Unified Approach for System-Level Generative Programming

Zhisheng Huang, Rajeev R. Raje,
Andrew M. Olson
Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, USA
{zhuang, rraje, aolson, csun}@cs.iupui.edu,
+1 317 274 5246/5174/9733

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu, +1 505 646 5286

Barrett R. Bryant, Carol Burt
Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, Alabama 35294-1170, USA
{bryant, cburt}@cis.uab.edu, +1 205 934 2213

Changlin Sun
Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, USA
csun@cs.iupui.edu, +1 317 274 5246

Abstract

Today's and future distributed software systems will certainly require combining heterogeneous software components that are geographically dispersed so that its realization not only meets the functional requirements, but also satisfies the non-functional criteria such as the desired quality of services (QoS). The Unified Approach (UA) incorporates the concepts of product line practice (PLP) and generative programming with the Unified Meta-component Model (UMM) to achieve automatic development, maximal reuse and seamless interoperation. The creation of a software solution for a distributed computing system (DCS), using the UA has two levels, the component level and the system level. In this paper, the system-level generative programming of the UA is described.

Keywords: Distributed Computing Systems, Heterogeneous Components, Quality of Services, Generative Programming, Generative Domain Model, Two-Level Grammar.

1. Introduction

As distributed computing becomes more and more crucial for the success of today's enterprises, there is an increasing need to develop software for a distributed computing system (DCS) in an effective and efficient way. A lot of distributed computing systems are still designed and built as single systems. This approach has the problems of large investment, long development cycles, difficulties in the system integration, and a lack of predictable quality. Generative programming [7] and product line practice (PLP) [19] help us to move the focus from the development of single systems to system families. The use of components to develop software for a DCS is consistent with the notions of generative programming and PLP. However, another challenge arises as component-based software development is applied to distributed computing. This challenge is an effect of the presence of multiple component models. Currently, different component models have been proposed, such as Java^{TM} Remote Method Invocation (RMI) [13], Common Object Request Broker Architecture (CORBA^{TM}) [11, 13, 17], and the Distributed Component Object Model (DCOM^{TM}) [10]. There are difficulties in bridging the components of different models, thus reducing the component reuse. The Unified Meta-Component Model Framework (UniFrame) research [14, 15, 16] is an attempt to unify the existing and emerging distributed component models under a common meta-model, the Unified Meta-component Model (UMM), for the purpose of enabling the discovery, interoperability and collaboration of components via a Unified Approach (UA). The UA is a UMM-based technique, which incorporates some ideas from generative programming and PLP. It replaces the manual search for, and adaptation and assembly of, heterogeneous and distributed components with automation. The aim is to develop a quality-oriented and time-to-market DCS with lower development and maintenance costs. The creation of a software realization of a DCS using the UA has two levels: a) the component level – component development and deployment, and b) the system level – automatic or semiautomatic system generation.

This paper describes the UA at the system level. The principles of generative programming, PLP and the UniFame are briefly described in the next section. Section 3 discusses, in detail, the system-level generative programming of the UA, which is illustrated by an example in section 4. The paper concludes in section 5.

2. Related work

2.1 Generative programming

The generative programming is concerned with bringing automation to the software development. In [7] the generative programming paradigm is defined as: "Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse). This model includes also a domain-specific software generator; b) given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)". The methods presented in [7] can be applied both "in the small", i.e., at the level of classes and procedures and "in the large", to develop families of large systems.

2.2 PLP

In 1997, the PLP initiative [19] was launched by the Software Engineering Institute (SEI) of Carnegie Mellon University. The intention was to help facilitate and accelerate the transition from the traditional single system development to sound software engineering practices using a product line approach. A software product line is defined to be a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a selected market or mission, and that are developed from a common set of core assets in a prescribed way [5, 6]. The SEI's PLP Framework is the first formal attempt to codify the comprehensive information about successful product lines. The idea behind this framework is to identify the different issues and practices relevant to establishing and running successful product lines in an organization. More information can be found on the PLP Framework website [20].

2.3 UniFrame

The UniFrame provides a framework for constructing a DCS by integrating the heterogeneous and distributed software components. It consists of the Unified Meta-component Model (UMM) and the Unified Approach (UA).

2.3.1 UMM. The recent shift in the focus of Object Management Group (OMG) to Model Driven Architecture (MDA) [12] is a recognition that bridging components to create DCS requires standardization of not only the infrastructure but also Business and Component Models. The UMM provides an opportunity to bridge gaps that currently exist in the standards arena. The core parts of the UMM are: components, service and service guarantees, and infrastructure. In UMM, components are autonomous entities. All components have well-defined interfaces and private implementation. In addition, each component in UMM has three aspects: a) computational aspect, b) cooperative aspect, and c) auxiliary aspect. Each component must be able to specify and guarantee the quality of service (QoS) offered. The headhunter [18] and Internet Component Broker (ICB) [15, 18] of the infrastructure are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous components (adhering to different models). The headhunter is responsible for searching and managing heterogeneous and geographically distributed components. The ICB acts as a translator between two heterogeneous components. An ICB itself is a component defined under the UMM. It achieves interoperability using the principles of wrap and glue technology [9]. An example of ICB is a Java – CORBA bridge, which bridges a component of Java RMI technology and a component of CORBA technology. For a detailed description of UMM, see [14, 15, 16].

2.3.2 UA. The UA is the UMM-based technique for the automatic production of a DCS. The creation of a software realization of a DCS using UA has two levels: a) the component level - components are designed and developed with UMM specifications (which are informal in nature [14]), tested and validated against the appropriate QoS, then deployed on the network, and b) the system level – a semi-automatic or automatic generation of a specific DCS product from a DCS family. The concepts of generative programming are applied at both levels in the UA. This paper describes the application of generative programming at the system level.

3. System-level generative programming of the Unified Approach (UA)

3.1 Core activities in the UA

The UA has four core activities for building distributed systems. These are: *generative domain engineering*, *component engineering*, *generative application engineering*, and *active distributed component management*. Their relationships are depicted in Figure 1. The development process is iterative and there are feedbacks during the first three activities. These four core activities span both the levels of UA: the component level and the system level. The first two activities, *generative domain engineering* and *component engineering*, corresponding to the domain engineering in [7], aim at maximizing the reuse of both the components and the software architecture. We distinguish between these two activities because they reflect the different levels in the UA. G*enerative domain engineering* is a system-level activity and the *component engineering* is at the component level. G*enerative application engineering* is another system-level activity. A*ctive distributed component engineerin*g is involved at both levels.

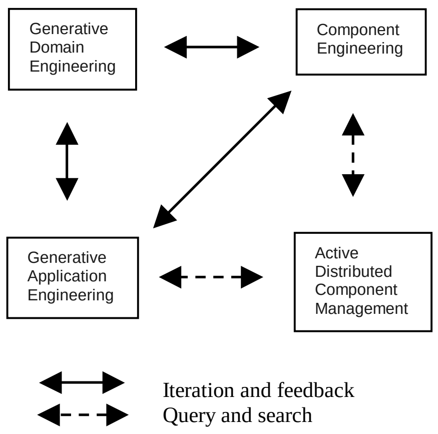
Generative Domain Engineering
Component Engineering
Generative Application Engineering
Active Distributed Component Management
Iteration and feedback
Query and search

Figure 1. UA core activities

Generative domain engineering consists of activities for identifying commonalities and variations of the system architecture of a DCS family. It is responsible for creating the generative domain model (GDM), which is discussed in 3.2, to represent a configurable system architecture. This architecture includes a set of abstract components as the guidelines for developing reusable concrete components during *component engineering* phase. Each abstract component represents one component type and is defined with its UMM specification. This specification is natural language-like and includes both the functional and nonfunctional (such as expected QoS properties) aspects of a component [14]. This specification is then refined into a formal specification, based upon the theory of Two-Level Grammar (TLG) [4] and natural language specifications [3]. The GDM is the core software asset that results from *generative domain engineering*.

During *component engineering* phase, the abstract components are mapped to different component models to create concrete components. The concrete components are tested and validated against the appropriate QoS, deployed over the network, and then are discovered by the headhunters. It is worthwhile to note that the generative programming is also carried out in the *component engineering* phase of the UA.

Generative application engineering is the process of building a DCS based on a GDM. It is supported by the query processor (see explanation in section 3.4) and *active distributed component management*. During *generative application engineering*, a DCS is produced out of a DCS family in three steps: a) determining the target system and its architecture instance according to the system specification produced by the query processor; b) searching for concrete components for the target system via the headhunter; and c) assembling and testing the DCS according to the architecture instance to produce a workable distributed system that meets both the functional and non-functional requirements. The GDM is used to guide the system assembly and validation. The validation of the QoS requirements is carried out both by QoS composition rules [21], which specify how the system QoS or subsystem QoS can be composed from the QoS of its parts, and by the event grammars [1, 2], which are used as the basis for the system behavior models to trace events like executing a statement or calling a procedure. The example in the next section illustrates these steps.

Active distributed component management is the UniFrame resource discovery service (URDS) [18]. It offers the dynamic discovery and management of the heterogeneous software components and assists in the finding of the required components during the phase of the *generative application engineering*. These are achieved by headhunters, which are analogous to binders or traders in other models, with one difference - a trader is passive, while a headhunter is active. For details, see [18].

3.2 UA GDM

The key to automating the manufacturing of systems is a GDM, which consists of a problem space, a solution space, and the configuration knowledge mapping between them [7]. The problem space consists of the application-oriented concepts and features that application developers can use to express their needs. UA GDM contains a Design Space Model (DSM) to represent the common and variable properties of a software architecture and a set of abstract components as guidelines for creating reusable distributed components. The DSM is an important part of the problem space. DSM describes the configurable software architecture with feature notations as described in [7], but, additionally, classifies the architectural nodes that are divided into five types: *domain*, *system*, *subsystem*, *design* and *abstract component*. In the graphical representation of the GDM, these node types are represented by surrounding the name of each node with << >>, < >, ( ), { }, and [ ] respectively. Associated with each node type is a standardized description, such as the UMM description for an abstract component. With the introduction of node types, a configurable system architecture can be easily represented. The description associated with a node shows information such as the relationship between its constituents (its children in the DSM). A simple example of a DSM is described in the next paragraph. The solution space consists of concrete components developed during component engineering when abstract components are mapped to specific component models and implemented. The configuration knowledge includes, as stated in [7], illegal feature combinations, default settings, default dependencies, construction rules and optimization rules, etc. In UA, it also includes additional important knowledge, such as, QoS composition and decomposition rules [21], which help ensure the assembled distributed system meets not only the functional requirements but also the non-functional requirements.

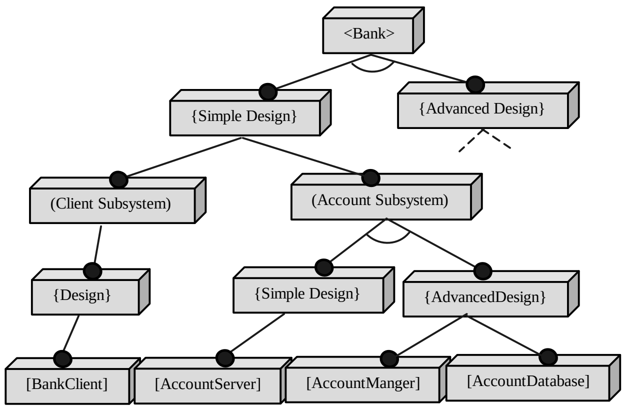

Figure 2. UA DSM for an account management system

Figure 2 shows a simplified example of a DSM for an account management system. In this DSM, two kinds of feature notations are used: mandatory and alternative. A node is mandatory if a simple edge ends with a filled circle touching it. This means this node is included in an architecture instance if and only if its parent is included. A set of nodes that is pointed to by edges connected by an arc forms alternatives. This means that if the parent of this set of nodes is included in an architecture instance, then exactly one node from this set is included in the architecture instance. The details of feature notations are indicated in [7]. The root of the example DSM is <Bank>, which indicates the specific type of account management system being considered. It has two different designs: a {Simple Design} and an {Advanced Design}. The details of the {Advanced Design} are omitted in the figure for simplicity. The {Simple Design} of the <Bank> has two subsystems: the (Client Subsystem) and the (Account Subsystem). These subsystems also can have more than one design that have different kinds of abstract components as shown in the Figure 2. Thus, this architecture can be configured, based on a customer's requirements, to create an appropriate architecture instance. One example of the customized architecture instance of this DSM is shown in Figure 3. Both the DSM and the architecture instance serve as the example in Section 4.

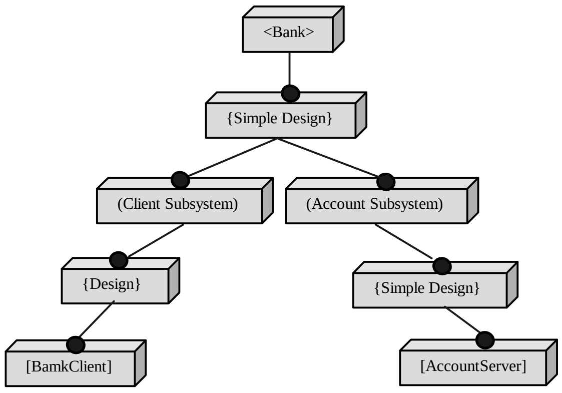

Figure 3. Architecture instance for an account management system

3.3 Language for ordering a DCS

Another important aspect of system level generative programming is how to express the query to order a concrete system out of a system family. [7] discusses the use of a domain specific language (DSL), which is a specialized and problem-oriented language, for placing an order. DSL could be a separate textual language, such as SQL, or it could be in a graphical notation. In general, there is a need for several different DSLs to specify a complete application. This makes the "order" complex. In UA, the ordering of a concrete system can be expressed in a structured form of natural language and then processed into TLG with the help of the query processor. TLG allows queries over the GDM to be expressed in a natural language-like manner, which is consistent with the way in which UMM is expressed. An example of a query for ordering a DCS is presented in Section 4.

3.4 UA generator

UA generator is a tool for realizing system-level generative programming. This generator is for system generation instead of component code generation. The architecture of he UA generator is shown in Figure 4. It consists of three functional modules: a generative domain model knowledgebase (GDMKB) producer; a query processor (natural language parser), which is responsible for translating natural language like orders into system specifications using TLG [8]; and an application producer which is responsible for assembling a DCS from a DCS family based on the UA GDM. The application producer implements the processing logic of the GDM. In our design, we separate UA GDM from the processing logic of GDM. The merit of this approach is that as a GDM evolves, the only thing that needs to be updated and maintained is the GDMKB. A simple generator for prototyping purposes has been designed and implemented with the logic of a multi-tiered architecture: client tier (web browser, HTML pages), web tier (web server, JSP/Servlet), business tier (application server, generator logic) and database tier (UA GDMKB). Experiments are underway with this prototype. The initial results indicate a good promise in a semi-automatic construction of simple distributed systems.

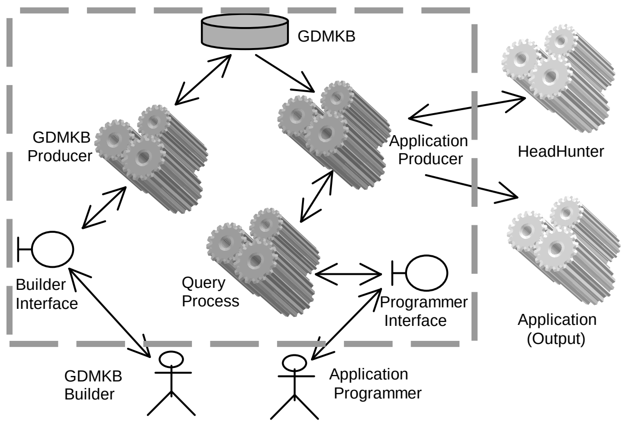

Figure 4. UA generator architecture

4. An example

In order to illustrate the process of the UA system-level generative programming, along with the functions of each of its constituents, a simple example of a bank account management system from the finance domain is described below. The DSM for this example is shown in Figure 2. This DSM constitutes four types of abstract components, [BankClient], [AccountServer], [AccountManager] and [AccountDatabase]. The goal is to assemble an account management system from the available concrete components of these abstract components using the corresponding generative domain model.

4.1 Determining the target system and its architecture instance

The general form of a query is to request the creation of a system that has certain QoS parameters. The name of the system is important in identifying the application domain. A sample query for the above example can be informally stated as: *Create a bank system for account management that has: end-to-end delay < 15 milliseconds and throughput > 2500 operations/second*. This query is parsed into a formal specification by the query processor. The generator checks the specifications against the GDM and may prompt for more information from the user, such as design option in this case (this is an iterative process to collect enough user requirements to determine the target system and its architecture instance).

Assume a simple design is specified for both the <Bank> and the (account subsystem) (certainly the UA generator provides the specifications from the GDM about the simple design and the advanced design so the application programmer can decide which one to choose). Then the generator can determine the architecture instance for the specified system and, thus, the required component types are also determined as seen in Figure 3. In this case, two types of component are needed to produce the desired system: [BankClient] and [AccountServer].

4.2 Searching for the concrete components

During this step, from the query and the available information in the DSM about the set of the required abstract components, searching criteria (for both functional and nonfunctional features) for each component type is created. In this example, the QoS of the two abstract components are set according to the QoS decomposition rules in this DSM: *1) component throughput > system throughput; 2) component end-to-end delay < system end-to-end delay*.

These decomposition rules provide the broadest range for the component QoS based on the system QoS. Thus the QoS criteria for both components are: a) throughput > 2500 operations/seconds; b) end-to-end delay < 15 milliseconds. Then the headhunters are contacted to search for the concrete components. If the found components are implemented with different technologies, the headhunters will also return the appropriate ICB. In this example, assume the headhunters discover the following concrete components for each of the required component types and the necessary ICB, Java-CORBA bridge (also a component type as described in section 2.3.1). Suppose these concrete components are implemented with different technologies: Java RMI or CORBA, and have different advertised QoS.

1. [BankClient]
   (a) BankClient - id: phoenix.cs.iupui.edu, technology: Java RMI, end-to-end delay < 10 milliseconds, throughput > 3000 operations/second
   (b) BankClient - id: lalo.cs.iupui.edu, technology: CORBA, end-to-end delay < 15 milliseconds, throughput > 2500 operations/second
2. [AccountServer]
   (a) AccountServer – id: swordmaster.cs.iupui.edu, technology: Java RMI, end-to-end delay < 5 milliseconds, throughput > 12000 operations/second
   (b) AccountServer – id: magellan.cs.iupui.edu, technology: CORBA, end-to-end delay < 1 millisecond, throughput > 8000 operations/second
3. [Java-CORBA bridge]
   Java-CORBA bridge – id: ericsson.cs.iupui.edu, technology: Java RMI, end-to-end delay < 1 millisecond, throughput > 10000 operations/second

4.3 System assembling and testing

Now the generator can assemble four systems (BankClient – AccountServer) from components found above. These four systems are distinguished by the implementation technology of its constituent components: Java RMI – Java RMI system, Java RMI – CORBA system, CORBA – Java RMI system and CORBA – CORBA system. The system QoS is composed from the QoS of the concrete components. The system QoS is used to select the final product. Assume the following composition rules for this example: *1) system throughput = min (component throughput); 2) system end-to-end delay = $\sum$component end-to-end delay.*

The system QoS of the four possible systems are listed below.

1. Java RMI – Java RMI system QoS
   system end-to-end delay < 15 milliseconds
   system throughput > 3000 operations/second
2. Java RMI – CORBA system QoS
   (including the Java-CORBA bridge)
   system end-to-end delay < 12 milliseconds
   system throughput > 3000 operations/second
3. CORBA – Java RMI system QoS
   (including the Java-CORBA bridge)
   system end-to-end delay < 21 milliseconds
   system throughput > 2500 operations/second
4. CORBA – CORBA system QoS
   system end-to-end delay < 16 milliseconds
   system throughput > 2500 operations/second

Based on the query and the analysis above according to the QoS composition rules, it is obvious the second system (Java RMI – CORBA) is the best. The first one (Java RMI – Java RMI) also meets the QOS requirement of the query. At this moment, the systems are chosen according to the advertised QoS of each component by QoS composition rules. The systems are further verified by the event grammars [2]. During system assembly, the code for carrying out event trace computations according to user-supplied test cases is also assembled. These test cases will be executed to verify that the assembled account management system does satisfy the system QoS specified in the query. If it does not, it is discarded. This verification process is carried out for each of the generated account management systems (the first two in the above example). The one with the actual best system QoS is chosen. If none of the systems meet the QoS criteria (as observed by an experimental evaluation), then the user may choose to modify the query and repeat the entire search, assembly and verification process.

5. Conclusion

The software solutions for the future DCS will require automatic or semi-automatic integration of software components, while abiding by the QoS constraints advertised by each component and the requirements on the system of components. This paper describes the system-level generative programming of the UA in the UniFrame that allows an effective and efficient assembly of heterogeneous and distributed software components to create a DCS out of a DCS family. The result of using the UniFrame and the associated tools (such as the UA generator) leads to the automation of DCS production while meeting both the functional and non-functional requirements of the DCS. Although a simple example is provided in this paper, the principles of the proposed approach are general enough to be applied to larger DCS.

6. Acknowledgement

This material is based upon work supported by the US Office of Naval Research under award number N00014-01-1-0746.

References

[1] Auguston, M. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 2nd Inernational Workshop on Automated and Algorithmic Debugging*, pages 277-291, 1995.

[2] Auguston, M., Gates, A., Lujan, M. Defining a Program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pages 257-262, 1997.

[3] Bryant, B.R. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 30-February 4, 2000, Canberra, Australia,* pages 24-30, January 2000.

[4] Bryant, B. R., Lee, B.-S. Two-Level Grammar as an Object-Oriented Requirements Specification Language, *Proceedings (CR-ROM) of 35th Hawaii International Conference on System Sciences, 2002,* page 10. http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDF documents/STDSLO1.pdf.

[5] Clements. P., Donohoe. P., Kang, K., Northrop, L. Fifth Product Line Practice Workshop Report. September, 2001. http://www.sei.cmu.edu/publications/documents/01.reports /01tr027.html.

[6] Cohen, S., Gallagher, B., Fisher, M., Jones, L., Krut, R., Northrop, L., O'Brien, W., Smith, D., Soule, A. Third DoD Product Line Practice Workshop Report. July 2000. http://www.sei.cmu.edu/publications/documents/00.reports /00tr024.html.

[7] Czarnecki, K., Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[8] Lee, B.-S., Bryant, B. R. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proceedings of SAC 2002, the 2002 ACM Symposium on Applied Computing, March 11-14, 2002, Madrid, Spain, 2002*, pp. 932-936.

[9] Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston, B.R. Bryant and B.K. Kin. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of the 12th IEEE International Workshop on Rapid System prototyping, pp.103-109, June 25-27, 2001, Monterey Beach Resort, California, USA,* IEEE Computer Society Press, 2001.

[10] Microsoft Corporation. DCOM Specifications, URL: - http://www.microsoft.com/oledev/olecom, 1998.

[11] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999. http://www.omg.org/cgi-bin/doc?orbos/99-02-05.

[12] Object Management Group (OMG). Model Driven Architecture: A Technical Perspective. Technical Report, OMG Document No. ab/2001-02-01/04, February 2001. ftp://ftp.omg.org/pub/docs/ab/01-02-04.pdf.

[13] Orfali, R., and Harkey, D. Client/Server Programming with JAVA and CORBA. The second edition. John Wiley & Sons, Inc., 1998.

[14] Raje, R. R., Bryant, B. R., Auguston, M., Olson, A M., Burt, C. C. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, Monterey, California, 2001,* pp: 109-119.

[15] Raje, R. R., Auguston, M., Bryant, B. R., Olson A. M., Burt, C. C. A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components. Submitted for publication to Concurrency and Computation, 2001.

[16] Raje R. R. UMM: Unified Meta-object Model. *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000, pp: 454-465,* Hong Kong, 2000.

[17] Seigel, J. CORBA Fundamentals and Programming. John Wiley & Sons, Inc., 1996.

[18] Siram, N. N. An Architecture for the UniFrame Resource Discovery Service. MS thesis. Indiana University Purdue University Indianapolis, 2002.

[19] Software Engineering Institute. The Product Line Approach Initiative. http://www.sei.cmu.edu/plp/plp_init.html.

[20] Software Engineering Institute. A Framework for Software Product Line Practice-Version 3.0. http://www.sei.cmu.edu/plp/framework.html.

[21] Sun, C., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., Huang, Z. Composition and Decomposition of Quality of Service Parameters in Distributed Component-based Systems. To appear in *Prodeedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2002).*