JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 5, No. 6, July–August 2006

Removing Redundant Boundary Checks in Contextual Composition Frameworks

Mircea Trofin, **John Murphy**, School of Computer Science and Informatics, University College Dublin

Currently, contextual component frameworks, such as Enterprise JavaBeans (EJB), provide for application modularity at the cost of reduced performance. This "endemic" is partly due to the fairly liberal execution of platform code at component boundaries. The task of such code is to satisfy boundary conditions that components have specified at deployment time. In some cases, the execution of this code becomes redundant. The overhead penalty in modular applications can be reduced or eliminated if this redundancy is removed.
We have developed a method for detecting and removing such redundancies. This method can be applied to optimize EJB application servers as well as other contextual composition platforms. We have evaluated our method on a prototype, and results indicate clear runtime improvements of optimized scenarios over un-optimized scenarios. We argue that it is now possible to develop modular contextually-composing applications that are also well-performing.

1 INTRODUCTION

Contextual composition frameworks allow components to specify boundary conditions describing properties that the runtime context must meet. Based on these conditions, component platforms execute services when the control is passed from a component to another one. The goal of these services is to ensure that the boundary conditions are met.

Examples of such frameworks include Enterprise JavaBeans (EJB) [1], Microsoft Transaction Server (MTS) [2] and CORBA Component Model (CCM) [3]. Typically, these frameworks are targeted at developing enterprise applications. For such applications, performance is an important system property, as it typically translates into the ability of serving an increased number of customers, and, as such, into revenue.

By their nature, the performance of applications developed on these frameworks depends both on characteristics of the constituent components, as well as those of the underlying platform, or application server. In a study [4] of two EJB application servers, it has been shown that most of the time required for fulfilling a client request is being spent executing platform code. Part of this platform code deals with checking and ensuring that component boundary constraints are being met. In EJB, for example, such boundary constraints include security restrictions and

Cite this document as follows: Mircea Trofin, John Murphy: *Removing Redundant Boundary Checks in Contextual Composition Frameworks*, in Journal of Object Technology, vol. 5, no. 6, July–August 2006, pages 63–82,
http://www.jot.fm/issues/issues_2006_07/article2

transactional isolation checks. To perform these checks, an EJB platform provides a security and a transactions service.

It has been previously documented [5] that removing the security service, for example, can increase the throughput of an application by almost 50%. This overhead can vary, of course, depending on a number of factors, such as application or platform characteristics. What is important, though, is that boundary checks do induce a significant runtime execution overhead. Therefore, reducing this overhead can be beneficial for performance. Characterizing the size of this overhead for an application and platform combination is possible but it is out of the scope of this paper.

Our goal is to show that it is possible to construct contextual composition platforms that automatically remove unnecessary inter-component overhead due to redundant services execution. We are addressing the problem generically with respect to the component technology (as long as composition is "contextual"), as well as the kinds of services handled.

Previously, we have presented a high-level overview of our approach [6], where we introduced some of the elements of our solution and described possible implementation strategies. The contributions of this paper are:

- a formal description of the services that can yield redundancies, together with a rigorous definition of redundancy
- a detailed description of an implemented prototype self-optimizing contextual composition platform
- a runtime characterization of this platform, which shows that the optimization method does not impose any processing overhead, once an application is optimized, at the cost of a (bounded) memory overhead. We also discuss aspects related to the convergence of an application to an optimized state (i.e. after which no further intervention from our method is required).

2 PROBLEM DESCRIPTION

We used EJB as start point, and thus the problem description is done in terms of EJB terminology.

In EJB, components are class-like entities that bind at runtime. An EJB component, or enterprise bean, is mainly formed out of: a **business interface**, an **implementation class**, a **home interface** and a **deployment descriptor**. The business interface is a java interface and it defines the services (functionality) that the component can provide to its clients. The implementation class implements the services defined in the business interface. The home interface is an interface through which clients can obtain instances of this component; the realization of this interface is typically delegated to the platform. Finally, the deployment descriptor specifies

meta-information about the component, including boundary conditions. The latter are provided as simple statements indicating, for example, the security credentials that a client must have in order to invoke a particular method, or the way a method will participate in transactions: for example, *"transaction required"* means that the method will participate in the client's transaction, or, if that is not supplied, a new transaction will be started for that method. It is important to note that boundary conditions are determined at deployment time, and do not change at runtime.

Each component has one home object associated, which implements the home interface and manages all that component's instances. The home object is registered with a naming service. To obtain a component instance, a client has to know the name of the corresponding home object. To avoid hard-binding through names, each client component has a local naming service. This allows client components to refer to other components' homes by local names. This naming service is associated with the current thread, by the platform, every time a component method is called. This ensures that the method code is naming-service agnostic: whenever a method looks up a component, it actually contacts the naming service currently associated with the thread.

At deployment time, these local names are linked to global names. It is important to note, however, that no inference can be made before runtime about the way components bind. Indeed, if a component has a set S of local names, it is unclear when an element from S will be used for binding, since we have no information about which methods will actually initiate bindings, and using which names. Moreover, the local naming mechanism can be bypassed, meaning that a client could simply request a component (via its home object) using a global name; this name could even be computed or obtained as a parameter at runtime. As a corollary, only at runtime one can understand how components bind.

A client request to an application built out of such components starts with the client obtaining the home object of a component, obtaining an instance from it, then invoking a business method on that instance. The platform ensures that, before the implementation of that method executes, all boundary conditions specified for that method are satisfied. As mentioned, a number of platform services are employed to achieve that. Next, the method executes, and in order to fulfill its responsibilities it might need to interact with another component. This is achieved using the same mechanism its client used.

As mentioned, in the EJB example, boundary conditions place constraints on the security and transactional environment that components run, in other words, they describe properties that the runtime context of components needs to exhibit. Since the associated platform services actuate these constraints, we call them context management services (CMS). At this stage, we consider only the case where CMS are the **only** means of altering the runtime context; this is consistent with a "pure" contextual composition scenario [7]. Cases where components can also interact directly with the context of their execution will be treated in our future work.

We refer to the platform code that ties a component to context management services as *gluecode*. Essentially, the gluecode is a proxy that wraps calls to a component's methods with calls to platform services, including CMS.

The interaction between the elements described so far is summarized in Fig. 1. Two components, A and B, interact as follows: a client calls *A::doSomething*, which, at a point, binds to B and calls *B::doSomethingElse*. We use C++ like method notations for brevity (i.e. *A::doSomething* means method *doSomething* of A). In the figure, *gcA* and *gcB* represent the gluecode associated with A and B; *homeB* is the home object associated with component B. The object labeled *thread* represents the thread in which this interaction takes place. Finally, *namingA* is the naming service associated with A.

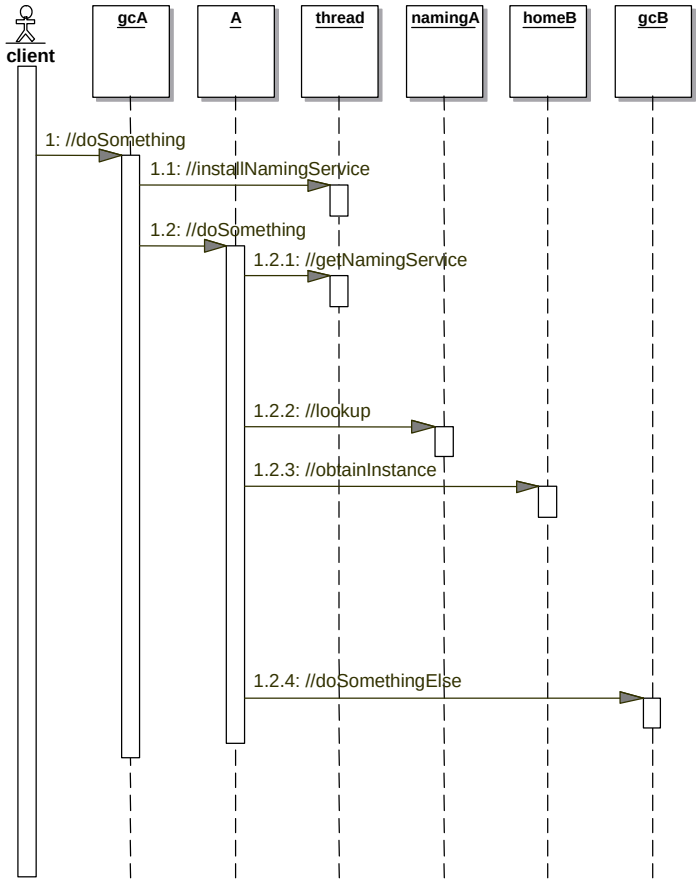
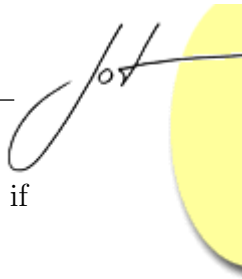


Figure 1: EJB binding mechanism

We skip the details of how the client contacted A, and show how the method *doSomehting* is being called on A. *gcA* acts as a proxy, and part of what it does is to install on the current thread *namingA* as a naming service. It also calls a number of CMS, but we have omitted that. Next, *A::doSomething* is called, which, at a point, needs to bind to B. It first obtains its naming service (*namingA*) from the thread, on which it performs a lookup, obtaining *homeB*. From the latter it requests a component instance, and calls *B::doSomethingElse*, which first needs to



pass through gcB, which is a proxy for B. The same interaction pattern follows, if *B::doSomethingElse* further binds to other components.

Depending on how components call each other in order to fulfill some client request, the case can be that the execution of some CMS becomes redundant. This entirely depends on the call scenario under discussion and on the actual boundary conditions. For example, if a component's method, $A :: m_1$, specifies *"transaction redundant"* and calls another component's method, $B :: m_2$, with the same constraint, no transactional CMS needs to be executed for the call of $B :: m_2$, as the boundary constraint of $A :: m_1$ ensures that there is a transactional context available. As we have seen, the execution overhead of CMS can impact performance negatively. Since some CMS can be redundant, their elimination will improve performance; by how much, it completely depends on the application in discussion, however, the mentioned results in [5] promise significant improvements.

We focus on the problem of redundant CMS detection and removal. The difficulty in achieving this is twofold. The first lies with applying existing redundancy elimination methods in the scenario in discussion. Redundancy removal has been extensively studied in the context of optimizing compilers [8]. An optimizing compiler can remove repeated computations of some expression, written in some language. To achieve this, it typically needs to have access to the complete data flow of a program. In the case we are studying, we are not dealing with computable expressions, but with a limited set of constraints expressed as statements; the semantics of these constraints are given in natural language (in the EJB specification); and complete data flow information cannot be generally assumed (security checks might be done remotely).

A second difficulty lies with the fact that components potentially participate in many types of interactions, which means that CMSs that are redundant in some interactions might not be so in others. This means that the gluecode of a component needs to be tailored to the particular scenario the component participates in. To the best of our knowledge, the current implementation strategy employed in all EJB platforms allows only for a one-to-one association between gluecode and component. We need to allow for multiple variants of gluecode for the same component, each optimized for a particular call scenario, executing a subset of the available CMS. Moreover, we need to employ a low-overhead runtime mechanism that selects the correct variant of gluecode depending on the current call scenario.

3 REASONING ABOUT REDUNDANCY

In order to reason about the redundancy of CMS, we need to define more formally the notions of CMS and boundary conditions. We also need to clarify what kind of call scenario information is required in order to decide upon redundancy.

A Model for CMS and Redundancy

We start by clarifying the notion of context. By context, we mean a set of attribute-value pairs associated with a thread. Depending on these values, the semantics of the operations performed within that thread change. For example, depending on the value of a transaction context attribute, a sequence of interactions with a database might or might not be part of a transaction with that database.

The values that each context attribute can take belong to a domain D. For example, the domain of the security context attribute, $D_{security}$, contains all security roles in some security realm. We additionally enforce that all domains contain a *null* element, indicating the absence of any value. In the transactions case, for example, *null* indicates that there is no transactional context available.

A Context Management Service is a service that is solely responsible for the management of the value of a particular context attribute. Thus, in the EJB example, there is a transactions CMS and a security CMS. The behavior of a CMS changes depending on boundary conditions. Essentially, a CMS can be modeled as a family of functions, all defined on the domain of the corresponding context attribute, and producing values on the same domain. Unmapped values are allowed, in order to indicate errors or cases that cannot be handled.

We consider the boundary conditions associated with a particular CMS as elements of a finite and discrete set. We associate to each boundary condition one and only one CMS function from the corresponding family of functions, e.g. security boundary conditions are associated with functions from the security CMS. For simplification, we will simply label CMS functions with both the name of the CMS, as well as the corresponding boundary condition. For example: $f^{required}_{transaction}$.

We illustrate the above concepts with an example. In EJB, for transactions, the set of boundary conditions B is

$$B_{transaction} = \{required, new, never, supported, mandatory, notsupported\}$$

as per specification [1]. The domain is the set $D_{transaction} = \{x \mid \text{x is a valid transaction id}\} \cup \{null\}$. All functions that make up the CMS service for transactions have the form $f : D_{transaction} \rightarrow D_{transaction}$. Since there are six elements in $B_{transaction}$, there will have to be six functions forming the CMS. Based on the EJB specification, we can exemplify with two of them, defined as in Fig. 2, where the f are the functions modeling CMS, and $x, y \in D_{transaction}$.

A call scenario through an application consists of the successive execution of CMS and component methods. The value of the context in a particular component method, m, is determined by the effects of the execution of the last CMSs (the ones that executed to satisfy the boundary constraints of m).

The only variable that determines the value of a context attribute in a particular method is the previous value of that attribute. In general, this means that the value of a context attribute at a particular point can be calculated as $(f_1 \circ f_2 \circ$

$$f^{required}_{transaction}(x) = \begin{cases} x & \text{if } x \neq null \\ y, y \neq null & \text{if } x = null \end{cases} \quad (1)$$

$$f^{never}_{transaction}(x) = \begin{cases} x & \text{if } x = null \\ undefined & \text{if } x \neq null \end{cases} \quad (2)$$

Figure 2: example CMS functions

$... \circ f_n)(input)$, where *input* is the value that the application client provides for that context attribute - see Fig. 3. This value could be, for example, some client-obtained security credentials for the security case, or a transaction id for transactions.

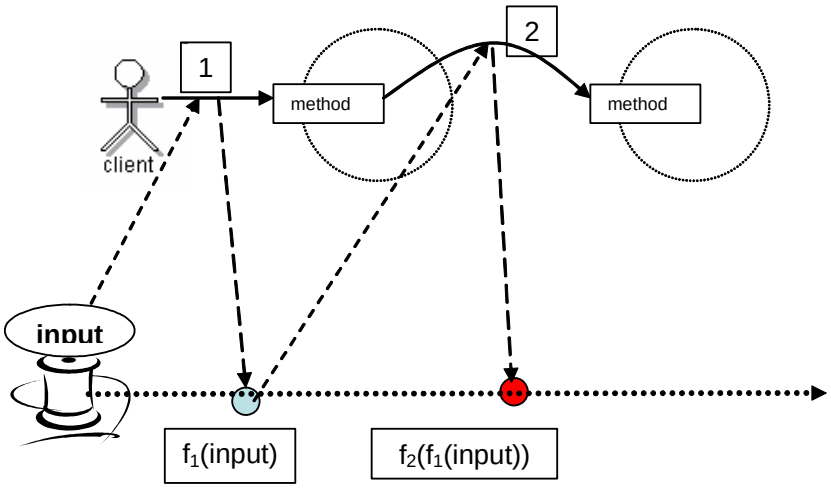


Figure 3: evolution of context along a thread

If, irrespective of *input*, one of the f_i always computes a static point (that is, $f(x) = x$), we deem f_i **redundant** in the corresponding call scenario.

Given a call scenario, and the boundary conditions of all methods involved, we can determine which CMS are redundant as follows: first, we substitute with corresponding CMS functions the boundary conditions (as per model). Then, we analyze each context attribute separately. For such an attribute, we start by assuming that the initial value is bound to its whole domain - in other words, we do not assume anything about its value. We then compute the image of the domain through the CMS first function in the scenario, (f_1 in Fig. 3). The image of a set S through a function f is the set $Img_f(S) = \{y \mid y = f(x), \forall x \in S\}$.

If the resulting set is a set of static points for the next function f_2, then f_2 is redundant. With this set, we proceed to the next CMS function in the scenario. Similarly, we compute the image of the set through the function, and assess whether this function is redundant. The process continues as such. Since we make no assumption over the initial value of the context, decisions regarding redundancy of CMS are guaranteed to be valid, for the given call scenario. Removing redundant CMS services from the corresponding call scenarios is also guaranteed to preserve

the semantics of the application, since the value of the context is guaranteed to be the same without the execution of the redundant CMS (from the definition of redundancy).

There are two problems that need to be solved at each step of the above analysis: one is how to compute the image of a function. The other is how to check that a set contains only static points for a function.

To our knowledge, there are no general algorithms for solving the first problem. We address this by constraining CMS functions by stating that their definition has to describe mappings from subsets to subsets; indeed, the example provided in Fig. 2 fit this description; in fact, we were able to specify as such all CMS functions that stem out of the EJB specification - for both transactions and security.

Is this constraint on CMS function definitions too restrictive? We believe not; the conjecture here is that, since boundary checks are declarative in nature (rather than formulaic), it is rather natural that the mappings they specify be not too detailed, at an element level, but rather at a subset level.

Representing Call Information: Binding Graphs

Binding graphs have first been introduced in [6], this article is simply summarizing the notion and refining their notation.

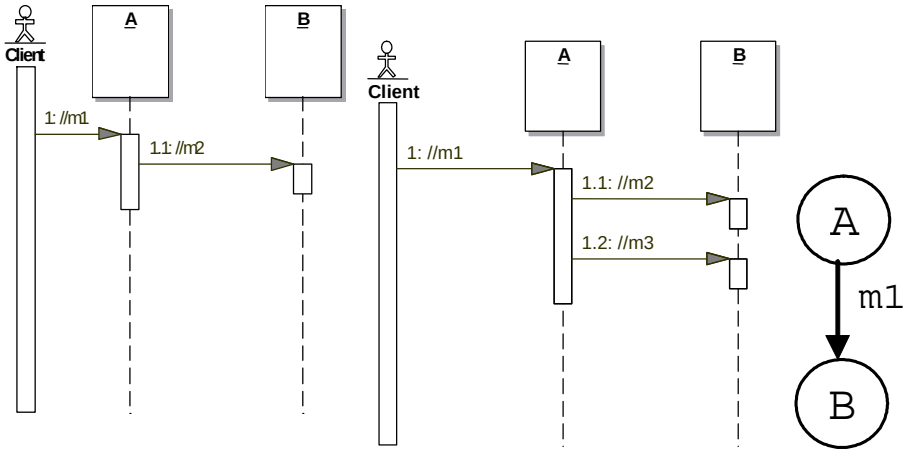


Figure 4: call scenarios that are equivalent from a binding perspective

Consider the very simple call scenarios in Fig. 4(a) and 4(b). In both figures, it is assumed that $A :: m1$ has already bound to B. In Fig. 4(a), $A :: m1$ calls $B :: m2$. In Fig. 4(b), $A :: m1$ calls $B :: m2$ and $B :: m3$. In either case, the context in which $m1$ executes constitutes the input for the CMS of B's methods. In fact, $A :: m1$ could call any of B's methods; from the point of view of analysing contexts, the only important factor is the fact that B is being bound to from $A :: m1$.

Rather than working with independent call graphs, we could use a refinement which would indicate only the methods that produce bindings to other components. The result is a binding graph, as shown in Fig. 4(c), which corresponds to our example in Fig. 4(a) and 4(b).

A binding graph is a tree where nodes represent components and arcs represent binding actions. All arcs are directed. The arcs are labeled with a method name. The method belongs to the parent node of the arc. The connection of two nodes A and B with an arc labeled m directed from A to B means that there is a request in method m of component A to bind to component B. The root of the binding graph represents the first component to be contacted by an outside client. All arcs emerging out of the root are labeled identically, as the binding graph captures only individual client request realizations.

If in Fig. 4, $B :: m2$ would bind to a component C, the binding graph in Fig. 4(c) would additionally contain a node C and an arc directed from B to C, labelled m2. For space considerations, we could not include a more complex binding graph example; we direct the reader to a previous publication [6].

4 SELF-OPTIMIZING PLATFORM

Based on the model of redundancy presented in section 3, we developed a technology that allows for the automatic self-optimization of an application server through the semantics-preserving removal of redundant CMS. To demonstrate this technology, we implemented, in Java, a prototype self-optimizing contextual composition platform. The platform automatically detects and removes redundant CMS. Detection and removal can be performed whenever binding graphs can be provided. In the most general case, this happens at runtime. As such, in Fig. 5, we present our solution utilizing a monitoring module that extracts runtime call information.

It is important to note that the purpose of the prototype is to showcase the technology required to construct platforms offering automatic detection and removal of CMS, and to perform measurements; it is not meant as a ready-replacement for existing technologies. The component framework the prototype platform realizes is reminiscent of EJB, in the sense that components bind using a naming service and that the services supported are transactions and security, with boundary constraints identical to the corresponding ones in EJB. The framework is simplified by not having multiple types of components, like EJB, but rather having a single, stateful type. Home objects are also generic, allowing only for the creation of instances and the finding of instances based on an identifier obtained at creation. These two simplifications reduce only the diversity of facilities provided, not their nature, and do not hinder our goal with the prototype.

The platform supports extension by deploying new CMS. A CMS "bundle" consists of: the service itself, its corresponding boundary checks expressed in a formal language, "hooks" into gluecode, and a declaration for the context variable that the

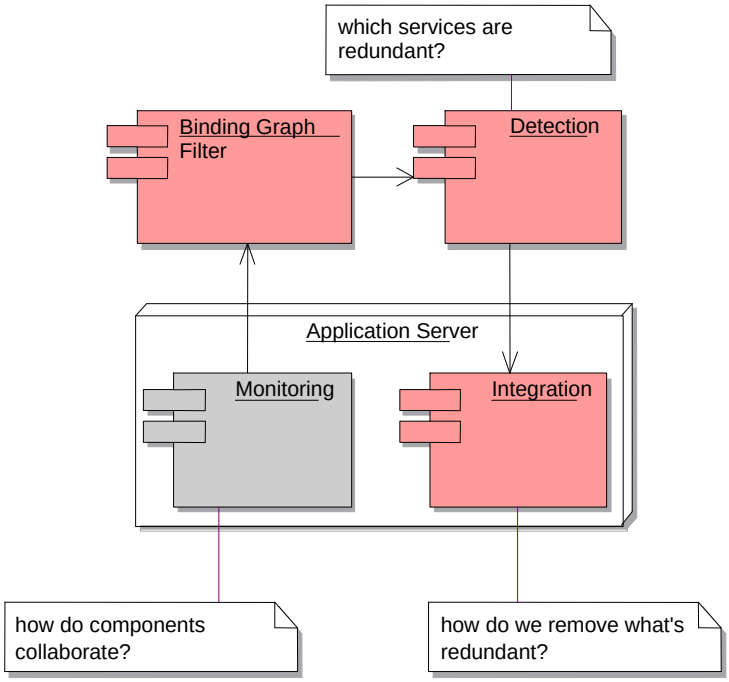


Figure 5: platform overview

service uses.

The optimization system operates much like a document processing system, where the document is a binding graph, as follows: at runtime, as clients make requests to the application, the **monitoring module** produces runtime information, and passes it to the **binding graph filter**. The binding graph filter applies an algorithm to extract the corresponding binding graph out of a call graph. Next, the **detection** module checks to see if this particular binding graph has been optimized before. If not, it attempts to determine which CMS are redundant for that binding graph. As a result, it populates the binding graph with its decisions, and passes it to the **integration service**. The integration service actuates these decisions, by adapting the platform in such a way so that, the next time a call takes place through the system, and components bind the same way (i.e. as per the binding graph), the call is performed without executing redundant CMS. The process is repeated each time another call takes place. Eventually, all binding graphs of the system are optimized. When that is achieved, the process can be stopped, by turning off monitoring, or disconnecting the binding graph filter from the monitoring service. This can be accelerated, thus greatly reducing any runtime overhead. Additionally, runtime optimization poses hot swapping problems, however, we are not focusing on these aspects.

Various component platforms, (EJB, for example), either offer monitoring solutions, or there are third party products offering this service. Research in this area has also been carried out [9]. We have focused our attention on the development of the detection and integration modules, which we are presenting in what follows.

Detection

The detection module processes one binding graph at a time. It starts by checking if such binding graph has been optimized already; if not, it attempts to detect redundant CMS.

The CMS model we have presented in Sect. 3 suggests that CMS functions behave like rules: based on a property of the input - the fact that it belongs to a set - the rules assert a property of the output - the fact that it belongs to another set. Our approach for automatic redundancy detection centers around a rule engine [10], pre-loaded with CMS functions rendered as rules. An example set of CMS rules, corresponding to the transactions case presented in 3, are given in Fig. 6 and 7.

```
(defrule trans-req-noctx
        (logical(transaction required ?m ?c))
        (not(transactionCtx))
        (not(done transaction ?m ?c))
=>
        (assert(transactionCtx))
        (assert(done transaction ?m ?c))
        (assert(verdict ?m ?c transaction not-redundant))
)

(defrule trans-req-ctx
        (logical(transaction required ?m ?c))
        (transactionCtx)
        (not(done transaction ?m ?c))
=>
        (assert(verdict ?m ?c transaction redundant))
        (assert(done transaction ?m ?c))
)
```

Figure 6: rules defining “transaction required” CMS function

```
(defrule trans-never-noctx
        (logical(transaction never ?m ?c))
        (not(done transaction ?m ?c))
        (not(transactionCtx))
=>
        (assert(done transaction ?m ?c))
        (assert(verdict ?m ?c transaction redundant))
)

(defrule trans-never-ctx
        (logical(transaction never ?m ?c))
        (not(done transaction ?m ?c))
        (transactionCtx)
=>
        (assert(done transaction ?m ?c))
        (assert(verdict ?m ?c transaction not-redundant))
        (assert(ERROR))
)
```

Figure 7: rules defining “transaction never” CMS function

The rules are written in Jess [11]. Jess is a java-written forward-chaining rule engine [10]. Its syntax is reminiscent of LISP. Typically, one interacts with Jess

by defining rules and facts, then starting the inference engine. Rules consist of a name (e.g. **trans-never-ctx**), a set of preconditions (before =>), followed by a set of actions (after =>) that are taken if the preconditions are met (i.e. the rule "fires"). Typically, preconditions are formulated around facts. Facts can be seen as LISP-like lists (e.g. *(done transaction ?m ?c)*). In a rule, facts can be described using variables (e.g. ?m). At runtime, a fact *(done transaction a b)* matches *(done transaction ?m ?c)* and ?m receives the value a and ?c receives the value b. In Jess, fact variables can have as value a java object, which allows for easy integration with a java application.

When Jess is started, all rules that have their preconditions matching the existing facts fire (in no particular order). As a result, new facts might be *asserted*. In turn, more rules might fire. When no rules can fire anymore, Jess stops. At this point, Jess offers mechanisms for introspecting the set of facts that were produced.

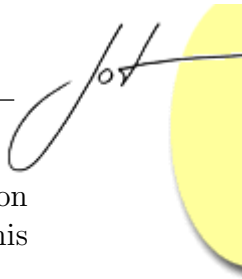
Consider Fig. 6, and refer to the corresponding function definition in Sect. 3. The two Jess rules correspond to the two branches in the functional definition. The first rule, *trans-req-noctx*, fires when the context value corresponds to *null*. As result, it produces a non-null transactional context (the *transactionCtx* fact).

The second rule, *trans-req-ctx*, treats the case a transaction context is available. In this case, we know that the transaction CMS is redundant, so the rule asserts that as a *verdict* fact.

In general, CMS rules operate based on properties of the context property they treat (e.g. transactional context in our examples). These properties describe which sub-domain the context belongs to at a point. For example, in Fig. 6, the firing of the first rule asserts that there is a transaction context, which corresponds in our model to $f(x) = y, y \neq null$.

In Fig. 7, an $ERROR$ fact can be asserted, for the *transaction never* boundary check, when a transactional context is present. Recall from Fig. 2 that the value of $f^{never}_{transaction}$ is undefined when a transactional context is available. The $ERROR$ fact indicates this case.

We can now describe the functionality of the detection module. With the rule engine initialized with all rules for all CMS, for the binding graph under processing, it starts by asserting into the rule engine the requirements of the first method (let's call it m), presented as facts. As effect, the rule engine fires the corresponding rules, at the end of which, we can know which CMS are redundant for m, and what can be known about the context at this stage. This information is associated with m in the binding graph. We continue by following through the binding graph, by picking a component that is being bound by m (let's call it B). We send all B's method requirements to the rule engine. The rule engine is run again. As result to rules firing, at this stage, we know which CMS need to be run and which are redundant for any of B's methods, *if* they are called from m. As before, this information is placed in the binding graph, associated with the corresponding methods. We continue parsing the binding graph as such.



At the end of the optimization process, the binding graph contains information indicating which CMS are redundant, for all involved components. We refer to this binding graph as *optimized binding graph*.

Integration

The goal of the integration module is to ensure that each subsequent time that a call matching an optimized binding graph passes through the system, the decisions made by the detection module are taken into account. The adaptations that are made in order to satisfy this condition have to also ensure that other calls involving the same components are not affected. Optimizations of different binding graphs have to not interfere with each other, even if these optimizations involve some of the same components. In other words, the optimization of the scenario $A :: m1$ calls $B :: m2$ calls $C :: m3$ has to "co-exist" with that for the scenario $B :: m2$ calls $A :: m1$ calls $C :: m3$, for example. We will refer to this as *non-interference*. It is important to observe that non-interference implies that, once a binding graph is optimized, it does not need revisiting.

To achieve this goal, the integration module employs a **gluecode generation service**, which creates a new, optimized, variant of gluecode for a component; and an **isolation service** which ensures that optimized gluecode variants participate only in the calls they are optimized for.

Optimized gluecode variants are generated using Velocity [12], based on the information provided by the detection module. For a component, the gluecode consists of the following parts: a generic *wrapper*, a number of *behavior* objects, and a number of *specialized homes*. The wrapper acts as a proxy to the component and is parameterized with behavior objects. A behavior object is a stateless singleton which corresponds to a particular optimization scenario; for each separate scenario, a separate behavior class is generated. It is behavior objects that actuate the decisions of the decision module. Associated with a particular behavior object is a specialized home object instance. The specialization consists of parameterizing it to produce component instances with the behavior parameter of the wrapper set to the home's associated behavior object. Note that all such home instances manage the same set of component instances, what differs is what behavior object they attach to these instances.

Consider a binding graph and the path from the root node to any node N. The ordered list of the arc labels constitutes the *predecessors* of N. The predecessors together determine the context in which any of the methods of N are being called from. The gluecode variant of N produced as result of optimizing this binding graph needs to be called whenever at runtime N's predecessors coincide with the ones in the binding graph. However, at runtime, information about the current call stack and, in effect, predecessors, is typically costly to obtain and verify.

We have developed an **isolation mechanism** that achieves the desired results

at a very low runtime cost. The mechanism leverages the name-based binding mechanism, and associates a separate naming service with each method of a component, for each gluecode variant. By default, such a naming service delegates a lookup request to the component-level naming service (as presented in Sect. 2). In turn, a default home object is returned. For components in scenarios that have been optimized for, these naming services can return specialized home objects.

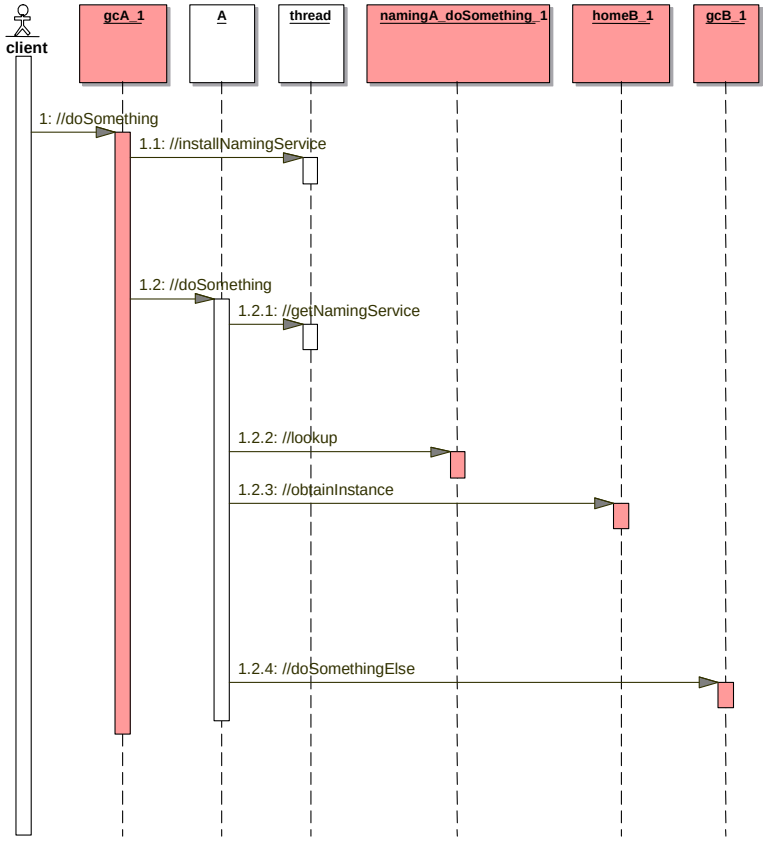


Figure 8: optimized flow

We can now show how the gluecode design is employed by the isolation mechanism to achieve the non-interference goal. At the time gluecode variants are created, corresponding method-level naming services are also generated and configured by the isolation service, as follows. Consider a binding graph and a member node labeled A. From this node, a number of arcs emerge, labeled $a_1, a_2, \ldots, a_n$. These arcs connect A with nodes labeled correspondingly as $N_1, N_2, \ldots, N_n$. Once gluecode variants have been created, assume that their associated homes are registered in the global naming service under the names $N_1^{'}, N_2^{'}, \ldots, N_n^{'}$, and $A^{'}$ for A. For each naming service ns_i associated with $(a_i, A^{'})$, the isolation service links naming requests made to N_i to the home $N_i^{'}$.

Let's assume that the scenario depicted in Fig. 1 has been optimized (refer to Fig. 8). For brevity, wrapper and behavior objects are represented together as gluecode objects. Notice how the client contacts an optimized variant of the gluecode of A,

gcA_1, which installs the naming service *namingA_doSomething_1*, which is the one associated with *A::doSomething*, in this scenario. As an effect, when binding to B is requested, *namingA_doSomething_1* returns the home *homeB_1*, the home object of B associated with this scenario.

If during the execution of an optimized call, a method requests a binding to a component that has not been included in the optimization, the method's naming service simply delegates that request to the global naming service, thus returning the default gluecode. This new scenario can be optimized afterwards, by appending it to the previously-optimized scenario. The details of the latter are out of the scope of this paper.

At runtime, a number of calls run concurrently in the system, and it is quite possible that a number of them correspond to the same binding graph. Since we have designed gluecode variants (behavior objects) to be stateless singletons, one instance can be shared by all component instances involved in separate interactions. Similarly, method-level naming services are shared by instances, as, although stateful, naming services do not change their state at runtime. This contains the space overhead required by our solution and ensures that it does not introduce any concurrency control.

5 RUNTIME CHARACTERIZATION

There are two aspects of our method that need to be discussed: one concerns the transition of a system to an optimized state, the other is the optimized system.

Transition to an optimized state

To this point we have shown a design that achieves our goal of developing a platform that automatically removes redundant CMS executions. We will now show how such a design performs at runtime.

We will first discuss the convergence of a system to an optimized state. Given that once a binding graph is optimized, it doesn't need revisiting, reaching the optimized state depends on whether all binding graphs in the system can be found, and, more importantly, if their number is bounded.

The number of binding graphs can be unbounded if some methods are recurrent. This is because we do not support cycles in binding graphs, at this stage. Recurrency is not typical in component systems, however, we intend to analyze this scenario as part of our future work.

The next issue is the cost of bringing the system to an optimized state. For that, we will characterize the cost of optimizing a single binding graph.

For a two-node binding graph, measurements are presented in Table 1. Time

Table 1: cost of optimization

metric	value (clock cycles)	physical time
detection	$\simeq 5000$	17 nanoseconds
gluecode generation	$\simeq 1.2 \times 10^7$	4 seconds
isolation	$\simeq 50$	-

Table 2: cost of optimization - sensitivity

metric	independent variable	cost increment (clock cycles)
detection	method	$\simeq 2000$
gluecode generation	node (component)	$\simeq 6 \times 10^6$

was measured using a precise timer, jtimer [13], capable of measuring time at the level of number of CPU clock cycles. For reference, we have also presented the physical time equivalent. Measurements were taken on a Pentium M, 1.7 GHz with 1GB of RAM. We used a Sun JVM 1.5, which we started with the *-server* and *-XX:CompileThreshold=5* options, in order to ensure that optimizations are performed to the full extent possible and as early as possible. All measurements were taken after the virtual machine had a chance to perform its own optimizations, which was ensured by running repeatedly the measured area of code. Due to operating system scheduler switches, it was not possible to obtain "stable" results when repeating measurements, and therefore we report them as approximative.

The results indicate that gluecode generation imposes, by far, the highest cost. This is because Velocity is IO intensive, and because the files generated need to be compiled. Other code generators will be investigated as part of our future work. Since applications converge to an optimized state, the cost of optimization can be ignored, if it can be performed very early; practical strategies for doing so have been previously described [6].

We further explored the sensitivity of the above costs to the complexity of the call graph. We have found that detection is sensitive to the number of methods to be processed, while gluecode generation is sensitive to the number of nodes in the graph. These are summarized in Table 2.

Optimized system

We measured the runtime characteristics of our prototype in terms of space (memory) overhead and cost of a method call, when no CMS services are executed (fully optimized case).

Space overhead is caused by the number of gluecode variants and increased nam-

Table 3: performance values

metric	value	**reference**	value
size of behavior object	10 K	**JVM memory footprint**	3.5 M
method call overhead	4 clock cycles	**java method call overhead**	4 clock cycles

ing service instances. To measure the space requirements of any of these, we applied the following procedure: before an object is instantiated, we force a garbage collection; next, we check the amount of free memory available, instantiate the object, and check again for the amount of free memory. The difference in free memory is the size of the object.

As it can be seen, the space impact of one gluecode variant represents a very small fraction of the overall java space requirements; besides, a typical EJB server has a memory footprint of around 60M, which makes this footprint negligible. In terms of time savings, there is no cost attached to an optimized method call, as it is identical to a regular java method call.

6 RELATED WORK

Operating Systems

Context switching optimizations were analyzed in the domain of operating systems (OS). For example, the authors of [14] optimize thread-related context switching overhead, by analyzing liveliness information of context elements (such as registers). In [15], 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.

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 our approach 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).

The JBoss application server [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.

7 CONCLUSIONS AND FUTURE WORK

We presented a method for reducing the overhead introduced by boundary checks in contextual composition frameworks. Based on a formalism of boundary conditions and associated context management services, a self-adapting component platform has been prototyped. This prototype shows that it is possible to achieve no-overhead inter-component calls, when such calls are optimized. The cost of optimization is

an increased memory overhead, which our measurements show as being negligible.

Our solution, although operating on a platform inspired from EJB, could be applied in other areas and other platforms exhibiting similar characteristics, and where space overhead is less important than processing overhead. Of particular importance is the fact that the platform can be extended to support different CMS. In EJB, for example, a number of efforts (JBoss, for example) intend to extend the number of services executed between components.

A liability of our solution is that incorrect rules in the rule engine could lead to incorrect decisions being taken, with loss of application semantics as an effect. However, since the definition of new CMSs is not expected to be a very frequent event, this issue should have a minimal impact. We are currently analysing the possibility of off-line rule verification.

REFERENCES

[1] Sun Microsystems. Enterprise JavaBeans Specification. http://java.sun.com/products/ejb/docs.html#specs, 2003.

[2] Microsoft Corporation. Microsoft Transaction Server Transactional Component Services. http://www.microsoft.com/com/wpaper/revguide.asp.

[3] Object Management Group. Corba Component Model. http://www.omg.org/technology/documents/formal/components.htm, 2002.

[4] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of OOPSLA 2002*, pages 246–261. ACM Press, 2002.

[5] Glen Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and Removing Performance Bottlenecks in Large Systems. In *Proceedings of ECOOP*, 2004.

[6] Mircea Trofin and John Murphy. A Self-Optimizing Container Design for Enterprise Java Beans Applications. In *Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004)*, pages 52–59, 2004.

[7] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, second edition, 2002.

[8] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[9] Adrian Mos and 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.

[10] Stuart Russel and Peter Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, 1995.

[11] Ernest J. Friedman-Hill. Jess, the Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess/, 2003.

[12] The Apache Jakarta Project. Velocity. http://jakarta.apache.org/velocity.

[13] Distributed Systems Group, Charles University in Prague. JTimer. http://nenya.ms.mff.cuni.cz/.

[14] Dirk Grunwald and 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*, 1996.

[15] Erik Johansson and Sven-Olof Nystrom. Profile-Guided Optimization Across Process Boundaries. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, 2000.

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

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

[18] Zahi Jarir, Pierre-Charles David, and Thomas Ledoux. Dynamic adaptability of services in enterprise javabeans architecture. In *Seventh International Workshop on Component-Oriented Programming (WCOP 2002), part of the 16th European Conference on Object-Oriented Programming*, 2002.

[19] The JBoss Group. JBoss Administration and Development Documentation - eBook - 3.2.1. http://www.jboss.org, 2004.

ABOUT THE AUTHORS

Mircea Trofin is a PhD research student at the School of Computer Science and Informatics, University College Dublin, Ireland. He can be reached at mtrofin@acm.org. See also http://pel.ucd.ie/~mtrofin

John Murphy is a Senior Lecturer of Computer Science at the School of Computer Science and Informatics, University College Dublin, Ireland. He can be reached at j.murphy@ucd.ie.