Split Context Model for Software Components

Wenxu Tong   Srikanth Krishnamurthy   David H. Lorenz
College of Information and Computer Science
Northeastern University
360 Huntington Avenue,
Boston, MA 02115, USA
lorenz@ccs.neu.edu

Abstract

The concept of a context is an integral part of a software component. Currently, most context models are restricted to single address spaces. When extending the concept of a context from a local computing model to a remote computing model, issues with respect to multiple address spaces need to be considered. Currently there is a dearth of structured ways of representing and evaluating the various alternatives in designing remote context models.

In this paper we address the issues pertaining to extending the component based models designed for a local computing model to a remote computing model. We address the issues of containment, membership and services by providing a structured classification of models. We discuss the concepts of equivalence, unification and matching of contexts and propose six split context architectures and discuss how requirement-based relaxation approaches increase the flexibility of models. We provide two implementation examples for two of our models to enhance understanding of the proposed models.

1. Introduction

The relationship to the environment is an integral part of the software component. The importance of the context is emphasized in the definition of a software component: "A software component is a unit of composition with contractually specified interfaces and *explicit context dependencies* only" [14]. Considering that there can be multiple component environments, it is necessary that the component's specification include context dependencies explicitly.

The concept of a context has formed the basis for component architectures. On the client side, the JavaBeans Runtime Containment and Services Protocol specification [7] details that "the container or embedding context not only establishes the hierarchy or logical structure, but it also acts as a service provider that components may interrogate in order to determine, and subsequently employ, the services provided by their context".

On the server side, the enterprise JavaBeans specification [5] emphasizes the importance of the EJB container in providing the environment in which EJB's execute. The container provides services to beans such as transaction processing, persistence, security, etc. The EJB can make a call to another EJB in the same container or in another container. EJB clients use a naming service (e.g., JNDI) to lookup another EJB, getting reference to an object that implements its home interface.

The concepts of containment, membership, and services are paramount in developing any component based architectures. Containment is the simple object composition technique where one object holds an exclusive reference to another [13]. A service is a contractually defined behavior that can be implemented and provided by some component for use by some other component, based solely on the contract [1]. A contract means an interface that precisely defines the syntax of a single behavior.

The term context instead of the term container is used for emphesizing that it prescribes the details of services, membership and containment. A context provides the necessary infrastructure and support for a component to exist and for the component to provide its own services to clients [4]. The component describes its requirements to its context, which may be embedded within another environment. This implies that a context can host another context or component provided that context or component fulfils the contract. From a programming standpoint, contracts are usually specified as interfaces. For proper functioning of the system, component respects the context's integrity and vice-versa. A breach of contract may result in the context rejecting the component or denial-of-service. A component leases services through mechanisms which may be available locally through its container.

Each component may depend on its environment to operate and for services. We use the term local computing (e.g., local object invocation), we mean programs that are confined to a single address space. In contrast, remote computing (e.g., remote object invocation) refers to programs that make calls to other address spaces, possibly on another machine [12]. In designing component based models in a remote environment, some of the issues to be considered are:

- Can we have virtual contexts in remote environments? Will a virtual context provide all the functionalities of a real context?
- Is it possible to abstract two contexts thriving on two physical systems and address spaces into one logical system across multiple address spaces?
- What mechanisms are used to facilitate inter-context communication?

In designing a distributed split-context model, the following fundamental questions must be considered:

- How to create a new local (remote) component in a remote (local) context?
- How to associate an existing local (remote) component, which may or may not be already contained in a local (remote) context, with a remote (local) context?
- How to unify two existing contexts already containing components into a split-context in a meaningful way?

Outline. Section 2 describes our classification and the various split-context models. Section 3 demonstrates two possible implementations of our proposals. Section 4 describes background, related work, and presents our conclusions.

2. Comparison of Centralized versus Distributed Context Models

In a centralized context model, the context and components are all local. In a distributed context model, the context or some of the components may be remote. For simplicity, we consider first a split-context model as the simplest kind of distributed model.

In this section we describe the traditional centralized context model, and compare it with six proposed models for splitting the centralized context model:

1. Proxy-based split context model (Sect. 2.3).
2. Delegate-based split context model (Sect. 2.4).
3. Hybrid split context model (Sect. 2.5).
4. Narrow-unionable split context model (Sect. 2.6).
5. Global-unionable split context model (Sect. 2.7).
6. Total distributed context model (Sect. 2.8).

2.1. Definitions and terminology

In comparing the proposed models, we use the following terminology.

Equivalence. The sense of equality relation between two context components can be defined to be either physical or logical. We say that two context components are *physically equivalent* if they are referring to the same actual memory address. Physical equivalence context components share a single object identity.

In contrast, models have the freedom of choosing the criteria of what *logically equivalent* means. Typically, two logical equivalent context components should have a matching signature and conform functionally. The differences among the six proposed models are largely due to the difference criteria set for what logical-equivalent means in these models.

The exact definition of signature and signature matching dependents on the chosen semantics. Conceptually, two signature matching entities should be intended for the same purpose and replaceable by each other. One possible definition of signature of a component can be the public interface of the context, including the signature of public set of properties and services. Here, a signature of a service is the signature of a method in the normal OO sense, including the service name and the argument list.

Unification of two contexts. Unify is a binary operation combining two contexts in a meaningful way, after which there is either a single context or two equivalent contexts containing the components contained in both contexts before the operation. This operation is commutative, but not associative. Physical equivalence and logical equivalence, correspondingly induce *physical unification* and *logical-unification*. For physical unification, after the operation, there is only one physical context, which contains all the components from both contexts. An example of a possible implementation could be to simply create a new context and put both contexts as components into this context. Another example could be adding one operand context as a component into the other operand context. Or it could be implemented as adding the components from each operand contexts into a split-context.

Physical-unification is straightforward, but logical-unification is more interesting and versatile. Since logical

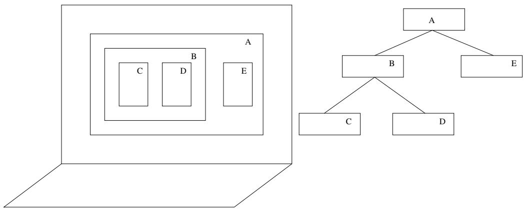

Figure 1. Containment relationship in the centralized context model can be represented as a tree structure. Every component is represented as a node. The components C and D are contained as children of the context component B, which is contained in the context A.

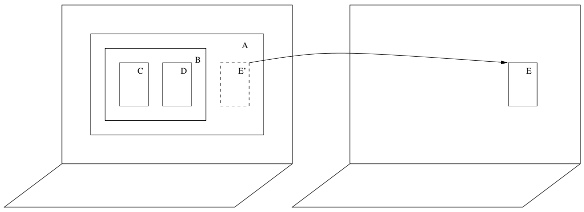

Figure 2. Proxy-based split context model: The remote component E is included in the local context through a reference E'.

equivalence is a model-defined criterion, the actual result of unification is also model dependent. Also, like shallow-copy and deep-copy, based on whether the unification stops at the direct components contained in the operand contexts or it is carried recursively into each layer of the components, there are *shallow-unification* and *deep-unification*.

Since context and containment relationship is fundamentally a tree structure (Figure 1), the parent field is a special case compared with all other fields and members of the context component. The different models differ in how they treat this special field.

2.2. Centralized context model

The centralized context model has the most stringent constraint that every component in the system must be local to the system and must be both physically and logically distinct from other components in the same system. An example of the centralized context model is Sun's current BeanContext model [10, 7]. All the BeanContext components are organized in a tree structure. A BeanContext may or may not have contained components. The contained components can be either BeanContext components or BeanContextChild component. Each BeanContext has only one containing BeanContext parent, except for the topmost one, which does not have a parent.

An important property of the centralized model is that every context component in the model is a distinct instance both logically and physically. They may share some features, like the containing BeanContext component, but apparently, each of their components has its own memory address and its own properties. Since the whole system is centralized in one host, there is no need to support more than one BeanContext component objects to be logically equivalent while occupying two memory locations in the standard centralized BeanContext model.

In sum, the standard centralized model has the following characteristics:

- All components are local.
- Each components has at most one parent.
- Logical equivalence is physical-equivalence.
- Unification undefined.

2.3. Proxy-based split context model

Since remote component is represented by a reference in local context, there is no need for two physical-distinct components to be logical-equivalent. Logical-equivalent in this model is still physical-equivalent.

In the proxy-based split model, all properties of the traditional centralized context model are kept, except for the restriction that every component has to be local. Instead, the model permits a local proxy/interface to an actual component on the remote side. Every service request to that component is sent through the proxy. The same mechanism is use for getting the result back (Figure 2).

This model supports unification. Physical unification is achieved by creating a proxy for one operand context and put it into the other context. It does not support more than one component objects to be logical-equivalent while occupying two memory locations in the model.

In this model, when a remote component is associated with a local context, a reference to the remote component is created and kept as a proxy in the local context. But the architecture of the remote component matters in implementation. One possibility is that the remote component is an orphan-component, i.e., does not have a parent. If that is the case, the local context may serve as a remote context for that component. On the other hand, if the remote component already has its own parent, then there are two cases to consider. In the first case, if the service requested by the local context from the remote component does not require the remote component to have knowledge of this local context, the remote component can still perform the task and send back the result. In the other case, if the task has to be carried out with the information of this local context, this model is insufficient and another model should be used.

The properties of this model are summarized as follows:

- Some components are remote.
- Every local components has at most one parent.
- Logical equivalence is physical equivalence.
- Service provided at the remote side is executed remotely.
- Physical-unification is achieved.

This is the simplest and the most straightforward split-context design. Applying this design to the BeanContext model, the only advantage gained in this extension is that now we can include remote BeanContext components in a local BeanContext and put its services into the local environment. By doing that, local BeanContext components can access services from the remote BeanContext components and since the remote components are now contained in this local environment, it can also access the services provided in this local environment. This model is useful in some simple but important applications like printing a local file on a remote printer.

2.4. Delegate-based split context model

In this model, two physical-distinct components (i.e., having two different memory addresses) are logical-equivalent if they are deep-matching, in other words, one is an exact copy of the other.

Conceptually, a remote component can be represented in a local context as a copy of that component. In this model, an exact delegate copy represents a remote component locally:

- Local, logical-equivalent copy of remote components
- Every local components has at most one parent
- Locally logical-equivalent is physical-equivalent
- Between local and remote components, logical-equivalent is deep-matching
- Service provided at the remote side is executed locally
- Physical-unification achieved

This model can have some performance gain by saving the network traffic caused by sending request and receiving the results. In some cases, this model may have a better performance compared to the proxy-based split context model. This is especially true if the state of the remote component does not change much and all of the service which that component provides can be performed by its exact local copy.

For example, a local context may have a divider component that does the division, and it also has a set of numeric data. Now, it wants to compute the average of the set of data. But the local context does not have an adder, which calculates the sum of a set of data. On the other hand, there is an adder in a remote context. In a proxy-based split context Model, the whole set of data has to be sent to the remote adder to get the sum and then the result is sent back and used to be divided by the number of numbers to get the average. There is an alternate way to carry out the task if we can make a copy of the remote adder locally and perform the summing locally. The delegate-based context model is designed for these situations.

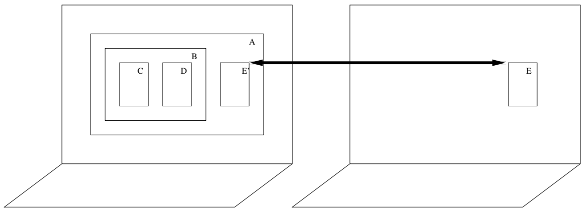

Figure 3. Delegate-based context model. A remote component (E) is included in a local context by a copy (E').

The delegate-based context model can provide some optimization to the simplest proxy-based split context model. It is useful in the case where a remote component is to provide some format changing service, or some mathematical analysis of local data.

Although we now allow two physically different component (the local copy and the remote component) to be logically equivalent, this model is functionally the same as the model that requires all objects to be physically and logically distinct. Since logical equivalence is defined here as deep-matching, one of the logically equivalent components can always be changed into a proxy without losing any functionality. Functionally speaking, this model is equivalent to the proxy-based split context model. The only gain in such a relaxation is just having some efficiency advantage in some cases. So, introducing only deep-matching is just some kind of optimization in implementation of our simplest model.

2.5. Hybrid split context model

If we mix the proxy-based split context and the delegate-based context model, we get a hybrid split context model. A local context in this model contains a remote context component through one of the following two ways:

- The remote component is represented by a proxy, in another words, a reference to the remote component itself.
- The remote component is represented by a local copy, which is a local object with different memory location

but deep-matching the remote component object. Extra mechanism must be implemented to keep the two objects always consistent, in a sense that they are always deep-matching. This hybrid model has the advantage from both proxy split context and the delegate-based context model.

While in most cases the above three models are good solutions for most situations, more relaxation is sometimes desired. Suppose we want to implement an application to play chess. Logically, there is only one chess-board, but physically, we need this component to exist on both hosts. In general, this is true for all visual components that represent the same object on more than one hosts, or processes. Moreover, some features of the visual component may be platform or host dependent.

Let us consider how we can apply logical-unification operation in a distributed context model. If we just use our deep-matching definition on 'logical-equivalent', we will face a difficult choice: what is the parent of the resulting context? Since we can only have one parent for each context, and since we use deep matching, then the unification operation cannot be defined if you want to operate on two contexts with different parents. Of course, one simple solution is in order to perform such operation, we use deep-unification, in another word, propagate the operation upward to unify the parent until we reach the topmost level of one context and by doing this, a local operation will be propagated to a global operation, which we think is not a very good approach. Another approach in extreme is to put another restriction disallowing the operation to be performed on two contexts with different parents. Apparently, this approach sounds too restrictive.

Since the problem comes from the constraint that each context component can have at most one parent and for two contexts being logical-equivalent implies somehow the parents of the two should be the same. So one way to solve the problem is to relax these constraints. The following models are based on different ways to relax this constraint.

2.6. Narrow-unifyable Distributed Context model

- Logical-unification operation is defined on this model
- Every components has at most one parent
- Parent field is excluded from the matching of two components
- Local context may be logical-equivalent to remote ones
- A remote component can be represented using model 3.2-3.4

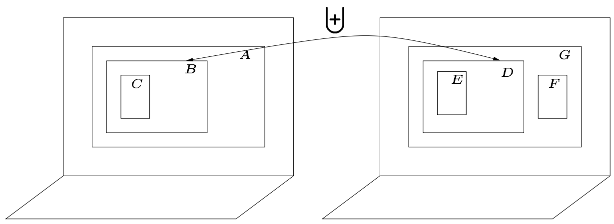

Figure 4. Before unifying.

In this model we still keep the constraints of letting one context have at most one parent, but relax the requirement of the parents to be the same for two components to be logically equivalent. Logical-equivalent is defined here as two components are deep-matching in all their members and operations, ignoring their properties, including parent fields. What this model provides is after the unification, two components merge into one component with the members and services within the components themselves, but not the environments they reside in, and the properties usually specific to the component itself.

A good analog for this in real world may be that in both Engineering College and the LAS College there is a computer science department and we want to merge them into one department so everyone within the two departments now can share the resources. But we don't want to merge two colleges and merge their resources as a result. So the rule is if you were from Engineering's computer science department, after the merge, you can access all the resource used to belong to the computer science department from LAS College plus what you could from the Engineering College, and if you were from LAS college and now you can access all the resource used to belong to the computer science department from the Engineering College plus what you could originally access from the LAS College. Also, the properties belonging to the original departments still remains unchanged for the departments.

So a Narrow-unionable Distributed Context model is a solution for the chess-board problem. It relaxes the constraint one step further than a delegate-based Context model by allowing two components to be logical-equivalent as long as they have every members and operations the same. In this model, two context components can be unified to let both components contain its original members as well as the members of the other components. But the two components keep their properties, including their environment, or containing context (parent) unchanged.

2.7. Global-unionable split context model

- Logical-unification and deep-unification is defined on this model

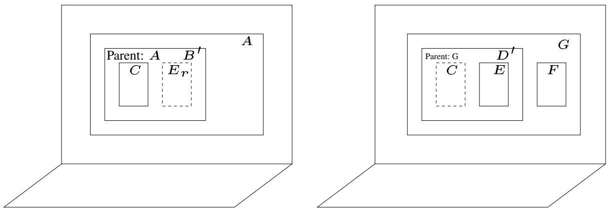

Figure 5. Narrow-unifyable Distributed Context model. To unify two contexts of Figure 4 (B and D), remote components (E' and C') are created into the local context, through either reference or copy. The two resulted contexts B and D still keep their original parents. (A for B and G for D.

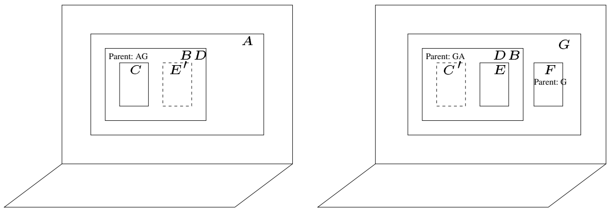

Figure 6. Global-unifyable split context model. To unify two contexts (B and D), remote components (E' and C') are created into the local context, through either reference or copy. The two resulted contexts BD and DB keep their original parents as the first context in the parent list and the context of the other operand as the second in the list (AG for BD and GA for DB) Notice that A and G themselves are not unified, F still does not have A in its parent list.

- Every component can have a list parents
- Logical-equivalent is deep-matching in members and signature-matching in operations and having the same set of values in parent fields
- Components must have same set of parents to be matching
- Local context may be logical-equivalent to remote ones
- A remote component can be represented using model 3.2-3.4

Logical-equivalent is defined as deep-matching in members and signature-matching in operations and having the same set of values in parent fields, with possibly different order.

A global-unionable split context model uses a list instead of a single value to represent the parent/contained-context. Two logical-equivalent components have the same set of components in their parent lists but possibly with different orders. This model guarantees the protocol matching of two logical-equivalent contexts.

The fact that Narrow-unionable Distributed Context model ignores the parents in unification may make the matching criteria too loose in some cases. In some application, users may want to let components in both contexts after the unification access all the service provided in both environments before the union, with some preference, like local service is preferred to the remote one if their protocols (signature) are same.

In this case, we introduce another model, global-unionable split context model. In this model we relax the constraint by allowing two context components to have a list of parents and we define 'logical-equivalent' of two components if their parent-lists contain the same set of elements, and all the their contained components are logical-equivalent in the same sense. But at run-time, the search of parent-list is order dependent, then we can provide some flexibility for contexts to have some local environment specific information as well as some guarantee to protocol conformance. We can either predefine or leave the choice to users/designers to specify which order (depth-first search/local-parent-first or breadth-first search/remote-peer-first) they want to use to find a service from the environment of the context component. In this model, the unification of two context components is for both components just appending the parent- list of the remote one to the parent-list of the local one, and adds the members of the remote components to the local member if they not exist already.

In the case as we mentioned in section 3.2, if a service by a remote component is requested, and the task requires context information from both local and remote context, this model is a good solution.

2.8. Total Distributed Context model

- Logical-unification and deep-unification is defined on this model
- Every component can have a list parents
- Logical-equivalent is signature-matching in members and operations and having the same set of values in parent fields
- Components must have same set of parents to be matching
- Each context may have local components and/or global components
- Global component can be shadowed by a signature-matching component, another implementation by its logical-equivalent peer

- Local context may be logical-equivalent to remote ones
- A remote component can be represented using model 3.2-3.4

A total Distributed Context model allows each of the logical-equivalent contexts to use a locally defined component to shadow a globally defined corresponding component, if they are signature matching. This model is the most flexible and the most complex one in our discussion.

If we go one step further than the global-unionable split context model by not only allowing the parent field of two logical-equivalent context components to differ, but also the contained component with the two logical-equivalent context components to differ as long as they are signature-matching, we get a total distributed context model. In this model, a locally defined copy of the component may have the same interface/signature as the remote component. But the implementation of each service can be either same or different to the remote one. Also, properly defined, even the membership can be the same or different. Local membership or service can shadow the signature-matching remote ones. This way, each logical context provides some kind of interface or super class with some default implementation, while each physical copy is a concrete implementation.

This design gives the distributed context model great flexibility. When a new component is added to one of the contexts, user can specify this component to be one of the following kinds: as a global/shadowable component, which allows all other remote copies of this component in a logical-equivalent context of this to access and shadowing it, or as a locally used, independent, and non-shadowable component. When unifying two contexts, the contained member components are unified in this way, if a shadowable component in one of the context does not have a signature matching counterpart defined in the other context, a copy or referrence will be created in the latter context. If there is a global/shadowable component in one context and a signature-matching global/shadowable counterpart also exists in the other context, each context keeps its own local copy after the unification. If the component is non-shadowable, then no matter whether there is a signature matching component in the other context, no remote copy of this context will be created in the other context.

Using this model, we can provide a mechanism that within a logical context, any services provided by a containing remote component can be accessed or shadowed with a possible local implementation with possible local features. Because these features can either be stored in a property field of this context, or as a shadowing member component which shadows a component provided by the other complex. At the same time, a component without local specific features can be declared as a shadowable component where global features and services are provided to another unifying context if there is no such feature or service defined there. On the otherhand, if the whole purpose of a component is for local use and it is not designed to be used anywhere else for some reason, it should be declared as a non-shadowable component. But this flexibility comes with a price that the behavior consistency may be lost in this model.

The global/shadowable and local/unshadowable is similar like the remote services and local services in EJB. Both global/shadowable components in our model and remote services are designed to be usable in another context, while the local/unshadowable components and local services are only for use in this context only. But in EJB model, every service has its own name. In EJB, an exact name and a naming service is needed to discover and invoke the service. Our model provides an overriding mechanism in service discovery which is not in EJB model.

The global/shadowable and local/unshadowable feature is also similar to the virtual and non-virtual method in C++ programming language. But it is more flexible in our model than typical inheritance in C++ or other object oriented languages. In a typical inheritance case, a subclass inherits a super class by inheriting all the super class's properties and methods. In object oriented design, if there are a group of related subclasses, a typical design would be abstract all of their common features and put that into a super class and let every subclass inherit from that super class. All the common features have to be implemented in the super class, and let special features reside in each subclass. The object oriented model is not symmetric in a sense that the super class is somewhat special to the subclasses. But in the total distributed context model, the context can be totally equal in a sense that each remote context component can have its own global/shadowable and local/nonshadowable properties and services. If it is declared as global/shadowable, then it can be accessed and shared by all its peer contexts of the same kind in another environment, as long as the later does not have a corresponding local property or service shadowing it.

A potential inconsistancy may arise in this model if we try to unify more than two contexts. Let us consider if we want to unify three context. If a shadowable components is defined in two of the three contexts but not the third one, after the unification, which of the two shadowable component should the third context have? We do not have a perfect solution for this. But we do propose two possible approaches to this inconsistancy. One way is the unification is a binary operation and it is not associative. So the order of the unification matters. Which of the shadowable components the third context gets depends on the order of the unification. This is a straightforward solution. Another approach is whenever you declare a component/property /service to

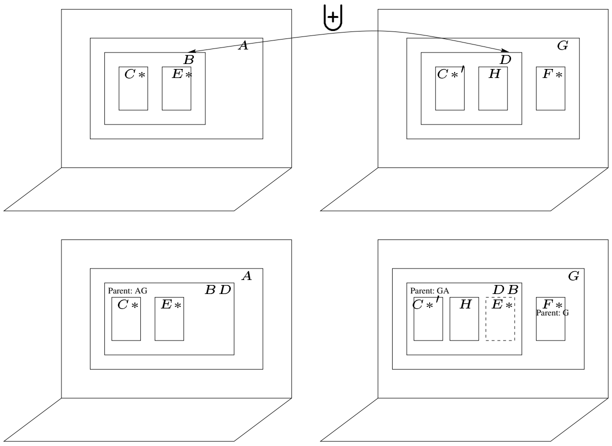

Figure 7. Total-unifyable Distributed Context model. $C*$ and $E*$ are shadowable components in context B. $C*'$ is shadowable component in context D and is signature-matching with $C*$. H is a local/non-shadowable component in D. $F*$ is a shadowable component in G. After the unification of B and D, the resulting context BD and DB each keeps their own shadowable component $C*$ and $C*'$ respectively. DB contains a remote component of $E*$ since it is a shadowable component in B and there is no signature-matching component in D. H is not a remote component for BD because it is local/non-shadowable. $F*$ is not a remote component for BD because we are unifying B and D where $F*$ is not a contained componentent is either of them.

be global/shadowable, you can specify whether it should be set as default in future unification. The rational is to only declare a component with only global information and generally useful even in other environments to be default shadowable. When unification is performed, it will check from all the signature matching properties/components/services of the contexts to be unified, there is at most one default global/shadowable defined. If there are more than one declared default properties /components/services, the user can either change one of the components signature, or randomly pick one as default, or simply not allow such unification. Either one of the approaches and a lot others can solve the consistancy problems.

This model is useful for our chess-playing example. We can create the board configuration component as a global component in one players context and let the context of another player either have a reference to this component or have a deep- matching copy of this component, while we also have another Display component to display the board according to the board configuration state. This Display component is just a signature-matching pair which provides the same kind of service but through different behavior in two different environment.

We realize in many BeanContext cases, especially in the traditional and centralized applications, the model proposed above may be more powerful, flexible and inevitably more complicated than needed.

In all above models, extra care has to be taken to ensure that a unification cannot be performed on a BeanContext component and its direct or indirect member, otherwise infinite loop will form in service discovery and possibly some other operations.

3. Implementation

All of the models in Section 2 are implementable using current technologies. In this section, we describe the implementation of the delegate-based split model and the narrow unionable context model. To implement this we have extended the Java BeanContext using CORBA (Common Object Request Broker Architecture) as the mechanism to facilitate inter-context communication. We have implemented a simple application in which a BeanContext can contain ColorBean components. Both BeanContext and the ColorBean has a color property, which can be set to different colors, choosing colors from a color palette provided by the BeanContext.

To make the model more interesting, we added a policy that restricts a ColorBean from being contained in a BeanContext if the ColorBean and the ColorBeanContext have the same color. If changing of the color of the component or the container causes a conflict between the container's policy or some of its contained components, the color change takes place, but the conflicting ColorBean is expunged from the container (and placed in the parent of the context). Also, adding a ColorBean into a container with the same color is disallowed. We shall now see how this application is implemented using two of our models.

3.1. Delegate-based split context model

In the implementation of the delegate-based split model,we built our ColorBean application on the system for membership synchronization, property uniform and service sharing. In our test,BeanContext1 has two ColorBean components, each has a color property. We constructed a new BeanContext2 and include the ColorBean components in BeanCoontext1 as BeanCoontext2's remote components. After the setup, we can change the color property of any ColorBean component from a BeanContext and the change in one BeanContext will be reflected in the other BeanContext as well.

The mechanism to keep the consistence and synchronization of the logical context is implemented as follows. Every component has a Hash-table keeping the copied information about itself. Each component has a flag to indicate whether it can be modified. If the state of the contained component needs to be modified, it will check and lock its status, record its current status, do the modification, and inform the modification to all of it copies. If every of its copies are updated successfully, the change is successfully committed and the lock was reset, otherwise, the modification has to roll back and restart the transaction in next try.

3.2. Narrow unifiable context model

In our Narrow unifiable model, BeanContext1 has ColorBean1 and BeanContext2 has ColorBean2, after the unification of BeanContext1 and BeanContext2, both have two ColorBeans. The color property of both ColorBeans can be changed from any of the BeanContext and changing the color property of the unified BeanContext will affect both physical BeanContext components as expected. If we add one more ColorBean component to one of BeanContexts, it will automatically appear in another context. If BeanContext1 and BeanContext2 has two different restrictions for adding new components, the new restrictions of the unified context is the unification of both sets. If other rules for combining restrictions is preferred, it can be easily set. Thus, we achieved the unification of membership, property and service of two BeanContexts after unifying them.

The implementation for this example is designed as follows. Each of context has a list to keep track of its components. A flag is associated with each component to indicate whether it belongs to local or the other side before the unification. After the unification, both context will have the components from the other side added into its own set. [1]

3.3. Lesson

Our implemented models and ColorBean test cases indicate that these split context models are implementable. We can associate/disassociate an already existing remote component into a local context. We can change properties of a remote component. To ensure consistency, we think the creation and destroying of a component should be issued locally. But if for some reason, the remote creation/destroy of a component is a desired feature, it can be easily incorporated into the model with suitable mechanisms. The remote components and contexts behave exactly as expected and the location of component and context is transparent to users. We can also use unification to make two split context into one logical contexts. The ColorBean example show that after the unification, two physically distinct components became logically equivalent. There is still some features not specified in the models, like if the unification of two components will cause the violation of their contained components, what should the expected behavior be. One way is to disallow such unification, another would be to expel the violating component. We think it may be more appropriate to leave these issues as implementation policies.

[1] Consistence and synchronization is taken care of through the same mechanism as in the Delegate-based Split model.

4. Discution, Related Work, and Conclusion

There is a growing trend in research emphasizing the importance of a context. The authors in [11] emphasize the importance of context by stating that *"A context can be considered to be everything that affects the computation except the explicit input and output."* The research efforts in the development for context models have been reflected in many industry and research frameworks and systems.

There have been research efforts to extend the BeanContext model to a distributed address space. Webflow [9] is a Java based architecture wherein CORBA is used to connect BeanContexts. Here a WebFlowContext object implements an extended BeanContext interface, maintains a persistent state, controls its children life-cycle, and is responsible for inter-JVM communications through a custom event model. This is similar to the Proxy-based split context model.

The EJB container provides classes that implement the bean's remote and remote home interface. JNDI [6] is used to lookup remote home interfaces and RMI is used to make client-side calls to EJB objects. This again is similar to the Proxy-based split context model. However the EJB container doesn't maintain hierarchical relationships with respect to its contained EJB components. The EJB specification provides for local and remote access of clients (which may be in the same or a remote container) using the same API. However, this depends on whether the client is termed local or remote. Local clients are not location independent whereas remote clients are.

While session Beans can be only one instance, entity Beans can have many instances. Here two EJBs from the same home interface and having the same Primary key are called identical. The two references needn't point to the same object in memory but all instances are considered to have the same identity.

In order to extend local contexts to remote contexts distributed computing mechanisms are used. We have used CORBA to facilitate inter-context communication. The EJB framework uses JNDI and RMI for enabling communication with a remote container or component. For our models, we classify the mechanisms to facilitate communication of split contexts:

- Remote invocation model:

  Stub and skeleton classes are used to provide object-oriented RPC mechanism. Usually a client makes a call for a desired service and gets back a reference.

The method called may be a part of a different address space. But this method is called as if it was locally available. Examples architectures are CORBA and RMI. CORBA [2]

In order for a context to find services located on remote contexts, mechanisms to locate this service must me available. In a local bean context, the hierarchical structure ensures that the child can query its parent to locate its services. When considering remote contexts, extending this would requires some feature by which services can be discovered. Two models can be considered to facilitate this feature.

- Static Services model:

  Similar to DNS, naming services can be used to lookup remote components operating in remote contexts. Consider the EJB architecture for example. Here JNDI is used to lookup an EJB component that returns the reference to an object implementing the EJB's home interface [5]. Another way to find a component is to use CORBA's Common Object Services (COS) [3]

- Dynamic Services model

  When the connectivity between contexts is assumed to be stable, the Static services model can be used to facilitate inter-context communications. However, if the services are registered dynamically and there is no guarantee of the connectivity available, service-oriented architectures using the Dynamic services model can be used. One architecture built on this model is JINI [8]. Here, lookup services are used to get the reference of another context. Services are leased for a particular duration. This architecture can be used to support fully flexible distributed systems.

When extending the concept of a context from a local computing model to a remote computing model, there is an unavailability of requirements-based classification models that discuss approaches, models, evaluating methodologies, alternatives, relative advantages and disadvantages.

In this paper we address the issues pertaining to extending the component based models designed for a local computing model to a remote computing model. We address the issues of containment, membership and services by providing a structured classification of models, depending on the issue addressed, the advantages, the requirements for each model and relaxations or removal of any constraints if required. The distributed computing problem described above has been an active area of research and deserves a separate extensive treatment.

References

[1] The choice is clear: Write to the javabeans component architecture. http://java.sun.com/products/javabeans/.

[2] Corba basics. http://www.omg.org/gettingstarted/corbafaq.htm.

[3] Corbaservices - persistent object service.

[4] Core j2ee patterns. SUN Microsystems press.

[5] The enterprise javabeans specification 2.0.

[6] Java naming and directory interface specification.

[7] The javabeans runtime containment and services protocol specification (v1.0). http://java.sun.com/products/javabeans/glasgow/beancontext.pdf.

[8] Jini network technology.

[9] E. Akarsu, G. C. Fox, W. Furmanski, and T. Haupt. Webflow: high-level programming environment and visual authoring toolkit for high performance distributed computing. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–7. IEEE Computer Society, 1998.

[10] JavaBeans Glasgow specification. JavaSoft. `http://-java.sun.com/beans/glasgow`.

[11] H.Lieberman and T. Selker. Out of context: Computer systems that learn about, and adapt to, and learn from, context. *IBM Systems Journal*, 39(3&4):617–631, 2000.

[12] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant. Note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, Nov. 1994. http://research.sun.com/techrep/1994/abstract-29.html.

[13] D. H. Lorenz and P. Petkovic. Design-time assembly of runtime containment components. In Q. Li, D. Firesmith, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems*, pages 195–204, Santa Barbara, CA, July 30-Aug. 4 2000. TOOLS 34 USA Conference, IEEE Computer Society.

[14] C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.