Using Runtime Information for Adapting Enterprise Java Beans Application Servers

Mircea Trofin[*]
Performance Engineering Laboratory
Dublin City University
mtrofin@acm.org

John Murphy[**]
Department of Computer Science
University College Dublin
j.murphy@ucd.ie

Abstract

Modern component-based technologies, such as Enterprise Java Beans (EJB), simplify system development by allowing developers focus on business logic, while system services are provided by an underlying application server. A class of system services, such as transactions or security, control the context in which components run.

The provisioning of such services can introduce performance overhead, as some system services might be executed redundantly. As EJB components bind dynamically, the determination that such an execution is redundant can be made only at runtime. We present a runtime mechanism for identifying and removing such redundant executions.

1. Introduction

Companies increasingly rely on component-oriented technologies, such as Enterprise Java Beans (EJB) [1], and Commercial Off-The-Shelf (COTS) components, in order to build large scale applications, reduce system development costs and capitalize on third party expertise.

Typically, component-based systems require an infrastructure that would support components, providing them with lifecycle services, intermediating component message interchange, etc. In the context of EJB, the infrastructure is referred to as an application server.

A trend in component-oriented technologies, especially the ones targeted at enterprise systems, is the separation of system-wide logic from business logic. That is, concerns such as security, transactional isolation, concurrency, or persistence (system concerns), are separated from concerns dealing with what actual services the application provides for its clients (business logic). This trend leads to a separation in responsibilities, as well: application server providers (vendors) are responsible for implementing system logic, leaving application developers with the responsibility of designing and implementing business logic. A module of such business logic is an EJB component, or bean.

Services such as transactions or security deal with the runtime context of a component, and they can be referred to as context management services. These services can be configured by means of deployment descriptors. Deployment descriptors are XML documents associated with each component, and include information indicating the configuration of context management services, on a per-method level. For example, for the security service, the configuration can indicate which user roles are allowed to execute the particular method.

Typical applications built on EJB include e-commerce and e-banking sites. Such applications are required to be highly available, while facing a potentially unbounded request rate. Another characteristic of these applications is that, while being multi-user, they tend to have little, if any, inter-user interaction, which makes the handling of various user requests highly parallel. In such cases, throughput is heavily impacted by the speed with which user requests are handled [2].

EJB applications are built by deploying EJB beans on an application server. The performance of such an application depends both on the characteristics of the developer-written code, as well as those of the application server. The code that ties components to the application server is called "glue code"; it acts as a proxy, calling application server services before and after calls to the component's methods. In some cases, glue code is referred to as "container code", however, since the concept of a container and the boundary between containers and application servers is not clearly separated in the EJB specification, we avoided using the term "container" in this paper.

Studies [3] have shown that a large proportion of the time spent to handle a client request is in fact spent within application server code. It is important, then, to optimize application servers in order to minimize their impact on performance.

Currently, the only means available for reducing the impact the application server has on performance is application refactoring [4]. Such refactorings can indeed improve performance, but at the cost of other system qualities, such as modularity or maintainability.

EJB components bind dynamically, at runtime. Based on how they bind, some context management services could be removed, in effect, minimizing the time spent within application server code. However, given the dynamic nature of EJB applications, the determination of

[*],[**] The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2002.

what can be removed has to be done using runtime information. A more detailed presentation of this aspect of EJB has previously been made [5].

We present a solution for the analysis and removal of redundant executions of context management services between EJB components on an application server. The execution of a context management service is deemed redundant if the goal it tries to achieve has already been achieved by a previous execution. For example, if a transaction context is available, and the control is passed to a method requiring such a context, no additional effort is required for providing this context.

The effect of execution removals is the generation of new glue code versions for a component.

Our solution is generic with respect to context properties, i.e. the solution is not applicable only to the transactions and security services available in EJB.

Note that our effort is not concerned with dealing with remote method invocations in EJB, and all inter-component method invocations described here happen locally, within the same virtual machine. We are also concerned only with the cases where contexts are managed by the application server; EJB permits "bean-managed transactions" for example, which is a case we do not treat.

2. Solution Overview

The solution consists of extracting runtime information from an application and combining it with static information, to generate decisions as to which context management services are redundant.

To deem an execution redundant in a situation, it is necessary to know the context in which that execution is performed. That is influenced by the call path followed to this point, and the context requirements the previous methods in the path had. However, this is not sufficient at all times. Suppose that method m1 of component A indicates that only "admin" users can call it, and method m2 of component B indicates that only "manager" users can call it. Suppose m1 requests a binding to B and calls m2. Now suppose that happens under the credentials of user "Joe", who is an admin, as well as a manager. In this case, only one security context check is necessary, before m1 is called.

This would mean that we can only deem redundant service executions in cases when these services do not take into account the runtime value of the context. However, in the case of the example above, if it were known that "all admins are managers", the security context check at m2 could be deemed redundant. We call this kind of information "application specific facts".

A second problem that needs to be solved is accommodating the fact that the same component might participate in different binding scenarios in which different services might be deemed redundant.

We will first describe the structure of the information required for our solution, and then describe a system that uses this information to optimize an application server.

2.1. Representing Runtime Information: Binding Graphs

Binding graphs are a refinement over the runtime data the monitoring service is producing. A binding graph reflects the order in which bindings took place when a system client request was handled.

A binding graph is essentially a tree. Any node has at most one parent. One node is associated with a component. Each node has a list of method elements. Each method element has a list of binding elements. A binding element contains at most one node. This structure is depicted in Figure 1.

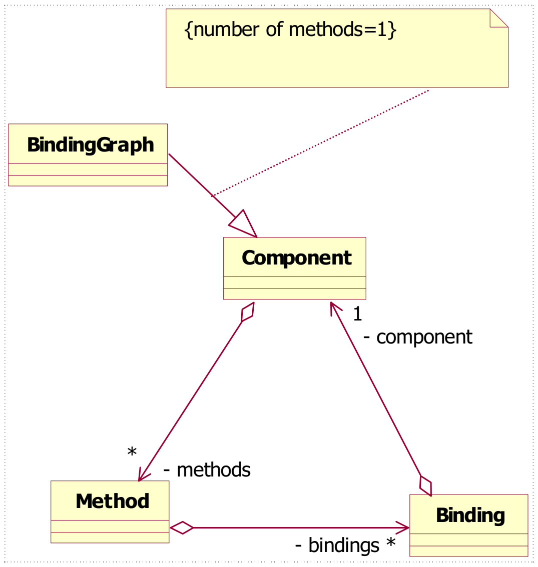

Figure 1 binding graph structure

An example of a binding graph is depicted in Figure 2. Nodes are illustrated as circles. An arc indicates the act of binding.

The root node has always only one element in the method set, as that is the method called by an external client. In our example (Figure 2), *method1* of component *A* eventually initiates a binding to *B* and then *C*. The order is not important, as both bindings happen in the context of *A*, and, since we assume that the context is not modified within a method, the contexts these bindings take place in are identical.

Next, *method1* of *B* initiates a binding to *D*. It is implied that *method1* of *B* was called by *method1* of *A*, since the binding arc ending in this node started there. Note how, because components *B* and *E* are being bound to twice, but in different contexts, therefore, they are being represented for each of those cases. For any node in the graph, the context it is bound in can be determined by identifying its parents, then tracing down the tree the binding process.

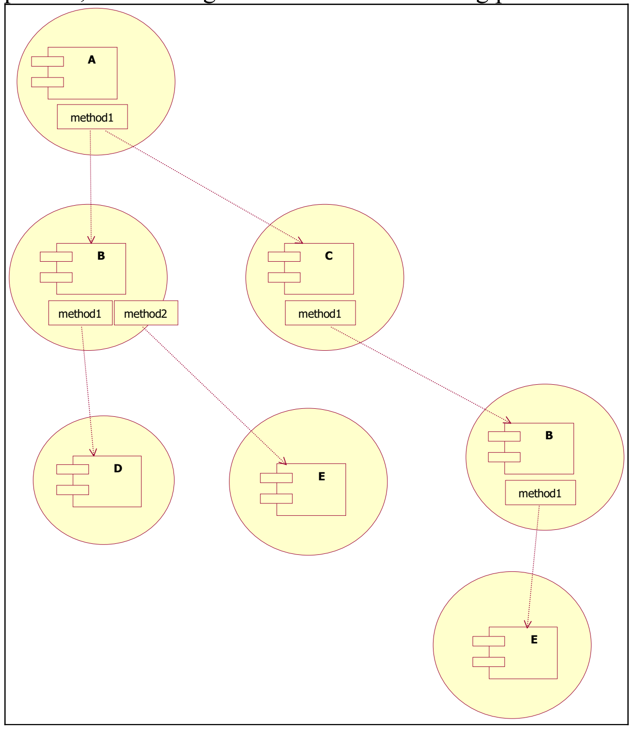

Figure 2 example of a binding graph

Note that a method element in a node describes the context in which calls to the methods in a child (bounded) node are performed. The purpose of the binding graph is to allow for the evaluation of the context in which components are used. For this reason, leaf nodes in a binding tree do not contain method elements, as they would not help evaluate anything (no further bindings). For the same reasons, if *method1* of component *A* called a method "*method0*" on component *B*, which did not lead to any further bindings, that information would not be represented in the tree.

In the case of a component calling its own methods, two possibilities exist: either the methods are called internally, without application server support (and no context management being performed), or through the application server, with context management. The latter would require a rebinding, which would appear in the binding graph as such. The former does not introduce any relevant information. Suppose *method1* of *C* calls *method2* of *C* internally, which in turn requests the binding to *B*. That binding still happens in the context of *method1*: since no application server support was used to call *method2*, no context management services are executed there.

2.1.1 Comparing Binding Graphs and Call Graphs

A call graph describes calls between various components in a system. Binding graphs filter out only those calls that lead to other components being bound. For this reason, more than one call graph can correspond to a single binding graph. Using our example in Figure 2, *method1* of component *A* can call some other methods on *B* after it binds to it, however, that is not important for our purposes, as they all happen within the context of *method1* of *A*. In fact, as it will be seen, what is optimized in this case is the complete glue code of *B*, given that any method *might* be called in the context of *method1* of component *A*.

2.2. Component Framework Rules

Deployment descriptors include information describing requirements placed on the context of execution by each method of a component. This information is encoded as configuration properties that affect the semantics of the execution of a corresponding context management service. For example, "transaction required" means that the method will be executed in the same transactional context as the caller, or, if that is not available, a new one will be created.

Currently, the set of possible configurations is published as part of the EJB specification and it's expressed in natural language. However, we can formally express them in a rule language, like Jess [6]. These rules describe how the context is transformed and whether something needs to be done to do that. For example, for "transaction required", the rule can indicate that, if no transaction context is available, the transactional context management service is to be run, and a new transactional context will be produced. We can refer to these rules as "component framework rules".

An example of such a rule is given in Figure 3.

```
(defrule transaction_required_noCtx
  (transaction required ?method)
  (not(transactionCtx))
  =>
  (assert (transactionSvc execute ?method))
  (assert (transactionCtx))
)
```

Figure 3 component framework rule example

The rule is written in Jess, a rule language similar in syntax to lisp. It describes what the "transaction required" configuration flag in any deployment descriptor means, in terms of executing the transaction context management service (transactionSvc) and in terms of the state of the context (transactionCtx). Here, the rule treats the case in which there is no transactional context available and so one has to be created. In order to achieve that, the transactions context management service has to be executed.

2.3. Context Requirements as Rule Engine Facts

Context framework rules determine a vocabulary that is used to describe the individual context requirements each method of a component. We will refer to such facts as context requirement facts. The translation between the syntax used for context requirements in deployment descriptors, and rule engine facts, is automatic. Translators can be reified using XSLT documents.

2.4. Application-Specific Facts

Relationships between security roles, as given in a previous example, constitute static information pertaining to a particular system. This information is encoded as facts, digestible by a rule engine. In our example, “all admins are managers” is such a fact. We will refer to these facts as application-specific facts.

2.5. Putting It All Together

The information in binding graphs, together with context requirement facts, describes a runtime scenario in terms of a succession of context requirements. Such information, together with application-specific facts, can be fed for processing by component framework rules in a rule-based engine. The output of the rules indicates which context management services need to be run. In other words, we have a mechanism for determining which services are redundant.

3. Solution implementation

Our focus is to develop a runtime optimization solution for application servers. It has to be easily integrated within existent application servers (R1). Extending it to support additional context management services should be done with minimal effort (R2). Implicitly, it is important to ensure that the overhead introduced by our solution does not exceed the performance improvements it generates (R3).

The optimization solution is able to analyze runtime information about an EJB application and decide in which cases context management service executions are redundant. This decision is based on both runtime information – binding grapsh, as well as static information pertaining to the system installation and the EJB framework – component framework rules, application-specific facts, and the information contained in deployment descriptors.

3.1. Overview

Our solution is implemented as an application server service, and consists of: a monitoring service that extracts runtime information from an application; a binding graph filter which extracts binding graphs from the runtime information produced by monitoring (Figure 4). An optimization coordinator controls the optimization of binding graphs by employing an expert system built on top of a forward chaining rule engine [7], such as Jess, which aggregates static and dynamic information and decides which services are redundant for a particular component. The glue code generator maps these decisions into the application server by generating specialized glue code variants. Finally, the call graph isolator ensures that glue code variants are called only in the situation they were optimized for.

Optimizations can be considered valid only for the period of time the set of components on an application server remains unchanged. Strategies for dealing with changes of the component set are under investigation; a trivial solution is to cancel all optimizations and start re-optimizing the system.

Our system is initialized with the set of component framework rules. Application-specific facts can be inserted, ideally pre-runtime, either manually, or automatically, if a facility is provided for that; however, this is outside the scope of our research.

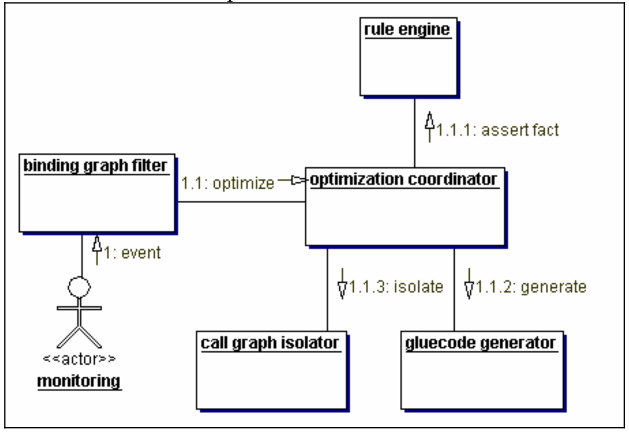

Figure 4 system overview

3.2. Overhead Considerations

Here we discuss aspects related to requirement R3. The optimization of a binding graph might be resource-consuming, but it occurs only once per graph. The overhead produced by our solution should be minimal, as most binding graphs should be optimized immediately after the application is started and serving requests. The more diverse the types of requests the server is presented with early-on, the faster the application will be fully optimized. Based on this observation, we can distinguish two different utilization scenarios of our solution (presented below). They differ in the period of time the optimizations take place. Since the active entity (the source of events) is the monitoring service, the differentiator between the two scenarios is the period of time the monitoring service is active.

3.2.1 Continuous Monitoring

In this scenario, monitoring is always active; therefore, optimizations can happen at any time. Since any new interaction is immediately optimized, the benefit is that all interactions end up optimized after the first time they are executed. The drawback is that monitoring imposes an overhead, which might not be desirable. This scenario is appropriate for the case in which the application under optimization is not well known, or in which monitoring is expected to be constantly turned on.

3.2.2 Training Period

In this case, monitoring is turned on for a period of time called training period, after which it's turned off. Therefore, optimizations can occur only during the training period. Ideally, the system would be exposed to as many different interactions as possible during this period, to minimize the number of un-optimized interactions left at the end of the training period. Insight into the system structure and behaviour is expected.

This scenario is appropriate for cases in which monitoring would not be normally turned on, and in which the application behaviour is well known. In such cases, it offers the benefit of having a fully optimized system (achievable during the training period), at no long-time extra performance cost due to monitoring.

3.3. Monitoring Service

The monitoring service extracts runtime events from an application, and makes them available to registered listeners. Such a listener is the Optimization Coordinator.

The development of this service is not part of our effort, as there are both academic [8] and commercial efforts in this area, which we can integrate with.

3.4. Binding Graph Filter

This component is tightly coupled to the monitoring service and processes whatever runtime information this service produces, extracting binding graphs. The tight coupling is due to the fact that there is no standard monitoring facility for EJB applications, and thus, the interface the various existing monitoring solutions offer needs to be adapted.

3.5. Optimization Coordinator

The optimization coordinator receives for processing one binding graph at a time from the binding graph filter. It maintains a set of binding graphs that it had optimized. Any binding graph is first checked against the optimized graph set. If it is not there (un-optimized), the binding graph is traversed depth first. It passes the context requirements of the method at the top to the rule engine, and then follows the first binding to the next node. Here, it passes all the context requirements of all the methods of the component associated with this node. At this stage, the rule engine decides, for each such method, which context management services are required.

The optimization coordinator invokes the glue code generator with these facts.

Next, the requirements of the methods are retracted, and we follow the next binding down by pushing the requirements of the method that owns the binding. The algorithm is presented in pseudocode in Figure 5.

```
Given RE, a rule engine

optimize (component c)
 for each method m in c

   push m's requirements in RE

   for each binding b in m
    c'= the component associated with b
    r=the set of requirements of the methods of c'
    push r in RE
    rc=get redundant context management executions
from rule engine, for c'
    generate glue code for c' given rc
    optimize(c')
    retract r from RE
   end

   retract m's requirements from RE

 end
end
```

Figure 5 optimization algorithm

Essentially, the algorithm generates a high-level specification of the glue code associated with a component, given a set of facts that can be known about the runtime environment that component might be run in.

3.6. Glue Code Generation

Requirement R1 governs the design of the integration between application server and the rule engine. There has to be minimal coupling between the rule engine and application server code, in particular, component glue code. However, we need to make some assumptions.

A strategy employed by some application servers, such as JOnAS [9], is to generate component glue code when the component is deployed. Usually, code templates are used, which are next run through a code generation engine, such as Velocity [10]. We developed our solution around the assumption that such a mechanism is being used.

The optimization coordinator has to use the information from the rule engine in order to generate specialized versions of glue code for each component. We opted to use a pre-processor solution. Within the code templates used to generate glue code, calls to context management services are tagged. Tagging can either be done with a technology such Velocity or XDoclet [11].

If the pre-processor is started with a set of properties, tagged areas of code can be excluded. Essentially, the code generation process is made aware of assumptions that can be made about the runtime environment of the code to be generated, which results in a customization of this code.

There has to be a mapping between facts produced by the rule engine and tags in the code. This can be ensured, as the "link" between these two is the set of component framework rules, which are available at the time glue code templates are developed.

3.7. Isolation of Call Graphs

In order to inject the optimized glue code back in the server, a major obstacle has to be overcome: the fact that the same component can participate in different interactions, which in turn can yield different glue code optimizations.

Our solution is to provide variants of glue code simultaneously, for the same component, and provide client components with a selection mechanism that allows them to pick the correct variant. A glue code variant of a component *A* is "correct" with respect to a client *C* in the following sense: consider the binding graph *B* that, through optimization, leads to the production of the glue code variant *GcV-A*, for component *A*. Let *B'* be the binding graph associated with the call path in which *C* is part of. If *C* has the same position (same parents) in *B'* and *B*, and *C* tries to bind to *A*, then *GcV-A* is the correct variant.

Refer to Figure 2. In that case, all external clients would bind to *A* via a variant of glue code dedicated to such clients. In this particular case, when *A*'s *method1* is called, the glue code installs a specialized naming provider. When an attempt is made to bind to *B*, this naming provider returns a reference to *B*'s glue code variant which optimizes for the current situation (i.e. *A*'s *method1* binding to *B*). This glue code variant of *B* installs a naming provider when *B*'s *method1* is called which "knows" which version of *D*'s glue code to chose; similarly, for *method2* and component *E*.

The call graph isolator requires the modification of the application server in order to allow for multiple glue code variants.

3.8. Extensibility

To extend our solution to support other context management services (R2), the component framework rule set has to be updated, and glue code templates need to be tagged accordingly. The ability of extending our solution is not so much targeted at EJB applications, as more to the migration of our solution to other frameworks, similar to EJB, such as CCM [12].

4. Related Work

4.1. Operating Systems

Context switching optimizations were analysed in the domain of operating systems (OS). For example, the authors of [13] optimize thread-related context switching overhead, by analysing liveliness information of context elements (such as registers). In [14], the authors attempt to avoid context switching incurred at inter-process communication.

There are two core differences between context switching optimizations in the OS area and our effort, which spawn from differences in problem domains. One lies with the entity that controls the context. In the OS case, the context of execution of a process is represented by a set of values (registers, stack pointer, etc) that belong to the process in the sense that it is the one that alters/controls them. The OS only saves and loads such values, but does not control them. In the components case, contexts are completely out of the scope of a component's control. The context is constrained outside the component's code, and is managed by the platform (application server). This allows for greater opportunities for analysis and optimization in the components case, as all the information related to context management can be made accessible by the platform to the agent performing the optimization.

The other difference lies with the composition of the context being managed. In the case of operating systems, this composition is "a given"; it typically consists of CPU registers. In general case we are focusing on, the composition of the contexts is variable.

4.2. Programming Language Compilers

The area of code optimization, including redundancy elimination, in the context of compilers, has been under extensive research and has achieved maturity. Currently, the vast majority of programming languages are compiled by compilers that make use of optimizers. In the case of interpreted languages, or languages that run over virtual machines, as the case is for Java, the virtual machine can provide an additional set of optimizations for a program.

Optimizations operate on information that is extracted from code, and, sometimes, on information related to the target platform. Typically, an intermediate representation is produced, on which optimization algorithms are run. The result is a modified representation, which is functionally equivalent to the first one, but optimizes for a particular aspect (i.e. space, time)

One requirement for redundancy elimination algorithms is that full data flow information be available [15]. In the case of EJB applications, this is generally impossible, as the execution can be distributed: for example, security checks could happen on a remote machine.

4.3. Optimization of Component Systems

The authors of [16] propose to optimize a component system at runtime. Their approach consists of recompiling an application built out of components, as interactions between components become apparent. The system is continuously evaluated and recompiled. Initial results indicated that a continuous evaluation-recompilation cycle is performance-detrimental.

The authors of [17] suggest that specialization scenarios for components be packaged together with components. The methods of specialization suggested are at the code level.

The most important difference between these approaches and ours is that code-level optimizations will miss out the semantics of context management services. We believe that our approach and the ones presented here can be applied conjointly, but they will optimize different aspects of the application in cause.

A number of authors propose that application servers offer facilities that would allow applications adapt to changes in their environment. An example is the work presented in [18]. Enterprise services tied to an EJB application server can be added/removed or altered. This is similar to what we propose, in a sense, as the effect of our optimizations is that the set of services that gets executed at inter-component calls gets altered. The difference lies with the scope of the alteration: in our case, it is specific to a particular interaction scenario in which a particular component participates, and is done in response to the discovery of redundant context management service executions; in [18], modifications affect all such interactions, and are performed as response to a change in the application environment (such as battery power or network conditions).

JBoss [19] offers the capability of adding or removing services provided by the application server for a component. Similar to the approach above, this capability has the shortcoming of affecting all interactions with that component. This approach cannot support the case in which the same component participates in different execution contexts.

5. Current Status and Future Work

5.1. Optimization Study

We conducted feasibility tests for our rule engine based optimization solution. We started by defining component framework rules for the transaction service, and extracting context requirements as facts from a set of components. No application-specific facts were used at this time.

We chose the transactions service for this test as it offers a larger array of configuration options, when compared to security.

The experience supports the current solution. The next immediate step is to include security rules, together with application-specific facts.

5.2. Call Graph Isolator Implementation

We have implemented a prototype call graph isolator on JBoss. We have used this prototype in order to gain more insight into the design implications related to it, as well as verify whether such a mechanism would introduce any overhead. A full discussion of the isolator is out of the scope of this paper. JBoss was used for this prototype purely for previous experience reasons. Since JBoss uses a reflective approach to glue code, it is not suited for implementing the rest of our solution; however, it proved sufficient for the purpose of this prototype.

5.3. Future Work

We intend to finalize a prototype optimization coordinator and engine, together with the corresponding set of rules for transactions and security services, as supported by EJB.

The next step will be to analyse the proposed glue code generation mechanism, in terms of technology used. Its applicability across various application servers will also be analysed. As we assume a particular glue code generation style in-place (template-based code generation), we will analyse glue code generation solutions for other cases - JBoss, for example, employs a reflective approach.

6. Conclusions

We presented the problem of determining which context management service executions are redundant for applications built on the Enterprise Java Beans component framework.

The proposed solution consists of aggregating static and dynamic information and producing variants of glue code that contain only context management service calls that are not redundant. Static information consists of component framework rules, context requirement facts, and application-specific facts. Dynamic information is encoded in binding graphs. The decision as to which service calls are redundant is made by a rule-based engine.

Glue code variants are produced by augmenting a currently employed method, template-based code generation.

Glue code variants are bound to the situation they are specialized for (i.e. a particular position in a call graph). A method has been presented and prototyped for ensuring that this binding is respected every time calls are passed between components.

7. References

[1] Sun Microsystems. “Enterprise Java Beans Specification”, http://java.sun.com/products/ejb/docs.html#specs

[2] The Middleware Company Case Study Team. “J2EE and .Net (Reloaded). Yet Another Performance Case Study”. http://www.middleware-company.com/casestudy/tmc-performance-study-jul-2003.pdf . June 2003

[3] E. Cecchet, J. Marguerite, W. Zwaenepoel. “Performance and scalability of EJB applications”. In Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) November 2002, Seattle, WA

[4] Brett McLaughlin. “Building Java Enterprise Applications Volume I: Architecture”. O’Reilly, 2002

[5] Mircea Trofin, John Murphy. “A Self-Optimizing Container Design for Enterprise Java Beans Applications”. The 8th International Workshop on Component Oriented Programming (WCOP), part of the 17th European Conference on Object-Oriented Programming (ECOOP). July 2003, Darmstadt, Germany.

[6] Sandia National Laboratories. Jess, the Rule Engine for the Java™ Platform.

[7] S. Russell, P. Norvig. “Artificial Intelligence. A Modern Approach”. Prentice-Hall, 1995

[8] Adrian Mos, John Murphy. “Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach”. In proceedings of The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC), September 2002, Lausanne, Switzerland

[9] ObjectWeb. “JOnAS: Java™ Open Application Server”. http://jonas.objectweb.org/

[10] The Apache Jakarta Project, “Velocity”. http://jakarta.apache.org/velocity/

[11] XDoclet – Attribute Oriented Programming. http://xdoclet.sourceforge.net/

[12] Object Management Group. “Corba Component Model” http://www.omg.org/technology/documents/formal/components.htm

[13] Dirk Grunwald, Richard Neves. “Whole-program optimization for time and space efficient threads”. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, Cambridge, Massachusetts. 1996

[14] Erik Johansson, Sven-Olof Nystrom. “Profile-guided optimization across process boundaries”. In Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, 2000

[15] Steven S. Muchnick. “Advanced Compiler Design Implementation”. Morgan Kaufmann Publishers, 1997

[16] A. Gal, P.H. Fröhlich, M. Franz. “An Efficient Execution Model for Dynamically Reconfigurable Component Software”. In Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming. June 2002, Malaga, Spain

[17] Gustavo Bobeff, Jaques Noye. “Molding Components Using Program Specialization Techniques”. In Eight International Workshop on Component-Oriented Programming (WCOP 2003) of the 17th European Conference on Object-Oriented Programming. July 2003, Darmstadt, Germany

[18] Zahi Jarir, Pierre-Charles David, Thomas Ledoux. “Dynamic Adaptability of Services in Enterprise JavaBeans Architecture”. In Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming. June 2002, Malaga, Spain

[19] The JBoss Group, “JBoss Administration and Development Documentation – eBook - 3.2.1”. http://www.jboss.org