Towards a Runtime Code Update in Java
an exploration using STX:LIBJAVA

Marcel Hlopko[1], Jan Kurš[2], and Jan Vraný[1]

[1] Faculty of Information Technology,
Czech Technical University in Prague
{marcel.hlopko, jan.vrany}@fit.cvut.cz
[2] Software Composition Group,
University of Bern
kurs@iam.unibe.ch

Abstract. Runtime Code Update is a technique to update a program while it is running. Such a feature is often used so the developer can modify an application without the necessity to restart the application and recover desired state after restart. This saves time and lowers costs. Furthermore, there are applications which cannot be stopped, such as air traffic control systems or telephone switches. Current virtual machines for Java programming language do not support non-trivial updates to the running code. We have modified STX:LIBJAVA – an implementation of Java virtual machine within Smalltalk/X – to support arbitrary changes to the running code. Beside changes to the fields and methods which are already supported by the tools such as JRebel or Javaleon, we also support unrestricted changes to the class and interface hierarchy. Our Runtime Code Updates scheme has been integrated into the Smalltalk/X IDE, thus providing interactive environment where a developer can modify a Java application while it is running.

1 Introduction

The ability to dynamically update the code of a running application is interesting for many domains. It can reduce downtime of long-running systems by eliminating the need for stopping, redeploying and starting the application again. There are applications that have to be maintained and improved but cannot be stopped. Financial transaction processors, telephone switches, air traffic control systems, are all examples of such applications.

Dynamic code updates can improve programmer productivity during programming by giving instant feedback without the need to wait for rebuild and deployment (Shan [12]). Kabanov and Vene [7] show that many of their clients have applications which take more than 15 minutes to rebuild. It is clear that support for dynamic code updates saves a lot of development time and reduces total cost of the software product. Furthermore, runtime code update can improve debugging efficiency by not forcing the programmer to restart the program and to recreate bug preconditions.

Updating the code of the running program has been researched in the past (Fabry [3]) and is still investigated today (Dmitriev [1], Kabanov and Vene [7], Orso et al.

[10], Redmond and Cahill [11], Subramanian et al. [13], Würthinger et al. [14]). Support for runtime code updates is common in VMs for dynamic languages, but not so common in VMs for statically typed languages such as Java (Ebraert and Vandewoude [2]). For example HotSpot VM[3]– the reference VM for Java– has only limited support for runtime code update. Currently, only changes to the method bodies are allowed.

There are approaches for some types of runtime code updates for Java including: JRebel[4] and Javaleon[5] – an application-level systems; Dynamic Code Evolution VM[6] – a modification of the HotSpot VM allowing runtime code changes; JVolve[7] – a solution based on the Jikes Research VM.

None of existing solutions supports all types of runtime code updates. HotSpot VM has not been developed with runtime code updates in mind and has to be modified to support this feature. Such modification requires a large amount of engineering work.

In this paper we present STX:LIBJAVA – a Java VM implementation for Smalltalk/X VM – which has been modified to support all types of runtime code updates for Java. We show solutions and implementation details which relate to the runtime code updates which may be relevant to all Java virtual machines.

The contributions of this paper are (i) presentation of the system supporting all types of runtime code updates for Java, (ii) identification of problems related to runtime code update support in STX:LIBJAVA and (iii) description of solutions to runtime code update problems in STX:LIBJAVA.

The paper is organized as follows: Section 2 describes the types of possible runtime code updates. Section 3 gives an overview of STX:LIBJAVA, a Java VM implementation used. In Section 4 we present our solutions and important implementation details. Section 5 discusses future work. In Section 6 we present related work and Section 7 concludes the paper.

2 Problem Description

2.1 Types of Runtime Code Updates

There are multiple types of runtime code updates, some of which are already implemented in the HotSpot VM, or provided by 3rd party tools executing at the application level, such as JRebel or Javaleon. More complex changes require modification of the HotSpot VM, as shown by Dynamic Code Evolution VM (DCE VM) project (Würthinger et al. [14]). Other relevant solutions make use of non-standard or research VMs, *e.g.,* JValve (Subramanian et al. [13]). A system providing full runtime code updates should handle all types of updates in Table 1, also containing a comparison of standard HotSpot VM, DCE VM, JRebel, Javaleon and JValve.

[3] http://openjdk.java.net/groups/hotspot/
[4] http://zeroturnaround.com/software/jrebel/
[5] http://javeleon.com/index.php
[6] http://ssw.jku.at/dcevm/
[7] http://www.cs.utexas.edu/ suriya/jvolve/

Feature	HotSpot	DCE VM	JRebel	Javaleon	JValve
Changes to method Bodies	✓	✓	✓	✓	✓
Adding/removing fields	x	✓	✓	✓	✓
Adding/removing methods	x	✓	✓	✓	✓
Adding/removing constructors	x	✓	✓	✓	✓
Adding/removing classes	x	✓	✓	✓	✓
Replacing superclass	x	✓	x	✓	x
Adding/removing implemented interfaces	x	✓	x	✓	x
Custom migration of changed instances	x	x	✓	✓	✓
Custom migration of changed classes	x	x	x	x	✓

Table 1. Comparison of HotSpot, DCE VM, JRebel and Javaleon features

2.2 Update of the Method Body

Out of all code updates, update of the method body is the simplest and most often used one. The signature of the method remains the same, only the code of the method is modified, for example after fixing simple bug.

As an example of this change, consider code shown in Listing 1.1. In `TicketController`, we modify the `buyButtonClicked` method. The changed method is shown in Listing 1.2.

```java
public class TicketController {

  private TicketView view;
  private TicketsSeller seller;
  private TicketValidator validator;

  ...

  public void buyButtonClicked() {
    Ticket ticket = view.getTicket();
    seller.sellTicket(ticket);
  }
}
```

Listing 1.1. Initial code before a method body update

```java
public void buyButtonClicked() {
  Ticket ticket = view.getTicket();
  if (validator.isValid(ticket)) {
    seller.sellTicket(ticket);
  } else {
    throw new RuntimeException();
  }
}
```

Listing 1.2. Code of the method after the update of the method body

2.3 Binary Compatible Update

Binary compatible update does not break the compatibility of the class with any existing code. Following types of updates fall into this category: adding a field[8], adding a method, adding a contructor, adding an implemented interface[9].

Following on our example in Listing 1.1, consider adding an arbitrary method. Adding this method does not break any existing code. No class depending on the `TicketController` has to be modified. But, there is an opportunity to modify other classes in the system to use newly added method (in *e.g.,* button click handler). This way the whole system can be improved and evolved at runtime.

2.4 Binary Incompatible Update

Binary incompatible update breaks compatibility with existing code. The following types of updates fall into this category: removing a field, removing a method, changing a signature of a method, replacing a superclass, removing an implemented interface. Such a situation has to be perceived and handled by the system. The runtime system can rollback the update or apply the update and throw an exception when incompatibility causes a problem.

Imagine we change the signature of the existing method. All dependent classes will keep invoking the class with the old signature, but there is no such method present in the updated class anymore. If the system allows such update, the `NoSuchMethodError` should be raised.

2.5 Updates of the Instance Format

Adding and removing a field poses an unique problem. The layout of the object changes. There may be live instances of the updated class. After the update, the instance format expected by the class is different to the instance format on the heap. And simply adding or removing the fields can bring the instance to the unexpected state not attainable by the normal execution.

2.6 Updates of the Class and the Interface Hierarchy

Updates to the class and interface hierarchy are the most complex. The methods and fields could be added or removed by updating the hierarchy therefore runtime system must be prepared for such change. From the point of view of the type safety the type correctness of the program could be broken by updating the hierarchy.

[8] Adding a field is more complex, as elaborated in Section 2.5.
[9] Adding an interface is also more complex change, as elaborated in Section 2.6

3 STX:LIBJAVA

Java is 2nd most used language nowadays having more than 17% community share[10]. Despite huge popularity, Java still lacks a runtime and development environment offering dynamic code reloading, interactive and incremental compilation. Also these features are foundations of high programming productivity of Smalltalk developers (Shan [12]).

STX:LIBJAVA is an implementation of the Java virtual machine built into the Smalltalk/X environment. In addition to providing the infrastructure to load and execute Java code, it also integrates Java into the Smalltalk development environment including browsers, debugger and other tools.

STX:LIBJAVA aims at providing fully compatible Java VM implementation, which is also capable of full interoperability with Smalltalk and vice versa. More about the architecture and interoperability features can be read at Hlopko et al. [6].

3.1 Architecture of STX:LIBJAVA

In this section we will briefly outline STX:LIBJAVA's internal architecture.

Unlike other projects which integrate Java with other languages, STX:LIBJAVA does not use the original JVM in parallel with the host virtual machine, nor does it translate Java source code or Java bytecode to any other host language. Instead, the Smalltalk/X virtual machine is extended to support multiple bytecode sets and execute Java bytecode directly.

Java runtime classes and methods are implemented as Smalltalk `Behavior` and `Method` objects. In particular, Java methods are represented as instances of subclasses of the Smalltalk *Method* class. However, they refer to the Java bytecode instead of the Smalltalk bytecode. Execution of the Java bytecode is implemented in the virtual machine. In the same way that the Smalltalk bytecode is handled by the VM, the Java bytecode is interpreted and/or dynamically compiled to the machine code (jitted).

The main disadvantage of our approach (as opposed to having a separate original JVM execute Java bytecodes) is that the whole functionality of the Java virtual machine has to be reimplemented. This includes an extensive number of native methods, which indeed involve a lot of engineering work. However, we believe that this solution opens possibilities to a much tighter integration which would not be possible otherwise.

3.2 Reference Resolving

Among other information, Java classfile contains a **Constant Pool**, a pool of constants and references used within the class (Lindholm and Yellin [9]). Constant Pool Reference (CPR) can be of following types: ClassRef, MethodRef, InterfaceMethodRef, FieldRef and StringRef. For example, ClassRef consists only of a single string constant, which contains a Fully Qualified Domain Name (FQDN) of the referenced class (Lindholm and Yellin [9]). MethodRef consists of a class ref, which identifies the class containing the method, and the *name and type* of the method.

[10] According to http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Every class is compiled into separate classfile. In order for this class to be used by the application, the classfile must be loaded and class needs to be **linked**. Superclass and superinterfaces are (if not already) loaded and linked before the class is linked. Final static fields have to be initialized to their constant value, and the static initializer of the class must be called (possibly setting the values of nonfinal static fields). After that the class is installed into the class registry.

```
public static String getNameAndParams() {
  return String.format(
    "%s?useUnicode=true",
    databaseName);
}
```

Listing 1.3. An example of the method requiring further resolving

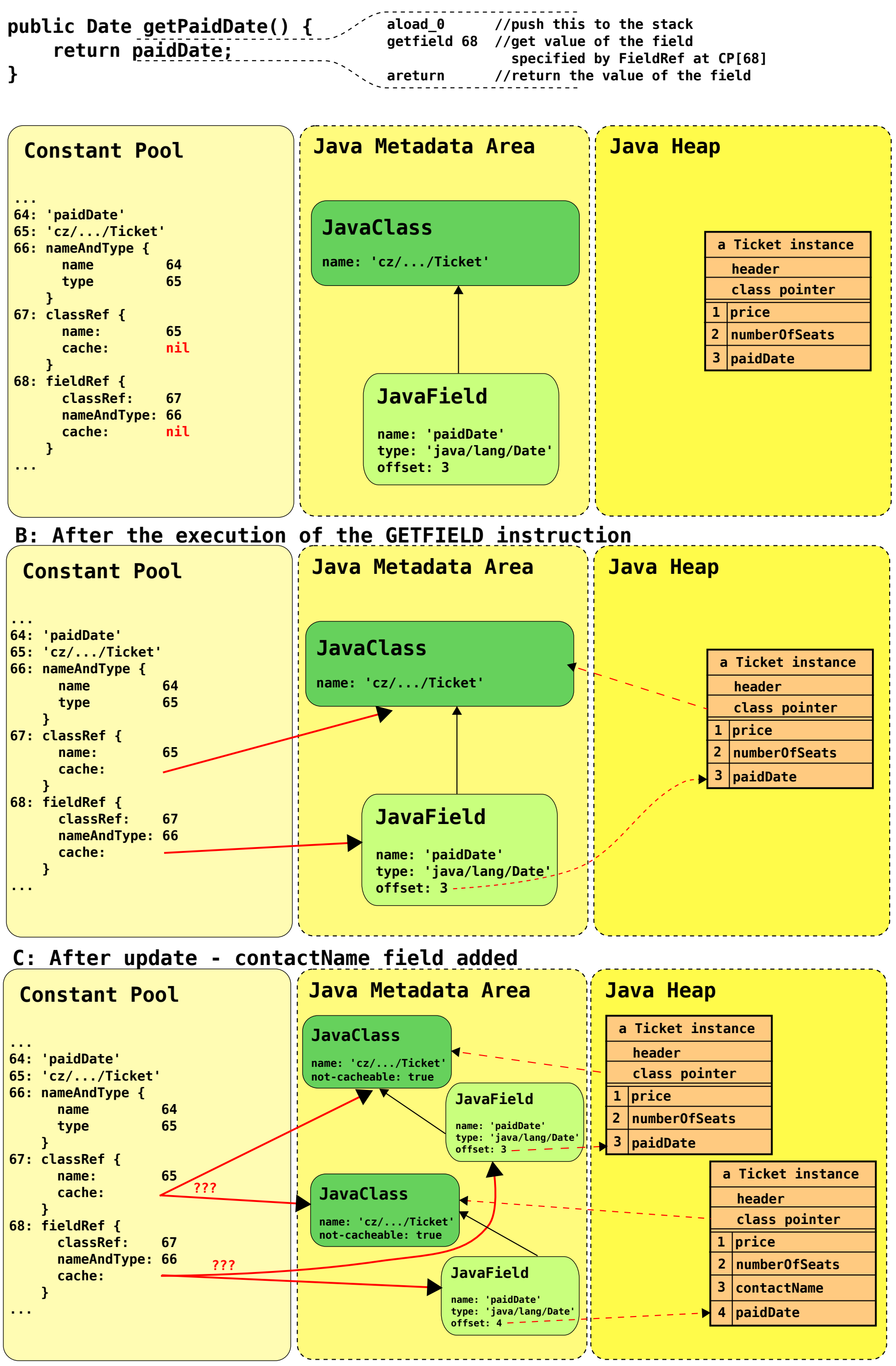
References are not only resolved during linking, they may be resolved also later in the runtime. Consider the method at Listing 1.3. The method contains references to the `String` class and its `format` method (used for string interpolation), but the method is not called during static initialization of the class. These references are therefore not resolved. They will be resolved, when the first invocation of the `getNameAndParams` method occurs. This lazy resolving scheme is used by STX:LIBJAVA (and HotSpot VM, DCE VM, Jikes RVM[11], JVolve and others).

Now consider the state of the VM as shown in Figure 1. In the top left corner the source code of the currently executed method is shown – the `getPaidDate` method of the `Ticket` class. In the top right corner the bytecode of the `getPaidDate` method is shown. 3 sections follow, first showing the state of the VM before the execution of the `GETFIELD` instruction, second showing the state after the `GETFIELD` was executed. The last one will be explained in Section 4.4. In each of sections on the left the constant pool of the `Ticket` class is shown, in the middle the Java Metadata Area of the VM with `Ticket` class and its field `paidDate` are shown. On the right the Java heap is shown, currently containing only one instance of the `Ticket` class. The instance consists of the header containing various fields needed by the runtime, garbage collection, synchronization etc. These are not relevant to our problem. Instance also contains a class pointer, pointing to the Java Metadata Area, where a runtime Java class representation resides. Finally, instance contains a slot for every field it should have. In our case, the `Ticket` class only has 3 fields.

Consider the state of the VM from the Part A at Figure 1. ClassRefs and FieldRefs (and the not shown MethodRefs as well) in the constant pool contain a cache field. Initially, the field is empty, but when the reference is resolved for the first time, cache is filled. Next time a reference needs to be resolved, cached value is returned immediately, without the need to lookup the class, method or field, which greatly improves the performance.

When the execution advances, `GETFIELD` instruction is to be executed. The instance which field is to be loaded is already on the stack, pushed by previous `ALOAD_0` instruction. The reference identifying the field is stored in the constant pool at the index

[11] http://jikesrvm.org/



```
public Date getPaidDate() {
    return paidDate;
}
```

```
aload_0       //push this to the stack
getfield 68   //get value of the field
                specified by FieldRef at CP[68]
areturn       //return the value of the field
```

```
...
64: 'paidDate'
65: 'cz/.../Ticket'
66: nameAndType {
       name          64
       type          65
    }
67: classRef {
       name:         65
       cache:        nil
    }
68: fieldRef {
       classRef:     67
       nameAndType:  66
       cache:        nil
    }
...
```

```
...
64: 'paidDate'
65: 'cz/.../Ticket'
66: nameAndType {
       name          64
       type          65
    }
67: classRef {
       name:         65
       cache:
    }
68: fieldRef {
       classRef:     67
       nameAndType:  66
       cache:
    }
...
```

Fig. 1. State of the STX:LIBJAVA in different resolving situations

68 (68 is given as an argument to the instruction directly in the bytecode). Situation after the processing of the instruction is shown in Part B at Figure 1. A `cache` field of the ClassRef at index 67 points to the runtime representation of the `Ticket` class, `cache` of the FieldRef points to the runtime representation of the `paidDate` field of the `Ticket` class. In our situation, `Ticket` class has 3 fields, and the offset of the `paidDate` field is 3. The `GETFIELD` instruction will retrieve instance data from the heap and will jump to the 3rd slot, pushing the found value to the stack.

4 Implementation

Currently, runtime code update support in STX:LIBJAVA operates at the level of a class. Due to the absence of an incremental compiler for Java, the whole class needs to be recompiled on each update. Incremental compiler is able to compile single method, as opposed to whole Java compilation unit, as is the case of the Standard `javac` compiler – a compiler currently used by STX:LIBJAVA. When update occurs, the difference between the old and new versions of the class is found and the update is handled depending on the type.

In this section we will revisit the types of runtime code updates. For each type we will describe how STX:LIBJAVA supports the update and what were the problems affecting the implementation.

4.1 Update of the Method Body

Updates of the method bodies (described in Section 2.2) are easiest to manage. No existing code is broken and nothing on the heap has to change.

The update could cause changes in the constant pool by *e.g.,* introducing new constants or references in the method body. Therefore constant pool of the new version of the class completely replaces the old constant pool. Also, bytecodes of all methods must be replaced, as the indices into the constant pool may not be valid after the update.

In the case of Figure 1, when `getPaidDate` method changes, the constant pool of the `Ticket` class is replaced, and the bytecodes of all its methods are replaced. The change does not affect the `TicketController` or other classes which depend or use the `Ticket` class.

There may be running invocations of the updated method. These invocations are not modified and are left intact. There are attempts to replace the method as it is running (Kabanov and Vene [7], Subramanian et al. [13]) by analyzing the update and migrating the code of the method in the middle of its execution. However there are situations, when the method cannot be transparently updated (Subramanian et al. [13]). There is a large amount of engineering work involved in such an approach, and the results are not always predictable by the programmer. STX:LIBJAVA therefore does not modify running methods in favor of always accepting method body updates and predictable behavior. Therefore only new invocations are affected by the update, similarly to the behavior of the HotSpot VM (Dmitriev [1]).

4.2 Binary Compatible Update

In STX:LIBJAVA, binary compatible updates (described in Section 2.3) are handled similarly to the update of the method body. Whole constant pool and bytecodes of the methods have to be replaced. In addition, new method must be installed into the class.

Another subtle issue may arise when dealing with overloaded methods. By adding a method, an existing method could be overloaded. Java compiler is responsible to choose best-matching method based on the static types of its arguments. There may be classes in the system, which need to be recompiled and updated in order to correctly choose one of overloaded methods.

By default, STX:LIBJAVA ignores these situations. The behavior can be compared to the situation, when the application is compiled against a certain version of a particular library, and then providing different version at runtime. However, dependent classes can be recompiled when explicitly requested.

4.3 Binary Incompatible Update

The essential problem of this type of update is that it breaks existing code. An application can try to invoke non-existing methods or to store a value to non-existing field. In general, there are 2 solutions: **Allowing the update**, and throwing an exception, when removed method is accessed. This corresponds to the behavior of the HotSpot VM in the situation mentioned in Section 4.2, when a different version of a library is provided in runtime. When a method existing in compile-time but non-existing in run-time is invoked, the `NoSuchMethodError` is thrown. **Detecting the incompatibility** and not allowing the update at all (possibly rolling back other already applied updates). This approach has an advantage that the system is always in compatible state.

In STX:LIBJAVA, these updates are allowed even when they break existing code. When a legacy class tries to invoke a method, which has been removed by the update, an exception is thrown. The exception can be caught and the problem can be fixed by updating the system in runtime, *e.g.,* by implementing missing method or by updating the code calling removed method.

4.4 Update of the Instance Format

When a field is added or removed by the update, the instance format changes. In Section 3.2 we showed, that the instance on the heap contains among other unrelevant fields a single slot per declared field. After the layout changing update the fields declared by the class would be different to what currently living instances contain. Additionally, as there are resolved fields stored in the `cache` field of the FieldRef, the offset stored in resolved field may no longer be valid. Existing classes which have already cached the resolved field may access incorrect or unexisting slot, resulting in heap corruption and in abnormal application termination. Therefore this issue must be addressed by the system. There are 2 solutions – migrating the instances with update or allowing multiple versions of a class to coexist and leave old instances intact.

Updating the class and migrating existing instances is the standard approach taken by other dynamic code evolution projects for Java (Gregersen and Jørgensen [4], Kabanov and Vene [7], Subramanian et al. [13]), also due to the limitations of the HotSpot

VM and Java class loading design decisions (Liang and Bracha [8]). The main problem of this solution is how to ensure that the migrated instances are in the correct state. Solutions such as leaving added or changed fields uninitialized or initialized to the default value are not practical and can cause the instance to be in a state not achievable by normal program execution. This approach should be and usually is accompanied with the ability to specify custom migration logic (Gregersen and Jørgensen [4], Kabanov and Vene [7], Subramanian et al. [13]).

Allowing multiple versions of a single class opens the possibility to leave old instances to live with old class version, and new instances to use new class version. This is the approach taken by STX:LIBJAVA and by DCE VM (Würthinger et al. [14]). When a class is updated in such a way that the instance format changes, the old class version is removed from the Java class registry (whis is used to lookup Java classes in STX:LIBJAVA), but as the class is still referenced by the old instances it is not garbage collected until all instances die naturally. New class version is installed into the registry and all new instances are created using new class version after the installation. Advantage of the approach is that the programmer is not forced to provide migration methods for each update and still all the instances are in the valid state. In case the instances need to be migrated immediatelly with the update, the developer can explicitly provide a custom migration method and all instances will be migrated.

The migration must happen atomically, as there may be Java threads working with affected instances. The standard solution is to wait for all threads to reach and wait in a **safe-point** (Gregersen and Jørgensen [4], Kabanov and Vene [7], Subramanian et al. [13]). Thread is in a safe-point when it can yield its execution – on every method invocation, backward-jump and method return. After the migration, all threads are resumed.

Having more than one class version in runtime brings up an issue. As shown in Figure 1, FieldRefs in the constant pool, once resolved, cache the field so when the reference is accessed next time, the resolving phase can be skipped. When there are multiple versions of a single class, the offset of the field can vary depending on the class version of the current instance at runtime.

To illustrate this problem, consider Part C in Figure 1. After the update, which has added `contactName` field to the `Ticket` class, there are multiple versions of the class, as shown in Java Metadata Area. Both these versions contain `paidDate` field, but in the new version, the offset of the field is 4, on contrary to 3 in old version. To solve this problem, classes, which have multiple coexisting versions, are marked as non-cacheable. When a FieldRef is resolved, the marked classes are detected and then the resolved value is not cached. This way the resolving is performed every time.

While solving the problem, this solution is not perfect. As field access is very common operation, STX:LIBJAVA keeps the track of all old class versions in the *weak array*[12]. Everytime a reference for non-cacheable class is resolved, the weak array is checked. If the array contains only single item, it means there are no old class versions anymore, the class is marked as cacheable and resolved value can be cached again. Therefore the performance is hindered only for a limited time after the change. The concrete performance measurements have not yet been taken and are part of the future work, as mentioned in Section 5.

[12] items in the weak array are free to be collected by the garbage collector

4.5 Update of the Class and Interface Hierarchy

As described in Section 2.5, two issues must be addressed in order to handle updates to the class and interface hierarchy.

In STX:LIBJAVA, we handle these updates similarly to updates of the instance format (Section 4.4), by allowing both old and new versions of the class to coexist. In this case, the type safety of the existing instances is not broken.

Type safety can be broken, when old instances are migrated by the update, and still stored in fields of static type not type compatible with new type information. Gupta et al. [5] states that the consistency problem is undecidable. Therefore in STX:LIBJAVA it is a responsibility of the migration logic provided by the programmer to ensure that the system will be in consistent state. When an update causes an error later in the runtime, `NoSuchFieldError`, `NoSuchMethodError`, or `ClassCastException` are raised.

5 Future Work

As the runtime code update support in STX:LIBJAVA is still under development, it has not been tested in the real world yet. We expect and believe, that it would be possible to evolve a Java project from initial version (*e.g.,* first commit in the source code repository) to the latest version without the need to restart the application. Similarly, there is no performance analysis of the runtime code update support in STX:LIBJAVA done yet.

A working incremental compiler for Java would enable us to implement fully interactive environment for Java where it would be possible to start the application before it is completely finished, and to implement missing pieces in the runtime incrementally. A work on such compiler is therefore also part of our future plans.

Currently, all instances are migrated immediately, blocking the execution of all threads. While this is a solution taken by many (Gregersen and Jørgensen [4], Kabanov and Vene [7], Subramanian et al. [13]), lazy migration of instances would be beneficial for long-running applications. We are currently evaluating a solution where instances would be migrated when they are resolved.

6 Related Work

6.1 JVM HotSwap

JVM Hot Swap is a feature of Java HotSpot VM which enables, albeit limited, dynamic code updates. HotSpot VM is capable of handling changes to method bodies after implementation of the first stage of Dmitrievs (Dmitriev [1]) four-step plan in the Java platform. However, other steps have never been implemented, due to the amount of the engineering effort needed. As elaborated by Dmitriev [1], HotSpot VM has been designed for Java and with performance in mind, which makes runtime code update implementation difficult. On contrary to STX:LIBJAVA, other types of updates are not supported.

6.2 Dynamic Code Evolution VM

Dynamic Code Evolution VM (DCE VM) is a modification of the Java HotSpot VM that allows dynamic class redefinition at runtime. It allows for arbitrary code evolution changes including changes to class and interface hierarchy.

DCE VM suffers from an issue, when the dynamic type of variable does no longer match its static type, a problem described at Section 4.5 STX:LIBJAVA does not suffer from this issue and allows for all types of dynamic code changes without breaking the execution. These situations would end up raising a `NoSuchMethodError`.

6.3 JRebel, Javaleon, Javadaptor and other

There is a group of application level tools which operate on unmodified HotSpot VM, while providing runtime code update support for various types of change. These solutions make heavy use of HotSwap feature together with Java class loading. In general, some kind of proxy service is installed and this service dispatches to concrete class versions. By adding a proxy layer these solution bring certain overhead to the normal execution, even when there are no updates applied. Because proxies

JRebel is commercial tool providing support for dynamic changes including adding/removing methods and fields, but lacks the ability to dynamically change parents hierarchy (Kabanov and Vene [7]).

Javaleon is another commercial product allowing arbitrary changes to the running Java application, including changes to the parent and interface hierarchies, but updates take effect at the granularity level of components, thus Javaleon is usable only when an application is developed on top of a component system, such as NetBeans Platform or Eclipse Rich Client Platform. In order to operate, Javaleon have to preprocess the whole standard library and all 3rd party libraries used by application.

Javadaptor takes a slightly different approach. Instead of loading classes with new class loader, it performs renaming of the classes, therefore allowing them to be loaded by the existing class loader. Similarly to Javaleon and JRebel, they use bytecode manipulation and dynamic proxying in order to achieve runtime code update.

The big advantage of these tools is that they execute on the standard, unmodified HotSpot VM, allowing the runtime code updates to be used without the need to update deployment machines. However, despite large amount of engineering effort put into these projects (Kabanov and Vene [7]), they cannot be compared to customized VM feature and(or) performance-wise.

7 Conclusion

Runtime code update as a technique to update a program while it is running has proven to be valuable feature of a runtime system improving development and debugging speed while lowering the costs and downtime of long-running applications. In this paper, we presented runtime class update support in STX:LIBJAVA, a Java VM implementation for Smalltalk/X VM. To our knowledge, STX:LIBJAVA is the only publicly available virtual machine for Java that supports all types of runtime code updates. Runtime code update support enables the STX:LIBJAVA to become the first fully interactive development environment for Java.

Bibliography

[1] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, pages 14–18. Citeseer, 2001.

[2] P. Ebraert and Y. Vandewoude. Influence of type systems on dynamic software evolution. In *the electronic proceedings of the International Conference on Software Maintenance (ICSM'05) Badapest Hungary*. Citeseer, 2005.

[3] Robert S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, 1976.

[4] A.R. Gregersen and B.N. Jørgensen. Dynamic update of java applications - balancing change flexibility vs programming transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112, 2009.

[5] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22 (2):120–131, 1996.

[6] M. Hlopko, J. Kurš, J. Vranỳ, and C. Gittinger. On the integration of smalltalk and java. *Proceedings of the Intenational Workshop on Smalltalk Technologies (IWST)*, 2012.

[7] J. Kabanov and V. Vene. A thousand years of productivity: the jrebel story. *Software: Practice and Experience*, 2012.

[8] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *SIGPLAN Not.*, 33:36–44, October 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/286942.286945. URL http://doi.acm.org/10.1145/286942.286945.

[9] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, The (2nd Edition)*. Prentice Hall, Santa Clara, California 95054 U.S.A, 2 edition, 4 1999. ISBN 9780201432947. URL http://java.sun.com/docs/books/jvms/.

[10] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658. IEEE, 2002.

[11] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. *ECOOP 2002 - Object-Oriented Programming*, pages 29–53, 2006.

[12] Y.P. Shan. Smalltalk on the rise. *Communications of the ACM*, 38(10):102–104, 1995.

[13] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. *Dynamic Software Updates: A VM-centric Approach*, volume 44. ACM, 2009.

[14] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852764. URL http://doi.acm.org/10.1145/1852761.1852764.