JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2004; **00**:1–7

Research

A component-based approach to online software evolution

Qianxiang Wang*,†, Junrong Shen, Xiaopeng Wang, and Hong Mei

Institute of Software, School of Electronics Engineering and Computer Science, Peking University, 100871, Beijing, China

SUMMARY

Many software systems need to provide services continuously and uninterruptedly. Meanwhile, these software systems need to keep evolving continuously to fix bugs, add functions, improve algorithms, adapt to new running environments and platforms, or prevent potential problems. This situation makes online evolution an important issue in the field of software maintenance and evolution. This paper proposes a component-based approach to online software evolution. Nowadays component technology has been widely adopted. Component technology facilitates software evolution, but also introduces some new issues. In our approach, an application server is used to evolve the application, without special support from the compiler or operating system. The implementation and performance analysis of our approach are also covered. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: software component, online evolution, component implementation, component interface, J2EE, application server

1. Introduction

Many software systems, especially mission critical software systems, need to provide services continuously and uninterruptedly. In addition, the emergence of new computing paradigms (e.g., grid computing and service-oriented architectures) and new applications (e.g., e-business) are extending the scope of such non-stop software systems.

*Correspondence to: Qianxiang Wang, Institute of Software, School of Electronics Engineering and Computer Science, Peking University, 100871, Beijing, China
†E-mail: wqx@pku.edu.cn

Contract/grant sponsor: National Natural Science Foundation of China; contract/grant number: 60103001
Contract/grant sponsor: National Grand Fundamental Research Program of China; contract/grant number: 2002CB31200003
Contract/grant sponsor: National High-Tech Research and Development Plan of China; contract/grant number: 2003AA115430

Meanwhile, those software systems may still have defects because of time-to-market pressures, limited development technologies and resources (such as funding and human resources) [1]. As one example of Lehman's E-type software [2], these non-stop software systems need to keep evolving continuously to fix bugs, add functions, improve algorithms, adapt to new running environments and platforms, or prevent potential problems. So, online evolution is becoming an important feature of many widely used software systems today.

The concept of *online evolution* was first introduced in some mission-critical areas such as telecommunication, avionics, and industry control [3, 4, 5]. To mission-critical systems, shutting down a system means high cost (think of shutting down a cooling pipeline) and is even life-threatening (think of shutting down a patient's life-support system). According to a Yankee Group report, banks can lose as much as US $2.6 million per hour and brokerages as much as US $4.5 million per hour from downtime [6]. Initially, because old systems had less supporting software, online evolution was mostly hardware oriented, with software playing an auxiliary role during the evolution process. However, as software covers a more significant role in information systems (i.e., software-intensive systems), it plays a more and more important role in today's online evolution approaches [5].

Online software evolution approaches can fall into two categories: hardware-based approaches and software-based approaches [5]. The hardware-based approach is also called the redundant hardware based approach, which is common but costly and complex. In this approach, "a secondary machine is loaded with new code, passed the necessary state from the primary, and 'switched over' to become the primary system" [7]. Compared with the hardware-based approach, the software-based approach is relatively cost effective and flexible, because it needs few hardware supports and less human intervenes. Many software-based approaches have emerged for the online evolution of procedure-based systems, object-oriented systems, and component-based systems in the past decades.

In recent years, component technologies have been well developed, such as Enterprise Java Beans (EJB) of Sun [8], Component Object Model (COM) of Microsoft [9], and CORBA Component Model (CCM) of the OMG [10]. In component-based systems, the dependencies between components are controlled through well-defined interfaces. And components are more independent from each other compared with procedures in procedure-based systems and with classes/objects in object-oriented systems. Moreover, at runtime, these components also have their own life cycles and runtime environments. This loose-coupled feature facilitates online software evolution; updating one component in such a structure has less effect on other components.

This paper presents a component-based approach to online software evolution, and the implementation of our approach is based on the Java programming language and J2EE-compliant Application Servers. The initial research results of this approach were published in ICSM (International Conference on Software Maintenance) 2002 [11]. This paper updates our research results systematically.

The remainder of this paper is organized as follows. Section 2 explains the concept of component-based online software evolution. Section 3 introduces the technical background of our approach. Section 4 presents the online evolution of component implementation. Section 5 presents the online evolution of the component interface. Section 6 introduces some results from the experimental use of our approach. Section 7 discusses related work. Section 8 lists conclusion and future work.

2. What is component-based online software evolution?

To date, there has been plenty of research on software maintenance and evolution. Ned Chapin *et al.*'s work [12] on the classification of software maintenance and evolution can be viewed as a roadmap in this field. Nevertheless, an acknowledged definition for online software evolution is still lacking [13]. In this section, we will try to give a definition of online software evolution, and discuss characteristics and challenges of component-based online software evolution.

2.1. Online software evolution

Literally, online software evolution should be a kind of software evolution. In this paper, we follow the definition of *Software*, *software maintenance* and *software evolution* given in [12]. Software means the non-hardware part of a system being implemented, including associated documentation which is the human-readable text and/or graphics specifying or describing software. Software maintenance refers to activities and processes of modifying existing software. Software evolution refers to the application of software maintenance activities and processes that generate a new operational software version with changed customer-experience functionality compared to the functionality of a prior operational version. Our definition of online software evolution is presented here.

Online software evolution is a kind of software evolution that updates running programs without interruption of their execution.

The key words of this definition are "running programs" and "without interruption".

Running programs are the target of online software evolution. Evolving a software system usually means modifying the programs and the associated documentation (e.g., requirements, data layouts, configuration management listings, regression test plans) [12]. Since the goal of online software evolution is to improve the behavior of the running software, it is costly and unnecessary to change the program and the documentation simultaneously. Thus, research on online software evolution focuses on the evolution of the program instead of both of them.

Without interruption is the constraint from the end user. Requests should not be refused or cancelled owing to the evolution, although the quality of service during the evolution process may decline a little, e.g., the response time may become a little longer.

While traditional offline evolution approaches can be applied to both system software (e.g., operating systems and application servers) and application software, it is difficult to evolve system software while keeping it running[14] [15]. In this paper, we will consider only the evolution of application software, which can get support from system software.

2.2. Component-based online software evolution

A *component* is an independent module that undertakes some specific functions in a software system. A component consists of a *component implementation* and a *component specification*. The component implementation is the code that implements the functions of the component, while the component specification describes how to assemble, use and manage the component. For example, an EJB component comprises Java classes, Java interfaces, and a XML based deployment descriptor. Java classes are the component implementation, and others are the component specification. According to Crnkovic *et al.* [16], a complete component specification usually includes functional properties,

extra-functional properties (quality attributes), use cases, and so on. A key part of the component specification is the *component interface*, which refers to the specification of the component's access point, i.e., a list of operations and attributes. The implementation of a component must be consistent with the specification of the component, especially the interface of the component.

For object-oriented programs, all objects (instances) are created from classes (types). Similarly, for component-based software, we must also separate two kinds of component implementation at runtime: component types, which mean the *class* code in the component package, and component instances, which mean the objects created from those classes. Usually, one component type corresponds with multiple instances. Obviously, only component instances can respond to the requests from clients.

In general terms, component-based online software evolution includes the following actions: component addition and deletion, component implementation update, and component interface update, etc. However, by acknowledging the following constraints, we think it reasonable to reduce the scope of these actions.

The constraint is that online evolution should keep the service promise of the old version component to the clients. The new version of the component should be backwardly compatible with its old version. In the context of service oriented architectures, this constraint becomes more important because all clients wish both the quality and the content of the service to be stable. Therefore, we will consider neither deleting components nor deleting operations from the interface, which might cause the violation of the service promise constraint. So only adding components, updating component implementations, and adding new operations to the component interfaces will be considered in this paper. Meanwhile, as the process of online adding components is similar to the online update of component interfaces (see Section 5), this paper will only discuss the online update of component implementations and the online update of component interfaces (adding new operations to component interfaces) in Section 4 and Section 5, respectively.

2.3. Problems of component-based online software evolution

Updating a component's interfaces usually involves updating its associated implementations, so the fundamental problem of the online evolution for component-based software is how to update the component implementations during runtime. In particular, if a component-based software system is developed with object-oriented technologies (for example, an EJB component is implemented with several Java classes, interfaces, and deployment descriptor), the problem of updating the component implementation will be partly transformed into the problem of swapping objects, although updating the component is more laborious than swapping objects.

Feng *et al.* [17] summarized three important issues for dynamically swapping objects: 1) *Referential Transparency Problem*. Sometimes the object to be swapped may not have enough knowledge to notify its clients of the replacement. 2) *State Transfer Problem*. The state of the object to be swapped, including its components' states (e.g., attribute values) and its current execution status, should be transferred to the new object to keep the application running consistently. 3) *Mutual Referential Problem*. Swapping an object may require swapping multiple related objects and the order of swapping may be important because modules may depend upon each other.

In addition to the problems mentioned above, component-based online evolution should consider more issues as well. Firstly, swapping a component usually means swapping a series of related classes and objects, so it may be not as atomic to swap a component as it is to swap an object, and the swapping

time may be much longer. Secondly, new requests may be sent during the evolution process, although the evolution process usually cost little time. If there are some new requests to the component being swapped, we have to consider blocking these requests and processing them after the evolution. Thirdly, the component to be evolved may have unfinished requests or be enrolled in a transaction. In this situation, the component cannot be swapped; also it is needed to ensure the semantic consistency of the transaction.

3. The foundation of our approach

One of the factors affecting the ability of online software evolution is the runtime structure of the software, which determines whether a functional entity can be managed and evolved easily at runtime. This is a program language and runtime platform related issue as the constructing entities vary from languages and platforms. For example, in order to support online evolution of software developed with structural programming languages (e.g., Pascal, C), the application should have: (1) modularized software abstractions (i.e., procedures and modules), which can easily be replaced; and (2) supports to control the behavior of the procedures or the modules. The first factor depends on the programmer, and the second one usually depends on the runtime platform. Traditional system software, such as operating systems, provides little support to online evolution. In recent years, some new runtime platforms, such as application servers, are being enhanced in this direction. Our approach is based on some important mechanisms of programming languages and application servers introduced below.

3.1. Support from programming languages

Online evolution mechanisms are usually language-specific. Different programming languages provide different facilities for online evolution. In this section, we will take Java language for example to show how to use the Java language-specific features to support the implementation of online evolution. Java provides a lot of mechanisms such as class loader and reflection that make programs flexible and replaceable at runtime.

3.1.1. *Class loader*

Class loader is one powerful mechanism for dynamically loading and linking software elements at runtime on the Java platform. It has the following four features that benefit online evolution: lazy loading, type-safe linkage, user-definable class loading policy, and multiple namespaces [18]. The Java class loader is used to load Java classes, the unit of software distribution of Java language that follows the *class file format* [19], into the JVM (Java Virtue Machine) platform. The JVM platform provides some default class loaders such as *ClassLoader*, *SecureClassLoader*, and *URLClassLoader*. The JVM also allows users to develop their own subclasses of *ClassLoader* to load and manage corresponding classes. The class loader uses a *delegation model* to search for classes and resources. Each instance of the class loader has an associated parent class loader except the *bootstrap class loader*. When called upon to find a class or resource, a class loader instance will pass the search for the class or resource to its parent class loader before attempting to find the class or resource itself [20].

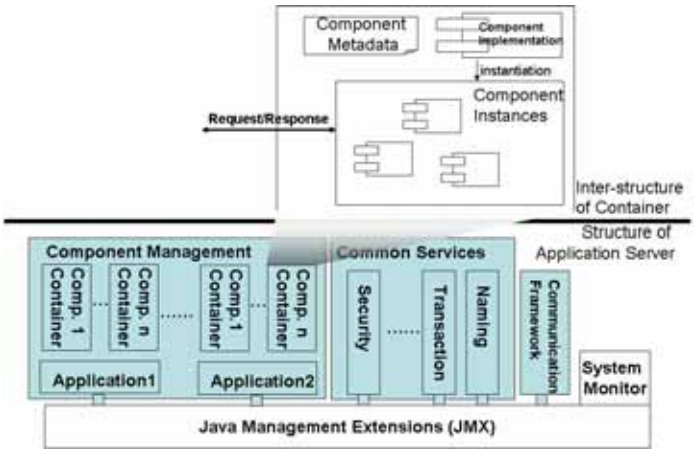

Figure 1. The structure of a typical application server.

3.1.2. Reflection

Reflection is another important built-in mechanism of Java. Through a type-safe and secure API, the reflection mechanism enables Java code to discover and inspect information about the fields, methods and constructors of loaded classes. This mechanism also enables Java code to use reflected fields, methods, and constructors to operate their underlying counterparts of objects [21].

The reflection mechanism provides the following supports for component-based online evolution. Firstly, we can use reflection to inspect the status of the component being evolved. The reflected information is very important for reasoning about the status of the evolution process and for determining the next action. Secondly, reflection can be used to transfer component states such as the values of object attributes, security information, and transaction contexts. These states can be acquired and transferred to the new components through reflection.

3.2. Support from application servers

Application Servers, especially J2EE-compliant application servers (for example, Weblogic, Websphere, JBOSS, PKUAS etc.), are the most widely used platforms for building enterprise applications in recent years. An application server usually provides a component running environment, especially a series of component containers, and a set of common services such as Naming Services, Transaction Services, and Security Services. The structure of a typical application server is shown in Figure 1. In this section, we will introduce component containers and the management framework, which are the most important mechanisms for online evolution.

```
public interface ShoppingCart extends EJBObject {
    public void addItem(String itemID);
    public void setLocale(Locale locale);
    public Collection getItems();
    public void deleteItem(String itemID);
    public void updateItemQuantity(String itemID, int newQty);
    public Double getSubTotal();
    public Integer getCount();
    public void empty();
}
```

Figure 2. The interface of the component *ShoppingCart*.

3.2.1. Component containers

A component container can be viewed as the runtime environment of a component. As the broker located between the component and its clients, a container is responsible for the following tasks: 1) the connection between the component and its clients; 2) the life cycle management of the component; 3) the mapping between the component interface and its implementations; 4) coordination of transaction, persistency and reference for components and so on [8]. With the container's support, both the component and its clients are transparent to each other.

In J2EE based application, EJBs are the core components that implement the business rules of the application. There are three kinds of EJBs in J2EE: *session bean* (including *stateless session bean* and *stateful session bean*), *entity bean* (including *bean managed persistence* and *container managed persistence* according to the persistence mechanism) and *message driven bean*. The application server usually implements different containers for different kinds of components, respectively.

3.2.2. The management framework

Java Management Extensions (JMX) define an architecture, the design patterns, the APIs, and the services for application and network management and monitoring in the Java programming language [22]. In J2EE application servers, all the common services must register themselves into the management framework at start-up time, and all the J2EE applications must register themselves into the management framework during deployment time, so that all of them can be located, monitored and controlled at runtime.

4. Online update of component implementations

Let us start by considering the update of the implementation of one stateful component *ShoppingCart* of Java Pet Store (JPS) 1.3.1 [23], which is powered by Sun Microsystems, contains 15 EJBs, and is freely available as the sample application that conforms to J2EE specification. Component

Table I. Different client behaviors in three evolution stages. In this table, "+" means there are some client requests, for example, adding or removing some products to or from the shopping cart, calculating the total price, etc; "-" means there are no client requests in this stage.

Category	Before Evolution	During Evolution
A	-	-
B	+	-
C	-	+
D	+	+

ShoppingCart maintains the contents of an individual client's shopping cart. The interface of the *ShoppingCart* is listed in Figure 2.

Suppose we want to update the component *ShoppingCart* in order to upgrade its pricing algorithm *getCount*, so as to attract more clients. What problems might we face? Note that if we had predicted the possible changes of the pricing algorithm, we would use *Strategy* pattern to make the algorithm replaceable; but if we had not predicated so at design time, we may need to evolve the delivered component at runtime. For every user, the corresponding *ShoppingCart* instance maintains a collection of *CartItem* objects in a private HashMap, and each *CartItem* is implemented as a conventional Java object that stores both the attributes of an item and the quantity of that item the user wishes to order. We can divide different updating situations into four categories as shown in Table I.

A) There is no client request before the evolution, which means the shopping cart instance is not corresponding to any specific client and contains no order information such as ordered product items, so we only need to update the component type, and do not need to consider component instances.
B) There are some client requests before the evolution, which means there are some instances whose states should be preserved and which should be updated together with the component type. So we need to duplicate the state of the instances of the old *ShoppingCart* component into the instances of the new version, and then let the new ones serve the clients.
C) There are some client requests while the online evolution is in progress, so it is necessary to block and cache the client requests during the evolution and respond to these requests using the new version of *ShoppingCart* after the evolution to ensure the updating process is transparent to the clients.
D) Of the above four categories, D is the most complicated one. To update the component *ShoppingCart*'s implementation at runtime in this situation, we need to 1) update the implementation codes of this component, 2) transfer the states from the instances of the old version to the instances of the new version, 3) block client requests during the evolution and process these requests after the evolution.

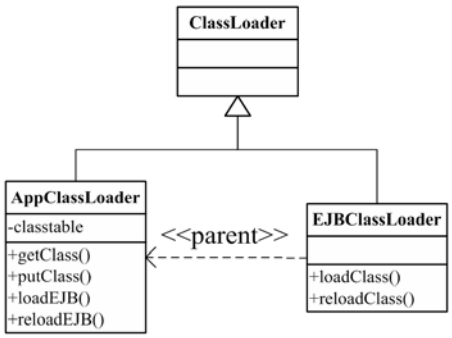

Figure 3. Class hierarchy of class loaders to support online evolution.

Generally speaking, To evolve one component implementation at runtime, we need to handle the following issues: 1) replacing the component type; 2) replacing the component instances; 3) blocking new requests. In the remainder of this section, we will introduce these issues separately.

4.1. The replacement of a component type

As mentioned in the former sections, to upgrade a component type, we need to replace all of the related Java classes: unload/disable old version classes and load new version classes.

The first step of replacing the component type is to develop a new version implementation of the component according to the evolution requirement. There are some special constraints for the new implementation of the component. Firstly, the new implementation should be consistent with the component interface. Secondly, the attributes of the new implementation must be a superset of those of the old version implementation. The second constraint comes from the following fact: as mentioned in Section 3.1.2, we use Java reflection API to retrieve attribute values and duplicate them to the corresponding new objects. If we hope that the container can retrieve and transfer the component states to the new component instances automatically, we must have a corresponding attribute in the new implementation for every attribute in the old version implementation.

Once the default JVM class loader introduced in Section 3.1.1 loads some classes from the file system, it will refuse to load these classes again. So the default JVM class loader does not support the replacement of classes required by online evolution. To facilitate online evolution, we developed an enhanced class loader which can reload a class into the memory, by inheriting the default JVM class loader. Figure 3 shows the class hierarchy of the class loaders. An *AppClassLoader* object is being created when deploying a J2EE application. This *AppClassLoader* uses a hash table named *classTable* as the cache to store loaded classes. Two methods, *getClass* and *putClass*, which are provided by *AppClassLoader*, are used to access classes stored in *classTable*. Methods *loadEJB* and *reloadEJB* are used to load and reload all the classes. Class *EJBClassLoader* is responsible for loading and reloading classes (i.e., class binary code in Java). Method *loadClass* loads the code of classes and stores them into *AppClassLoader*'s *classTable*. Method *reloadClass* is used to load the new class code,

find the class which has the same name in *classTable*, and fill the *classTable* with the new class code. If the *EJBClassLoader* fails to find the file of a class, it will pass the request to its parent class loader *AppClassLoader* through the delegation mechanism explained in Section 3.1.1.

For old classes, we just leave them unused and let them to stay in memory. They may be still useful in some special cases, such as if the evolution process has to roll back because the evolution process was not successful [24].

4.2. The replacement of component instances

The replacement of component instance is the most delicate part of online evolution. Replacing component instances means replacing all of the related objects. Our solution to the online replacement of component instances is based on the evolution-oriented component execution state management (life cycle). Here, the term "state" does not mean the component states such as attribute values that need to be transferred from the old version to the new version as mentioned above. Instead, it means to which situation the component instance belongs. Based on the life cycles of EJB instances specified in [8], we introduce one new life cycle of component instances to facilitate the online evolution of EJBs. The states in our approach are a little different from the states defined in EJB specification. For example, the "passive" state in EJB specification relates with no instances, so we do not need to consider that state here. Meanwhile, just like the life cycles of an EJB defined in EJB specification, life cycles of different kinds of component are different. In this section, we only illustrate the component states of a stateful bean from the perspective of online evolution (see Figure 4). Other kinds of EJBs are simpler than stateful beans: they need address neither the issue of transferring the states of the old instances nor the issues related to transactions.

When the evolution process starts, one component instance may belong to one of the following execution states: pooled, ready, session, and execution. Component instances in different execution states require different handling mechanisms to evolve.

- **Pooled**: In this execution state, component instances have been created from the component type and cached in the *component instance pool* provided by the component container. These instances have not been initialized, so we need just to delete the old instances in the *component instance pool*, create new instances, and refill the pool with the new ones.
- **Ready**: In this execution state, in response to the client's *Create* requests, the component instances in the "pooled" state will be initialized and enter the "ready" state. Here "ready" means the components instance can be, but haven't been invoked. To replace the component instances in this state, we need not only delete the instances in the instance pool and refill the pool with new ones, but also to use the new instances to replace the old ones that have specified relationships with clients, and to maintain these relationships.
- **Session**: In this execution state, the component instance has stored some data concerning the session between the client and the server. We must transfer these session data (component states such as attribute values) from the old instance to the new one. This process is implemented through the reflection mechanism described in Section 3.1.2.
- **Execution**: A client's invocation to some methods of a component instance will take the instance into the "execution" state. Component instances in this state cannot be replaced, since each execution of a method is viewed as an atomic action and can not be interrupted in EJB. For

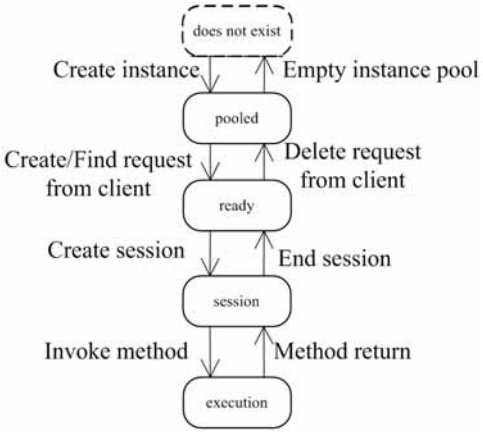

Figure 4. Component states of stateful bean from perspective of online evolution.

service oriented software, the executing time of one method is limited, so we can just wait until the instance finishes the execution of current method and returns to "session" state. Meanwhile, if the component instance is evolved in some transaction, we should not replace the component instance until the transaction is submitted or rolled back, so as to keep the consistency of the transaction.

We use a *component instance manager* to manage these component instances. As one of the key parts of component container, the instance manager is responsible for maintaining the execution states of the instances, caching instances, and forwarding client requests to specific instances and so on. The instance manager provides a *getInstance* method to the container, from which the container can get the reference of the expecting component instance.

The instance manager also maintains the identifier of the component instance, which is one important issue during online evolution. For stateful session beans, the identifier should be carefully maintained, because the identifiers record the one to one relationships between component instances and clients. For stateless session beans, the identifier is generated while creating the component instance, and both the identifier and the instance are stored in a hash table. After evolution, the stored identifier will not be changed although the instance has been replaced. For entity beans, the identifier is the primary key stored in its associated database, so the evolution process will not influence the value of the identifier either.

In the JVM, all the unreferenced objects will be collected and destroyed automatically by *garbage collector*. Thus, replacing the objects which are the instances of the old class just means creating new objects of the new class and moving all of the data related with the old objects to their new counterparts. We do not need to remove the old objects explicitly from memory.

Because all the classes of one component are loaded together by one *EJBClassLoader* and managed by one container, there are no referential transparency problems, as mentioned in Section 2.3, between

objects inside the same component. Meanwhile, objects in different components cannot reference each other directly, but through containers and naming services, so the referential transparency problem between objects of different components disappears as well.

4.3. Blocking new requests

During the updating process, the clients may keep sending requests to the component. As various old versions and new versions of component types and instances may co-exist in a system promiscuously, newly submitted requests have to be blocked to keep the system states consistent. On the other side, if we simply block the whole component during the evolution process, the longer response time will give a higher cost to the evolution.

In our solution, we employ a mechanism named *fine-granularity requests blocking*, which can enhance the performance during the evolution process. When a certain component type is being replaced, we block only "Create" requests which will create new component instances, until the replacement transaction of that component type is finished. When a certain component instance is being replaced, we block only requests to that component instance, until the replacement of that one is finished.

We implement blocking of client requests by locking the instance queue which contains the references of the available component instances, and by providing a request queue to cache the blocked requests. Obviously, these two queues will increase the response times to requests. However, as the class reloading process for the component type and the state transfer process for component instances are usually very fast, and we adopt the *fine-granularity requests blocking* mechanism, the overall throughput of the application should still be acceptable to clients.

5. Online update of component interfaces

According to the discussion in Section 2.2, online evolution must keep the service promise to the old clients. That means, for the evolution of interfaces, the operations in the new interface should be a superset of those in the old interface. In this section, we discuss how to add an operation to an interface and how to add a new interface at runtime.

Before evolving an interface at runtime, there are some auxiliary tasks to do: (1) Analyze the new functional requirements to find out which components are related to the requirements based, on the traceability between the functional requirements and the components; (2) Design a new interface that supports the new functional requirements for the component; (3) Develop a component implementation according to the new interface. To design a new interface, we just need to remember that the operations in the new interface should be a superset of those in the old interface. For developing component implementations, we have two choices: (1) We can develop a new implementation that supports both old functions and new functions (see Figure 5 (b)). (2) If the source code of the old implementation is not available or we want to keep the old code in the new system, we can develop a new implementation that supports only the new functions, while the the old implementation is responsible for the old functions (see Figure 5 (c)). The approach of Figure 5(c) is similar to the aggregation technology of COM [9].

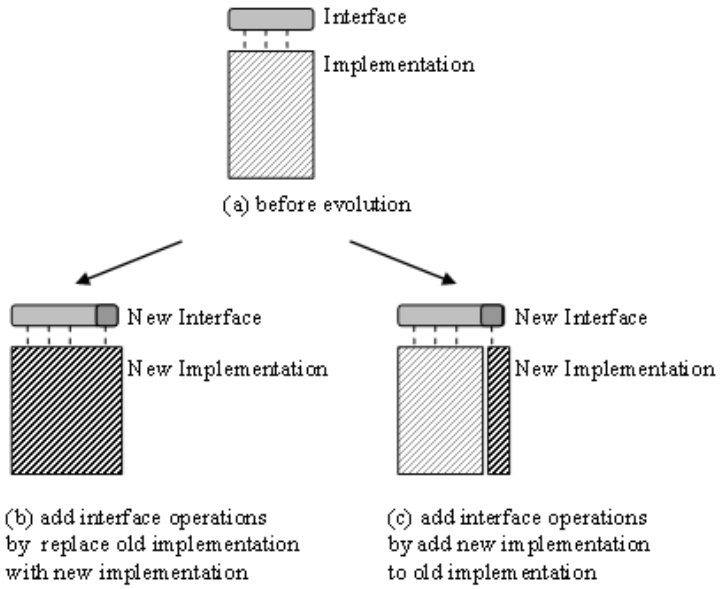

Figure 5. Two different implementations for the new interface.

When both the new interface and the new implementation are prepared, we can start the online interface evolution process as follows:

1. **1) Update component implementations**. If a component implementation was developed according to Figure 5(b), update the component implementation by employing the approach introduced in the previous Section 4. If the component implementation was developed according to Figure 5(c), we can just load the corresponding new classes.
2. **2) Update the mappings between client requests and servicing procedures**. The component container contains a hash table that maintains all of the mappings. So the only thing we need to do is to update this hash table.
3. **3) Generate and load into memory a new skeleton of the new component based on the new interface**. The skeleton is used to handle the request messages from the clients and process, and forward these requests to service procedures.
4. **4) Re-establish the reference relationships between the new component implementation and other components or resources**. This is needed only if the evolution of the new interfaces affects other parts of the software, such as other components or resources.
5. **5) Republish the new interface to the server**. This is needed if the old interface has been published to some naming and directory server (such as *UDDI* for Web Services). Then clients

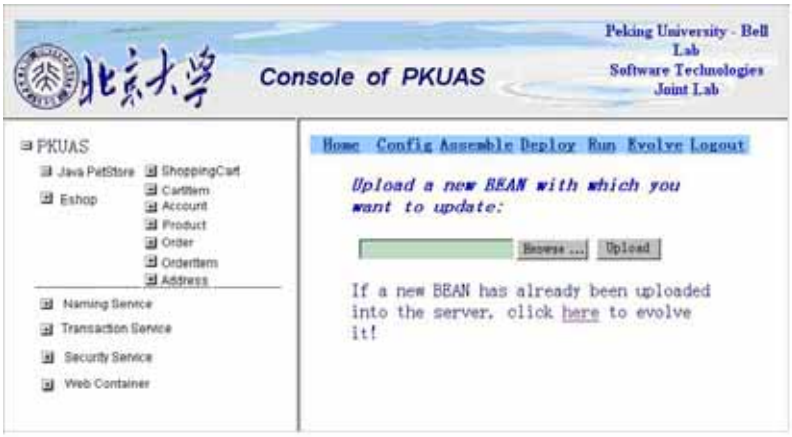

Figure 6. The user interface for evolving a component (select the new component type code).

can download the new version of the component interface, and send requests according to the new interface.

Now let us consider how to add a new component interface at runtime. Actually, it is essentially the same as adding a new component. When both the component interface and the component implementation have been developed, the following main steps are executed, a process that is the same as the process of deploying a component of an application:

1) Create a component container and add it to the application's container list.
2) Load all the necessary classes and description files of the component.
3) Create and initialize some instances of the new component.
4) Publish the new component interface.

6. Demonstration and performance analysis

In this section, we present some data about the results of our experiments with our online evolution approach. We implemented the proposed approach in PKUAS (Peking University Application Server), a J2EE-compliant application server which is developed by authors' research group. PKUAS provides nearly all of the functionalities specified in J2EE v1.3 [25] and EJB v2.0 [8]. In PKUAS, different containers are responsible for different components, and one container holds all the instances of some special component. Such an organization of the containers facilitates the management and the online evolution of components. We will demonstrate the process of evolving an EJB component supported by a web page-based administration console, and analyze the performance of the containers provided by PKUAS.

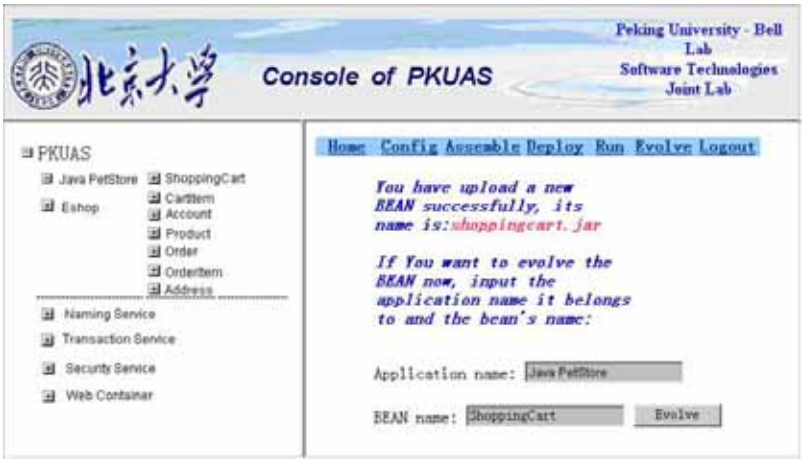

Figure 7. The user interface for evolving a component (input the name of the component).

6.1. Web page-based administration console

We have developed a web page-based administration console which provides management functions such as deploying or un-deploying an application, evolving a component, configuring environment resources, and monitoring system status. To update a component, the administrator needs to logon to the console as shown in Figure 6, select a new version of the component (i.e., a new version of *.jar* file for J2EE), and press the "Upload" button to upload the component from the local file system to PKUAS's component depository.

After uploading the component, PKUAS's console will ask the administrator to input the names of the application and the component to be updated, as shown in Figure 7. If the administrator starts the evolution process by pressing the "Evolve" button, PKUAS will uncompress the *.jar* file, analyze the type of required evolution (for example, evolve only the component implementation? or evolve both the component interface and the implementation?), and then use the approaches introduced above to execute the evolution process.

6.2. Performance evaluation and analysis

Supporting online evolution imposes a number of costs to software systems. At update-time, the container needs to validate the conformance of the new version component, unload the old component and load the new version, block incoming client requests, transfer component states and so on. At runtime, for each request, the container needs to perform some extra operations, such as checking whether a update procedure is in progress. In this section, we present some results of our experiments that measure the influence of these costs.

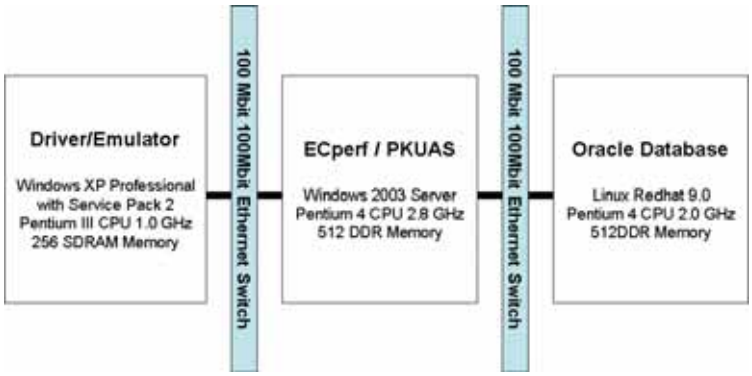

Figure 8. ECperf Experiment Setup

Our experiment consists of two parts. Firstly, we evaluate the runtime overhead of our online evolution solution by comparing the performances of containers that support online evolution and containers that do not. Secondly, we evaluate the update-time overhead by evaluating the amount of time that an update takes and calculating to what degree components slow down when being upgraded.

We use ECperf [26] powered by Java Community Process as the application performance test bed. ECperf is an Enterprise JavaBeans (EJB) benchmark to measure the scalability and performance of J2EE servers and containers. It stresses the ability of EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc. ECperf can be viewed as a typical three-tier J2EE application which covers four business domain: customer domain (implemented by 7 EJBs), manufacturing domain (10 EJBs), supplier domain (7 EJBs) and corporate domain (3 EJBs). ECperf measures EJB containers' performance through many metrics, such as response time, transaction time, and an overall metric named **BBops/min** (the average number of successful **B**enchmark **B**usiness **OP**eration**S** per **min**ute completed during the Measurement Interval) which is the primary overall metric of ECperf benchmark and is calculated by adding the metrics of the OrderEntry Application in the Customer Domain and the Manufacturing Application in the Manufacturing Domain as: **BBops/min = (Transactions + Workorders)/min**. In our experiment, we use **BBops/min** to measure the overhead of our online evolution solution.

6.2.1. Runtime overhead

Compared with the standard implementation of EJB containers, the runtime overhead is only incurred through the use of a update lock named *WriterPreferenceReadWriteLock*, which is acquired before every update starts and released after the update ends. So before every attempt to require one EJB, the corresponding container will always check the lock (using statement *update lock.readLock().acquire()*) to see whether the EJB is being updated. To evaluate the exact performance effect of our online evolution solution on the application server, we implemented two kinds of containers: one supports online evolution, and the other does not.

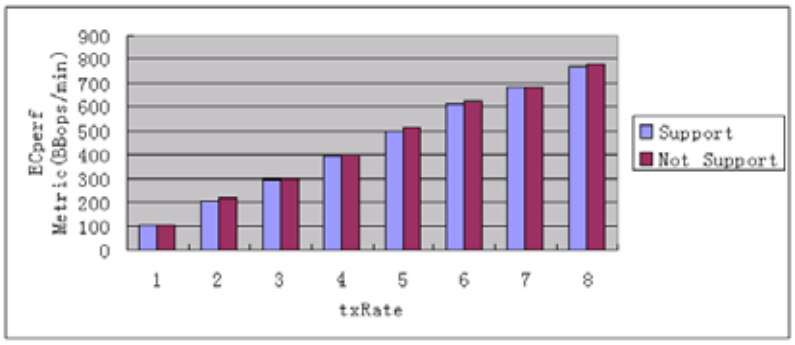

Figure 9. Performance measurements of both containers that support online evolution and containers that do not support online evolution.

Our experimental setup is shown in Figure 8. We use three machines: one machine runs ECPerf client *Driver* including three kinds of agents (OrdersAgent, MfgAgent and LargeOLAgent which simulate the EJB clients and run in separate threads), one machine runs PKUAS with two kinds of containers, and one machine runs the Oracle database management system. These machines are connected with Fast Ethernet (100Mb/s). The software and hardware configurations of them are also listed in Figure 8.

We deploy ECperf benchmark in two kinds of containers, and measure their corresponding BBops/min in different client request pressures. ECperf simulates the typical transactions in e-business applications with a lot of clients sending requests simultaneously and repeatedly. The generation rate of requests (called txRate) determines the number of concurrent client calls. The higher the txRate is, the higher throughput the J2EE application server achieves in the measurement until the server is overloaded.

Figure 9 shows the results of our measurements: when the value of txRate grows from 1 to 8, the performance measured by BBops/min grows from 101.33/102.93 to 768.73/777.33; the performance difference between containers supporting online evolution and containers not supporting is very slight (1.96 % on average). So we can argue that the runtime overhead of our solution to online evolution is acceptable.

6.2.2. Update-time overhead

Our solution also imposes an update-time overhead to block and cache incoming requests during update, unload old classes, reload and validate classes of new components, transfer component states, update component instance pools, unlock the container and process blocked requests.

To evaluate the update-time overhead and its influence, we choose three EJBs: entity bean *OrderEnt*, stateless session bean *OrderSes*, and stateful session bean *CartSes*, and measure their evolution time, respectively. To make our experiment accurate, we build an evolution client which is responsible for firing the evolution commands repeatedly; the interval between two evolution commands is about 4000 milliseconds, which is much longer than the average evolution time to make sure that these evolution

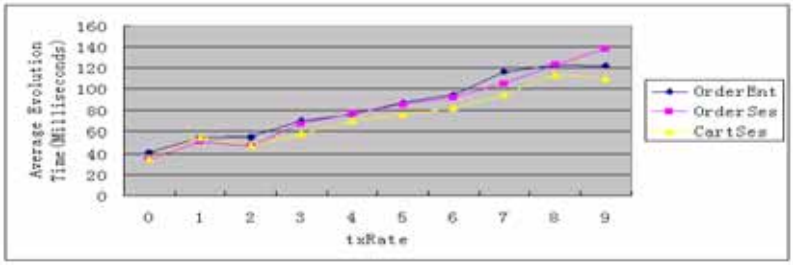

Figure 10. Average Update Time of EJB *OrderEnt*, *OrderSes* and *CartSes* for Different Client Requests

processes are independent and do not influence each other. As there may be network latency, we do not use the client to measure the evolution time; instead, we use the container to calculate the time for each evolution. For each degree of request pressures (by setting the value of txRate), we evolve each EJB 100 times, and then calculate the average evolution time.

The result of our measurement is presented in Figure 10. From the result, we can see that the higher the value of txRate is, the more average time will be used to evolve an EJB. There are many reasons for this. Firstly, the J2EE application server needs to allocate a large amount of resources, especially CPU cycle times and memory sizes, to serve so many client requests that it only spends a fraction of resources for evolution. Secondly, more client requests mean that there are more active components whose state needs to be transferred from the old version to the new version.

Another observation from the result is that the evolution time for each EJB is not so high, from about 30 milliseconds to about 150 milliseconds. Although it is nearly equivalent to the response time of each ECperf transaction such as *AddOrder*, *ModifyOrder*, we can still claim that the average evolution time is tolerable because evolutions are just rare cases in a real enterprise application. Moreover, the result also gives us a hint that it could be recommended to evolve a component when there are few client requests (for example, at mid-night for most business applications) because the evolution time increases when the number of client requests grows.

6.2.3. Results summary

Our experiments show that, at runtime when no update happens, the overhead (about 2%) of our approach due to checking the update lock is tolerable. When an update begins, the overhead becomes significant from 30 milliseconds to 150 milliseconds as the value of txRate grows from 1 to 9. But as updating a component is infrequent, the infrequent pause is at worst inconvenient to a fraction of users, but not harmful to the system in the long term.

To reduce runtime overhead, PKUAS supports both kinds of containers simultaneously and lets the component developer or the application deployer determine whether the component can be updated. They just need to declare clearly in the component's deployment descriptor file using the XML tag "$< evolution > true < /evolution >$" to notify PKUAS to provide the appropriate kind of container.

To reduce update-time overhead, the validation (conformance checking) step can be ignored because Java has provided runtime type-safety mechanisms; if the new version does not conform to its former version, Java will report exceptions such as *NoSuchMethodException* and *IllegalArgumentException* which can be cached and handled by PKUAS's default exception handling mechanism at runtime.

7. RELATED WORK

In this section, we discuss related approaches. These approaches can be organized from either a temporal or a spatial perspective. In this section, we will list some of them from the temporal perspective, and then discuss more of them from the spatial perspective.

As temporal perspective, we mean the time when code for online evolution was woven into the target software. This may occur at different phases of software life cycle, such as the coding phase, the compiling phase and the deployment phase. Different weaving phases require different supports at different transparency levels.

- At the coding phase, it is helpful and easy to modify the program source code to enable its online evolution through some specific language features (class renaming, code rewriting, etc., see [27] for examples) or some design patterns (proxy pattern, delegation pattern, etc. [28]). Support for online evolution at this phase is flexible and does not depend on specific system software, so it is transparent to system software. But it introduces too much redundant code and will increase the maintainer's burden.
- At the compiling phase, Orso *et al.* introduced their implementation technology of "application transformation" in [27]. Codes for online evolution were woven into the application, after the business rules were successfully completed. If we consider code for online evolution as one aspect, we find that this approach is similar to the method taken in AOP (aspect oriented programming) [29]. Compiling phase weaving is transparent to the application.
- At the deploying phase, weaving code for online evolution into the application was undertaken by the system software. Supporting online evolution at this phase is easy and is transparent to the application. It can be undertaken by system software such as a special version of the JVM. If it is undertaken by middleware such as an application server, it will also be transparent to the the JVM and the operating system.

Phase selection for weaving code for online evolution is a trade-off between flexibility and efficiency. As mentioned above, our approach weaves code for online evolution at the deploying phase.

A spatial perspective of previous research focuses on the granularity of the software entity to be modified when evolving the application. These research results can be divided into three main categories: evolution of procedure-based systems, evolution of object-oriented systems, and evolution of component-based systems. Roland T. Mittermeir argued that researchers must consider the size (or complexity) of the software that evolves [13]. For a procedural-oriented program, the possible evolution granularity includes the whole application, the procedure, the abstract data type, etc. For an object-oriented program, the possible evolution granularity includes the application, the object, the method (function), and so on.

7.1. Online evolution of procedure-based systems

Mark E. Segal *et al.* [5] presented a prototype system named PODUS (Procedure-Oriented Dynamic Updating System), developed at the University of Michigan and enhanced at Bellcore. In PODUS, a program is updated by loading the new version of the program, replacing each old procedure with its corresponding new procedure, and changing the binding from its current binding in its current version to that of the new version during execution. Another main contribution of [5] is that it compares a number of studies previous to 1993 on the dynamic modification of dynamic updating systems (mainly procedure-based or module-based systems) based on features such as hardware requirements, system-support requirements, language requirements, granularity, changes supported, update time, and so forth. This comparison provided us with an overview of many famous research/industry projects in this area previous to 1993, such as Argus powered by MIT, and Conic by Imperial College.

In the late 1990s, typical research projects during this period included [7], [30], [31], and [32]. Michael Hicks *et al.* [7, 30] presented a new approach for C-like languages that provided type-safe dynamic updating of native code based on dynamic patches that contained both the updated code and the code needed to transition from the old version to the new. Patches are mostly generated automatically and are applied using dynamic-linking technology. Duggan's dynamic updating approach [31] was based on a reflective mechanism for dynamically adding type sharing constraints to the type system, realized by programmer-defined version adapters in the run-time. Lyu *et al.* [32] provided a solution for making the new version procedure work correctly in the address space of the old version process.

7.2. Online evolution of object-oriented systems

The emergence of object-oriented paradigms has driven many researchers into the field of online evolution of object-oriented systems. Robert Laddaga and James Veitch claimed that dynamic object technology, which combines object-oriented programming technology with dynamic linking and updating capabilities, can be applied to runtime updating problems [33]. The characteristics of dynamic object technology include: 1) it is ideal for runtime configuration without needing to access source code; 2) it uses techniques of reflection, meta-data, and meta-object protocols; and 3) it permits ongoing evolution, even in systems that are completely operational. A series of research papers published in [34] also covered the application of dynamic object technology.

Feng *et al.* [17] discussed the basic problems for object hot-swapping, as previously mentioned in section 3.1. Generally speaking, the techniques for online evolution of object-oriented systems are mostly programming-language specific. Here we introduce some typical work specified for the Java language. M. Dmitriev [35] split the runtime evolution for Java into four stages including: 1) changing method bodies only, 2) binary compatible changes, 3) arbitrary changes, except the changes to instance format and 4) arbitrary changes. Malabarba *et al.* [36] presented an approach that introduced dynamic classes to Java and modified the standard JVM by adding a dynamic class loader. Type correctness was checked when an upgrade was compiled. This approach required the user to install a modified JVM that broke the Java law "write once, run everywhere." Alessandro Orso *et al.* [27] presented another proxy-classes based approach and a supporting tool named DUSC (Dynamic Updating through Swapping of Classes) that required no support from the runtime system, especially the JVM. The approach allowed for updating a running Java program by substituting, adding, and deleting classes.

In 2003, the OMG adopted a specification, "online upgrade" of CORBA objects [24], to help server objects provide continuous service to their users without interruption or suspension of service. This standard borrowed many important technologies, such as commit, rollback, and group servers, from fault tolerate technology.

7.3. Online evolution of component-based systems

M. M. Lehman and J. F. Ramil [37] examined the relevance of the eight software evolution laws in the context of component-based software in general and COTS-intensive systems in particular. Shusaku Iida *et al.* [38] presented an algebraic specification-based formal approach to handling software evolution in component-based software systems based on the two aspects (functional and non-functional aspect) of software evolution. Both of them can be viewed as general discussions on component-based software evolution.

To evolve a single component online, there are different approaches suitable for software having different architectural styles. Nico Janssens *et al.* [39, 40, 41] studied hot swapping of components for producer/consumer based systems. They presented a mechanism to obtain a safe state for unanticipated reconfiguration with minimal interference to the rest of the system, and with minimal contribution from the programmer. M. Zenger [42] proposed a type safe prototype-based model for component evolution as an extension to Featherwight Java with a set of primitives to build, extend, and compose software components dynamically.

There are also many systematic approaches for component-based software evolution, including software architecture based approaches, and agent-based approaches. Plasil, F. *et al.* [43] presented SOFA (SOFtware Appliances) architecture, the SOFA component model and its extension, DCUP (Dynamic Component UPdating), which provided a small set of well scaling orthogonal abstractions (easily mapped to Java and CORBA) to support dynamic component updating in running applications. Postma *et al.* [44] described an approach called 3RDBA to facilitate replacing a key component in a long-living architecture, with an architecture systematically gathering all information needed to make well-founded decisions regarding evolution. The approach consisted of exploration, consolidation, and migration cycles, and each cycle contained four steps: Requirements, Design, Build, and Analyze. Martin L. Griss and Robert R. Kessler [45] treated software agents as next generation flexible components that support dynamic evolution of features and autonomic self-managing.

Besides the general approaches above, researchers have also studied component-based online evolution in different application areas, such as e-Business, networked applications, embedded systems, distributed systems [4, 46, 47] and so forth. Ross Gardler *et al.* [48] presented an approach to semi-automated component-based evolution of eBusiness support systems by linking business strategy with software structure using mappings between business process patterns and software patterns. Jyrki Akkanen *et al.* [49] focused on the major evolution steps, their rationale, and their outcomes, taking the evolution of an in-house developed network editor as their example. Yves Vandewoude *et al.* [50] discussed the component evolution in a Java-based embedded system named SEESCOA and presented an approach which used the concept of ports to redirect messages between components, loaded classes including version information, and updated all component instances at once.

In 2002, Sun JCP released a JSR for Continuous Availability [51] to support online upgrades, but it was withdrawn in 2003. On the contrary, Java Management Extensions [22] has been kept very

active since 1998. It is used to implement flexible and dynamic management solutions for components, services, and resources, and involves related runtime management issues.

8. Conclusions and future work

In this paper, we contributed one definition of online software evolution and one approach for component based online software evolution, which focused on two issues: online update of component implementations and online update of component interfaces. The experiment shows our approach is feasible, and the cost of performance in our approach is acceptable. In particular, our approach adheres strictly to the assumption that online evolution should keep the service promise to clients.

For online software evolution, there are many related issues that are beyond the scope of this paper. For example, how can the correctness of the evolved software system be assured? How can the online evolution process be rolled back in case the process fails or the user changes his mind? We are planning to extend our approach from bottom-level implementation issues up to the high-level issues in the following three directions.

1. **How to choose a component to evolve.** Through some architecture evaluation methods (either formal, analytical models and tools, or the judicious use of rules of thumb and past experience), the maintainer can determine more easily the proper component to be evolved. Our work on runtime software architecture (RSA) [52] is the first step to this direction. RSA can be viewed as a snapshot of the running software system plus a set of controllable points, to model the structure and behavior of the runtime system to be evolved.
2. **How to evaluate one evolution.** To ensure the correctness of evolution, testing is necessary. Online Testing is one kind of testing that can be done while the systems are in use. In reference [53], we explained what online testing is, what it concerns, and the difference between online testing and offline testing. In addition, we also introduced some key problems, such as how to reduce interference to the normal services of the system, how to improve the efficiency of online testing, and so on.
3. **How to make the evolution process more intelligent and autonomic.** The solution proposed in this paper is mainly manual, but for large complicated applications that may contain thousands of components, a software system needs the ability to adjust themselves automatically. Our research on self-adaptive software [54] is ongoing.

REFERENCES

1. Orso A, Liang D, Harrold M, Lipton R. Gamma system: continuous evolution of software after deployment. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA2001)*. ACM Press: New York NY, 2002; 163–169.
2. Lehman MM. Laws of software evolution revisited, *Proceedings of the 5th European Workshop on Software Process Technology (Lecture Notes in Computer Science vol. 1149)*. Springer-Verlag: Berlin, 1996; 108–124.
3. Keymeulen D, Iwata M, Kuniyoshi Y, Higuchi T. Online evolution for a self-adapting robotic navigation system using evolvable hardware. *Artificial Life* 1998; **4**(4):359–393.
4. Kramer J, Magee J. The evolving philosophers problem: dynamic change managemeng. *IEEE Transactions on Software Engineering* 1990; **16**(11):1293–1306.
5. Segal ME, Frieder O. On-the-fly program modification: systems for dynamic updating. *IEEE Software* 1993; **10**(2):53–65;

6. Ganek AG, Corbi TA. The dawning of the autonomic computing era. *IBM Systems Journal* 2003; **42**(1):5–18.
7. Hicks M. Dynamic software updating. *Doctoral dissertation*. Computer and Information Science Department, University of Pennsylvania: Philadelphia PA, 2001; 221pp.
8. Sun Microsystems. Enterprise JavaBeans(TM) specification 2.0. *Sun Developer Network Enterprise JavaBeans Technology Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2005; 572 pp. Http://Java.sun.com/products/ejb [10 September 2004]
9. Rogerson D. *Inside COM*. Microsoft Press: Redmond WA, 1997; 376pp.
10. Object Management Group. CORBA component model (CCM) 3.0. *Object Management Group Working Group Specification*. Object Management Group, Inc.: Needham MA 2002; 434 pp. Http://www.omg.org/cgi-bin/apps/doc?formal/02-06-65.pdf [10 September 2004]
11. Wang QX, Chen F, Mei H, Yang FQ. An application server to support online evolution. *Proceedings International Conference on Software Maintenance (ICSM2002)*. IEEE Computer Society: Los Alamitos CA, 2002; 131–140.
12. Chapin N, Hale JE, Khan KM, Ramil JF, Tan W-G. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2001; **13**(1):3–30.
13. Mittermeir RT. Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools. *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Springer-Verlag: Berlin, 2001; 114–121.
14. Kishan A, Lam M. Dynamic kernel modification and extensibility. *Technical Report of the SUIF group*. Department of Computer Science, Stanford University: Stanford CA, 2002; 26pp.
15. Soules CAN, Appavoo J, Hui K, Wisniewski RW, Silva DD, Ganger GR, Krieger O, Stumm M, Auslander M, Ostrowski M, Rosenburg B, Xenidis J. System support for online reconfiguration. *Proceedings of the General Track: 2003 USENIX Annual Technical Conference (USENIX03)*. USENIX Press: Berkeley CA, 2003; 141–154.
16. Crnkovic I, Hnich B, Jonsson T, Kiziltan Z. Specification, implementation, and deployment of components. *Communications of the ACM* 2002; **45**(10): 35–40.
17. Feng N, Ao G, White T, Pagurek B. Dynamic evolution of network management software by software hot-swapping. *Proceedings IEEE/IFIP International Symposium on Integrated Network Management (IM2001)*. IEEE Computer Society: Los Alamitos CA, 2001; 63–76.
18. Liang S, Bracha G. Dynamic class loading in the Java virtual machine. *ACM SIGPLAN Notices* 1998; **33**(10):36–44.
19. Lindholm T, Yellin F. *The Java(TM) virtual machine specification (2nd edition)*. Addison-Wesley Publishing Co.: Reading MA, 1999; 473pp.
20. Sun Microsystems. Java document for classLoader. *Sun Developer Network JDK Document*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 1 pp. Http://java.sun.com/j2se/1.3/docs/api/java/lang/ClassLoader.html [10 November 2004]
21. Sun Microsystems. Java document for reflection. *Sun Developer Network JDK Document*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 1 pp. http://Java.sun.com/j2se/1.3/docs/guide/reflection/[10 November 2004]
22. Sun Microsystems. Java management extensions specification. *Java Community Process Specification JSR-000003*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 166 pp. http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html [10 November 2004]
23. Sun Microsystems. Java pet store demo 1.3.1. *Sun Developer Network Java Pet Store Demo 1.3.1 Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 1 pp. http://Java.sun.com/blueprints/code/jps131/docs/index.html [10 November 2004]
24. Object Management Group. Online upgrades specification. *Object Management Group Working Group Specification*. Object Management Group, Inc.: Needham MA 2003; 94 pp. Http://www.omg.org/docs/ptc/03-08-07.pdf [10 November 2004]
25. Sun Microsystems. Java 2 platform, enterprise edition specification. *Sun Developer Network J2EE Specification Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 174 pp. http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf [10 November 2004]
26. Sun Microsystems. J2EE ecperf homepage. *Sun Developer Network Ecperf Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2003; 1 pp. Http://Java.sun.com/j2ee/ecperf/index.jsp [10 November 2004]
27. Orso A, Rao A, Harrold M. A technique for dynamic updating of Java software. *Proceedings International Conference on Software Maintenance (ICSM2002)*. IEEE Computer Society: Los Alamitos CA, 2002; 649–658.
28. Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Co.: Rading MA, 1995; 395pp.
29. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM, Irwin J. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (Lecture Notes in Computer Science, vol. 1241)*, Mehmet Akşit and Satoshi Matsuoka (ed). Springer-Verlag: Berlin, 1997; 220–242.
30. Hicks M, Moore JT, Nettles S. Dynamic software updating. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York NY, 2001; 13–23.
31. Duggan D. Type-based hot swapping of running modules. *SIGPLAN Notices* 2001; **36**(10)):50–61.

32. Lyu J, Kim YJ, Kim Y, Lee I. A procedure-based dynamic software update. *Proceedings International Conference on Dependable Systems and Networks (DSN2001)*. IEEE Computer Society: Los Alamitos CA, 2001; 271–284.
33. Laddaga R, Veitch J. Dynamic object technology. *Communications of the ACM* 1997; **40**(5):36–38.
34. Laddaga R, Veitch J(eds). Special section on Dynamic Object Technology. *Communications of the ACM* 1997; **40**(5); 39–69.
35. Dmitriev M. Towards flexible and safe technology for runtime evolution of Java language applications. *Proceedings Workshop on Engineering Complex Object-Oriented Systems for Evolution*, in association with OOPSLA, October 2001. Http://www.experimentalstuff.com/Technologies/HotSwapTool/runtime-evol.pdf [10 November 2001]
36. Malabarba S, Pandey R, Gragg J, Barr E, Barnes JF. Runtime support for type-safe dynamic Java classes. *Proceedings of the 14th European Conference on Object-Oriented Programming (Lecture Notes in Computer Science, vol. 1850)*. Springer-Verlag: Berlin, 2000; 337–361.
37. Lehman MM, Ramil JF. Software evolution in the age of component based software engineering. *IEE Proceeding Software* 2000, **147**(6); 249–255.
38. Iida S, Futatsugi K. Formal approach for handling software evolution in component-based software developments. *Proceedings International Symposium on Principles of Software Evolution*. IEEE Computer Society: Los Alamitos CA, 2000; 262–271.
39. Janssens N, Michiels S, Mahieu T, Verbaeten P. Towards hot-swappable system software: the DiPS/CuPS component framework. *Object-Oriented Technology, ECOOP 2002 Workshop Reader (Lecture Notes in Computer Science, vol. 2548)*. Springer-Verlag: Berlin, 2002; 73-74. Http://research.microsoft.com/ cszypers/Events/WCOP2002/06. Janssens.ps [10 June 2002]
40. Janssens N, Michiels S, Mahieu T, Verbaeten P. Towards transparent hot-swapping support for producer-consumer components. *Proceedings 2nd International Workshop on Unanticipated Software Evolution (USE2003)*. University of Warsaw: Warsaw, Poland, 2003; 9–16. Http://www.cs.kuleuven.ac.be/cwis/research/distrinet/resources/publications/40739.pdf [10 November 2004]
41. Janssens N, Michiels S, Holvoet T, Verbaeten P. A modular approach enforcing safe reconfiguration of producer-consumer applications. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM2004)*. IEEE Computer Society: Los Alamitos CA, 2004; 274–283.
42. Zenger M. Type-safe prototype-based component evolution. *Technical report IC/2002/014*. School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne: Lausanne, Switzerland, 2002; 33pp. Http://citeseer.ist.psu.edu/zenger02typesafe.html [10 November 2002]
43. Plasil F, Balek D, Janecek R. SOFA/DCUP: Architecture for component trading and dynamic updating, *Proceedings 4th International Conference on Configurable Distributed Systems (ICCDS1998)*. IEEE Computer Society: Los Alamitos CA, 1998; 35–42.
44. Postma A, America P, Wijnstra JE. Component replacement in a long-living architecture: the 3RDBA approach. *Proceedings 4th Working IEEE/IFIP Conference on Software Architecture (WICSA2004)*. IEEE Computer Society: Los Alamitos CA, 2004; 89–100.
45. Griss ML, Kessler RR. Achieving the promise of reuse with agent components. *Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications (Lecture Notes in Computer Science, vol. 2603)*. Springer-Verlag: Berlin, 2003; 139–147.
46. Hauptmann S, Wasel J. On-line maintenance with on-the-fly software replacement. *Proceedings 3rd International Conference on Configurable Distributed Systems*. IEEE Computer Society: Los Alamitos CA, 1996; 70–80.
47. Sridhar N, Pike SM, Weide BW. Dynamic module replacement in distributed protocols, *Proceedings 23rd International Conference on Distributed Computing Systems (ICDCS2003)*. IEEE Computer Society: Los Alamitos CA, 2003; 620–627. Http://citeseer.ist.psu.edu/sridhar03dynamic.html [10 November 2004]
48. Gardler R, Mehandjiev N. Supporting component-based software evolution. *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (Lecture Notes in Computer Science vol. 2591)*. Springer-Verlag: Berlin, 2002; 103–120.
49. Akkanen J, Kiss A, Nurminen JK. Evolution of a software component-experiences with a network editor component. *Proceedings of 6th European Conference on Software Maintenance and Reengineering (CSMR'02)*. IEEE Computer Society: Los Alamitos CA, 2002; 119–125.
50. Vandewoude Y, Berbers Y. Run-time evolution for embedded component-oriented systems. *Proceedings International Conference on Software Maintenance (ICSM2002)*. IEEE Computer Society: Los Alamitos CA, 2002; 242–245.
51. Sun Microsystems. J2EE APIs for continuous availability. *Java Community Process Specification JSR-000117 Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2001; 1 pp. http://www.jcp.org/en/jsr/detail?id=117 [10 November 2004]
52. Wang QX, Huang G, Shen JR, Mei H, Yang FQ. Runtime software architecture based software online evolution. *Proceedings 27th International Computer Software and Applications Conference (COMPSAC2003)*. IEEE Computer Society: Los Alamitos CA, 2003; 230–235.

53. Wang QX, Quan LN, Ying FC. Online test of web-based services, *Proceedings 28th International Computer Software and Applications Conference (COMPSAC2004), vol 2*. IEEE Computer Society: Los Alamitos CA, 2004; 166–169.
54. Shen JR, Wang QX, Mei H. Self-adaptive software: cybernetic perspective and an application server supported framework. *Proceedings 28th International Computer Software and Applications Conference (COMPSAC2004), vol 2*. IEEE Computer Society: Los Alamitos CA, 2004; 92–95.

AUTHORS' BIOGRAPHIES

Qianxiang Wang received a PhD degree in Computer Science from NorthWestern Polytechnic University, Xi'an, China, in 1997. He is currently an associate professor at Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His current research interests include: software engineering, software evolution, network based software, and service oriented computing. His email address is: wqx@pku.edu.cn.

Junrong Shen received a Bachelor degree in Computer Science from Peking University, Beijing, China, in 2003, and is currently working toward the Master degree under the direction of Professor Hong Mei at Peking University-Bell Labs Software Technologies Joint Lab. His current research interests include: software architecture, component technology, and application server. His email address is: shenjr@sei.pku.edu.cn.

Xiaopeng Wang received a Master degree in Computer Science from School of Electronics Engineering and Computer Science, Peking University, Beijing, China, in 2004. He is currently a software engineer at BEA compony. His current research interests include: software engineering, application server, and software component technology. His email address is: xpwang@bea.com.

Hong Mei received a PhD degree in Computer Science from Shanghai Jiao Tong University, Shanghai, China, in 1992. He is currently a full professor at Institute of Software, School of Electronics Engineering and Computer Science in Peking University. His current research interests include: software engineering, software reuse and software component technology, software production technology, and programming language. His email address is: meih@pku.edu.cn.