MODIFYING OBJECTS IN RUNNING JAVA PROGRAMS

YASUHIRO SUGIYAMA
Department of Computer Science
Nihon University
Koriyama, Fukushima 963-8642
Japan

sugiyama@ce.nihon-u.ac.jp

ABSTRACT

Software systems have become playing more and more important roles in human daily life. Even a temporary stop of such software systems gives inconvenience and/or damage to human activities. The primary goal of my research is to develop a mechanism that allows software engineers to fix, enhance, and maintain such software systems without stopping the execution. My approach towards the goal is based on the version management of objects. DVMS is a runtime version management system that allows objects in a running program to be replaced with new versions that include bug fixes and enhancements. Using DVMS, new versions of an object can be created without loosing the state of older versions, and versions of an object can be switched in a consistent manner with versions of other related objects.

KEYWORDS

Software Maintenance, Software Evolution, Version Management, Java

1. Introduction

Software systems have become playing more and more important roles in human daily life. Even a temporary stop of such software systems gives inconvenience and/or critical damage to human activities. However, software systems often stop due to periodic maintenance, or system updates for bug fixes or functional enhancements. It is quite desirable that software systems can be maintained and fixed without stopping their execution. The primary goal of my research is to develop a mechanism that allows software engineers to fix, enhance, and maintain software systems without stopping the execution. In other words, my research goal is to develop a mechanism that allows software systems to evolve without stopping their execution.

Recent computer programs written in object-oriented programming languages, like C++ and Java, are composed of many objects. I am currently working on a mechanism that allows objects in programs to get new functionality during the execution. Bugs in objects may be fixed while the program is running. My approach for software evolution is based on the version management. In [18], I introduced the indirect dynamic linking mechanism that allows versions of library functions in dynamic linking libraries to be replaced at runtime. However, this approach is not suitable for managing the evolution that involves some data.

I extended the indirect dynamic linking mechanism to objects in which versions of objects can be replaced during the execution. In [19], I briefly described the idea of my runtime version management of objects for software evolution. I allow objects to have multiple versions. Versions of objects to be used in a program can be selected and changed while the program is running. Consequently, objects may be replaced with new versions that have new functionality. Objects with bugs may be replaced with new versions that fix the bugs during the execution.

This paper will give the details of my runtime version management system DVMS (Dynamic Version Management System) for the Java language [6], focusing on the following two issues: (i) how I switch versions of an object in a consistent manner with versions of other related objects; (ii) how I produce new versions reusing old versions.

This paper is organized as follows. In the next section, I will briefly describe my approach for runtime version management of objects. In the third section, I will describe the details of my version management system DVMS for Java objects. The fourth section will give some discussions on related works. I will conclude this paper with some evaluation and issues to be considered in the future.

2. An Overview of the Runtime Version Management of Objects

My approach for runtime software evolution is based on the runtime version management of objects. When programmers need to make changes to an object for

evolution, including functional enhancements and bug fixes, I do not allow them to make structural and behavioral changes to the object directly. Instead, I ask the programmers to create a new version of the object that includes the any changes for the evolution, and I allow them to switch the original object to the new version while the entire program is running. As *Figure 1* shows, an object in a running program may be replaced with a new version that has new functionality. An object with bugs may be replaced with a new version that fixes the bugs during the execution. New objects to be accessed by the new versions may be added to the running program during the execution. I guarantee that any clients, which access the object with multiple versions, will see as if the identity of the object does not change even if the versions to be used may change.

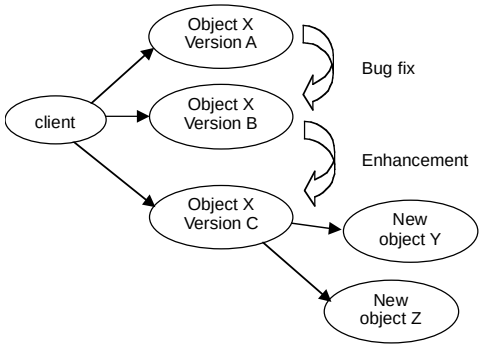

Figure 1: Switching Multiple Versions of an Object

In many cases, bug fixes or enhancements of a software system involve changes of only a small part of the system. Some objects in the software system may be modified, while others are unchanged. Even in the modified objects, some methods may be modified to fix bugs, while other methods may be reused without any changes. New methods and instance variables may be added.

I allow programmers to reuse parts of the original object in various ways when they create a new version. Programmers may simply reuse methods and instance variables of the original object in the new version. Programmers may override the definition of methods in the original object to fix bugs in the methods in the new version. Methods in the new version may invoke methods in the original object, and may refer to the instance variables in the original object.

However, only the reuse of definitions of methods and instance variables is not sufficient. In many cases, it is quite inadequate if programmers simply produce new versions of objects independently from existing ones. I also allow programmers to reuse the internal state of objects. The value of an instance variable in a new version may be copied from the corresponding instance variable in the original object. The value of another instance variable in the new version may be computed from the value of related instance variables in the original object.

Here, I would like to address two remarks on my approach. First, I do not intend to develop a mechanism that allows programmers to debug new versions in running programs. I assume programmers develop and test a new version of an object in some other safe place. I offer programmers a mechanism to replace the buggy object in a running program with the new versions, which have been tested out, without stopping the execution of the running program.

Second, when I talk about changes to an object, I do not intend to mention simple state changes, like changes of the value of instance variables that can be performed by invoking methods permitted on the object without stopping the running program. My major concern is in behavioral and structural changes to objects that involve changes to the source code of the objects and the reproduction of the objects, like changes to the definition of methods, and addition of new methods.

3. The Dynamic Version Management System

This section describes the details of the runtime version management system DVMS (Dynamic Version Management System) for the Java language. The DVMS itself is written in Java. DVMS is being developed with JDK 1.3 on the Solaris operating system.

3.1. Defining Versions

As objects are created from their classes in Java, in order to create a new version of a given object, the class of the new version needs to be defined first. Then, the new version of the object is created as an instance of the new class. The class of the new version is also called the *version* of the original object's class. A version of an object is an instance of a version of the original object's class.

In DVMS, a version of a class is created as a subclass of the class. Consequently, versions of a class are defined in an incremental fashion. New versions inherit instance variables and methods from the original version. New versions may have new instance variables and methods for structural and behavioral enhancements. Some existing instance variables and methods in the original class may be overridden in the new version for fixing bugs. However, I do not support simple hiding or removing of instance variables and methods in the current implementation.

To make the runtime replacement of versions manageable, I restrict the way of defining versions to the use of subclasses. Although it is desirable that any type of modification is supported for versions, arbitrary changes

will make a mechanism for switching versions quite hard to implement. Furthermore, using subclasses is a quite standard way to extend classes in object-oriented languages including Java. I followed this standard.

Please note that a change of versions of a class is usually transparent to its client classes, if the new versions do not change their external interface. In this case, client classes can be used without any change. However, as versions are defined by subclasses, the external interface of versions may evolve. New methods and instance variables may be added. When the external interface of a class is modified in a new version, its client classes also need to be revised in order to use the new version of the class.

A class, which is a subclass of another class, may have its own subclass in Java. Consequently, a version of a class may have its own version, which is again a version of the original class. Consequently, a *sequence* of versions of a class may be defined in an incremental fashion. In a version sequence, when a class is a version of another class, the latter class is called a *predecessor* version of the former class, while the former class is called a *successor* version of the latter class.

A class may need to have two or more immediate successor versions as the class may be enhanced in two or more different ways for different types of enhancements. My definition of versions naturally supports this type of enhancements. As a class may have two or more subclasses, the class may have two or more versions simultaneously. Consequently, programmers may create *branches* of a version sequence. A new branch is created when programmers create an immediate successor version of a class that already has one or more immediate successor versions.

DVMS uses simple naming convention to identify multiple versions of classes. The class name of a version of a given class is obtained by appending the *version name* to the name of the original class separated by an underscore. Given a class Apple, for instance as *Figure 3* shows, Apple_0, Apple_1, Apple_2 are version 0, 1, and 2 of the class Apple respectively, where 0, 1, 2 are version names. Any alphabet and number can be used in version names. For instance, Apple_unix, Apple_mac, Apple_windows are three versions of the Apple class. The original class Apple is aliased to the version 0 of the class by default.

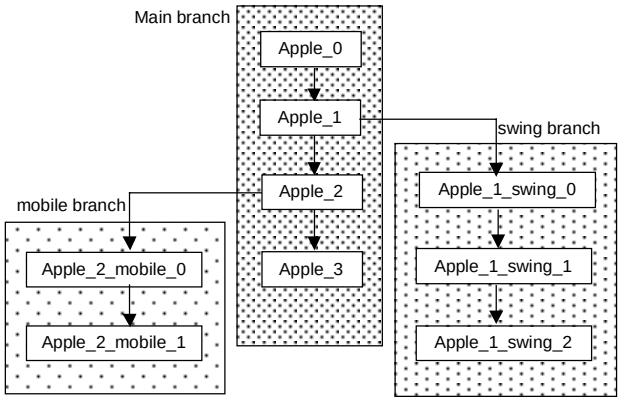

Figure 3: Identifying Versions and Branches

The class name of the versions in a branch is obtained by appending the *branch name* and the version name to the class name of the version from which the branch is originated. For instance, in *Figure 3*, when a new version is created for Apple_1, which already has a successor version Apple_2, a new branch is created. Assuming that the name of the branch is swing, versions in the swing branch are named like Apple_1_swing_2, which is a version 2 in the swing branch.

3.2. An Overview of DVMS

The regular Java Virtual Machine (JVM) is not capable of understanding multiple versions of objects. DVMS is an execution environment for Java programs that include objects with multiple versions.

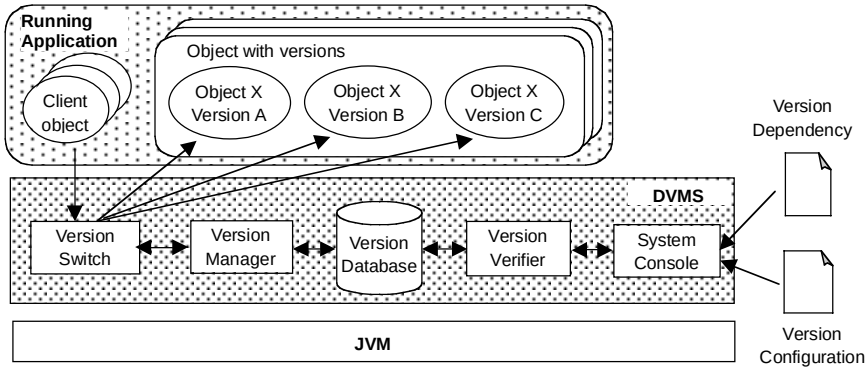

Figure 2: The Architecture of DVMS

Figure 2 shows an overview of DVMS. DVMS has five major components: version switch, version manager, version database, version verifier, and system console.

Version switch: the version switch accepts a request from a client object to an object with versions, and forwards the request to an appropriate version, consulting with the version manager.

Version manager: the version manager keeps track of the versions being used for each object, as well as classes, in the running application. It also takes care of creating a new version of an object when it is used for the first time. It also keeps track of availability of versions of individual classes, as well as dependencies among versions of two or more related classes.

Version database: Although the version manager keeps track of the runtime information, like the name of versions of objects being used, in its memory, DVMS has a small persistent database to store the static information, like availability of versions and dependencies among classes.

Version verifier: the version verifier checks to see if the selected versions are consistent with respect to the dependencies among related classes stored in the version database.

System console: the system console accepts various requests from the system administrator of the running application. The system console offers a command line user interface as well as a graphical user interface.

3.3. Executing and Switching Versions

In order to start executing a program, DVMS needs to know the versions of classes in the program to be used. A *configuration* is a collection of pairs of a class name and the version name of the class in the program. The system administrator of the program notifies DVMS the configuration to be used by the program through the system console. Usually, the console accepts configurations in files. Configurations may be defined interactively using commands in the system console.

```
// Configuration
Dialog=2
Button=5
Slidebar=stable
default=10
```

Figure 4: A Configuration File

Figure 4 shows a typical example of a configuration file. Each line in a configuration file is a pair of a class name and the version name of the class in the program, separated by an equal sign. This example states that the version *2* of the class *Dialog*, the version *5* of the class *Button*, and the *stable* version of the class *Slidebar* will be used. It is not necessary to specify versions of all the classes in the program. Instead, a *default version* may be specified. In *Figure 4*, *default* is not a class name. It designates a default version. If a specific version of a class is not specified in a configuration, but a default version is specified, the default version will be used. In the case above, the version of classes other than the three classes listed in the configuration will be 10.

In order to switch a version of an object to another version, the system administrator needs to notify DVMS a new configuration that includes the change. The system administrator needs to notify DVMS only the new versions of classes to be changed. Versions of other classes will be unchanged.

After a new configuration is given, the system administrator directs DVMS to use the new configuration. Then, the versions of the instances of the classes in the running application will be switched to the versions specified in the new configuration. The version manager will carry out the creation of new versions.

3.4. Keeping Versions Consistent

DVMS allows two or more classes have multiple versions simultaneously. When two or more classes have multiple versions, it will frequently happen that a version of a class can be used only with a specific version of some other classes, but not with other versions. Consistency among versions of classes needs to be maintained.

```
// Dependency
Dialog=3: Button>=6 Slidebar!=2
```

Figure 5: A Dependency File

I provide a mechanism to define *dependencies* among versions of classes. Dependencies are separately specified from configurations. I define dependencies among classes in the notation in *Figure 5*. This dependency tells DVMS that the version 3 of the *Dialog* class can be used only when: (i) the version of the *Button* class is 6 or its successor versions, and (ii) the version of the *Slidebar* class is not 2. Like configurations, dependencies are usually specified in a file. They may be defined interactively using commands in the system console.

Whenever a new version of a class becomes available, the availability of the version and its dependency must be given to DVMS through the system console. The version verifier will check to see if the supplied dependency does not include any inconsistency within itself, like cyclic dependencies. It also checks to see if the supplied dependency is consistent with the dependencies stored in

the version database. If everything is fine, the dependency will be added to the version database.

When DVMS is directed to use a new configuration, the version verifier checks to see if the new configuration is consistent with respect to the dependencies stored in the version database. If the new configuration is not consistent with respect the dependencies, the versions of objects will not be changed, and an error will be reported to the system console.

For instance, if a running program is currently in the configuration of *Figure 4*, and the system administrator of the program wishes to switch the version of the class *Dialog* to 3, the new configuration will be: *Dialog*=3, *Button*=5, *Slidebar*=stable, *default*=10. This configuration is not consistent with the dependency in *Figure 5*, because the version of *Button* is neither 6 nor its successor version. The version of *Dialog* will not be switched to 3. If the system administrator tries to change the version of *Button* to 6 at the same time, the dependency will be met. Then the version of *Dialog* and *Button* will be changed simultaneously.

3.5. Safety

Please note, for safety reasons, the version switch does not change versions of *active* objects. I call objects are active when they are executing their methods, or are being accessed their instance variables. A change of the definition of a method, which is being executed, could result in a disaster. Due to this restriction, the change of versions of objects in DVMS may show some delay for active objects, which might result in some inconsistency. The active objects remain in the old versions, while non-active objects are switched to the new versions. For some classes, such inconsistency is quite acceptable, while such inconsistency may result in a trouble for other classes. Although, with DVMS, versions of individual objects cannot be specified, dependencies can be used to control the behavior of the version switch on individual objects. DVMS accepts dependencies among objects as well as classes.

For instance, remember the last example in the previous section 3.4. If the system administrator tries to change the versions of *Dialog* to 3 and the version of *Button* to 6 at the same time, the dependency in *Figure 5* will be met. Then the version of *Dialog* and *Button* will be changed simultaneously. However, if active instances of *Button* exist, their version does not change until they finish their current activities. Consequently, the version 3 of instances of *Dialog* will co-exist with the version 5 of instances of *Button*.

To prevent this, the system administrator may use the dependency in *Figure 6*. The class name enclosed in a bracket denotes its active instances. This part of the dependency is meaningful when the active instances of *Button* exist. If no active instances of *Button* exist, this part of the dependency will be ignored. The version of the class *Dialog* can be switched only when: (i) the version of active instances of the *Button* class is 6 or its successor versions if they exist, (ii) the version of non-active instances of the *Button* class is 6 or its successor versions, and (iii) the version of the *Slidebar* class is not 2. The first requirement will not be met if active instances of *Button* remain in the version 5. Consequently, the version of *Dialog* as well as *Button* will not be changed, even if the system administrator tries to change the version of *Dialog* and *Button* simultaneously.

```
// Dependency
Dialog=3: [Button]>=6 Button>=6 Slidebar!=2
```

Figure 6: A Dependency File Involving Instances

Please note if the system administrator changes the version of *Dialog* and *Button* in a sequential fashion, the result will be different. Assume the system administrator tries to change the version of *Button* to 6 first, and then the version of *Dialog* to 3. In this case, the change of the version of non-active instances of *Button* will take effect immediately, but the version of active instances of *Button* will remain unchanged. Then the change of the version of *Dialog* will not carried out, if the version of the active instances of *Button* is still 5. On the other hand, if the system administrator first tries to change the version of *Dialog* to 3, this request will not be carried out at all.

3.6. Reusing Versions

A new version may be created preserving the internal state of its predecessor version. I use *derivation* [20] that allows incremental refinement of objects without loosing the internal state of objects. Derivation avoids migration and some other expensive conversion tasks from older versions to newer versions.

3.6.1. Derivation

In many object oriented programming languages, including Java, C++, Smalltalk, objects are created from their classes by instantiation. In these languages, classes are templates to produce their instances. The definition of classes includes instance variables and methods to be included in their instances.

Derivation is a mechanism to create instances of a class, like instantiation. Derivation and instantiation differ only when programmers create instances of a class that is a subclass of some other classes. In instantiation, an instance of a class includes all the instance variables defined in the class in addition to the instance variables inherited from its superclasses. All the instance variables and methods,

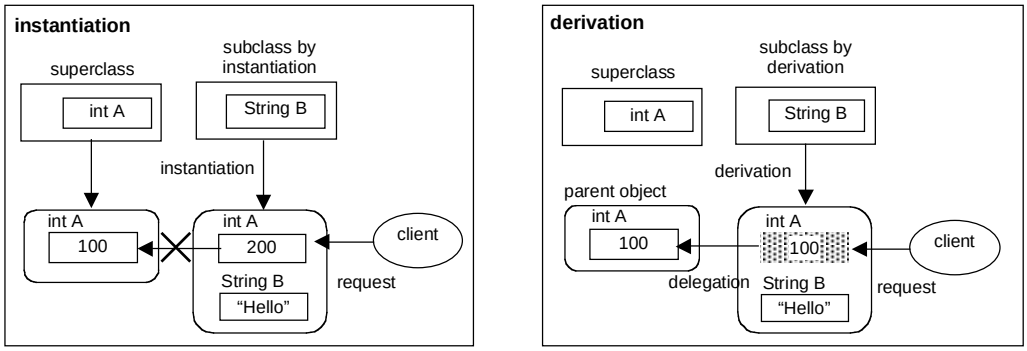

Figure 7: Instantiation and Derivation

including the inherited ones, are available locally in the created instance.

In derivation, as *Figure 7* shows, each instance of a class includes only the instance variables defined in the class, but not those defined in its superclasses. Instead, when programmers create an instance of a class by derivation, they need to select an instance of the superclass as the *parent object* of the instance being created. When the instance receives a request for its instance variables and methods that are not locally available, the instance forwards the request to the parent object that has the requested instance variables and methods. Consequently, users of the instance will see as if all the instance variables and methods of the superclasses are locally available. The mechanism to forward requests is called *delegation* [12].

If programmers create a new version of an object by instantiation, the new version has its own instance variables independently from the original version. Consequently, the values of the instance variables of the original object cannot be reused unless the new version explicitly copies the values from the original version.

I use derivation to reuse the internal state of an object in its successor versions. As a version of an object is an instance of a subclass of the original object's class, the new version may be created by derivation, selecting the original version as its parent object. If programmers need not reuse the state of the original object, they may create versions by instantiation.

3.6.2. Syntax

I extended the syntax of Java so that derivation can be defined in Java programs. I adopted a specific notation suitable for version management, instead of supporting the derivation in a generic form.

In *Figure 8*, class A is a subclass of B, and class Button_6 is a subclass of class Button_5. Instances of the class A will be created by instantiation, while instances of the class Button_6 will be created by derivation from some existing instances of the class Button_5. The keyword extends denotes that instances of the subclass are created by instantiation. The keyword revises indicates that instances of the subclass are created by derivation. To distinguish these types of subclasses, I call them as *subclasses by instantiation* and *subclasses by derivation* respectively. All the syntactical rules, except for the ways of creating instances, are applied to the both subclasses equally.

I developed the derivation compiler derc to translate Java programs with derivation into the regular Java notation. The derivation compiler derc is a preprocessor to the Java compiler javac. The derc takes a Java source file that includes the extended notation for derivation, and translates it into regular Java source code.

As far as the syntax for derivation is concerned, I would like to make the following two remarks. First, the creation of instances by derivation cannot be stated in the regular Java notations. However, when programmers create a new version of a class as a subclass by derivation, they are not required to create its instances explicitly. The version manager will carry out the creation of instances transparently. So I do not need a new notation for such statements for DVMS.

```
//   Subclass by Instantiation
class A extends B {
            ....
}

// Subclass by Derivation
class Button_6 revises Button_5 {
      ....
      // a new method is added
      pubic void newMethod() { .... }
      ....
}
```

Figure 8: Two Types of Subclass Definition

```
//   inform derc to compile with version 6 of Button
import Button_6;
class Dialog_3 revises Dialog_2 {
     // myButton is defined in the original version like
     // protected Button myButton;
     ….
     // must be compiled with version 6 of Button
     myButton.newMethod();
     ….
}
```

Figure 9: Revising a Client Class

Second, as versions are defined by subclasses, the external interface of versions may evolve. New methods and instance variables may be added. However, when the external interface of a class is modified in a new version, the client class also needs to be revised in order to use the new version of the class. Furthermore, the new version of the client class needs to be compiled assuming that it is used with the new version of the class, which the client accesses. Otherwise, the compiler will complain that the new method and/or instance variables are not available. I extended the semantics of the *import* statement of Java for this purpose. A version name of a class can be specified in import statements. *Figure 9* shows a skeleton of a typical client class *Dialog* whose configuration was given in *Figure 4*. The source code will direct derc to compile the version 3 of the *Dialog* class with the version 6 of the *Button* class by the import statement.

3.7. Applying DVMS

The current goal of my research is to verify the effectiveness of DVMS for software evolution. DVMS is implemented in Java, and has several overheads for managing versions at runtime. Consequently, DVMS is not so efficient as the regular JVM. Due to this limitation, the current application area for DVMS is the version management of relatively large grain objects. I do not intend to version control all the kinds of objects in a program. I tried to use DVMS for evolution of several Java applications satisfying this requirement.

A Database Server: The database server consists of two big objects: an object that accepts requests from a web server in a sequential fashion, and an object that accesses a relational database. I successfully used DVMS to fix a bug in a method in the database object.

RMI applications: RMI [8] is a Java framework for distributed objects. The current implementation of DVMS does not include direct support for version management of distributed objects. However, as far as distributed objects based on Java RMI are concerned, I have verified that my version management mechanism reasonably works. I used DVMS to maintain versions of a server object.

4. Related Works

Evolution of software is one of the most active research areas. A series of the international workshops and conferences has been held [1] [2]. In this paper, I addressed two different issues to be resolved for software evolution: switching versions in a consistent fashion, and reusing older versions in newer versions. My previous attempt [18] and Hercules [7] emphasize on the switching mechanism for multiple versions of software components, but these systems are not suitable for the evolution of software components that keep their own state, like objects. I introduced the version management to dynamic linking libraries. Library functions are allowed to have multiple versions that can be switched during the execution. The architecture of my current system has been established in this attempt. Hercules has a built-in mechanism, called the constraint evaluator, which allows versions to be switched in a safe way. However, simple switching among multiple versions results in loss of the internal state of old versions. As I stressed in this paper, a mechanism to switch to a new version preserving and reusing the older versions is quite essential and desirable.

On the other hand, several database systems focused only on the side of reusing capability. Schema evolution [5] and schema versioning [22] are two different attempts to allow schema changes while databases are running. In schema evolution, a schema can be directly modified while the database is running. Whenever a schema is modified, instances of the schema need to be migrated to the instances of the new schema. In schema versioning, direct modification of a schema is not allowed. Instead, modification to a schema results in the creation of a new version of the schema, like DVMS. Schema versioning requires explicit migration of instances from the old version to the new version of the schema. The cost of migration is quite high compared with the cost of derivation.

As far as underlying mechanisms for reuse of versions are concerned, I would like to compare my system with traditional version control systems, Java, DCOM, and prototyping. The traditional version control systems, like SCCS[16], RCS[21], and DSEE [11] use deltas to save the disk space to store multiple versions of a single source file. The use of derivation in the version management of objects has some common characteristics with deltas. Both deltas and derivation store successive versions in an incremental fashion.

Although, Java does not include direct support for evolution of objects, it has some support for versions of classes [10]. Serialization of objects to persistent data storage has a potential problem with changing class definition. If a class is modified after its instances are

serialized, the serialized instances cannot be de-serialized any more, by default, with the modified class. However, some changes like adding a new method does not interfere with the de-serialization of the instances. The tool serialver is provided as part of the JDK to generate an identifier called a Stream Unique Identifier (SUID) of any classes that support serialization. Java allows programmers to mark the modified class with the SUID of the original class to tell the virtual machine that the modified class is compatible with the original class. Consequently, although it is a responsibility of the programmers to validate the compatibility of the changes, an instance of a class can be converted to an instance of the modified class by serializing and de-serializing the instance successively.

Although DCOM does not have direct support for evolution, DCOM offers a primitive mechanism for reusing components [9]. As DCOM provides a binary level interface for constructing distributed systems, reusing mechanisms at the source code level, like inheritance and derivation, cannot be implemented. Instead, DCOM allows a capability of a component to be reused in some other component by delegation.

Prototyping [12] is a mechanism for object creation in some object-oriented programming languages that do not have classes, including Self [4]. Prototyping allows programmers to create an object by specifying differences to an existing object, called the prototype of the new object. New objects can be created reusing existing objects. In prototyping, objects also use delegation to access methods and instance variables in the objects' prototype. Derivation may be considered as a hybrid mechanism of instantiation and prototyping.

5. Summary and Future Works

In this paper, I presented my runtime version management system DVMS for runtime software evolution. I put focus on my way of managing consistency among versions of objects, and my way of creating new versions of objects reusing older versions. Although the current application area of DVMS is limited to the management of relatively large grain objects, I was able to verify the effectiveness of DVMS.

The current implementation is still in its preliminary stage, and includes various limitations. I will summarize some, but not all, of the limitations. I plan to remove these limitations in future releases.

Safety: Although the current implementation of DVMS includes some support for changing versions in a safe way, the support is not sufficient. The version switch performs the changes of versions only when the given dependencies are met. However, the safety control mechanism based on dependencies heavily relies on the system administrator who writes the dependencies. If the system administrator makes some error in some dependency, the running system would crash. I am currently working on a tool that generates dependencies by analyzing the source/byte code of the running program for safe version switching.

Synchronization: It is also critical for safety that the system administrator chooses the right time to direct DVMS to change versions. Especially, Java supports multi-thread programming. A method of an object may be invoked from two or more client objects simultaneously in different threads. In such cases, synchronization of version changes among related objects is critical for safety version management. However, it is impossible to require the system administrator to choose the right time for version changes in the order of milliseconds. I am working on a mechanism that allows the system administrator to schedule and synchronize switching versions of objects in a systematic way.

Efficiency: The current implementation is written in Java that runs on top of the Java Virtual machine using the dynamic class loading mechanism of Java [13]. But in order to reduce the runtime overheads for version management, I am planning to include the mechanism to the Java Virtual Machine.

Topological changes: DVMS addresses behavioral and structural changes of objects. However, there is another type of software evolution, topological changes. Interconnection among objects or components may be changed for evolution. Several systems, including Regis [14], Polylith [17], and ArchStudio [15], emphasize on this type of software evolution. I am planning to work on this type of software evolution.

REFERENCES

[1] Proceedings of the International Workshop on the Principles of Software Evolution 98, April, 1998.

[2] Proceedings of the International Workshop on the Principles of Software Evolution 99, July, 1999.

[3] http://www.omg.org

[4] O. Agesen et al, The Self 4.0 Programmer's Reference Manual, Sun Microsystems, 1995

[5] F. Bancilhon, et al, The Design and Implementation of O2, An Object-Oriented Database System, *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1998.

[6] K. Arnold and J. Gosling, *The Java Programming Language*, Addison Wesley.

[7] J. E. Cook and A.D. Jeffrey, Highly Reliable Upgrading of Components, *ICSE99*, pages 203-212, IEEE, 1999.

[8] T. B. Downing, *Java RMI: Remote Method Invocation*, IDG Books, 1998.

[9] G. Eddon and H Eddon, *Inside Distributed COM*, Microsoft Press, 1998.

[10] R. Englander, *Developing Java Beans*, O'Reilly and Associates, 1997.

[11] Leblang, D. B., R. P. Chase Jr. and G. D. McLean Jr. The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. *Proceedings of the IEEE Conference on Workstations*, pages 266-280, IEEE, November, 1985.

[12] H.Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, *Proceedings of the OOPSLA'86*, pages 214-223, 1986.

[13] S. Liang and G. Bracha, Dynamic Class Loading in the Java Virtual Machine, *Proceedings of the OOPSLA'98*, pages36-44, ACM, 1998,

[14] J. Magee, N. Dulay, and J. Kramer, Regis: A Constructive Development Environment for Parallel and Distributed Programs, *Proceedings of the International Workshop on Configurable Distributed Systems*, 1994.

[15] P. Oreizy, N. Medvidovic, and R. Taylor, Architecture-based Runtime Software Evolution, *ICSE98*, pages 177-186, 1998.

[16] Rochkind, M. J., The Source Code Control System. *IEEE Transactions on Software Engineering*, vol. SE-1, no.4, pages 364-370, December, 1975.

[17] J. M. Purtilo, The Polylith Software Bus, *ACM Transactions of Programming Languages and Systems*, vol. 16, No.1, pages 151-174, 1994.

[18] Y. Sugiyama, Runtime Software Evolution based on Version Management, *IWPSE98*, pages 98-102, April, 1998.

[19] Y. Sugiyama, A Mechanism for Runtime Evolution of Objects, *IWPSE99*, June, 1999.

[20] Y. Sugiyama, Producing and Managing Software Objects in the Process Programming Environment OPM, *APSEC'94*, December, 1994.

[21] Tichy, W. F. RCS - A system for Version Control. *Software - Practice and Experience*, vol.15, no.7, pages 637-654, July, 1985.

[22] Zdonik, S. B. Version Management in an Object-Oriented Database. *International Workshop on Advanced Programming Environments*, pages 405-422, 1986.