Dynamic Deployment of Java Applications

Tobias Ritzau
Linköping University
tobri@ida.liu.se

Jesper Andersson
Växjö University
jesan@ida.liu.se

Abstract

Producing high quality software, on time, and keeping costs within reasonable bounds have been three major goals from the very beginning of software engineering as an engineering science. Hardly ever are deployed systems either error-free or fully functionally satisfactory. The result is that, once brought into operation, systems undergo a series of "patches", "fixes", modifications, and changes.

At the same time, increasingly many functions dependent on software in business, industry, and at home depend on being immediately available. Having these services unavailable due to updating the software is annoying for users. And there are other types of systems – real-time systems – that cannot afford to be taken down, for instance, telecommunication systems, command-and-control systems, and any other systems requiring continuous operation.

This paper presents the Java Distributed Run-time Update Management System (JDRUMS). JDRUMS is meant to deal with the above mentioned problems, specifically for Java. It provides functionality for the introduction of new versions of existing Java classes on the fly, while preserving the internal state of objects.

1 Introduction

Producing high quality software, on time, and keeping costs within reasonable bounds have been three major goals from the very beginning of software engineering as an engineering science. To achieve this, several theories, methods, and tools have been developed, all supporting some part of the total idealized design process. Even though software developers have a wide range of tools and techniques available today, there are still severe problems. These are most directly associated with the increased complexity inherent in modern software systems. As a result, hardly ever are deployed systems either error-free or fully functionally satisfactory. The result is that, once brought into operation, systems undergo a series of "patches", "fixes", modifications, and changes. In a high pressure environment this would result in, not only system unavailability, but quite often a severe diminishment in logical clarity of the total system. This degradation continues throughout the often quite long operational cycle of a system as it is modified. Such modifications can be either incremental changes in functions or the adding of functions to adapt it to new operational requirements. In order to reduce operational costs, if not to considerably extend the life of the system, it is most important to design systems so that maintenance enhancements preserve the logical clarity of the system. This has to be kept in mind when recommending adding an additional consideration, dynamic reconfiguration, to the architectural design repertoire. At the same time, increasingly many functions dependent on software in business, industry, and at home depend on being immediately available. Having these services unavailable due to updating the software is annoying for users. And there are other types of systems – real-time systems – that cannot afford to be taken down, for instance, telecommunication systems, command-and-control systems, and any other systems requiring continuous operation.

The idea of dynamically modifying certain lower level elements of an application is not new. In 1976, Fabry [4] presented a technique for replacing types dynamically. Over the years several strategies have been developed and implemented, spanning the range from hardware solutions with redundant processors to software based systems supporting distributed applications. A more detailed overview of previous work is to can be found in Segal et al. [15].

This paper presents the Java Distributed Run-time Update Management System (JDRUMS), developed jointly by Linköping University and Växjö University. JDRUMS is meant to deal with the above mentioned problems, specifically for Java. It provides functionality for the introduction of new versions of existing Java classes on the fly, while preserving the internal state of objects. Conceptually the updating is supposed to be coherent to the user of the Java application, although in our implementation the added overhead might in some cases be noticeable. JDRUMS is under development and the current version provides basic functionality required at run-time by dynamically evolving Java applications.

Future developments include several functions supporting distributed Java applications, including synchronized, distributed updates and a configuration tool for managing run-time evolution.

The reminder of this paper is organized as follows. Section 2 discusses run-time deployment and how this can be supported at the architectural level of design [14, 16]. In Section 3 we present the JDRUMS and its architecture. Finally, in Section 4 we conclude and point out some interesting findings from our work.

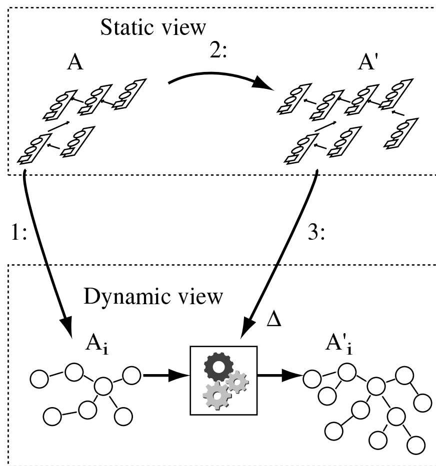

Figure 1: Dynamic deployment

2 Dynamic Deployment

The Software evolution process, or software maintenance in other words, is triggered by request for modifications. These changes are typically classified into four groups corrective, perfective, adaptive, and preventive [7]. Corrective changes are modifications to existing functionality, while perfective changes introduces new functionality to the system. Adaptive maintenance aims at adaptation of a system to changes in the execution environment, while preventive takes actions that will simplify or remove future, additional change requests.

The maintenance process follow the tracks of a traditional development process. Normally an impact analysis, where effect of changes are predicted, is followed by change implementation, where the required changes are applied to the system. Finally, the new version of the system is released and deployed.

A Dynamic Deployment System (DDS), as seen in Figure 1, changes the final step in a traditional software maintenance activity. Normally, the system will be taken down, the new software installed, and finally the system will be taken back up again. This procedure can be really annoying for users of the system, especially if they are heavily dependent on the functionality that the system offers. In larger companies and cooperations, these *down-times* can effect the revenue.

The alternative, dynamic deployment, will introduce modifications to a system without affecting the execution. Below we present some scenarios were dynamic modification of executing applications really add value to the different stake holders, users, procurers, and developers.

2.1 Dynamic Deployment Scenarios

2.1.1 Scenario 1: Web Server

A security hole has been found in a web server. Since availability is essential, an administrator of the server would not like to take the server down to perform the update during the day. Instead the system is taken down during the night when the users are not that active. This has two main disadvantages: the system must still be shut down and the system is vulnerable during the time it is not yet patched.

Using dynamic update, the server can be updated during peek hours. Using JDRUMS the performance will suffer for a while, but the system will always be available. The security patch can also be installed as soon as it is available.

2.1.2 Scenario 2: Cellular Phone

During maintenance a severe bug is found in the software of a cellular phone model. Since this model has been sold to hundreds of thousands of customers, it is hard to update all phones. One solution would be to collect all phones. This would require a big work effort and a huge amount of money. An alternative is to let the users update their phones themselves. The problem with this approach is that all users are not capable of doing that. The best solution is maybe to keep quiet about the bug and hope that it is not exposed.

Using dynamic update this would be a small problem. As soon as the bug is fixed the “patched” version can be distributed. The service provider broadcasts a message to the phones telling them that a new version of their software is available and the phones updates them selves without the user’s knowledge. The phones that can not be reached when the message is broadcasted are noticed as soon as they are switched on.

2.2 Existing Approaches

Dynamic updating systems is by no means a new field of research. Several strategies and implementations have been tested, although none have revolutionized dynamic updating by offering a solution cheap or practical enough for anyone to use. They all have some short-comings. This section will present some of the more important progress in the field, categorized by the strategies which they follow. If not otherwise specified, the description of the different systems are compiled from Beitz [3], Segal and Frieder [15] and Gupta [5].

2.2.1 Hardware-based Redundant Systems

There exist, as mentioned earlier, hardware-based solutions which allow dynamic reconfiguration of applications by replacing the existing system with an updated one running on a secondary machine. Such a system is often operating in conjunction with a hardware-based system for fault-tolerance, but it is still difficult and expensive [15] to provide the functionality for migrating state between machines.

2.2.2 Abstract-data Type-Based Systems

In the early 1970’s the notion of data abstraction gave the idea of the possibility to exchange the implementation of

the type during runtime. This is possible due to the fact that the implementation of an abstract datatype is hidden behind its interface. So, even after the change, the type can be expected to work as if nothing had happened.

Fabry [4] presented ways to implement such a system, which he called the *dynamic type replacement* system. In addition to being transparent, he also makes a demand of the updating mechanism. It should not be necessary to stop the whole system and modify all instances at once; at worst, individual instances should be made temporarily unavailable. It would otherwise make any real-time demands impossible in larger systems. This is achieved by waiting with the introduction until the time that the new version is actually used.

Fabry's solution uses indirect references, i.e. pointers to an intermediate reference, which in turn points to the most recent versions of the code. Since no changes to the interface is allowed, the problem of incompatibility is avoided. If code currently executing is being updated it will continue to work on the old code segment, until next time, when the newer version is used. Fabry's system, however, is limited to operating systems which allow capability-based [5, 4] addressing. Changes to the structure of programs is not possible at all, only the implementation of individual types.

2.2.3 Client-Server Systems

The client-server model works much in the same way as that for abstract data-types, only in a bigger scale. The granularity of components are increased from datatypes to server-modules. The interface to the server never changes, only the implementation of the different services it provides. A general drawback using the systems which follow this approach is the size of the components being replaced. Even a small change will lead to an entire server being unavailable, for the duration of the update.

2.2.4 Module-Based Systems

In this approach programs consists of a number of independent building-blocks, or modules, connected together using a separate configuration manager. To reconfigure the system no modifications are necessary to the individual modules, since the connections between them are handled by the manager rather than hard-coded references. The main drawback is that special knowledge is demanded on both programmer and system manager. This is due to the fact that the modules must be adapted to the configuration system, and that knowledge of how to control the configuration manager is necessary.

2.2.5 Procedure-Based Systems

Procedure-based systems provide functionality for exchanging individual procedures in languages like C and Pascal. This is a more general code restructuring strategy than just changing the implementation of abstract data types. The granularity is often significantly smaller than that of the previous approaches.

2.2.6 Architecture-Level Systems

Dynamic evolution and dynamic deployment, "dynamicism" or "dynamism", have earned a growing interest from the software architecture community. Luckham and Vera [10], define *dynamicism* in the Rapide language as the capability of modeling architectures in which the number of components, connectors, and bindings may vary when the software system is executed. Medvidovic and Taylor [11], describe *dynamism* as an aspect of configurations. Configurations that allow replication, insertion and removal, and reconnection of architectural elements, statically and dynamically, demonstrate the dynamism property.

Extended specifications in Wright [1, 2], add event based reconfiguration to ordinary Wright specifications. This technique provides the means for exhaustive static analysis. Nevertheless, there are several limitations on the expressiveness of dynamic aspects in Wright. Wright does not support the deployment activity, instead the focus is on earlier phases, such as modeling and testing of architectures.

Darwin [8, 9] is another architectural description language, where the implementations supports dynamic aspects of an architecture. The intended application area is that of distributed systems. Darwin provides support for dynamic reconfiguration at run-time in the form of a reconfiguration manager. A supervisor can send directives in a script language to the system that invoke dynamic reconfiguration. In a Darwin implementation, every element can be either added, removed, or replaced dynamically but from external tools only.

An approach similar to that in Darwin is used for architectures implemented in the C2-style. The C2-framework introduces components and connectors as implementation entities. A component or connector corresponds to a class in the implementation language. At run-time system maintainers use a shell-like tool ArchTool [12, 13]. ArchTool implements a language where the manager can express modifications (e.g. add a component) and constraints. Applications in the C2-style are aware of the architecture, and the meta-information is directly accessible from ArchTool.

3 The JDRUMS Environment

The goal of JDRUMS is, as stated earlier, to allow dynamic reconfiguration of executing Java applications. Not only should we, through the JDRUMS interface, be able to affect the application at a mere implementation level, we should also be able to affect it at a structural level and even at an architectural level. For instance, it would be possible to switch from using a merge sort algorithm to a quick sort algorithm, thus making a change at the implementation level. A change at the structural level of an application implies changes in the relationships between software components, or in this case, between Java classes. Finally, at the architectural level, a more radical reconfiguration in

the fundamental design of an application can also be carried out.

The JDRUMS environment, depicted in Figure 2, consists of:

- A modified Java Virtual Machine (JVM), especially adapted to be able to change classes and objects on the fly.
- An interface, which allows for outside intervention of the JVM.
- A toolkit that can be used for creating conversion policies of Java classes, reconfigure a single JVM, synchronize reconfiguration of multiple JVM's running in a distributed environment.
- A communication layer which will work as a middle tier in between the interface and the toolkit.

The communication layer must be used when accessing the JVM from the outside. It allows for interaction with a JVM from a quite different location. JDRUMS has two communication protocols that be used in the communication layer, RPC and JINI, and it can easily be extended for other protocols. However, JINI is the solution to be preferred. JINI produces a clean Java solution and has built security policies for accessing the JVM. Also, JINI provides functionality such that performing distributed reconfiguration becomes simplified.

3.1 Conversion Process

JDRUMS introduces the ability to migrate components through updates while preserving the consistency within the JVM. For this purpose we have laid down a strategy that defines how the mapping will be performed. In JDRUMS the term "mapping" refers to the procedure of converting the internal state of a Java class/object into a state that correspond to another class/object.

The mapping has been realized through a special Java class that is generated with a conversion tool.

3.1.1 Strategy

To be able to perform a conversion it is necessary that we have all the information about the component which we intend to update. This is done by consulting the running JVM. Alternatively we could have inspected a non executing entity, e.g. the actual source code. The drawback doing this is that in some cases you might not have access to this entity. We also need information about the component which we intend to update the JVM with. The information gathered so far can be used to generate a skeleton describing attributes of both components. It is up to the person with knowledge about the components to put some meat to the skeleton, i.e. he has to write the actual procedures describing the mapping of attributes. This constitutes, together with the updated component, the *conversion package*.

When the conversion package has been created it is sent to the JDRUMS-enabled JVM through the communication layer. The JVM has been specifically equipped so that it uses the conversion package to reconfigure the running application. This will be further discussed in section 3.2.5.

3.1.2 Conversion Class

The JVM needs access to some formal rules of how it should perform the mapping of Java classes and objects. One way to do this is to invent some language, understood by the JVM, describing the relation between fields and methods of old and new instances. However, this approach introduces some additional problems. For instance the JVM has to parse a "second" language, which makes it unnecessarily complex. Also, the person who puts together the mapping rules has to learn this mapping language. Another way to do it, and also the strategy used in JDRUMS, is to use pure Java as the conversion language. A simple class can be used to formalize the conversion rules. Now the only language involved in the reconfiguration process will be Java. The JVM will have no trouble understanding this conversion class, meaning that no excessive functionality will need to be added to the JVM. However, there is one concern; the conversion class has to be compilable. As we shall discover later this might complicate things.

The conversion class has to be constructed in such a manner that the modified JVM understands what to do with it. For instance, it has to know what fields refer to the old and the new version of the class/object. We can, unfortunately, not simply refer to the old and new instances in our conversion class since they in fact have the same identity. To solve this we let the conversion class contain two inner classes - `Old` and `New`. One reflecting fields and methods of the old, still running, instance and one reflecting the new instance. For example, by referring to `Old.foo` the field named `foo` in the running instance can be reached. The same philosophy applies for methods. The conversion class must also contain information that the JVM can use to carry out the actual mapping. We simply add two methods, `convertClass()` and `convertObject()`, that uses fields and methods in the above mentioned inner classes to make assignments, like this; `New.bar = Old.foo`. These methods are called respectively when a class instance and an object is converted. This is further described in section 3.2.5.

Back to the little concern raised previously. What would happen if the class we intend to reconfigure has some inner class? Since there is no direct connection between the converter class' internal class `Old` and the actual old instance, there can be no references to its inner classes. This problem is solvable and will be further discussed in section 3.4.

Another related problem is that there might be cases where we would want to reach fields and methods in the superclass of the class intended to be reconfigured. One way to solve this is by letting the `Old` and `New` classes inherit the same classes as their counterparts.

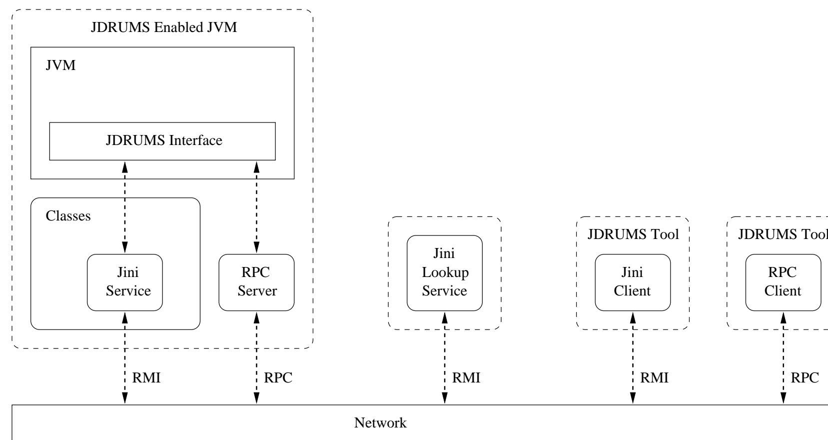

Figure 2: The JDrums environment

3.1.3 Tool

JDRUMS provides a tool for automatically generating a conversion skeleton class, which includes all imports, fields and methods that are used to refer to the old and new version of a specific class. A class might have a more or less complex relationship to other classes. The generating tool should in any case figure this out and reflect it in the conversion class.

The tool is connected to the JDRUMS-enabled JVM through the communication layer and can thereby retrieve information about the old class that is to be updated. Also, it gets information about the enhanced, new class, from a local source.

By automatically generating a conversion class, we minimize the risk of using non-matching fields and methods in our mirror of the old and new class. For instance if the wrong field is used, the compiler will complain.

3.2 Reconfiguration Process

When the needed conversion classes have been created, they are packaged together with their respective class into a conversion package. Finally this is sent to the JVM through the communication layer. This type of configuration can be seen as transparent reconfiguration, where neither the user nor the developer need to be aware of this capability. This implies that any application can be subject to updates.

The JDRUMS does, however not restrict the configuration process to be carried out in this manner. Another way of doing it is to write the application so that it directly uses the JDRUMS interface discussed in section 3.2.1. This way of reconfiguring applications can either be planned or self-governed, depending on how the application is designed.

3.2.1 Interface

To be able to reconfigure running applications, the JVM needs an interface that provides such functionality. The JDRUMS has one internal interface which is used to reflect and affect the internal state. Around this there is an external, pure Java interface which can be used from the running application itself as well as from the communication layer.

3.2.2 Distributed Reconfiguration

Today it is common with distributed applications, and the trend is even more systems of this kind. In Java a distributed application will reside on one or more JVMs which must be JDRUMS-enabled if reconfiguration should be possible. Reconfiguration in these applications becomes a more complex task since there must exist synchronization of the update carried out among the components. Otherwise, the updates must be carried out at exactly the same time, both on the client and on the server, in order to keep the state of the application consistent.

By using JINI in the communication layer it becomes an easy task for the reconfiguration tool (see section 3.2.6) to keep track of all components in a distributed application. Currently, the JDRUMS-system has no mechanism for distributed, synchronized updates. In section 3.4 our plans for improved distributed reconfiguration through JDRUMS is discussed.

3.2.3 Static Procedures

Internally Java classes are represented by a descriptor record containing, among other things, the class name, a reference to its superclass, and lists of its fields and methods. Each such record is referenced to by another smaller record, a handle. An object is also referenced through such a handle, but instead of containing a descriptor record, the object consists of the actual data of its attributes. A handle also contains a reference to a method block, through which the class of the object can be reached.

The JDRUMS-enabled JVM has some additional data members in these records. Each class have a reference to a conversion class, which would be null if none is needed.

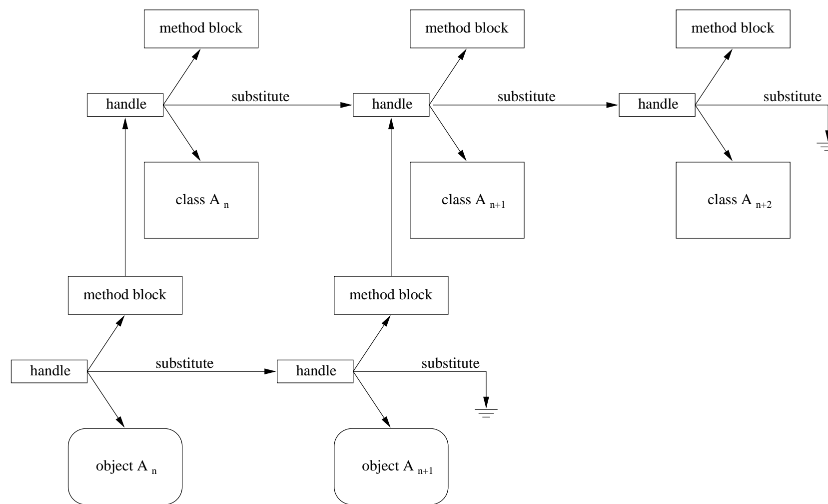

Figure 3: Static structure of objects and classes.

The handles have a *substitute* field, containing a reference to a newer version of its data. This could either be an updated class or a migrated object, depending on what the handle refers to. How these additions are used throughout the reconfiguration is described in the following two subsections.

3.2.4 Dynamic Procedures

A Java program consists of a series of instructions, or *opcodes*, which are interpreted during execution. The interpreter is stack based, i.e. a stack is used to store temporary data, pass references to methods and store the return value of methods and operations. This has to be taken into consideration when reconfiguring the components, as the stack could contain references to old objects or classes.

When an opcode in which an object of a old class is used, the JVM has to discover this and act accordingly. Since we cannot guarantee that the references on the stack are up-to-date at dereference time, the JVM assures that an object of the most recent version is used through the following two steps:

1. If the dereferenced object handle has a substitute reference, follow this repeatedly.

2. If the object's class has a substitute reference, migrate the object into an instance of the newer class. Do this repeatedly.

Migration is done by creating an instance of the new class, and initialize this using the old object and the conversion procedure described in section 3.2.5.

If this strategy is allowed to continue indefinitely, the response time will increase as the chain of new objects and classes grows. To avoid this, the source of the object reference is updated to the most recently migrated version of the object. This is done at load-time, i.e. when a reference is gathered and put on the stack. There might be a newer version of the class, but since the object will be migrated before use, this can be skipped for now. The important thing is that the chain of different object versions will not grow indefinitely, and old versions can eventually be garbage collected.

When the JVM finds a reference to a field or method in another class, it is resolved. This means that the symbolic reference is translated to a direct reference. For optimization purposes the symbolic reference is then replaced by the resolved one. This constitutes a problem when the component containing the referenced field or method have been updated. The JDRUMS-enabled JVM first has to make sure that the direct reference refers to a current component. If not, the JVM will have to re-resolve it.

3.2.5 Conversion Procedures

When the JVM receives a *conversion package*, as described in section 3.1.1, it is unwrapped and the contained classes are introduced to the running environment one by one. The reconfiguration of a single class follows the scheme:

1. Rename the old version; two classes are not allowed to have the same name.

2. Load the new version as if it were a regular class and attach it to the substitute reference of the old class.

3. Load the conversion classes and attach them to the new class (section 3.2.3).

4. Copy the old class static fields to the conversion class' internal class `Old` (section 3.1.2).

5. Also copy the static fields to the `New` class of the conversion class, by name-type matching. This allows implicit mapping.

6. Execute the Java method `convertClass()` in the conversion class. Here explicit mapping is performed (section 3.1.2).

7. Initialize the new class' static fields using their mirrored counterparts in the `New` class of the conversion class.

A similar scheme is used when an object migration is triggered as described in the previous section.

1. Create an instance of the new class.
2. Set the new instance as the substitute for the old object.
3. Copy super-data, i.e. attributes in the common superclass.
4. Copy the old fields to the conversion class' internal class `Old` (section 3.1.2). Static fields are copied from the old class, which might be out of date.
5. Also copy the fields to the `New` class of the conversion class, by name-type matching. Static fields are copied from the new version of the class.
6. Execute the Java method `convertObject()` in the conversion class (section 3.1.2).
7. Initialize the fields using their mirrored counterparts in the `New` class of the conversion class.

Methods are yet to be addressed. They are also to be mapped, much in the same way as fields. This will allow making calls to methods, in either the old class/object or the new versions, during the execution of the mapping. At the time of writing method mapping is yet to be implemented. The possibility of exchanging objects that are currently running is a difficult problem to solve. This lies in the domain of future JDRUMS, which is presented in section 3.4.

3.2.6 The JDrums Reconfiguration Tool

To help with the packaging of classes, and the gathering of their inner classes and conversion classes, the JDRUMS environment includes a special tool. It has a graphical interface in which you can easily combine one or more classes into a package. The order in which the classes are to be introduced to the JVM can be selected, although this functionality will be automated in a future version.

The tool also displays a list of the JDRUMS-enabled JVMs in the neighborhood. If such a JVM is started or shut down, this change will immediately be presented if the tool is running. This functionality relies on the JINI protocol, on which the communication layer is built as described in section 3. Some additional information, such as which applications are currently executing on which hosts, is also presented. One or more JVMs can be selected and used as targets when shipping the *conversion package*.

3.3 The Phone-Book Example

As an example of the conversion procedure we choose a simple example. The application is a simple personal phone directory. In the initial implementation the application uses a Java `Vector` for storing the directory entries. As the directory grows, the linear search algorithms used on this data-structure, gets worse in performance. An alternative implementation, which uses a hash-table is developed for the new release.

The source code for the conversion class can be found in Figure 4. Note the representation of the `Old` and `New` classes.

This example shows how both class attributes and attributes of individual objects can be modified. The `className` and `version` attributes are included for this purpose only. We see two conversion procedures, one for the class members and one for object members. In this example, the second method is more interesting. When the dictionary reference, `phonelist`, is accessed after the update, a hash table is created, and each object from the vector implementation is inserted into the new structure.

We can also see an example of implicit and explicit conversions. In the conversion procedure for class members, the `className` attribute is implicitly copied into the new object, while the `version` attribute (which is to be changed) must be addressed explicitly.

3.4 Future JDrums

Currently JDRUMS supports functionality for gathering enough information to map fields. You could also, in a convenient way, package and send classes to JVMs in a distributed environment. The JDRUMS-enabled JVM is then able to update classes, and migrate their existing instances to the newer versions. There is still much work to be done to offer full functionality. We now present some improvements which will further increase the power of the system.

Mapping of functions, in addition to fields, is desirable. This would allow them to be used during the mapping. Currently, as shown in Figure 4, we only map the `Old` and `New` classes. The problem with function mappings is that we will have to include all methods that are applicable on the two classes respectively. This can, under some circumstances, result in an code-explosion. Mapping of methods does not increase the power of the current mapping, although it could decrease the amount of code duplication needed for the conversion.

The limitations of the current conversion implementation is not being able to reach members in the mapped class' super-data and not being able to map classes containing inner classes. Reaching the super-data could for instance be achieved by having the `Old` and `New` classes inherit the same class as the classes they mirror. Another solution might be to include a "super" member which is an instance of the super classes. You might want to convert a class into a class with a different superclass. In the

```
static class Old {
  static String className;  //Static members
  static String version;
  static Vector phoneList;  //Object members
}
static class New {
  static String className;  //Static members
  static String version;
  static Hashtable phoneList;//Object members
}
public static void convertClass(){
  New.version = "v1.1";
}
public static void convertObject(){
  New.phoneList = new Hashtable();
  for(int i=0;i<Old.phoneList.size();i++){
    Entry contact=(Entry)Old.phoneList.elementAt(i);
    New.phoneList.put(contact.firstName+contact.lastName,contact);
  }
}
```

Figure 4: Source for the conversion class

current implementation the super-data of the new instance would remain uninitialized, which seldom is the desired result.

When updating a class, the process of migrating objects is not only necessary for direct instances of the class, but also for instances of any subclass. In the current implementation, such migrations are never triggered. The solution is rather simple; do not only check the substitute fields of the current class, but also of its superclass and its superclass' superclass and so on. Since this is a rather time-consuming check which is performed for several opcodes we choose not to implement it unless we find a faster control structure to support the check.

The harder problem of updating active objects, i.e. objects which are currently executing, is left to be investigated in the future. For now the JVM will simply have to wait for the running stacks to be empty of the class before updating it.

Currently JDRUMS has the ability to perform distributed reconfiguration, although it lacks the earlier mentioned synchronization functionality. This is satisfactory as long as the interface between the distributed components remain untouched.

There are a number of ways to improve the overall performance of the JDRUMS-enabled JVM. As it is now none of the critical sections, i.e. the implementation of the opcodes, are written in assembler. Also, the support for just-in-time compilation has been disabled. There are also ways of enhancing the actual updating process. For instance by loading and preparing as much as possible before triggering the updates.

4 Conclusions

In this paper we have presented the JDRUMS environment which supports dynamic updating of Java applications. Providing this type of capability in future system development environments will give system maintainers access to cutting-edge technology for transparent updates of applications. The types of systems we foresee in the future are distributed on traditional computers and on small mobile units. Applications will also become an integral part of other standard devices, such as consumer appliances. This new area for software products requires more extensive support for the final activity in a system maintenance phase, the deployment. The possibility to upgrade software on remote devices while these are still running will simplify distributing the software and installing the upgrades. Software vendors will no longer require end-users to retrieve and install updates.

As we pointed out in Section 3.4, our system still needs many improvements. Still our initial experiments and the results from these are promising. Improved performance, support for updates of distributed applications, and improved tool support are high on our work-list. Our approach is novel but still capable of performing quite powerful updates. Future work also includes more complex functionality, such as support for updates of active objects, i.e. replacing methods that can be found on the call-stack. The formal basis for such work has been developed by Gupta and colleagues [6], but how to implement this framework efficiently is still an open question.

Acknowledgements

This paper is based upon work sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), The Excellence Center in Computer Science and Systems Engineering (ECSEL) at Linköping University, and Växjö University.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying Dynamism in Software Architectures. In

G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems Workshop*, 1997. Available at `http://www.cs.iastate.edu/~leavens/FoCBS/`.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, March 1998.

[3] A. Beitz. Dynamic Software Reconfiguration. PhD. Thesis proposal. Available at `http://www.dstc.qut.edu.au/~ashley/phd/ls.html`.

[4] R. Fabry. How to Design A System in Which Modules can be Changed on the Fly. In *Proceedings of International Conference on Software Engineering*, pages 470–476. IEEE-CS Press, 1976.

[5] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[6] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions and Software Engineering*, 22(2):120–131, February 1996.

[7] J Hartmann and D.J. Robson. Techniques for Selective Revalidation. *IEEE Software*, 7(1):31–36, January 1990.

[8] J. Kramer and J. Magee. Dynamic Structure in Software Architectures. In *Proceedings of the fourth ACM SIGSOFT symposium on Foundations of software engineering (FSE'96)*, pages 3–14. ACM, ACM-Press, October 1996.

[9] J. Kramer and J. Magee. Self Organising Software Architectures. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *Joint Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96)*, pages 35–38. ACM, 1996. SIGSOFT'96.

[10] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 12(9):717–734, Sep. 1995.

[11] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In M. Jazayeri and H. Schauer, editors, *Software Engineering – ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer Verlag, September 1997.

[12] P. Oreizy. Issues in Runtime Modification of Software Architectures. Technical Report 35, Department of Information and Computer Science, University of California, Irvine, CA 92697, August 1996.

[13] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*. ACM, ACM-Press, April 1998. To appear.

[14] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(4):40, Oct 1992.

[15] M. E. Segal and O Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[16] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.