remotely. It locates the EVE service requested using service name resolving and executes the service using a defined interface called 'IEveService'.

The following sections present an evaluation service and an OCL profile validator service. These services are concrete 'filters' referring to a pipe-and-filter architecture. Thus, it is allowed to combine filters in arbitrary sequences (service chains) with the EVE framework. The MOCL service, for example, is normally used twice within such a service chain (see section 5).

4 Example Application

To show how EVE actually performs in an environment, we present a walkthrough in a realistic scenario employing an EVE Evolution Service: The UseCase-to-Interface Method by Jansson [10] involves adding an interface for each association between an actor and a use case. If we do not want to generate all these interfaces by hand, we can define a rule, which describes how associations between actors and uses cases map to interface elements. A specific evolution service (ES) can encapsulate an algorithm that executes this rule.

The example in figure 2 traces a single execution of ES in the framework. A developer initiates a service execution in the ArgoUML modeling tool. The service is subsequently invoked remotely over the Jini network:

1. The model instance is extracted from the modeling tool via a Plug.
2. The Plug marshals the model instance into an XMI stream.
3. A network connection is established and ES is looked up using a service listener. The service listener communicates with all available lookup services. `NetFrame` components transparently handle these complex tasks and forward the service's invocation to the server implementation.
4. ES returns a modified model instance. Additionally, it provides feedback about the service execution (not shown).
5. Finally, the model instance is reintegrated into the model inside the tool. If ES only deletes model elements, this is relatively easy to accomplish. Integrating new model elements is challenging because XMI only contains information about model structure but not about diagrams.

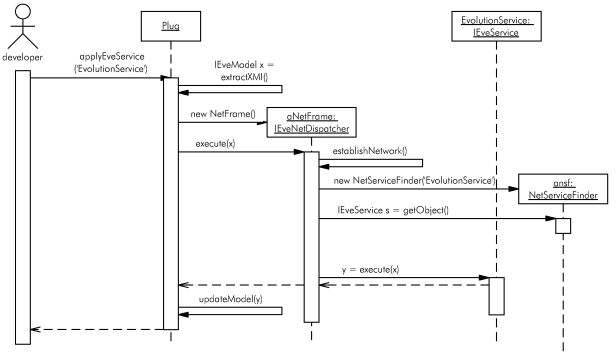


Fig. 2. Simplified Evolution Service execution

5 The OCL Profile Validator

The previous example tacitly assumes that execution of the ES somehow preserves internal integrity of the model. How can EVE guarantee this property? Static constraints of the metamodel, i.e. associations between classes like 'UseCase', 'Interface', and their cardinalities are preserved through the repository itself. Netbeans Metadata Repository, which implements the Java Metadata Interface (JMI), can only import and export data compliant with a MOF structure. Well-formedness rules however escape the check, because the repository does not interpret the OCL constraints. For plain-vanilla UML models, this omission does not pose a great problem. But if Profiles are added, the situation changes. All domain constraints are expressed as well-formedness rules, because with the current version of UML it is not possible to define extensions to the metamodel, which introduce new static integrity constraints. So to reliably work with models based on Profiles, a metamodel-based OCL-validator is necessary. This validator checks OCL constraints specified by a profile against a model instance. Because, these profile constraints are written against the metamodel, normal OCL tools can't accomplish this task. Thus, EVE contains a service, called the MOF OCL validator (MOCL) that handles constraints against the metamodel. This service is used twice in every model transformation cycle, as shown in figure 3. To prevalidate the model, to ensure only consistent models are handed to the ES, and to post-validate the model, to ensure the ES has worked correctly.

5.1 Design Challenges

Although writing the design of interpreters and parsers is a well-established area, a number of less obvious issues made the development of MOCL difficult. Below, we describe how the following aspects have been handled in our implementation.

Variability of OCL. Since OCL was integrated into the UML specification, four versions of the language have been published. Since the language or

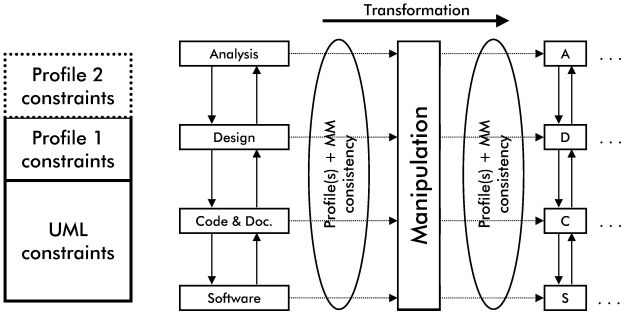


Fig. 3. Validation-Transformation Cycle

version of constraints is not standardized by the UML, every language version could possibly appear in every model.

Variability of Metamodels. The class structure of the UML metamodel, especially that of the extension mechanism has changed with every revision of the UML standard. The interpreter needs to navigate in and extract Profile information from models.

5.2 Creating an Adaptable Parser

The grammar of OCL versions is not very amenable to the use of parser generators. Although the specification in different versions claims to use a notation based on JavaCC, it carries a number of limitations: It is not context-free, does not define a start-symbol, does not include literals for Boolean values, precedence rules for numerical comparison are not expressed.

A number of these difficulties have already surfaced in the pioneering implementation of the OCL parser and compiler described in [9]. There, as in our implementation, variability is retained through the application of SableCC [6], a parser based on the application of the Visitor pattern [7], which allows traversal code to be maintained independently of the grammar specification. However, since true polymorphic visitors are difficult to realize [13]and composition of visitors is not trivial [14], changes in the grammar propagate to the implementation, albeit in a limited fashion. Therefore, MOCL implements OCL 1.5, which is the maturest released version of OCL and also reasonably compatible. Prospectively, the component will be updated in the coming year to work with OCL 1.6, which should resolve many of the problematic issues.

5.3 Adapting to Different Metamodels

OCL statements can only be executed in the context of an object-oriented model, so access to models must be provided. The challenge in the design of the context checker lies in the different interpretation of connections between OCL Basic

Types and the Model Types. To resolve this MOCL defines a minimal interface to metamodels, which consists of the following elements:

ModelFacade is used to access model elements and enumeration by their qualified name. A ModelTypeDescriptor describes model elements returned by it.

ModelTypeDescriptor describes a model element or enumeration. It provides information about the name, the predefined OCL super-type, the position in the type hierarchy, and defined properties. The name identifies the element. The OCL supertype defines the available primitive functions. Conformance with other elements is derived from the type hierarchy position.

AssociationEnds must be treated explicitly because evaluation navigation results in different types. Navigating an association with multiplicity one or zero results in a Set containing one target element. If it is adorned with "ordered" result, it results in a Sequence. The interface consequently returns the type, upper multiplicity, and adornments of the association.

This interface is implemented by a generic MOF-bridge, which delegates calls to a JMI compliant repository. Thus the validator can access any MOF compliant model. During the course of the implementation, a number of inconsistencies, idiosyncrasies and loopholes has come to light, which in themselves merrit a second paper. As the tip of the iceberg a number of the well-formedness rules defined in the UML standard are actually mistyped or cannot be executed. Hopefully the availability of MOCL will contribute to the improvement of those aspects of quality, as well as to the productivity of software engineers.

6 Conclusion and Outlook

Today, proprietary modeling tools dominate the market. Developers are tied to tools used in their company. Services like code generation or reverse engineering are only available in a subset of these tools. Currently, the lack of open model exchange is a common drawback in the application of these tools: Models cannot be used as an abstract basis of system development, cannot be exchanged in a market, or handled with standardized tools and services. Existing services in different tools cannot be shared, as they are not portable and cannot be reused. If iterative services like the ES are moved out of the CASE tools, the interactive user tools are free to take on a new character. In addition to their traditional role as the static drawing board of the system architect, they grow into interactive specification aids for the dynamic properties of models. They can help the modeler by generalizing scenarios into behavior or by playing through the state space of the model.

The EVE framework pursues the objectives of model reuse and separation of modeling tools and modeling services in order to shift the focus of system development to the design level. MOCL as a core component enables validation of arbitrary OCL constraints on MOF models, particularly those using UML profiles. EVE ist to be extended to make use of OCL 1.6 and due for stable

release at the begining of next year. The EVE framework has been implemented at the Technical University of Berlin. A prototype is available. The MOCL service is currently developed and will be available in February.

References

1. T. J. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions. In T. J. Biggerstaff and C. Richter, editors, *Software Reusability*, volume I — Concepts and Models, chapter 1, pages 1–17. acm press, 1989.
2. Jr. Frederic P. Brooks. *The Mythical Man Month.* Addison-Wesley, anniversary edition, 1995. ISBN: 0-201-83595-9.
3. Dan Chiorean. Using OCL beyond specifications. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, volume P-7 of *LNI*, pages 57–68. German Informatics Society, 2001.
4. Tony Clark, Andy Evans, and Stuart Kent. The specification of a reference implementation for the unified modeling language. *L'OBJET*, 7(1), 2001.
5. Frank Finger. Design and implementation of a modular OCL compiler. Diplomarbeit, Technische Universität Dresden, Fakultät für Informatik, March 2000.
6. E. Gagnon and L. Hendren. Sablecc – an object-oriented compiler framework, 1998.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
8. Wai Ming Ho, Jean-Marc Jquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT: An extendible UML transformation framework. In *Automated Software Engineering*, pages 275–278, 1999.
9. Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, 44(1):51–69, July 2002.
10. Par Jansson. *Use Case Analysis with Rational Rose.* Rational Software Corp, Santa Clara CA, 1995.
11. J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies. Model processing tools in UML. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 819–820, Los Alamitos, California, May12–19 2001. IEEE Computer Society.
12. Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, 18, 1999.
13. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.
14. Joost Visser. Visitor combination and traversal control. In *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 270–282, 2001.

Distributed Composite Objects: A New Object Model for Cooperative Applications

Guray Yilmaz[1] and Nadia Erdogan[2]

[1] Turkish Air Force Academy, Computer Eng. Dept., Yeşilyurt,
34149 İstanbul, Turkey
g.yilmaz@hho.edu.tr
[2] Istanbul Technical University, Electrical-Electronics Faculty,
Computer Engineering Dept., Ayazaga,
80626 İstanbul, Turkey
erdogan@cs.itu.edu.tr

Abstract. This paper introduces a new programming model for distributed systems, distributed composite objects (DCO), to meet efficient implementation, transparency, and performance demands of distributed applications with cooperating users connected through the internet. DCO model incorporates two basic concepts: composition and replication. It allows the representation of an object as a collection of sub-objects and enhances the object distribution concept by implementing replication at the sub-object level and only when demanded. DCOBE, a DCO-based programming environment, conceals implementation details of the DCO model behind its interface and provides basic mechanisms for object composition, distribution and replication of object state, consistency management, concurrency control and dynamic deployment of restructured objects.

1 Introduction

With the increasing use of Web technology for Internet and Intranet applications, distributed computing is being increasingly applied to large size computational problems, especially to collaborative applications. The shared-object model is an attractive approach to structuring distributed applications. In this paper, we propose a new object model, *distributed composite objects (DCO),* that extends the shared object paradigm to meet efficient implementation, transparency, fault tolerance and enhanced performance demands of distributed applications, cooperative applications in particular. We also present the software layer that supports the DCO model, a middleware between a distributed application and Java Virtual Machine, called ***D**istributed **C**omposite **O**bject **B**ased **E**nvironment (DCOBE).* DCOBE offers services that facilitate the development of internet-wide distributed applications based on the DCO model [9].

The DCO model incorporates two basic concepts. The first one is *composition*, by which an object is partitioned into *sub-objects (SO)* that together constitute a single *composite object (CO).* The second basic concept is *replication*. Replication extends the object concept to the distributed environment. Sub-objects of a composite object

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 106–115, 2004.

are replicated on different address spaces to ensure availability and quick access. Decomposition of an object into sub-objects reduces the granularity of replication. To a client, a DCO appears to be a local object. However, the distributed clients of a DCO are, in fact, each associated with local copies of one or more sub-objects and the set of replicated sub-objects distributed over multiple address spaces form a single distributed composite object.

Development of a distributed collaborative application on the internet is not an easy task because the designer has to handle issues of distribution, communication, naming, in addition to the problem at hand. DCOBE middleware facilitates the development and integration of distributed collaborative applications as it hides all implementation details behind its interface. In the following sections of the paper, we present a brief overview of the DCO model and DCOBE. Next, we describe a typical application that can benefit the model we propose: a real-time collaborative writing system. An evaluation of DCOBE performance and related work are described in the following sections.

2 Distributed Composite Object Model

The distributed composite object model allows applications to describe and to access shared data in terms of objects whose implementation details are embedded in several SOs. Each SO is an elementary object, with a centralized representation, or may itself be a composite object. Several SOs are grouped together in a container object and form a composite object.

The implementer of the CO distributes its state among multiple SOs and uses them to implement the features of the CO. The clients of the CO only see its interface, rather than the interfaces from the embedded SOs. Therefore, from the client's point of view, a CO is a single shared object, accessed over a well-defined interface. He is not aware of its internal composition and, hence, has no explicit access to the SOs that makeup its state. This restriction allows for dynamic adaptation of COs. The implementer may add new SOs to extend the design, remove existing ones or change their implementation without affecting the interface of the CO. Thus, dynamic adaptation of the object to changing conditions becomes possible, without effecting its users.

The proposed model relies on replication. SOs of a composite object are replicated on different address spaces to ensure availability and quick local access. A CO is first created on a single address space with its constituent SOs. When a client application on another address space invokes an operation on a CO which triggers a method of a particular SO, the state of that SO only, rather than that of the whole CO, is copied to the client environment. With this replication scheme, SOs are replicated dynamically on remote address spaces upon method invocation requests. The set of SOs replicated on a certain address space represents the CO on that site. Thus, the state of a CO is physically distributed over several address spaces. Active copies of parts, or whole, of a composite object can reside on multiple address spaces simultaneously. We call this conceptual representation over multiple address spaces a *distributed composite object* (DCO).

Fig.1. depicts a DCO that spreads over three address spaces. It is initially created on *Site2* with all of its sub-objects (SO1, SO2, and SO3), and is later replicated on

two other sites, with SO1 on *Site1*, and SO1 and SO3 on *Site3*. The three sites contribute to the representation of the DCO. The set of address spaces on which a DCO resides evolves dynamically as client applications start interactions on the target CO.

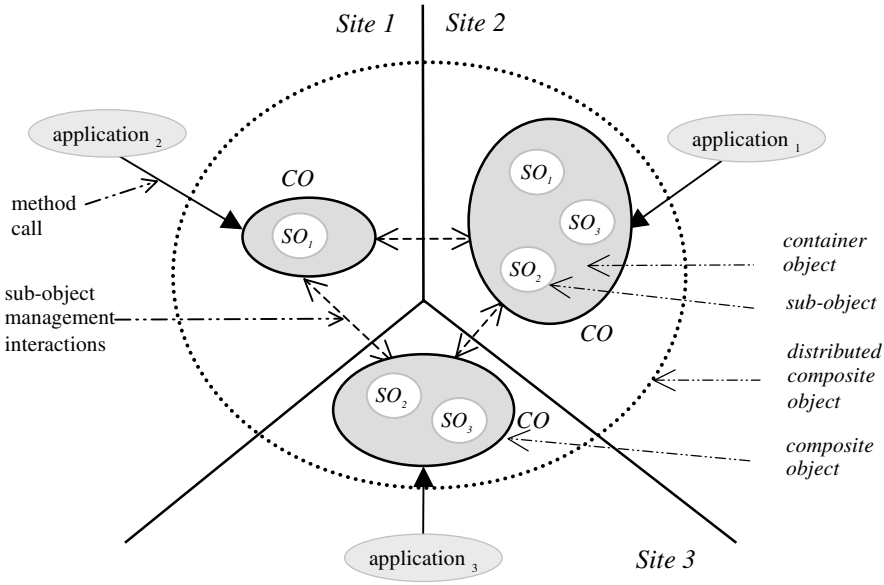


Fig. 1. A distributed composite object that spreads over three sites

The DCO model, with the support of the DCOBE middleware, conceals all implementation details behind its interface, as expected of the object-oriented programming paradigm. Clients of a DCO are isolated from issues dealing with distribution and replication of object state, the underlying communication technology, management of consistency of object state and management of concurrency control.

Management of Sub-objects: We have defined an enhanced object structure to deal with implementation issues and thus provide the object implementer with complete transparency of distribution, replication and consistency management. This new structure includes two intermediate objects, namely, a *connective object* and a *control object*, which are inserted between the container object and each target sub-object.

Connective and control objects cooperate to enable client invocations on DCOs. A connective object is responsible for client to object binding, which results in the placement of a valid replica of a SO in the caller's address space. A control object is a wrapper that controls accesses to its associated replica. It implements coherence protocols to ensure consistency of sub-object state. A client object invocation follows a path through these intermediate objects to reach the target sub-object after certain control actions.

The control object is located between the connective object and the local SO and exports the same interface as the SO. It receives both local and remote invocation requests and applies them on the local SO. Consistency problems arise among SO replicas on different address spaces when they want to modify object's state concurrently. The control object is responsible for the management of consistency of object state and concurrency control. It implements certain coherence and access synchronization protocols before actually executing the method call on the SO. The system uses *entry consistency* [1] for memory coherency.

There are two approaches in the synchronization of write accesses to objects so that no client reads old data once a write access has been completed on some replica: *write-update* and *write-invalidate* [5]. Write-update broadcasts the effects of all write accesses to all address spaces that hold replicas of the target object. In the write-invalidate scheme, on the other hand, an invalidation message is sent to all address spaces that hold a replica before doing an update. Clients ask for updates as they need them. DCO model implements both coherence protocols. The object implementer chooses the one which suits the requirements of his application the best and the control object is generated accordingly by the class generator. The implementer may also specify different coherence protocols for different SOs of a composite object for enhanced performance. The control object implements a method invocation in three main steps:

i) Get access permission*:* This step involves a set of actions, possibly including communication with remote control objects, to obtain access permission to the sub-object. The control object recognizes the type of the operation the method invocation involves, either a write operation that modifies the state of the object or a read operation that does not, and proceeds with this information. It is blocking in nature, and further activity is allowed after the placement of a valid sub-object copy in the local address space if one is not already present (a local implementation is not created before it receives its first call or the current replica may have been invalidated).

ii) Invoke method: This is the step when the method invocation on the local sub-object takes place. After receiving permission to access the target sub-object, the control object issues the call which it had received from the connective object.

iii) Complete invocation*:* This step completes the method invocation after issuing update requests for remote replicas on the valid list to meet the requirements of write-update protocol. After the call returns, the control object activates invocation requests that have blocked on the object. The classical multiple-reader/single-writer scheme is implemented, with waiting readers given priority over waiting writers after a write access completes and a waiting writer given priority over waiting readers after the last read access completes.

A class generator that has been developed in the context of this work is used to generate connective and control objects' classes automatically from interfaces of SOs. Hence, the SO is the only object that the implementer has to focus on. The others are generated automatically, according to the coherence protocol specified by the implementer.

3 DCOBE Middleware

A DCO-based programming environment, depicted in Fig. 2, hosts various numbers of applications dispersed on several nodes, interacting and collaborating on common goals and shared data. DCOBE conceals implementation details of the DCO model behind its interface, allowing users to concentrate merely on application logic rather than on issues dealing with activity on a distributed environment.

DCOBE provides the basic mechanisms for object composition, distribution and replication of object state, consistency management, concurrency control and dynamic deployment of restructured objects. It is a middleware architecture that is implemented on a network of heterogeneous computers, each capable of executing Java Virtual Machine. Its place in the software hierarchy is between a Java application and the JVM. DCOBE architecture consists of two main components that handle the core functionalities of the middleware: a system-wide coordinator (*DCOBE Coordinator–DC*) and a server component (*DCOBE Server-DS*) on each node which participates a DCO-based application in the distributed environment.

DC is the component that initializes the DCO based execution environment and coordinates the interaction of DCOBE Servers. It runs as a separate process, which is explicitly started at a predefined network address such that it may be accessible by all servers that participate the environment. It has a remotely accessible interface that allows distributed DS's to request services from it. When a DS is started, DC supervises a handshake protocol ensuring that each DS is initialized knowing the address of every other DS participating. Being unique makes DC very critical as it plays a major role in the system. In order to protect the system against failures, we have added a secondary DC as a backup unit to the DCOBE architecture.

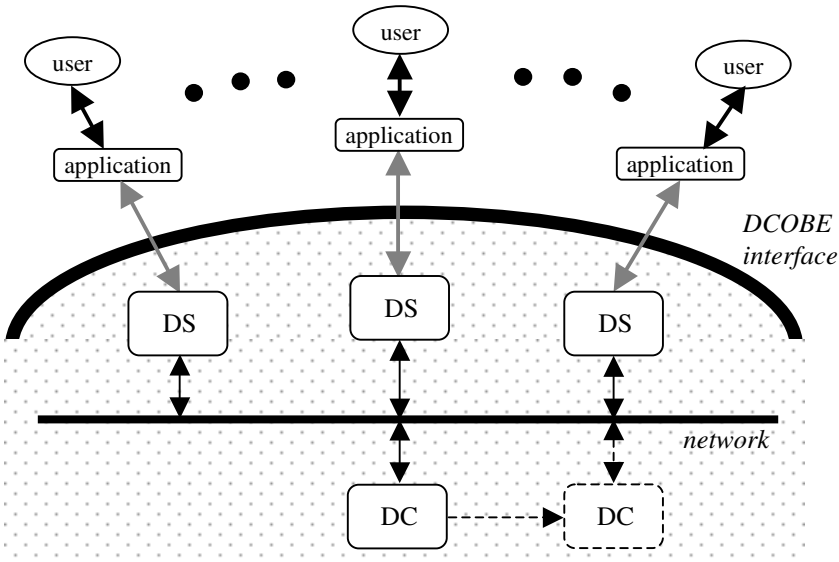


Fig. 2. Schematic view of DCOBE middleware

The main goal of DS is to provide execution support for DCO objects. A DS is actually instantiated within the context of each application and provides facilities that implement the DCO model. As a DS is integrated in each client application, the application can directly perform method calls as both are in the same address space. DSs on remote address spaces cooperate to process application requests and to ensure consistency of the replicated DCO state.

4 An Application: Collaborative Book Writing

Collaborative book writing on the internet is not a new approach. There are several academic and industrial studies on this issue [2, 6, 8]. Our goal is to show how the DCO model facilitates the design of the distributed application, reducing significantly the overall time for development by taking care of distribution, replication, consistency, concurrency and communication issues.

The application aims to develop a web-based collaborative environment that allows several physically dispersed people work together to produce a document, to view and to edit shared text at the same time. Collaborative writing involves periods of synchronous activity where the group works together the same time, and periods of asynchronous activity, where group members work at different times. The authors need to be able to read and update any displayed document content. They also require rapid feedback on their actions. The system has the following characteristics:

- *Low response time:* Response to local user actions is quick (as quick as a single-user editor) and the latency for remote user action is short (determined mainly by external communication latency).
- *Distributed environments:* Cooperating authors reside on different machines connected by different communication networks.
- *Unconstrained collaboration:* Multiple users may concurrently and freely edit any part of the text at any time.

4.1 Storage of Text Objects

In this application, the text of a book is a persistent document. It is represented by a single DCO, namely `book`, which is shared among authors. As shown in Fig. 3, `book` consists of a collection of related SOs, each representing a specific part of the book text: its contents, preface, index, references, and chapters denoted by sections and sub-sections. A chapter may include several sections, and these sections may further be divided into sub-sections.

The replicated architecture of the composite object model plays an important role in achieving good responsiveness and unconstrained collaboration. The shared sub-objects are replicated at the local storage of each participating author, so updates are first performed at the local address space immediately and then propagated to remote sites, according to the consistency protocol. Multiple authors are allowed to access any part of the shared text. A read type of method in the user interface results in the loading of the target sub-object into the local memory if it is not already present or the present copy has been invalidated meanwhile. Any number of updates may be performed on the local replica, and the new content is submitted with a write type of

method invocation. This action may be performed whenever desired, after a word has been changed, or after the author has been working several hours on the content. Since simultaneous updating operations on shared parts of the books by different authors may conflict, the consistency protocol (write-invalidate is chosen for this application) ensures consistency. While authors carry out these operations, they will not be aware of composite and distributed structure of the object they are working on.

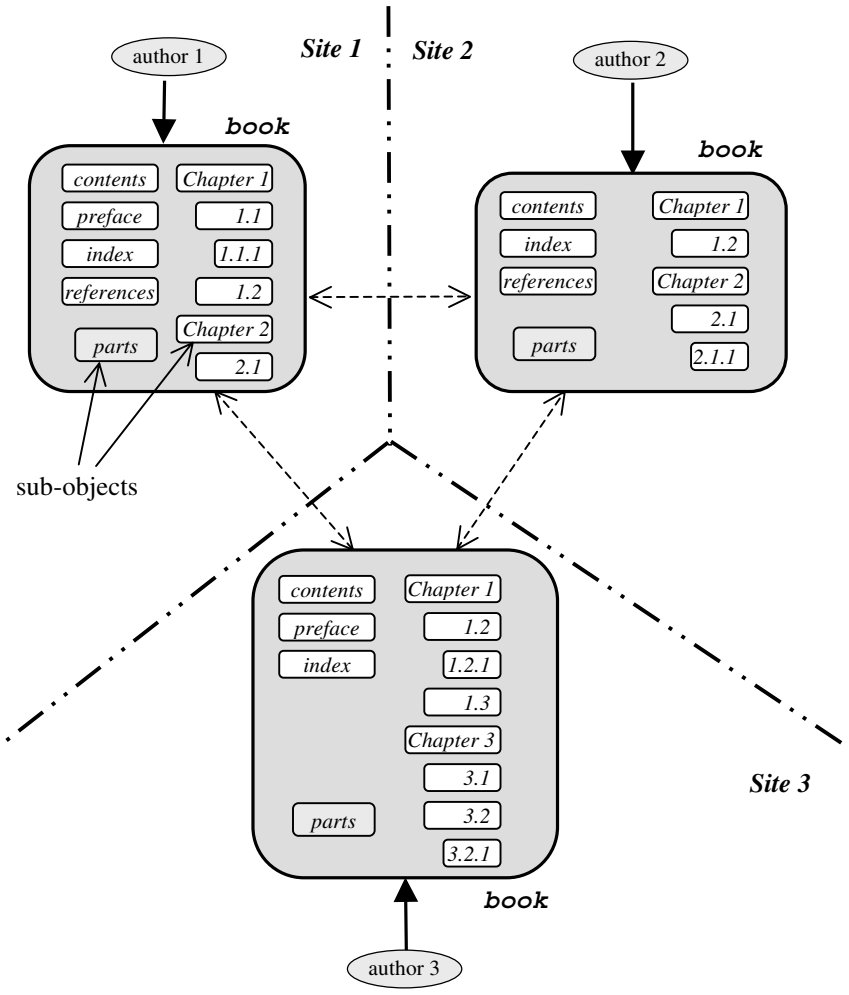


Fig. 3. Composite structure of ***book*** as accessed by three distinct authors

An interesting characteristic of the composite object `book` is that, initially at the start of the application, as no part of the book has been written yet, the set of sub-objects that will represent various parts of the book text is empty. As work progresses, the object `book` evolves, sub-object by sub-object, as new parts of the book text are added to the composite object. A special sub-object keeps track of the sub-objects

thus added to the composite object in a table, where it stores the name of the part, a string, with a reference to the connective object associated with it.

4.2 Author Interface

Authors access `book` via a GUI which contains methods that fall into three groups:

Content based methods: They involve the reading, writing, or editing of the content of a part of the book text. Presently, a simple class *editor* implements the functionality required, providing a text-based editing environment with filing facilities. As further work, we plan to integrate standard editing tools into the framework to provide authors with an enhanced working environment.

Attribute based methods: They involve adding, modifying, or retrieving the attributes of a part, such as other authors' opinions, last modification date, etc.

***Document based methods*:** They involve listing, adding, removing, renaming, or searching of parts or index entries. The contents sub-object is updated automatically if the invocation of a method requires such a modification.

5 Performance Evaluation

This section presents an evaluation of DCOBE middleware and its services. We measured the cost of basic operations to better understand system's behavior. The experiments were performed on 5 PC desktops based on Pentium III processors (1000 Mhz, 256 MB RAM), connected through a 10 Mbit Ethernet, running Windows 2000 and JDK 1.3.1. Measurements were repeated 10 times and the reported results are the arithmetic averages of these measurements. Table 1 presents the results.

Table 1. Measured costs of the basic operations in DCOBE

Operation		Time
Registering a DCO with a user defined name on DS and DC.		1.25 ms
Lookup and load operations for container and connective objects of a DCO, respectively, by a new client application.		3.15 ms
Loading of control object and sub-object onto the client site, on the first method invocation request on a connective object.		2.65 ms
Obtain access permission on a sub-object	(from local control object)	0.00002 ms
	(from remote control object)	0.75 ms
Method invocation	(if object is not cached)	0.0006 ms
	(if object is cached)	0.0001 ms
Update and invalidate a sub-object		
invalidate other sub-objects	(for i number of sub-objects)	i*0.75 ms
update other sub-objects	(for i number of sub-objects)	i*0.95 ms

In order to evaluate DCOBE performance, we compared the timing results of basic operations of DCOBE with those of Java RMI. We used the object instance we had used for DCOBE with measurements in RMI, as well. First, an RMI object was

initiated on a node. Then, remote method invocations were performed on that object from a different node. The results of the measurements are illustrated in Table 2.

Binding operation of the RMI object on a server registry and the lookup operation of that object's stub from the remote server and similar operations in DCOBE are executed only once. Measurements show that the durations of these operations are almost similar in both systems. With RMI, every method invocation is forwarded to a remote site. However, in DCOBE, an invoked object is replicated on the requesting site with the target sub-object(s) and then invocations are carried out locally. Therefore, if the read access to write access ratio of method invocations in an application is high, DCOBE is expected to perform better than Java RMI.

Table 2. Measured costs of the basic operations in Java RMI

Operation	Time
Binding of the RMI object on a server with a user defined name	2.15 ms
Looking up of the RMI object's stub from the remote server	1.75 ms
A remote method invocation (with parameter)	0.5 ms
A remote method invocation (no parameter)	0.4 ms

6 Related Work

Our work has been influenced closely by the SOS [3], Globe [7], and Jgroup [4] projects, which support state distribution through physically distributed shared objects. The SOS system is based on the Fragmented Object (FO) model [3]. The FO model is a structure for distributed objects that naturally extends the proxy principle. FO is a set of *fragment objects* local to an address space, connected together by a *connective objects*. Fragments export the FO interface, while connective objects implement the interactions between fragments. The lowest level of the FO structure, the connective object, encapsulates communication facilities. Even though the work hides the cooperation between fragments of a FO from the clients, the programmer of the FO is responsible to control the details of the cooperation. He has to decide if a fragment locally implements the service or is just a stub to a remote server fragment.

One of the key concepts of the Globe system is its model of *Distributed Shared Objects* (DSOs) [7]. A DSO is physically distributed, meaning that its state might be partitioned and replicated across multiple machines at the same time. All implementation aspects are part of the object and hidden behind its interface. For an object invocation to be possible, a process has to bind to an object, which results in placement of a local object in the client's address space. A local object may implement an interface by forwarding all method invocations, as in RPC client stubs, or through operations on a replica of the object state. Local objects are partitioned into sub-objects, which implement distribution issues such as replication and communication, allowing developers to concentrate on the semantics of the object. Jgroup [4] extends Java RMI through the group communication paradigm and has been designed specifically for application support in partitionable distributed systems. ARM, the Autonomous Replication Management framework, is layered on top of Jgroup and provides extensible replica distribution schemes and application-specific recovery strategies. The combination Jgroup/ARM can reduce significantly the effort

necessary for developing, deploying and managing partition-aware applications. None of these projects support the composite object model and caching is restricted to the state of the entire object. However, the DCO model allows the representation of an object as a collection of sub-objects and enhances the object distribution concept by implementing replication at the sub-object level, providing a finer granularity. To the best of our knowledge, there is no programming framework that supports replication at the sub-object level.

7 Conclusion

This paper presents a new object model, *distributed composite object*, for distributed computing along with the design of a middleware architecture, DCOBE, that provides basic mechanisms to deal with issues related to activity on distributed environments. The proposed model, with the support of DCOBE, allows for collaborative design and control of distributed applications. DCOBE, being implemented on JVM, provides an environment that works on heterogeneous platforms. The key benefits of the proposed object model are distribution transparency, ease of application development, conserved bandwidth consumption, and dynamic adaptation and deployment of shared objects. We plan to enhance the model by introducing data persistency and capability-based access control policies to prevent unauthorized access to objects.

References

1. Carter J.B., Bennett J.K. and Zwaenepoel W., Techniques for reducing consistency-related communication in distributed shared-memory systems. ACM Transactions on Computer Systems, Aug. 1995, 13(3), pp. 205–243.
2. Fish R.S., Leland M.D.P., Kraut R.E., Quilt: A Collaborative Tool for Cooperative Writing, In Proc. of ACM Int. Conference on Office Information Systems, vol. 9, pp. 30–37, 1988.
3. Makpangou M., Gourhant Y., LeNarzul J.P. and Shaphiro M., Fragmented Objects for Distributed Abstractions, in T.L. Casavant and M. Singhal (eds.), Readings in Distributed Computing Systems, IEEE Computer Society Press, pp. 170–186, 1994.
4. Meiling H., Montresor A., Babaoglu Ö., Helvik B. E., Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing, Technical Report UBLCS-2002-12, Dept. Comp. Sci. Univ. of Bologna, Italy, Oct. 2002.
5. Mosberger D., Memory consistency models, Operating Systems Review, 17(1), pp. 18–26, Jan. 1993.
6. Pacull F., Sandoz A., Schiper A., Duplex: A Distributed Collaborative Editing Environment in Large Scale, In Proc. of ACM Conference on Computer Supported Cooperative work (CSCW), North Carolina USA, 1994.
7. Steen M.V., Homburg P. and Tanenbaum A.S.: Globe: A Wide-Area Distributed System, *IEEE Concurrency, 7(1),* Jan.-Mar., pp. 70–78, 1999.
8. Tammaro G., Mosier J., Goodwin N., Spitz G., Collaborative Writing is Hard to Support: A Field Study of Collaborative Writing, The Journal of Collaborative Computing 6, Kluwer Academic Publisher, pp. 19–57, Netherlands 1997.
9. Yilmaz G., Distributed Composite Object Model for Distributed Object-Based Systems, PhD Thesis, Istanbul Tech. Univ., Institute of Science and Technology, Istanbul, Turkey, May 2002.

A Java-Based Uniform Workbench for Simulating and Executing Distributed Mobile Applications*

Hannes Frey, Daniel Görgen, Johannes K. Lehnert, and Peter Sturm

University of Trier
Department of Computer Science
54286 Trier, Germany
{frey|goergen|lehnert|sturm}@syssoft.uni-trier.de

Abstract. Spontaneous multihop networks with high device mobility and frequent fluctuations are interesting platforms for future distributed applications. Because of the large number of mobile devices required for any detailed analysis, it is nearly impossible to deploy prototype applications yet. In this paper, a comprehensive approach is presented which supports experiments ranging from pure simulation of several thousand mobile devices over hybrid scenarios with interaction among simulated as well as real life devices up to dedicated field trials. Part of this paper are also conclusions drawn from experiences with a first prototype version of a self-organized auction system for ad-hoc networks.

1 Introduction

Supporting mobile devices is central to any next generation networking technology. Future paradigms such as pervasive computing and ubiquitous computing [1] are characterized by countless numbers of small and thus mobile devices that communicate with neighboring peers using wireless transmission techniques. Groups of these mobile devices may form highly dynamic ad-hoc networks at any given time. Singlehop ad-hoc networks are state of the art, e.g. in the WLAN infrastructure mode by means of access points, and allow devices to reach a conventional wireful network with one hop of wireless communication. Spontaneous multihop networks with a high device mobility and frequent fluctuation require self-organization as a primary design principle to cope with the anticipated frequency of change. As a consequence, decisions within a mobile device can only be based on local information, e.g. the state of the device itself and its current neighborhood. In such an environment, most of the goals of a distributed mobile application are achieved by synergy through altruistic behavior of all involved devices. Successfully evaluating self-organizing applications for multihop ad-hoc networks requires too many mobile devices as can be provided at the moment. Additionally, the number of volunteers needed to run these experiments is hard

* This work is funded in part by DFG, Schwerpunktprogramm SPP1140 "Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme".

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 116–127, 2004.

to obtain even in a university environment and they are even harder to orchestrate for the sake of reproducible scientific results. A promising approach is to provide a uniform workbench supporting experiments ranging from pure simulation of several thousand mobile devices, over hybrid scenarios with interaction among simulated as well as real life devices up to dedicated field trials as proofs of concept. From this uniform workbench, a development process can be derived that starts with implementing, testing, and evaluating algorithms and applications in a purely simulated environment first. In a second step, dedicated mobile devices can be cut out of a simulation run and be transferred to a real mobile device in order to deal with real user interaction and to evaluate the user experience. In a final phase, specific field trials can be defined and executed on a number of mobile devices. The hybrid nature of this approach leads to the additional requirement, that the simulator must be capable to achieve real-time execution behavior even in the case of several thousand mobile devices.

In the remainder of this paper, a Java-based implementation of such a uniform workbench for multihop ad-hoc networks is presented. The next section starts with a discussion of related work. The structure of the subsequent sections follows the aforementioned three phases of the development process. In section 3, the simulator part of the workbench is presented. Experiences gained during the evaluation of a test application show, that the required real-time behavior can be achieved for a sufficiently large number of mobile devices. Section 4 shows how dedicated mobile devices can be attached to a simulation run. The next section discusses the transfer of a simulated application on a number of mobile devices in order to carry out live experiments. Finally, in sections 6 and 7 conclusions drawn from experiences with a first self-organized auction system for ad-hoc networks are discussed.

2 Related Work

This work presents a development environment for mobile applications running in a large scale mobile multihop ad-hoc network. The environment divides the design process of mobile applications in three parts, simulation, emulation and execution on real devices. The CMU wireless extension to ns2 [2] and GloMoSim [3] are the commonly used tools for simulating protocols in mobile multihop ad-hoc networks. Both simulation environments focus on a detailed simulation of protocol layer 1 to 4. This high level of detail increases the computational complexity of the simulations. Thus, only a small number of devices can be simulated in acceptable time.

Emulation is used in order to allow user interaction and to allow existing applications to be tested in a simulation environment without the need of managing hundreds of real devices. An emulation platform using ns2 is proposed in [4]. It is an extension of the Vint/NS simulator [5] for emulation of wired networks using the wireless extension to ns2 [6]. This emulation platform scales well up to about 100 devices simulated by ns2. Other emulation approaches forego the simulation of mobile devices but emulate only node mobility and the resulting network behavior. Mobility data can be created synthetically as proposed in [7] or captured from real life test runs as done in [8].

Network experimentation using a testbed is useful to prove the applicability of a protocol or application in a real life scenario. Ad-hoc network examinations in such experiments were performed by [9] and [10] or [11] for single hop networks. An obvious disadvantage of using a testbed to examine protocol or application characteristics is that results are not reproducible and that it takes great efforts to manage the large number of real mobile devices. In order to allow more reproducible results [12] proposes users walking around by following the instructions of a scenario script running on each mobile device.

3 Simulation of Mobile Applications

The main focus of the simulation and evaluation platform presented in this paper is to ease the development and simulative evaluation of applications for mobile multihop ad-hoc networks. A high abstraction level based on an object-oriented design in Java enables the developer to concentrate on application design without worrying about technical details of the simulator. Instead of aiming to simulate the lower network layers as exactly as possible, the focus is on the simulation of the topological properties of the ad-hoc network. The combination of a high abstraction level and the limitation on the topological properties reduces the computational complexity of the simulation and allows the simulation of a high number of devices while maintaining short simulation times.

Additionally, the high abstraction level allows to implement the same abstractions in the simulation environment as well as on real hardware platforms. Applications developed and tested in the simulation environment can easily be transferred to the real hardware platform (see section 5). Thus, field trials are based on well-tested code. Traditional simulators require a complete rewrite of the application in order to port it to a real hardware platform.

The simulation environment is based on a three-layered architecture. The first layer abstracts the operating system of the mobile device, the positioning method and the wireless network technology and communication protocols. The application makes up the second layer, while the simulated user behavior controlling the application is placed on the top layer.

The operating system abstraction provides the application with information about the device, e.g. the current position and moving direction. The neighbor discovery service informs the application whenever another device enters or leaves the communication range of the device running the application. Additional information like the current position and direction of other devices is available from the neighbor discovery service, too.

Applications never access the network interface of a device directly, but use a collection of communication protocols instead. By using these existing communication protocols or defining new ones, applications can communicate with other applications running on different devices. Instead of bit-optimized messages simple Java objects are sent over these protocols. By defining a "simulated size" attribute for each message, the bandwidth used can still be simulated correctly while the development of algorithms and applications is easier. Sending objects as messages simplifies the reception of messages as well because the message objects "know" which particular method to call in the receiving application.

Mobility information of the devices in the simulation is generated from mobility models. The simulation environment defines mobility models in terms of three basic operations: devices enter and leave the simulation, and they move directly from one point to another with constant speed. Sequences of these base operations are sufficient to approximate all mobility models.

Given the network sending radii for all devices, the link calculator component computes the potential communication links between all devices for arbitrary mobility models. When a device starts a new movement, the link calculator can compute the potential communication links with all other devices for the *whole* movement. This reduces the number of comparisons with other devices significantly. The results of the link calculator can be saved and used in further simulation runs (even with a different application), thus resulting in an additional speed-up.

The network model of the simulation is exchangeable. The respective implementation has access to all simulated devices and receives the link information from the link calculator. Therefore, the network implementation can decide whether a potential link calculated by the link calculator will actually result in a communication link between devices. This allows to implement rather simple network models like an unbound network as well as complicated statistical models, which take into account the device density and other disturbances of the wireless network.

The simulation environment offers a comprehensive set of visualization tools. An application for example, may visualize its internal state, or the network implementation may draw a graph of all communication links. Visualization is based on a canvas supporting basic drawing operations like lines, rectangles, polygons, ellipses, images and text. Components use collections of geometric shapes drawing to this canvas to visualize themselves. Due to this generic approach the output of the visualization is not limited to windowing systems, but can be saved as video files or postscript figures. The current version provides canvas implementations using Java Swing/Java 2D, SWT[13], OpenGL[14,15] and PostScript.

In order to allow a better evaluation of the scalability and performance of the simulation environment, a sample message dissemination application has been tested. This application was simulated with the Random Waypoint Mobility Model [6] and an unbound network implementation for 10 minutes. The sending radii of the mobile devices are uniformly distributed between 10 and 100 meters and the moving speed of the mobile devices is uniformly distributed between 0.8 and 1.0 meters per second.

Two series of measurements were done on a Pentium IIIm with 1.2GHz and Java 2 SDK 1.4.2. The first one measured the execution speed for an increasing number of devices on a fixed simulation area of $500m \times 500m$, thus increasing the average device density in each step. The second measurement increased the size of the simulation area while increasing the number of devices, thus keeping the average device density the same. Table 1 clearly shows that device density is an important factor for the performance of the simulation environment.

In case of a constant device density the simulation environment scales well with the number of devices: for a doubled number of devices the execution time is

increased by a factor of 3.3 at most. With increasing device density the execution time for a doubled number of devices is increased by a factor of 5.2 at most. In both cases the real-time simulation of a high number of devices is possible: for the constant device density a simulation of 6400 nearly twice as fast as real-time is possible.

If the mobility and connectivity data is precomputed, the execution times decrease significantly. Especially in the case of constant device density the performance gain is dramatic. A simulation run with 12800 mobile devices is finished nearly 12 times as fast (96.6s instead of 1139.9s). For the measurement with increasing device density the execution time is reduced by 56% (451.9s instead of 1030.4s).

#devices	execution time	execution time (precomputed)
50	0.5s	0.5s
100	0.8s	0.8s
200	2.5s	2.0s
400	10.4s	7.6s
800	44.4s	28.3s
1600	198.8s	113.2s
3200	1030.4s	451.9s
6400	n/a[1]	n/a[1]
12800	n/a[1]	n/a[1]

(a)

#devices	simulation area	execution time	execution time (precomputed)
50	$250m \times 250m$	0.6s	0.7s
100	$354m \times 354m$	1.0s	1.1s
200	$500m \times 500m$	2.3s	2.0s
400	$707m \times 707m$	6.4s	4.3s
800	$1000m \times 1000m$	15.9s	9.0s
1600	$1414m \times 1414m$	41.5s	16.2s
3200	$2000m \times 2000m$	122.1s	28.1s
6400	$2828m \times 2828m$	344.8s	49.3s
12800	$4000m \times 4000m$	1139.9s	96.6s

(b)

Table 1. Execution times for the example simulation: (a) on a simulation area of $500m \times 500m$, (b) with an average device density of 0.0008 devices per m^2.

Using a Java2D canvas implementation, visualization in real-time (or even faster) is possible up to 1600 devices in the case of increasing device density. For constant device density the maximum number of devices for a real-time visualization is 3200. Further speed-ups are possible by using precomputed mobility and connectivity data.

4 Hybrid Simulation Platform

The hybrid simulation platform allows real mobile devices to be attached to a running simulation by using RMI [16] over a network connection. This enables users with dedicated external devices to interact with the simulated application.

[1] The available RAM (384 MB) of the machine used for the measurement was not sufficient to run the simulation. This is due to the high device density and the resulting number of events.

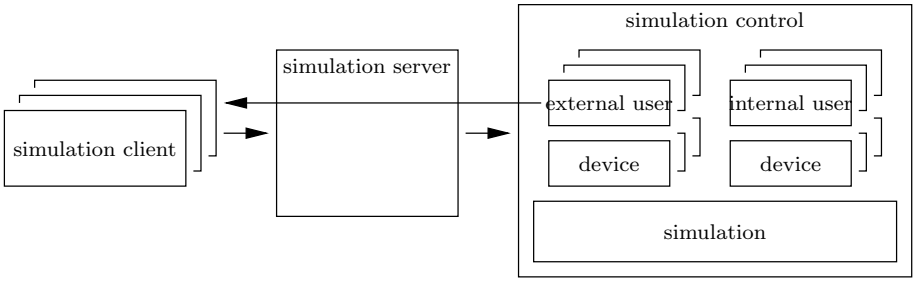


Fig. 1. Client server architecture of the hybrid simulation approach.

The hybrid approach is advantageous over evaluating on real hardware solely, since no hundreds of real devices are needed and have to be configured/managed to run the application in a real scenario. Furthermore, the visualization component of the simulation environment allows visualization of the current global state regarding the considered application. This is profitable for debugging code and to get a better insight into the application behavior. Finally, it is possible to replay, start at a certain time or speed up the simulation. This allows easier evaluation and demonstration of the considered application. Even though the application is running in a simulation environment, nevertheless one gets a feeling about the application running in real life.

Figure 1 depicts the client/server architecture of the hybrid evaluation platform. Simulation server and simulation control are running on the server side. Simulation clients are running on real mobile devices communicating with the running simulation over a wireless connection or on traditional workstations using a stationary network.

The simulation server is running as an independent thread registered as an RMI service. Any client message is processed by the simulation server first, i.e. the server waits for client logins and any user input which is passed to the simulation control in advance.

The task of simulation control is to run the simulation as a second thread on the server side. Initially, it creates a simulation instance with simulated users only. Each internal user controls one device. Additionally, simulation clients can be registered to the simulation control afterwards. The simulation control provides an additional device and an external user instance for the new client. By passing client input to the device, the external user controls the application in the same manner as an internal user. Since client input produces asynchronous events during the simulation run, synchronization regarding the event queue of the simulation is needed in order to achieve atomic processing of each event handle method. Additionally, any application output is passed to the client by using a callback object created by the simulation server. The core simulation is not aware of external clients, since by using the external user object, input and output regarding a certain client is treated in the same manner as for a

common simulated user. Thus, there is no need to change any code from the pure simulation to the hybrid simulation platform.

Finally, the simulation client is responsible to provide a front-end for the client to control its assigned mobile device and application part in the current simulation run. In general, it consists of an application GUI replacing the simulated user behavior and providing information about the application state. When porting an application from the pure simulation to the hybrid simulation platform, this is the only part where additional code has to be written.

5 Mobile Applications on Real Devices

Switching between a simulated environment and true hardware requires the same interfaces and the same platform behavior. Therefore a hardware abstraction layer is needed, which comprises all functionality of the simulated platform, such as network communication, neighbor discovery, positioning and the same event-based programming model including system timers and message events. Additionally, a user interface is needed to enable users to interact with the application. By following the proposed application development process, this GUI is already implemented for the hybrid testing environment. The platform is implemented for devices such as PocketPC or notebooks running a JVM. These devices must be able to communicate with others over IEEE802.11b [17] and to identify their current position by using a GPS unit.

The network communication part is based on UDP/IP unicasts and broadcasts. One hop communication uses these primitives directly. All higher level protocols such as position-based routing or topology-based routing are based on these simple primitives and are already implemented for the simulation.

Beaconing is used to detect adjacent devices. The beacon contains the device address, its current position, its moving direction and additional device information. To detect bidirectional connections, devices answer broadcasted beacons with an empty unicast message. Neighbor information is cached and provided to the application and attach/detach events are communicated if needed.

To provide positioning information only GPS is supported up to now. Other positioning services such as [18,19] are also possible. GPS provides spherical (longitude/latitude) coordinates but the application expects Cartesian coordinates. The projection used in the platform is a variant of a Gauss-Kruger projection with a shifted zero-point.

Up to now, messages sent by the applications are directly passed to the send system without any bit optimization. The message objects are serialized by the standard serialization mechanism provided by Java. If needed, this could be replaced with a more efficient method.

Since the simulated applications are inherently event-driven, the real platform must provide the same event-driven programming model. This is achieved by multithreading and operating system timers (normally also implemented as threads). Due to the single-threaded nature of the simulation environment, the application code does not care about concurrency errors. To avoid application

changes, precautions have to be made. The platform uses one thread to execute the application handlers, the events from the other threads are passed to it over an event queue. Contrary to the simulation, the platform events have no ordering, therefore no assumption should be made in case of contemporaneity or nearly contemporaneity of independent events.

In the simulated environment simulated user behavior is needed. On real hardware this simulated user behavior could still be used for testing purposes, e.g. to reproduce phenomena observable in simulations. The common uses for the implementation on real hardware are on the one hand field trials with real user interaction on the other the normal use of this application in a real environment. Therefore a GUI is needed allowing the user to interact with the application. Since a GUI for this application is already implemented and tested with the hybrid environment it can also be used for the real hardware. Porting the GUI is fairly easy done by omitting the remote link to the application achieved by RMI.

6 Case Study: UbiBay

The development environment presented in this paper has been successfully used to implement UbiBay, a self-organizing auction platform for mobile multihop ad-hoc networks. UbiBay is based on SELMA[20], a middleware platform for mobile ad-hoc networks. This middleware platform uses the marketplace pattern [21] as its main communication method and provides applications with all the components needed to implement this pattern. A marketplace is a small region of the ad-hoc network where a high device density is likely. The basic idea of the marketplace pattern is that users send their requests and offers to the marketplace where the negotiations take place. This increases the probability that matching peers are able to find each other and negotiate in an efficient manner. Requests, offers and negotiation results are sent to the marketplace and back using position-based routing. Negotiation results use a home zone of the user to find back to the originating device. This home zone is a small region in the ad-hoc network where the owner of a device will be found with a high probability, e.g. the user's house.

Communication at the marketplace is limited to the marketplace area and thus does not increase the network load outside of the marketplace. Due to the higher device density at the marketplace and its relatively small size, only a few hops are needed to communicate with other devices at the marketplace. Instead of simple flooding a neighbor knowledge broadcast[22] is used for marketplace communication. Additionally, devices at the marketplace can use topology-based routing for unicast communication.

UbiBay consists of three kinds of requests: auction, bid, and information requests. When a user wants to start an auction, he will send an auction request to the marketplace containing the description of the auction, the starting bid and the duration. The auction request stays at the marketplace until the auction is over. By sending information requests to the marketplace, a user gets information on all auctions currently running. In order to bid on an auction, the user sends a

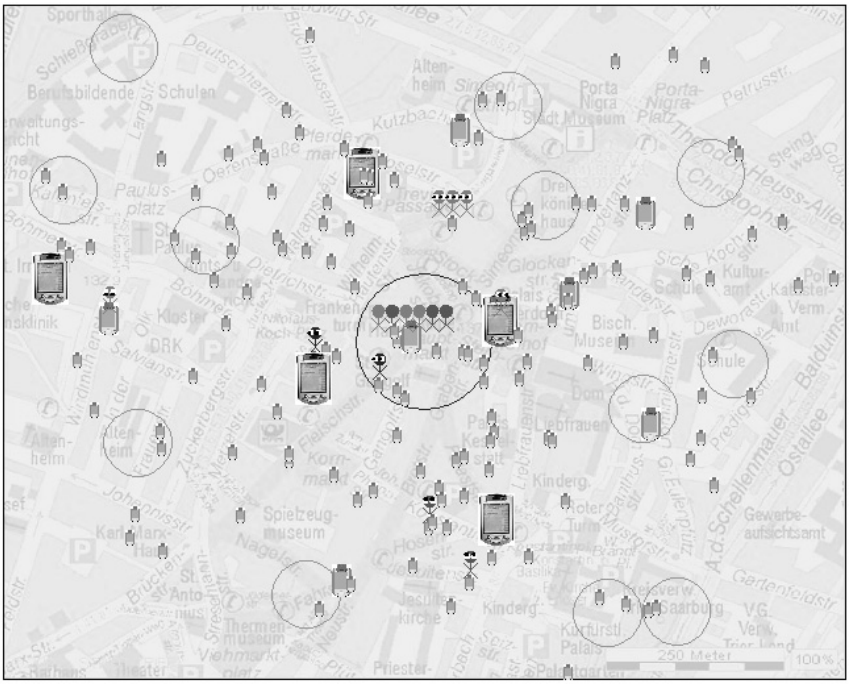


Fig. 2. Visualization of the simulated UbiBay application.

bid request with his maximum bid to the marketplace. At the marketplace, the bid request is matched with other bid requests for the auction. The bid request will stay at the marketplace until its maximum bid is exceeded or the auction is over. Both outcomes result in a notification message to the user.

UbiBay was implemented as a simulated application using the simulation environment described in section 3. Using a simple simulated user behavior auctions where simulated and the visualization features of the simulation platform were used to debug the application during development and to get a better understanding of the application behavior (see Fig. 2).

Based on this simulation application, a version for the hybrid simulation platform was developed. The application code of UbiBay did not change at all, but a suitable external user class was developed and plugged into the simulation server. Additionally, a graphical user interface was developed to allow the external clients to interact with the UbiBay application. See Fig. 3 for a screenshot. SWT was used as the widget set, because the graphical user interface was to be used on HP iPAQs in the next step. Other widget sets like AWT, Swing etc. are either not available on the mobile Java platform, do not allow rich user interfaces, or do not perform well enough.

The graphical user interface allows the user to send information requests to the marketplace. Results are displayed in a table together with a status icon. The user can view details of an auction, bid on an auction or start a new auction.

Fig. 3. Graphical user interface of UbiBay.

Auctions are removed from the list if they are over and the user has not been involved.

Using the hybrid simulation platform with workstations in a fixed network proved to be very valuable. Testing an application by using a real graphical user interface is completely different from simulating an application with simple user behavior. Many (small) bugs emerged not until human users tested the application in the hybrid environment.

The last step in the development of UbiBay was the implementation of the application on real mobile devices. PocketPCs with Microsoft Pocket PC 2002, GPS receivers and the IBM J9 VM were used as the platform. Due to the fact that the execution platform provides the same abstractions as the simulation environment, the UbiBay application code was transferred "as is" to the execution platform. Transferring the graphical user interface to the mobile device required only minor modifications. The sizes of some GUI elements had to be tweaked and the RMI part of the client code was no longer necessary.

7 Conclusions

The proposed development platform for applications in mobile multihop ad-hoc networks takes advantage of three different environments. The simulative approach enables the testing of algorithms and applications with lots of devices. The hybrid approach allows testing with real user interaction in a simulated environment. Finally, the real platform implementation is needed to verify the simulation results in the real world and test the final application in field trials. Using the development platform also leads to a development process for applications in mobile multihop ad-hoc networks. Starting with an application in the simulation environment, the complete application code can be reused in the hybrid platform, extended by a graphical user interface. Both the user interface and the application code again can be reused with minor modifications when testing

the application on real devices. Following this development process reduces the implementation overhead to a minimum due to code reuse.

Both the development process and platform have been tested during the implementation of the UbiBay application and SELMA, a middleware platform for mobile multihop ad-hoc networks. Based on first simulation results the UbiBay application was tested both in the hybrid environment and on real hardware. Tests in the hybrid environment proved to be very valuable, because the user interface and behavior of the application could be thoroughly tested, which led to several improvements.

References

1. Weiser, M.: The computer for the 21st Century. Scientific American **265** (1991) 94–104
2. Fall, K., Varadhan, K.: The ns manual. The VINT Project – A collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC (1989-2003)
3. Zeng, X., Bagrodia, R., Gerla, M.: GloMoSim: A library for parallel simulation of large-scale wireless networks. In: Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS-98), Los Alamitos, IEEE Computer Society (1998) 154–161
4. Ke, Q., Maltz, D., Johnson, D.B.: Emulation of multi-hop wireless ad hoc networks. In: The 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000). (2000)
5. Fall, K.: Network emulation in the VINT/NS simulator. Proceedings of the fourth IEEE Symposium on Computers and Communications (1999)
6. Broch, J., Maltz, D., Johnson, D., Hu, Y.C., Jetcheva, J.: A performance comparison of multi-hop wireless ad hoc network routing protocols. In: Proc. of the 4th ACM/IEEE Int. Conf. on Mobile Computing and Networking (MobiCom'98). (1998) 85–97
7. Zhang, Y., Li, W.: An integrated environment for testing mobile ad-hoc networks. In: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing. (2002) 104–111
8. Noble, B., Satyanarayanan, M., Nguyen, G.T., Katz, R.H.: Trace-based mobile network emulation. In: Proceedings of SIGCOMM 97. (1997) 51–61
9. Maltz, D., Broch, J., Johnson, D.: Lessons from a full-scale multihop wireless ad hoc network testbed. IEEE Personal Communications Magazine **8** (2001) 8–15
10. Ramanathan, R., Hain, R.: An ad hoc wireless testbed for scalable, adaptive qos support. In: Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2000). Volume 3. (2000) 998–1002
11. Agrawal, P., Asthana, A., Cravatts, M., Hyden, E., Krzyzanowski, P., Mishra, P., Narendran, B., Srivastava, M., Trotter, J.: A testbed for mobile networked computing. In: Proceedings of 1995 IEEE International Conference on Communications (ICC '95). (1995) 410–416
12. Lundgren, H., Lundberg, D., Nielsen, J., Nordström, E., Tschudin, C.: A large-scale testbed for reproducible ad hoc protocol evaluations. In: 3rd annual IEEE Wireless Communications and Networking Conference (WCNC 2002). (2002)
13. Eclipse Project - Universal ToolPlatform: SWT: Standard widget toolkit (2003) Available from `http://www.eclipse.org/platform/index.html`

14. Eclipse Project - Universal ToolPlatform: SWT: experimental OpenGL plug-in (2003). Available from
`http://dev.eclipse.org/viewcvs/index.cgi/ checkout /platform-swt-home/o%pengl/opengl.html`
15. Segal, M., Akeley, K.: The OpenGLm graphics system: A specification (version 1.4)(2002). Available from
`http://www.opengl.org/developers/documentation/version1_4/glspec14.pdf`
16. Sun Microsystems, Inc.: Java remote method invocation specification (revision 1.8) (2003) Available from
`ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`
17. International Standard ISO/IEC 8802-11: Information technology – telecommunications and information exchange between systems – local and metropolitan area networks – specific requirements – part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications (1999)
18. Bulusu, N., Heidemann, J., Estrin, D.: GPS-less Low Cost Outdoor Localization For Very Small Devices. IEEE Personal Communications Magazine **7** (2000) 28–34
19. Hamdi, M., Capkun, S., Hubaux, J.P.: GPS-free Positioning in Mobile Ad-Hoc Networks. In: Proceedings of HICSS, Hawaii (2001)
20. Frey, H., Görgen, D., Lehnert, J.K., Sturm, P.: Selma: A middleware platform for self-organizing distributed applications in mobile multihop ad-hoc networks. Submitted to PERCOM 2004 (2003)
21. Görgen, D., Frey, H., Lehnert, J., Sturm, P.: Marketplaces as communication patterns in mobile ad-hoc networks. In: Kommunikation in Verteilten Systemen (KiVS). (2003)
22. Williams, B., Camp, T.: Comparison of broadcasting techniques for mobile ad hoc networks. In: Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC). (2002) 194–205

Seamless UML Support for Service-Based Software Architectures*

Matthias Tichy and Holger Giese

Software Engineering Group, Department of Computer Science
University of Paderborn, Germany
{[mtt|hg]}@uni-paderborn.de

Abstract. The UML has become the *de facto* standard for the analysis and design of complex software. Tool support today includes the generation of code realizing the structural model described by class diagrams as well as code realizing the reactive behavior of the dynamic model described by statecharts. However, the CASE tool support for service-based architectures and especially later process phases addressed with component and deployment diagrams is rather limited. In this paper a seamless support of the whole service life cycle of service-based software architectures by means of UML is presented. We present the employed concepts to support the design of services (including full code generation), to compose applications out of services at design time, and to deploy services at run-time. Additionally, we describe our realization of these concepts in form of a CASE tool extension and a run-time framework.

Keywords: Development methodologies for UML, service-based architectures, design of distributed Java applications.

1 Introduction

Open service-oriented software architectures [1,2,3] have received considerable attention as approach to overcome the maintainability problems of large monolithic software. UML [4] CASE tool support for the mentioned approaches for service-based architectures is, however, usually rather restricted. As these service-based approaches traditionally have focused on language mappings to C++ or Java rather than design notations such as UML, currently support for them is most often found in programming environments. Therefore, *generic* UML CASE tools are used for the analysis and design and thus no support for service composition or deployment in dynamic service-based architectures is provided.

Those UML CASE tools can generate source code based on the design results for the structural model described by class diagrams to improve maintainability.

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

N. Guelfi, E. Astesiano, G. Reggio(Eds.): FIDJI 2003, LNCS 2952, pp. 128–138, 2004.

Some more elaborated tools can also generate code for the reactive behavior of the dynamic model described by statecharts. Unfortunately, the generated source code must be manually adapted by the developer to comply with the special requirements of service-based architectures.

Therefore, the manually realized implementation usually differs from the original design in many aspects. Since the design documents are not changed accordingly when implementing the system the maintainability of such systems becomes deteriorated. The current design is not correctly documented any more and the lost traceability information must be re-engineered by looking into the source code when a re-design is required. Build-in support for roundtrip engineering by some tools tries to overcome the traceability problem for direct changes of the implementation. However, this approach is restricted to high level model information which is still contained in the implementation.

The sketched maintainability problems of service-based software result from software adjustments usually done in the later phases (composition and deployment) to comply with the requirements of service-based architectures. Thus, only the *seamless* support of the whole software life cycle for service-based systems and especially the later phases can prevent the described deterioration of maintainability.

In this paper we present an approach for a seamless UML support for the complete life cycle of service-based systems. UML diagrams are used for all phases in the life cycle. The seamless support manifests itself in specific extensions of the used UML diagram types for service-based architectures, since all specific information due to the nature of service-based architectures must be included in the models. Additionally, the results of prior phases are used as inputs to later phases. Based on the respective UML diagrams, full source code for the service implementation and description files for their composition and deployment are generated by our realization of this approach in form of a CASE tool extension of the Fujaba Tool Suite[1]. Additionally a framework is provided which executes the specified services in a fault-tolerant manner. Jini [1] is used as a representative for a service-based architecture. Due to space restrictions we mainly focus in this paper on the later life cycle activities (Service Realization, Service Composition, and Service Deployment Planning). Nevertheless, we briefly address the System Design activity to highlight the seamless integration with the later activities.

In the following Section 2, we present our approach for service-based architectures. Afterwards, we review in Section 3 the existing body of work and how the presented approach differs. The paper is closed with some final conclusions and an outlook on future work.

2 Seamless Support

Figure 1 shows the activities of the life cycle for service-based systems, which diagram types support the different activities, and the artifacts an activity gen-

[1] www.fujaba.de

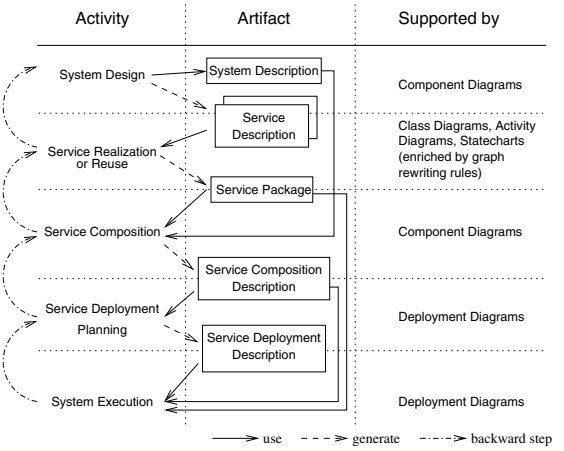


Fig. 1. Activities during the service life cycle

erates or uses. The different activities of Figure 1 correspond to the subsections of this section.

In Section 2.1 the first life cycle activity is described. The topic of this activity is the analysis and design of the system, which should be realized as a service-based architecture. Here, we show how a system (our running example) can be partitioned into several services. This decomposition is described by a UML component diagram. As a result the required services, their interfaces, and connections, which are specified within the component diagram, result in a System Description and multiple Service Descriptions.

We proceed with a description of how one service is realized in Section 2.2. In our approach we use UML class diagrams for the modeling of the structure. For the modeling of behavior we use activity diagrams and statecharts enriched by graph rewriting rules. Based on these design diagrams the implementation resp. the full source code for the service is automatically generated and packed into a Service Package, which is later executed by the run-time environment.

In Section 2.3, service composition is described. We use UML component diagrams to specify the required restrictions for the composition of services and store them in a specific Service Composition Description. The composition of services at run-time by the run-time environment respects the specified interface types and additional attribute restrictions.

The planning of service deployment using UML deployment diagrams is then presented in Section 2.4. For the usage in service-oriented architectures, we have added requirements of node characteristics to deployment diagrams. The planned configuration is then stored in a Service Deployment Description.

In Section 2.5 we present the execution of the specified services by a Jini-based run-time environment. The information provided by the earlier activities (Service Packages, Service Composition Description, and Service Deployment Description) is used for the correct deployment and online-binding. Using the plan-

ning deployment diagrams the current situation of a running system is visualized online to support administration.

2.1 Service Design

During the System Design activity the different services which form the service-based system must be determined. During the identification of services UML component diagrams are used. The approach proposed in [5] can be used for the identification of the different services. After the initial set of services are identified their connecting (provided and used) interfaces must be defined. The definition of interfaces includes the declaration of their operations. After the interfaces have been defined the different services are connected via their provided and used interfaces to complete the specification. Thus in the resulting initial component diagram the identified services, their interfaces, and their connections are shown.

Based on the resulting component diagram a System Description is generated which contains information about all services and their connections. For each service contained in the component diagram a Service Description is generated which contains the specification of the service and its interfaces.

Throughout this and all following sections we show the application of our approach using the service-based version and configuration system DSD (Distributed Software Development) [6] as running example. For the sake of a clearer presentation we show only a subset of DSD containing some basic services.

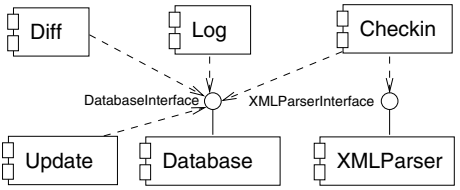


Fig. 2. DSD example

After the initial analysis step several services have been identified (see Figure 2). A Database service and an XML parser service are two of the main services in DSD. Both services are used by the services which provide the version and configuration functionality. In the considered subset of DSD all services use the Database service in order to gather information about the user and its roles in the different development projects. The Checkin service additionally uses the XML parser to read some intermediate file which is generated during the commit process. The information stored in this intermediate file is then written into the database via the database service for long-term storage.

After the identification of the different services and the definition of their interfaces, the realization resp. reuse of services follows.

2.2 Service Realization or Reuse

The above mentioned Service Descriptions contain the services and their interfaces. It is possible that there are already some standard services available which can be used in place for some specific services in the system. For example the XML parser and the database service are likely already available and can be simply reused. If no standard service is available, the service must be realized. For this service the name and the defined interfaces are extracted from its Service Description. Based on this information an initial class diagram for that service is generated.

This initial class diagram contains a main class and the different interfaces the service provides or uses. The different interfaces are marked by stereotypes ≪ProvidedInterface≫ resp. ≪UsedInterface≫. These stereotypes are later used for the generation of the Service Description contained in the Service Package to differentiate between these two kinds of interfaces. Since the service will be executed by a run-time environment, an initial draft of a helper class is generated, too. This helper class provides some service specific support to the run-time environment.

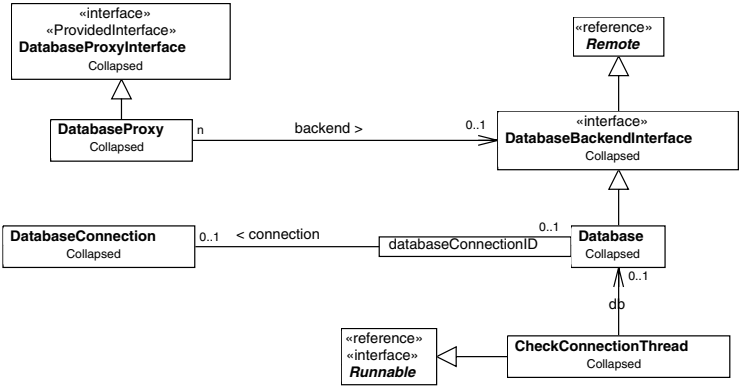


Fig. 3. Simplified class diagram of the database service

The initial class diagram provides a starting point for the developer. For the structural part the initial class diagram must be extended by the developer. Figure 3 shows a simplified class diagram of the database service which has been extended by the developer. Note the ≪ProvidedInterface≫ stereotype attached to the DatabaseProxyInterface interface.

For the behavioral part of the implementation the Fujaba Tool Suite provides additions to basic activity diagrams and statecharts in form of graph rewriting rules [7,8]. Graph rewriting rules are a powerful design notion for the specification of changes on the object structure. The Fujaba Tool Suite provides code generation for class diagrams, activity diagrams, statecharts, and the graph rewriting rules additions [7,8]. By the use of these diagrams, their well-defined semantics

[9] and the provided code generation the separation between design and implementation is lifted. Thus, the complete behavioral specification of a service is designed using UML and the full source code implementing the design is generated. Therefore, the maintainability of the resulting services is greatly improved compared to manually written or after code generation manually adapted services.

After the realization of the service is finished, the source code is generated and compiled. The resulting service class files and its generated Service Description, which includes information about the service, its interfaces and the helper class, are packaged into a Service Package. When all needed services have been developed and Service Packages have been created, the composition to a service-based system using component diagrams follows.

2.3 Service Composition

To provide seamless support the initial component diagram developed at the beginning (see Section 2.1) is the starting point for the service composition. Now this initial component diagram is refined to reflect the logical structure of the service-based design of the system and to include the implemented services. Our approach targets service-based architectures in which services are not assembled in monolithic applications but form a loosely coupled system of services. These services have no hardwired connections but connect themselves dynamically by the use of online binding. The connection is based on their interfaces. Due to the dynamic nature of service-based architecture we explicitly support *dangling* used interfaces during the service composition activity. Those dangling used interfaces are connected to provided interfaces of other independently deployed services during run-time. In dynamic service-based architectures richer semantic information about the provided and used interfaces is required. Thus, we have added support for the setting of attributes to provided interfaces and adding of attribute restrictions to used interfaces. Our approach is extensible w.r.t. concepts for specifying and matching the behavioral meaning of interfaces (e.g. [10])

Figure 4 shows a part of the DSD system which describes the composition of the Checkin, Database, and XML parser service. Since the Checkin service should connect to an already available XML parser service during run-time, the XML parser service itself is not part of the component diagram.

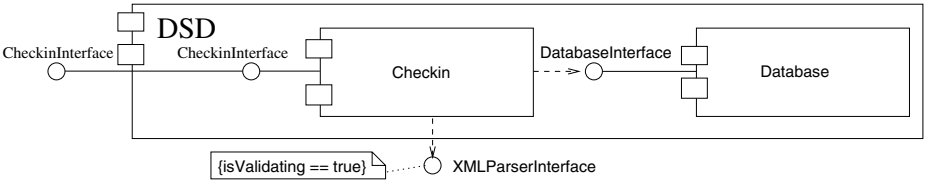


Fig. 4. Service composition

In our example the checkin service must be connected only to XML parsers which have the ability to validate the XML file's conformance to an XML schema. Therefore we add a corresponding restriction `isValidating==true` to the used interface. The run-time environment respects these additions to the interfaces and only connects services where the following conditions are met: 1) the provided interface is the same or derived from the used interface, 2) the attribute restriction expression of the used interface evaluates to true based on the attributes of the provided interface. In our example the database service offers its service via its interface only inside of the compound DSD service. For this encapsulation we use service groups provided by Jini. The specification defined in the component diagram is written to the Service Composition Description which will be used in the Service Deployment Planning activity and for the execution by the run-time environment.

2.4 Service Deployment Planning

After specifying the composition of the services in the next step the deployment (physical mapping) of the services contained in the Service Composition Description has to be planned. In this planning stage the required properties of computation nodes to execute the services are specified using UML deployment diagrams.

Deployment diagrams show the relation between services and nodes. According to the current UML specification [4] this relation means that the service will be executed on that node. Especially for service-based architectures it is rather useful to describe the characteristics a node must have to be able to execute a service. Thus, the deployment plan includes more degrees of freedom which can be utilized by the run-time environment. In our realization the set of nodes, which have the needed characteristics, is defined by boolean expressions on node attributes. The list of node's attributes includes but is not limited to: hostname, jdk version, operating system, memory, ip address.

For the sake of a clearer presentation of the set of nodes, on which a service can be executed, the service is allowed to be connected with more than one node via a deploy-edge in the deployment diagram. If a service has multiple deploy-edges, a disjunction is used to build the final expression out of the boolean

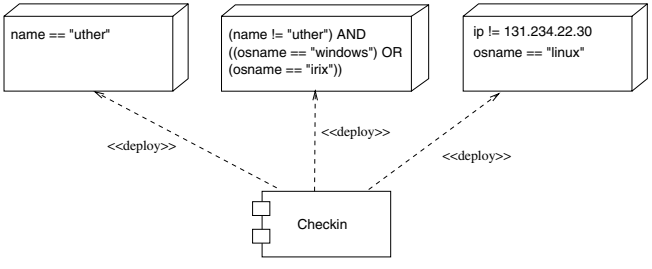


Fig. 5. Deployment planning diagram

expressions defined for each node. As you can see in Figure 5, two types of specifying a set of nodes are possible. The first type uses a conjunction of a set of attribute restrictions. In the diagram they are displayed vertically as in the right node. Since in some situations this is not convenient a second type is possible, where an arbitrary boolean expression based on the attribute restrictions can be specified as in the middle node. A Service Deployment Description results from the Service Deployment Planning activity. The actual deployment of the services by the run-time environment is based on this description.

2.5 System Execution

After the planning has been finished, the administrator uses the run-time environment to execute the system of services, modeled by UML diagrams in the previous steps (cf. Figure 1), in a fault-tolerant manner. For each service contained in the component diagram specified during the service composition step (stored in the Service Composition Description), the run-time environment looks for a computational node which satisfies the constraints specified in the deployment diagram (stored in the Service Deployment Description). Then, the run-time environment loads the compiled source code of the service, which has been generated from the structural and behavioral design diagrams, from the Service Package and executes the service on that node. After that, the run-time environment connects the interfaces of the different services according to the constraints specified in the component diagram in the service composition step. Finally, it supervises the execution of all started services and ensures the availability of the services in case of failures. A more detailed description of this run-time environment can be found in [11].

The executed system of services is visualized using the deployment planning diagrams. In this context all nodes which are in the system are displayed including their actual characteristics as well as all services which are executed in the system. Here the services have connections only to the nodes they are currently executed on. See Figure 6 for a screenshot of the Fujaba Tool Suite displaying the current deployment situation (Note, that the deployment edges now have ≪deployed≫ stereotypes).

3 Related Work

The presented goal of full life cycle support with the UML is very much in line with the model driven architecture (MDA) initiative [12] of the OMG. The MDA claims that full platform-independence is possible by model compilers for the design models. In contrast to this ambitious concept, the presented work focuses on the question how complete UML based life cycle tool support for the specific case of service-based architectures can be achieved.

Cheesman and Daniels in [5] only cover the component specification process, whereas we also cover the later phases (implementation support and especially composition and deployment issues for service-based architectures) and provide

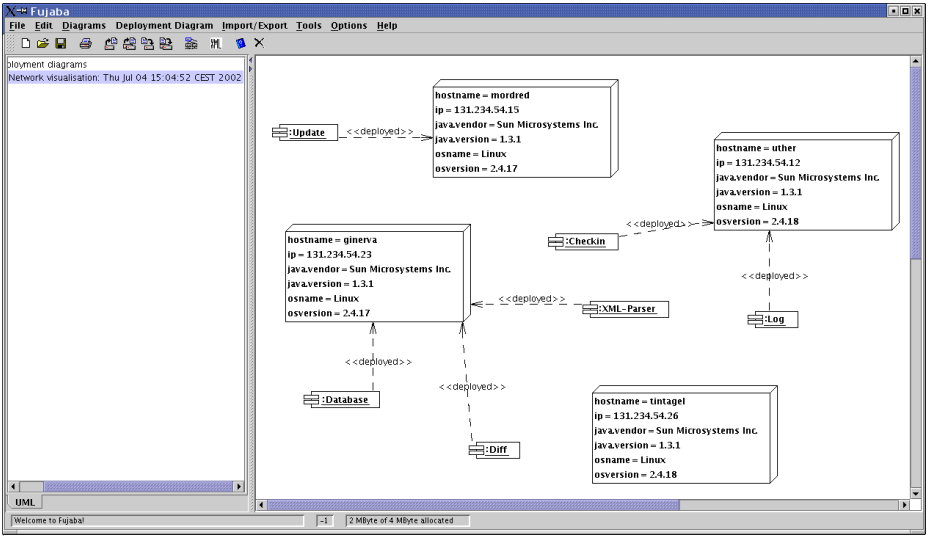


Fig. 6. Deployment visualization diagram

tool support. Additionally, they do not take the inherent dynamics (spontaneous networking and online binding) of service-based architectures into account.

Baresi et al. in [13] also use graph rewriting rules in their process for service-based architectures. They focus mainly on the collaboration part of the system and verify that the different collaborations at run-time can indeed be reached by the application of the graph rewriting rules. Our application of graph rewriting rules is targeted at the service implementation not their interconnection. We restrict ourselves to basic composition constraints (type and attribute-conditions) in order to connect the service instances during run-time automatically.

For Jini [1] a reasonable approach for life cycle support of services has been developed in the RIO project [14]. The specification of the service composition via interfaces is done only by an XML file. No UML support and no code generation are provided. The RIO framework itself provides a tool for the visualization of the deployment situation which uses a proprietary graphical representation of service instances. Therefore, support for the life cycle is restricted to composition and deployment, only.

4 Conclusions and Future Work

In this paper we proposed an approach which supports the complete life cycle of service-based architectures by the use of UML. It offers a higher level of maintainability of the resulting service-based system due to the seamless use of UML, the direct generation of full source code, and the direct usage of the UML component and deployment diagrams by the run-time environment to execute the service-based system. Our approach takes the special characteristics of service-

based architectures into account. Especially in the later activities our approach offers added value to the developer compared to other approaches.

We are currently further developing our approach to support the UML 2.0 superstructure final adopted specification [15]. In this specification the addition of ROOM [16] elements like capsules, protocol state machines etc. is proposed. We plan to use protocol state machines, describing the dynamic characteristics of interfaces, in addition to the proposed attribute restrictions to check at run-time, whether the connection of two services via their interfaces is correct.

Acknowledgments. The authors wish to thank Sven Burmester, Matthias Meyer, and Daniela Schilling for comments on earlier versions of the paper.

References

1. Arnold, K., Osullivan, B., Scheifler, R.W., Waldo, J., Wollrath, A., O'Sullivan, B.: The Jini(TM) Specification. Addison-Wesley (1999)
2. Microsoft: Microsoft .NET: Realizing the Next Generation Internet. Technical report, Microsoft (2000) White Paper.
3. Sun Microsystems: Sun[tm] Open Net Environment (Sun ONE) Software Architecture. (2001)
4. OMG: Unified Modeling Language Specification Version 1.5. Object Management Group, 250 First Avenue, Needham, MA 02494, USA. (2002)
5. Cheesman, J., Daniels, J.: UML Components, A simple process for specifying component-based software. Addison-Wesley (2000)
6. Gehrke, M., Giese, H., Tichy, M.: A Jini-supported Distributed Version and Configuration Management System. In: Proc. of the International Symposium on Convergence of IT and communications (ITCom2001), Denver, USA. (2001)
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. LNCS 1764, Springer Verlag (1998)
8. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22^{nd} International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
9. Zündorf, A.: Rigorous Object Oriented Software Development. Habilitation thesis, University of Paderborn (2001)
10. Giese, H., Wirtz, G.: The OCoN Approach for Object-Oriented Distributed Software Systems Modeling. Computer Systems Science & Engineering **16** (2001) 157–172
11. Tichy, M., Giese, H.: An Architecture for Configurable Dependability of Application Services. In: Proc. of the Workshop on Software Architectures for Dependable Systems (WADS), Portland, USA (ICSE 2003 Workshop 7). (2003)
12. Gokhale, A., Schmidt, D.C., Natarajan, B., Wang, N.: Applying model-integrated computing to component middleware and enterprise applications. Communications of the ACM **45** (2002) 65–70
13. Baresi, L., Heckel, R., Thöne, S., Varro, D.: Modeling and Validation of Service-Oriented Architectures: Application vs. Style. In: Proceedings of the ESEC/FSE 03, September 1 5, 2003, Helsinki, Finland, ACM Press (2003)

14. Sun Microsystems: RIO - Architecture Overview. (2001) 2001/03/15.
15. OMG: UML 2.0 Superstructure final adopted specification. Technical Report ptc/03-08-02 (2003)
16. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. (1994)

Model Generation for Distributed Java Programs

Rabéa Boulifa and Eric Madelaine

INRIA Sophia-Antipolis,
(rabea.boulifa, eric.madelaine)@sophia.inria.fr

Abstract. We define a behavioural semantics for distributed Java applications and a procedure for building finite models of their behaviour. The models are synchronized networks of labelled transition systems, structured in a compositional way, and capturing the communication events between distributed objects. Our procedure is based on the analysis of the method call graph. If the set of observable events is made abstract enough to be finite, then our procedure is guaranteed to terminate and to compute a finite model.

1 Introduction

The use of formal verification techniques to enhance the reliability of software systems is not yet widely accepted, though there are a number of methods and tools available. Ideally, one should be provided with a development environment including high level languages for the specification of the system, and automatic tools for checking the implementation against the specification. At the heart of this is the ability to generate, from the code of an implementation, a model that is both precise enough to encompass the property we want to prove, but small enough to be manageable by the tools (and at least have a finite representation).

The first concern is naturally related to the work on abstract interpretation [7], that can be used to abstract away from concrete data values. It is also related to *slicing* techniques [3,10], that allows to restrict the analysis to the part of the program effectively related to the property.

A number of tool developments have been done recently, for example in the Slam project [2] for analysis and verification of C programs, or in the Bandera project [6] for sequential or multi-threaded Java programs. Typically this kind of tools use static analysis techniques, coupled with abstraction and slicing, to produce a finite model that is fed to various model-checking tools. The whole process cannot be fully automatic, because the data abstractions, and othe rforms of approximations, are provided by the user.

We are developing a similar framework [4], dedicated to distributed Java applications, in which remote objects communicate by asynchronous method calls. We are interested in proving global properties of distributed applications, namely temporal properties capturing the significant events of the lifecycle of distributed objects : sending of remote method calls, receiving results from remote computations, selecting and serving requests from the local request queue. Distributed

N. Guelfi, E. Astesiano, G. Reggio (Eds.): FIDJI 2003, LNCS 2952, pp. 139–152, 2004.

applications fit naturally with compositional models, and we shall take advantage of this for structuring the models, thus keeping them much smaller, and also by using verification tools that make use of this structure.

We rely on a middleware called ProActive [5] is an example that provides the developer with a high-level programming API for building distributed Java applications, ranging from Grid computing to pervasive and mobile applications. ProActive has a formal semantics [8], that guarantees that programs behave independently of their distribution on the physical network; *active objects* are the basic units of activity, distribution, and mobility used for building these applications.

The verification tools we use to check our models are based on Process Algebra theories : models are communicating labelled transition systems [1], and their semantics is considered modulo bisimulation congruences. Their emphasis is on properties related with bisimulation semantics, including safety and liveness properties in modal branching time logics, and more generally equivalence of models of different level of refinement. They take advantage of congruence properties of the systems to avoid state explosion of the models. This approach allows us to build the models of our applications on a per-object basis, and also to specify their desired behaviour in a component-wise manner.

Our contribution in this paper is a behavioural semantics for Java/ProActive applications, given in the form of SOS rules working on the method call graph [9] of the active objects. Assuming that we have (e.g. by abstract interpretation) a finite enumeration of the active objects created during the application lifetime and a finite set of message labels, these rules give us a procedure for building a finite LTS model for each object class, and a synchronization network [1] representing the application. We guarantee that the obtained network is finite and that the procedure for computing the behavioural model terminates. We build compositional models to keep individual LTS as small as possible, and use the compositionality features of the semantics (bisimulation congruences) and of the checking tools to master the state explosion.

2 Background

2.1 ProActive

ProActive [5] is a 100% Java library for concurrent, distributed, and mobile computing whose main features are transparent remote active objects, asynchronous two-way communications with transparent futures, and high-level synchronization mechanisms. *ProActive* is built on top of standard Java APIs and does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor, or modified virtual machine. The target environment can be either a multiprocessor workstation, a pool of machines on a local network, or even a grid.

A distributed application built using *ProActive* is composed of a number of medium-grained entities called *active objects* which can informally be thought of

as "active components". Each active object has one distinguished element, the *root*, which is the only entry point to the active object. All the other objects inside the active object are called *passive objects* and cannot be referenced directly from objects which are outside of the active object.

Each active object has its own thread of control and has the ability to decide the order in which to serve incoming method calls that have been dropped into its pending requests queue. Method calls to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism called *wait-by-necessity*. Note that while asynchronous, there is a guaranty of delivery, and conservation of order for remote method calls; this is achieved thanks to a short *rendez-vous* phase at request sending.

ProActive provides primitives for dynamically creating and migrating active objects. In the context of this paper, we shall not elaborate on those dynamic features, but rather concentrate on the case where the active objects are already created and their topology is fixed.

2.2 Method Call Graphs

Our analysis focuses on applications in which the interaction between active objects is done by messages exchange: procedure or method calls (cf Figure 1). A model of program capturing these interactions is the method call graph. It abstracts away all data flow and focuses on control flow: which methods are called during execution and in what order.

Definition 1 *The* **Method Call Graph** *of an application is a tuple* $MCG \triangleq (id, V, \mu, \rightarrow^C, \rightarrow^T)$ *where*

1. *id is the main method name,*
2. V *is a set of nodes, each decorated by a node type in* $\{ent, call, rep, seq, ret\}$,
3. $\mu : V \rightharpoonup V$ *is a partial function mapping nodes (representing a program point where a future object is used) to the node where this future was defined,*
4. $\rightarrow^C \subseteq V \times V$ *are the call edges of MCG,*
5. $\rightarrow^T \subseteq V \times V$ *are the (intra-procedural) transfer edges of MCG.*

The node type indicates whether a node is the entry point *ent(id)* of method, a call *call(id)* to another method (local or remote), a reply point *rep(id)* to a remote method call, a sequence node *seq* (representing standard sequential instruction, including branching), or a node *ret* in which the execution of the method terminates, either normally or with an exception.

The domain of a method named *id* is the set of its nodes, $\mathcal{D}(id) = V$.

2.3 Labelled Transition Systems

As usual in the setting of process algebras and distributed applications, we give the behavioural semantics of programs in terms of labelled transition systems. The composition of LTSs representing the behaviour of individual active objects, into the global LTS of the application is expressed by synchronization networks [1]. We give here the notations that we use for those notions:

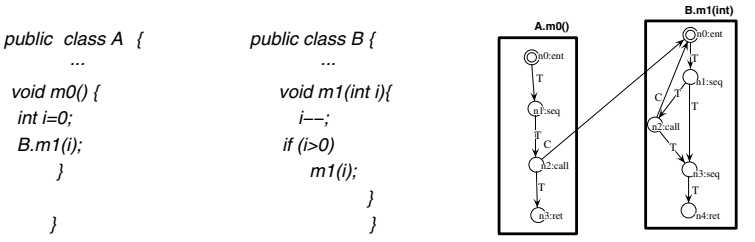


Fig. 1. Java program with the corresponding method call graph

Definition 2 Action. *The alphabet of actions is a set* $A = \{!m, m, ?m\}$ *representing method calls (local and remote), and reception of their results.*

Definition 3 LTS. *A labelled transition system is a tuple* $LTS \triangleq (S, s_0, L, \rightarrow)$ *where* S *is the set of states,* $s_0 \in S$ *is the initial state,* $L \subset A$ *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \subset S \times L \times S$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

The operator for composing LTSs is a synchronization network. We give its definition in a graphical form (as in [11]), before defining its semantics as a product on LTSs.

Definition 4 Net. *A net is a pair* $Net \triangleq (\mathcal{B},\ \mathcal{L})$ *where*

- $\mathcal{B}$ *is a set of Boxes, each box has a name* $Name(\mathcal{B})$ *and a set of labels* $Ports(\mathcal{B})$ *labelled by actions in* A,
- $\mathcal{L}$ *is a set of links between ports, each link having the form* $B1.p1 \xrightarrow{l} B2.p2$ *in which* $p1$ *is a port of box* $B1$ *and* $p2$ *is a port of box* $B2$ *and* l *an action in* A.

The set of port labels of the box B_i *and the set of actions of links* $\mathcal{L}$ *are respectively denoted* $Actions(B_i)$ *and* $Actions(\mathcal{L})$.

Given a Net with n boxes and their corresponding LTSs, we construct the global behaviour of the network, using the synchronization product of [1]. Basically, each of the links is represented by a synchronization vector in Arnold's setting.

Definition 5 Synchronization Product.
Given a Net $N \triangleq (\mathcal{B}_n,\ \mathcal{L})$ *and* n *LTSs* $\{(S_i,\ s_{0i},\ L_i, \rightarrow_i)\}_{i=1..n}$, *we construct the product LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *where* $S \subset \Pi_{i=1..n}(S_i)$, $s_0 = \Pi_{i=1..n}(s_{0i})$, $L \triangleq Actions(\mathcal{L})$,
$\rightarrow \triangleq \{s \xrightarrow{\alpha} s' |\ s =< s_1, \ldots, s_n >, s' =< s'_1, \ldots, s'_n > and\ \exists\ l \in \mathcal{L}.\ l = B_{i_1}.p_{j_1} \xrightarrow{\alpha} B_{i_2}.p_{j_2},\ s_{i_1} \xrightarrow{B_{i_1}.p_{j_1}} s'_{i_1},\ s_{i_2} \xrightarrow{B_{i_2}.p_{j_2}} s'_{i_2}, \forall i \neq i_1, i_2.\ s'_i = s_i\}$

3 Informal Semantics of ProActive Distributed Applications

In the current version of the ProActive middleware, an application is a (flat) collection of *active objects*, that can be dynamically created, migrated, and terminated. We shall associate a *process* to each *active object* of the application, and build a synchronization network to represent the communication between active objects.

We are not interested here in properties related to the location of objects in the physical network, so we can discard all information on location and migration. We are interested in the creation and termination of objects, but we shall limit ourselves in this work to a static and finite topology of processes, that can be approximated by static analysis techniques, based on the primitives for active object creation, e.g.:

$ProActive.newActive\ (Class,\ Args,\ Node)$

Static analysis will also give us, for each process, the set of services it offers (*public methods*) and the set of services it uses (*method calls* to other active objects), allowing us to build the synchronization network between the objects.

The model of an active object is the product of two LTSs, the first encoding (a bounded approximation of) its pending request queue, the other representing its behaviour, that the user has programmed in the special *runActivity* method. The construction of this behaviour includes the unfolding of local method calls (of the active object itself, and of local passive objects); the termination of this procedure is obtained by detection of (mutually) recursive method calls modulo a finite abstraction of the call stack (see next section).

We have said that the events of our models are the requests and responses of remote methods. Consider an object *co* (the current active object), calling method m (with a non-void return type) of a remote active object ro:

$x = ro.m\ (arg1,\ \ldots,\ argn);$

We model these calls by message exchanges (Figure 2) between the two processes: *co* emits a request $!Req\_m$, that is instantaneously received by ro's queue as $?Req\_m$. The remote object ro may eventually decide to serve this request.

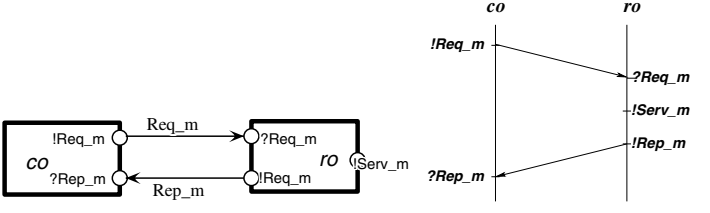


Fig. 2. Message exchanges between the processes co and ro

Starting the execution of the method call within object *ro* is modeled by a message *Serv_m* local to *ro*. And finally, when the method returns a result, *ro* emits a reply message !*Rep_m*, synchronously received by *co* as ?*Rep_m*.

From the *current object* point of view, the response is asynchronous : the *wait by necessity* semantics of *ProActive* says that the reply message can be received at any time between emission of the request and the first effective usage of the returned object, i.e. the access to a field or method of this object ; or an explicit awaiting of this result. We model this mechanism by interleaving the *response* message with the behaviour of *co* between these two points.

The *Request Queue* of an active object deals with arrival, service and disposal of requests. It is always ready to accept incoming messages. The synchronization with the *runActivity* LTS is done through the service primitives of ProActive, that allow the user to inspect and select specific requests in the queue.

4 Model Construction

An application is a set of active objects, modeled by LTSs in a compositional way. Active objects are connected by a synchronization network whose links represent the exchange of messages.

We use static analysis techniques to establish for each application :

- an enumeration of its active objects creation points and an enumeration of the links (the set of incoming and outgoing requests) between the objects,
- a method call graph for each active object class,
- a predicate *Active*(O) indicating for each object O belonging to this application whether O is a ProActive or a Java (passive) object.

For each active object class, or for each active object creation point, an application usually creates an arbitrary number of objects, dynamically. However, most interesting properties would involve only finitely many instantiations of any given active object class. So, we need to consider only a finite number of objects. The precision of the model (in term of number of different active objects, and granularity of observation on message contents) can be tuned depending on the proofs to be done.

Given a property to be proved, we require the user to provide an abstract interpretation for all data types in the application. This abstraction will retain only data information (active objects parameters and messages parameters) relevant to this property. In practice, tools like the Bandera abstract specification language can be used for defining this abstraction.

Property : Whenever all domains of this abstraction are finite, we obtain a network with finite topology, and the procedure in section 4.2 for computing the behavioural model is guaranteed to terminate.

4.1 Network Construction

Given a finite set of creation points, and a finite domain for active objects parameters, we obtain a finite active object enumeration, expressed as :

1. $\mathcal{O} = \{O_i\}$ a finite number of active object classes.
2. A finite number of instantiations for each class, depending on the possible values of the arguments passed at their creation, denoted $Dom(O_i)$.

For each such object, we build a Box $B(O_i, k)$, named by the object class O_i and an index k in $Dom(O_i)$.

Now, what we want to observe from method calls also depends on our abstraction. In the following we denote m a message containing : the method name, the destination object, the abstraction of the parameters.

Then for each active object class, we analyze the relevant source code (starting with the *runActivity* method of the object main class), to compute its ports and the corresponding links :

1. The set of public methods of the active object main class gives us the set of "receive request" ports $?Req\_m$ of its box.
2. We then identify in the code the variables carrying "active objects" values, using usual data flow analysis techniques, and taking into account their creation parameters. Amongst those objects, one is identified as the "current object", the others are "remote objects".
3. For each call to a method of the current active object, we add a "local" port m to the current box.
4. For each call to a method of a remote object, we add a "send request" port $!Req\_m$ to the current box, linked to the corresponding port of the remote object box, and if this method expects a result, a "response" port $?Rep\_m$.

The behaviour of an active object is programmed in its *runActivity* method ; it specifies how the requests arriving in the object *request queue* are served, and the requests that the object sends to others. The box of an active object is in turn structured in a runActivity LTS and a request queue LTS connected by a synchronization network. We build a box $B(\mathcal{A})$ for the runActivity LTS and a box $B(\mathcal{Q})$ for the queue. The ports and links computed between these boxes are determined by the set of public methods of this active object:

For each method m, a "send" port $!Serv\_m$ and a "receive" port $?Serv\_m$ are added on $B(\mathcal{A})$ and $B(\mathcal{Q})$ respectively, and a link between them. If m expects a non-void result, a "send" port $!Rep\_m$ is added on $B(\mathcal{A})$. An example of a generated Net is displayed in Figure 3.

The request queue of an active object runs independently of its body, and is always accepting arriving requests (of correct type). It is synchronized with the object body through the $Serv\_m$ actions, and with remote objects by their Req_m actions. The model of the queue is a LTS $\mathcal{Q}$ constructed in generic way by a graph grammar from the set of methods coming in it and its bound.

4.2 Behaviour Computation

In the following section we give a construction function $\Longrightarrow_A$ describing a traversal of the method call graph of an active object class (and all passive object classes/methods required), and the corresponding building of the LTS. This function is specified as a set of transition rules in SOS style:

$$\frac{\{Premise\}*}{< v = pattern,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \quad \Longrightarrow_A \quad < v',\ n',\ \mathcal{A}',\ \mathcal{M}',\ \mathcal{S}'_n,\ \mathcal{S}'_m >}$$

where

- $v = pattern$ is the current analyzed MCG node, together with its value,
- n is the last LTS node created,
- $\mathcal{A}$ the currently constructed LTS. The empty LTS is denoted by $\emptyset$, and the LTS reduced to a single node is denoted by this node itself,
- $\mathcal{M}$ is a mapping from MCG nodes already visited to LTS nodes,
- $\mathcal{S}_n$ a stack of MCG nodes; empty stack is denoted by [ ] and push on top of the stack is denoted by $v : \mathcal{S}$,
- $\mathcal{S}_m$ a method calls stack.

Definition 6 Sub-LTS. *A sub-LTS, of the LTS* $\mathcal{A} = (S,\ s_0,\ L\ , \rightarrow)$, *denoted* $\mathcal{A}^m_n$, *is the LTS* $\mathcal{A}'$ *defined by* $S' \subseteq S$, $L' \subseteq L$, $s'_0 = n$, $\rightarrow' = \{s_1 \xrightarrow{\alpha} s_2 \in \rightarrow | \exists x \in L^*.\ n \xrightarrow{x} s_1 \in \rightarrow^*$ *and* $\exists x' \in L^*.\ s_2 \xrightarrow{x'} m \in \rightarrow^*\}$

Definition 7 Interleaving operator. *Interleave a sub-LTS* $\mathcal{A}^{s_0}_{s_k}$ *with an action* α, *denoted* $\mathcal{A}^{s_0}_{s_k} \xrightarrow{\alpha}$, *is the pair a LTS and a node,* $\mathcal{A}'$ *defined by* $S' = \{s_l | s \in S\} \cup \{s_r | s \in S\}$, $L' = L \cup \{\alpha\}$, $s'_0 = s_{0l}$, $\rightarrow' = \{s_{1l} \xrightarrow{x} s_{2l}$ *and* $s_{1r} \xrightarrow{x} s_{2r} | s_1 \xrightarrow{x} s_2 \in \rightarrow\} \cup \{s_l \xrightarrow{\alpha} s_r | s \in S\}$

Definition 8 Connection operator. *Given two LTSs* $\mathcal{A}$ *and* $\mathcal{A}'$ *such that* $S \cap S' = \{s'_0\}$. *The connection of* $\mathcal{A}$ *and* $\mathcal{A}'$ *at* s'_0, *denoted* $\mathcal{A} \triangleleft \mathcal{A}'$, *is* $\mathcal{A}''$ *defined by* $S'' = S \cup S'$, $L'' = L \cup L'$, $s''_0 = s_0$, $\rightarrow'' = (\rightarrow \cup \rightarrow')$

We are ready now to describe the construction rules of the LTS corresponding to the active object's behaviour from the *runActivity* method MCG which encapsulates this behaviour. We start with:

$$< v = ent(runActivity),\ \emptyset,\ \emptyset,\ \mathcal{M},\ [\ ],\ [\ ] >$$

$$\frac{v_1 \rightarrow^T v_2 \quad fresh(init(n))}{\begin{array}{l}< v_1 = ent(runActivity),\ \emptyset,\ \emptyset,\ \mathcal{M},\ [\ ],\ [\ ] > \\ \Longrightarrow_A < v_2,\ n,\ n,\ \mathcal{M} \cup \{v_1 \mapsto n\},\ [\ ],\ runActivity : [\ ] >\end{array}} \text{(Ent\_Run)}$$

$$\frac{v_1 \rightarrow^C v_2 \quad v_1 \rightarrow^T v'_2 \quad \neg Active(O) \quad fresh(n')}{\begin{array}{l}< v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ v'_2 : \mathcal{S}_n,\ \mathcal{S}_m >\end{array}} \text{(L\_Call)}$$

The first method analyzed is the method named *runActivity*; at the entry point the initial LTS node is created (ENT_RUN rule).

For a call to a local method we construct a transition labelled with the name of this called method (L_CALL rule); afterwards the body of this method is inlined (its MCG is analyzed). The continuation node v_2' is save on the node stack for later analysis.

$$\frac{m \neq runActivity \quad Rec(v_1)}{< v_1 = ent(m),\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A}[n'/n],\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{(ENT1)}$$

$$\frac{m \neq runActivity \quad \neg Rec(v_1) \quad v_1 \rightarrow^T v_2}{< v_1 = ent(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2,\ n,\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_n,\ m : \mathcal{S}_m >} \text{(ENT2)}$$

where $Rec(v_1)$ is the predicate: $\mathcal{M}(v_1) = n' \wedge \exists name \in \mathcal{S}_m.\ v_1 \in \mathcal{D}(name)$

The next set of rules deals with the structure of local method invocation (method of passive objects of the current active object). Rule L_CALL has created a visible event corresponding to a method call, but we must now recognize recursive calls, using the mapping $\mathcal{M}$, and a finite abstraction of the method call stack (a method name appears only once in the stack). This is the role of the $Rec(v)$ predicate: if an entry node v has already been visited the $\mathcal{M}(v)$ is defined. And if its method name is present in the call stack, then we have detected a recursive call, and we build the corresponding cycle in the generated LTS. Otherwise the name of the method is pushed on the call stack, and we continue analyzing this method.

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad v_1 \rightarrow^T v_3 \quad \ldots \quad v_1 \rightarrow^T v_n}{< v_1 = seq,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n, \mathcal{S}_m > \Longrightarrow_A < v_2, n,\ \mathcal{A}, \mathcal{M} \cup_i \{v_i \mapsto n\}, v_3 : \ldots : v_n : \mathcal{S}_n, \mathcal{S}_m >} \text{(SEQ)}$$

$$\frac{\mathcal{M}(v_1) = n'}{< v_1,\ n,\ \mathcal{A},\ \mathcal{M}, v : \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A}[n'/n],\ \mathcal{M}, \mathcal{S}_n,\ \mathcal{S}_m >} \text{(LOOP-JOIN)}$$

Sequence nodes represent sequential instructions and branching. Thus no transition is constructed. In the case of branching, the alternative nodes are pushed onto the stack nodes for later visit (SEQ rule). The (LOOP-JOIN rule) applies to all types of nodes that already have been visited. Then the corresponding LTS node is substituted to the current LTS node, eventually creating loops or joins in the LTS.

At a return node, we check the nodes waiting on the node stack. If they belong to the same method we go and process them (RET1 rule). If not, we pop this method from the call stack and we remove its nodes from the mapping before

$$\frac{\mathcal{M}(v) = n'}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{(RET1)}$$

$$\frac{v \notin \mathcal{D}(m)}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ m : \mathcal{S}_m > \Longrightarrow_A < v,\ n,\ \mathcal{A},\ \mathcal{M} \backslash \mathcal{D}(m),\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{(RET2)}$$

continuing the analysis (RET2 rule). Note that those rules will not apply if the node stack is empty ; this case will terminate the LTS construction procedure.

$$\frac{v_1 \rightarrow^T v_2 \quad v_2 = call(m) \quad fresh(n')}{\begin{array}{l} < v_1 = call(serve),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Serv\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_n,\ \mathcal{S}_m > \end{array}} \text{(SERV)}$$

This is a call to one of the *service* primitives of the library, e.g. *blockingServeOldest(m)*. The service primitives allow to select a request in the queue and run it. The call graph constructed in this case has a first *call(serve)* node without a call edge (we do not represent the serve primitive itself), immediately followed by a *call(m)* node. So we simply construct a transition labelled !*Serv_m*.

$$\frac{v_1 \rightarrow^T v_2 \quad fresh(n')}{\begin{array}{l} < v_1 = rep(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Rep\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_n,\ \mathcal{S}_m > \end{array}} \text{(REP)}$$

When a request returns a non-void result, the control graph will have a special *rep* node following the *call(serve); call(m)* sequence. At this point we construct a transition labelled !*Rep_m*, the emission of response to the remote caller (REP rule).

$$\frac{v_1 \rightarrow^T v_2 \quad Active(O) \quad fresh(n')}{\begin{array}{l} < v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Req\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_n,\ \mathcal{S}_m > \end{array}} \text{(R\_CALL)}$$

In the case of the call to a remote method, the call graph has no call edge (because it was built separately for each active object). We construct a transition holding the label !*Req_m*, emission of the request named by the method name towards this remote object (R_CALL rule).

Processing futures. One interesting feature of *ProActive* is the notion of wait-by-necessity : it is only when the result of a method call (future value) is used

that this result is awaited. After completion of the LTS production with the preceding rules, we apply the (FUT rule) below at each future utilisation point: in the MCG the μ mapping links each utilisation point to its creation point.

$$\frac{\mu(v_1) = v_2 \quad n = \overline{\mathcal{M}}(v_1) \quad n' = \overline{\mathcal{M}}(v_2) \quad \mathcal{A}' = (\mathcal{A}^n_{n'} \xrightarrow{?Rep\_m})}{< v_1,\ \mathcal{A} > \Longrightarrow_F < v'_1,\ \mathcal{A} \triangleleft \mathcal{A}' >} \text{ (FUT)}$$

The effective return of the future value (the transition labelled $?Rep\_m$), is interleaved with the part of the LTS comprised between its definition point and its first utilisation points (FUT rule). $\overline{\mathcal{M}}$ is the phantom of the $\mathcal{M}$ mapping, and its is defined as the union of $\mathcal{M}$s at each step of the construction procedure.

5 Example

We illustrate our work on an example borrowed from work made with the Chilean administration to provide electronic procedures for their sales and taxes system. It involves software pieces that will run at different locations (government and companies) and communicate through the Internet. The specification involves both safety (protocols) and security (authentification, secrecy, ...) aspects.

The whole system is huge in size, and we shall only describe here a small part of the *Vendor* process, in charge of managing the invoice's *Stamps* delivered by the administration (SII) to a given Vendor. This Vendor subsystem is composed of 4 components; we concentrate here on the dialog between the "StampStock" component and the corresponding SII component : the vendor requires a number k of "stamps", and the SII will eventually answer by sending an authorization code for a number x of stamps.

We have proven a number of temporal properties on this system, for example that "it's not possible to cancel an invoice which has never been emitted" or " Every buyer which makes a purchase will eventually receive an invoice".

Now we give the code of a ProActive implementation of a StampStock object. It receives external *Stamp* requests. It is more specific than the specification : it asks for new stamps immediately after having received the previous stamps. The response from SII can arrive at any time, but will only be used when the stock is down to 0 (last `else` branch).

```
public class StampStock implements RunActive {
 public Boolean stamp() {
                stock--;
                return (new Boolean(true)); }
 public void runActivity(Body body) {
                Service service = new Service(body);
                boolean stampsRequested = false;
                Integer newStamps = new Integer(0);
                while (body.isActive()) {
                      if (!stampsRequested) {
                                  newStamps = sii.NewStamps();
                                  stampsRequested = true; }
                      else if (stock > 0)
                                  service.blockingServeOldest();
                      else {int ns = newStamps.intValue();
                            stock = stock + ns;
                            stampsRequested = false; }}}}
```

The model generated for the StampStock and SII classes, for a small instantiation of the object parameters (stock ¡= 2), is shown in Figure 3. The queue of StampStock has been arbitrarily bounded to 2, while the queue of SII can be provably bounded to 1. It can be shown that this implementation is correct versus its specification.

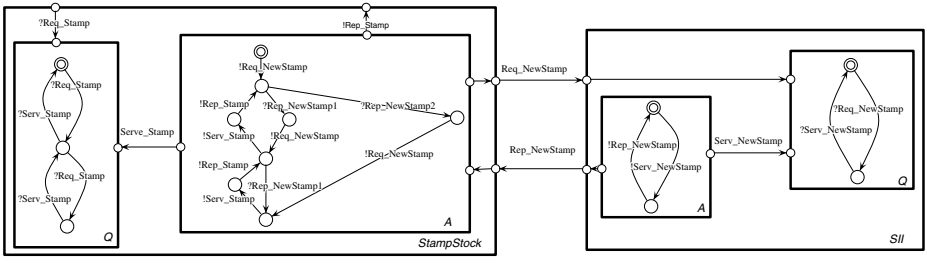


Fig. 3. Constructed models for StampStock and SII

6 Conclusion

We have sketched a method for constructing asynchronous models for distributed Java applications built with the *ProActive* library. The models are hierarchical labelled transition systems, suitable for analysis with verification tools based on bisimulation semantics. The behaviour of distributed objects is obtained by SOS-style rules, from the method call graph of the application. We ensure that thank to a finite abstraction we obtain a finite model.

We have illustrated this approach on a distributed implementation of the Chilean invoice and tax declaration system and we have checked some properties. We are working on relating our behavioural semantics with the operational model of ProActive [8], to get correctness results.

One of the crucial hypotheses used in this work is the ability to enumerate by static analysis a finite number of distributed objects. This allows us in practice to prove many useful properties. But there are now a number of verification tools, either automatic or partially automatic (e.g. Moped, Trex, or Harvey), that can deal with some kind of finite representations of infinite systems, or with parametrised systems. We are working on an extension of our intermediate model (LTSs and Nets) that supports parametrised systems ; the models will be more compact than in the fully instantiated version, and may be directly user with this new generation of tools.

We shall also extend the approach to take into account other features of the middleware, and in particular the primitives for group communication, and for specifying distributed security policies. Last but not least, ProActive active objects are also viewed as *distributed components* in a component framework. In the next version, it will be possible to assemble distributed objects to form more complex components. This will increase the impact of the compositionality of our model, and the importance of being able to prove that a component agrees with its (behavioural) specification.

References

1. A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.
2. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop*, LNCS 2057. Springer-Verlag, 2001.
3. D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1-50, 1996.
4. R. Boulifa and E. Madelaine. Finite model generation for distributed Java programs. In *Workshop on Model-Checking for Dependable Software-Intensive Systems*, San-Francisco, June 2003. North-Holland.
5. D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.
6. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. *Int. Conference on Software Engineering (ICSE)*, 2000.
7. P. Cousot and R. Cousot. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, Copenhagen, Denmark, LNCS 2404, pages 37–56. Springer-Verlag Berlin Heidelberg, 27–31 July 2002.
8. L. Henrio D. Caromel and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages.* ACM Press, 2004. To appear.

9. J. Dean D. Grove, G. DeFouw and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented*, pages 108–124, 1997.
10. J. Hatcliff M. Dwyer and H. Zheng. Slicing software for model construction. *Journal of High-order and Symbolic Computation*, 2000.
11. V. Roy and R. de Simone. Auto and autograph. In *Workshop on Computer Aided Verification, New-Brunswick*, pages 65–75. LNCS 531, Spring-Verlag, June 1990. also available as RR-4460.

Software Inspections We Can Trust

David Parnas

University of Limerick, Limerick, Ireland
david.parnas@ul.ie

Abstract. Software is devilishly hard to inspect. Serious errors can hide for years. Consequently, many are hesitant to employ software in safety-critical applications and all companies are finding correcting and improving software to be an increasingly burdensome cost.
This talk describes a procedure for inspecting software that consistently finds subtle errors in software that is believed to be correct. The procedure is based on four key principles:

- All software reviewers actively use the code.
- Reviewers exploit the hierarchical structure of the code rather than proceeding sequentially through the code.
- Reviewers focus on small sections of code, producing precise summaries that are used when inspecting other sections. The summaries provide the "links" between the sections.
- Reviewers proceed systematically so that no case, and no section of the program, gets overlooked.

During the procedure, the inspectors produce and review mathematical documentation. The mathematics allows them to check for complete coverage; the notation allows the work to proceed in small systematic steps.
Concurrent programs are especially difficult to inspect. We will talk briefly about recent research on how to apply this approach to concurrent programs.

Presenter. David Lorge Parnas is Professor of Software Engineering, SFI Fellow and Director of the Software Quality Research Laboratory at the University of Limerick.
Professor Parnas received his B.S., M.S. and Ph.D. in Electrical Engineering - systems and Communications Sciences from Carnegie Mellon University and honorary doctorates from the ETH in Zurich and the Catholic University of Louvain.
Professor Parnas won an ACM "Best Paper" Award in 1979, and two "Most Influential Paper" awards from the International Conference n Software Engineering, the 1998 ACM SIGSOFT "Outstanding Research Award", the "Practical Visionary Award" in honour of the late Dr. Harlan Mills, and the "Component and Object Technology" Award presented at TOOLS99. He was the first winner of the Norbert Wiener Prize from Computing Professionals for Social Responsibility and recently won the FifF prize from Forum Informatiker für Frieden und Verantwortung in Germany.

N. Guelfi, E. Astesiano, G. Reggio(Eds.): FIDJI 2003, LNCS 2952, pp. 153–154, 2004.

Dr. Parnas is a Fellow of the Royal Society of Canada and the Association for Computing Machinery (ACM) and is licensed as a Professional Engineer in Ontario. Prof. Parnas has also taught at McMaster University (where he was the first Director of the Software Engineering Programme), the University of Victoria, the Technische Hochschule Darmstadt, the University of North Carolina at Chapel Hill, Carnegie Mellon University and the University of Maryland. He has held non-academic positions at Philips Computer Industry (the United States Naval Research Laboratory in Washington, D.C. and the IBM Federal Systems Division. He has advised the Atomic Energy Control Board of Canada on the use of safety- critical realtime software at the Darlington Nuclear Generation Station.

J2EE and .NET: Interoperability with Webservices

Alain Leroy

Microsoft Corporation
alainler@microsoft.com

Abstract. The next generation of distributed computing has arrived. Over the past few years, XML has enabled heterogeneous computing environments to share information over the World-Wide Web. It now offers a simplified means by which to share process as well. From a technical perspective, the advent of web services is not a revolution in distributed computing. It is instead a natural evolution of XML application from structured representation of information to structured representation of inter-application messaging. The revolution is in the opportunities this evolution affords. Prior to the advent of web services, enterprise application integration was very difficult due to differences in programming languages and middleware used within organizations. The chances of any two business systems using the same programming language and the same middleware was slim to none, since there has not been a de-facto winner. These 'component wars' spelled headaches for integration efforts, and resulted in a plethora of custom adapters, one-off integrations, and integration 'middlemen'. In short, interoperability was cumbersome and painful. With web services, any application can be integrated so long as it is Internet-enabled. The foundation of web services is XML messaging over standard web protocols such as HTTP. This is a very lightweight communication mechanism that any programming language, middleware, or platform can participate in, easing interoperability greatly. These industry standards enjoy widespread industry acceptance, making them very low-risk technologies for corporations to adopt. With web services, you can integrate two businesses, departments, or applications quickly and cost-effectively.

Presenter. Alain Leroy has been working for large enterprises for about 12 years; the first 4 of them as a designer in an R&D department, and thereafter, as consultant and trainer for multinational organizations. He has spent most of his time focusing on large scale heterogeneous, distributed environments and the related technologies. He has contributed to projects for major banks and insurance companies, for the Belgian government and for many other industries. After 4 years at Microsoft working in the Consulting department first as Application Architect and finally as Solution Architect, he has taken over the responsibility of Principal Advisor for Platform Architectures.

N. Guelfi, E. Astesiano, G. Reggio(Eds.): FIDJI 2003, LNCS 2952, p. 155, 2004.

Author Index

Althun, Björn 23
Arciniegas, Jose L. 1

Bermejo, Jesús 1
Berner, David 33
Bettini, Lorenzo 12
Boulifa, Rabéa 139

Cerón, Rodrigo 1
Chabarek, Fadi 96
Chen, Jessica 48

Delamaro, Márcio 73
Dez, Bruno Le 33
Dokovsky, Nikolay 62
Dueñas, Juan C. 1

Erdogan, Nadia 106
Ewetz, Hans 86

Frey, Hannes 116

Gamatié, Abdoulaye 33
Giese, Holger 128
Görgen, Daniel 116
Guernic, Paul Le 33

Halteren, Aart van 62

Lehnert, Johannes K. 116
Leicher, Andreas 96
Leroy, Alain 155

Madelaine, Eric 139

Parnas, David 153

Ruiz, Jose L. 1

Sharif, Niloufar 86
Sturm, Peter 116
Süß, Jörn Guy 96

Talpin, Jean-Pierre 33
Tichy, Matthias 128

Vincenzi, Auri Marcelo Rizzo 73

Wang, Kun 48
Widya, Ing 62

Yilmaz, Guray 106

Zimmermann, Martin 23