Coping with API Evolution for Running, Mission-Critical Applications Using Virtual Execution Environment [*]

Bashar Gharaibeh, Tien N. Nguyen, and J. Morris Chang
Electrical and Computer Engineering Department
Iowa State University
{bashar,tien,morris}@iastate.edu

Abstract

Recent research showed that the majority of compatibility-breaking changes in a component-based, object-oriented software system are refactorings [5]. The software updating process in such system with the presence of heavily refactored application programming interface (API) is largely manual and disruptive to running, mission-critical applications, which are expected to run continuously without any interruptions. To address the online, API update issue for mission-critical systems, we have developed an online updating framework based on Virtual Execution Environment (VEE) such as Java Virtual Machine. The framework extends the VEE to take the API change log, which records changes to API components, and seamlessly apply proper updates to the running system without human intervention and without shutting down the system. This framework was implemented as an extension to Jikes Research Virtual Machine. Our experimental study showed that our online update tool requires little overhead while maintaining the consistent state of the updated running application.

1 Introduction

In component-based software development, software components and application programming interfaces (APIs) tend to be developed concurrently in different locations with asynchronous lifecycle. New versions of software components often change and so require applications that use the components to be changed. Typically, conflicts among software components are commonly encountered when integrating software components together. Detecting and resolving conflicts by incorporating changes and/or upgrading versions of components are crucial in component-based software maintenance. However, in practice, conflicts are usually resolved in an ad-hoc and manual fashion by all parties involved after it happened. Updating the software in a distributed environment is not trivial [20] because components in such system are interdependent to each other.

In studies on the evolution of component-based software and APIs, several researchers have examined the changes in real-life software systems [5, 23]. Recent research showed that the majority of compatibility-breaking changes between major releases of four popularly used APIs are *refactoring* changes [5]. Refactoring changes are not trivial in nature and many of them break the compatibility in component-based systems. For example, an API provides two classes: B inherits A. Assume that initially, A has a method m. In a client's code, his class C, which inherits from B, contains method calls to the method $m()$ in class A. B and C initially do not contain any method with the name m. Suppose that, in the new version of the API, a new overriding method m is added into class B. In the integration with the new API, all method calls in C would be directed to m in B, rather than to $A.m$. It means that a semantical error occurs. An observation here is that a refactoring operation performs on *static* information (i.e. it is source-to-source transformation), while the *dynamic* method dispatch is decided at the *run-time* level.

Speaking of dynamic method dispatch, we must mention about inheritance and polymorphism as key mechanisms for software reuse through API in object-oriented development. It is apparent that refactorings have made the original intention in polymorphism invalid, thus, creating the incompatibility and semantical errors between client code and the API. This phenomenon occurs only with object-oriented programs in which the API provides not only the functions, but also super classes for inheritance in the client's code. Because the refactoring changes are independently made to API and client code, the semantics of a program including both API and client's code can inadvertently change. In this

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0296131 (ITR) and 0219870 (ITR). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation."

paper, we make the following **contributions**. We present effects of several refactoring operations on the inheritance structure by showing potential semantic conflicts caused by API evolution. Based on their effects, we classify them into two groups: resolvable and unresolvable conflicts. For each resolvable case, we describe a procedure that can be used to resolve the conflicts and to update the running application to temporarily cope with API changes.

The next section shows an example on the effect of API evolution on the application's stability and correctness. Section 3 presents an overview of our approach. Section 4 discusses the effects of several refactoring operations and how to resolve them. The online update algorithm is provided in Section 5. Section 6 describes the implementation details within the Jikes Research Virtual Machine (RVM). Our performance evaluation is in Section 7. Related work is in Section 8, and conclusions are in the last section.

2 Motivation Example

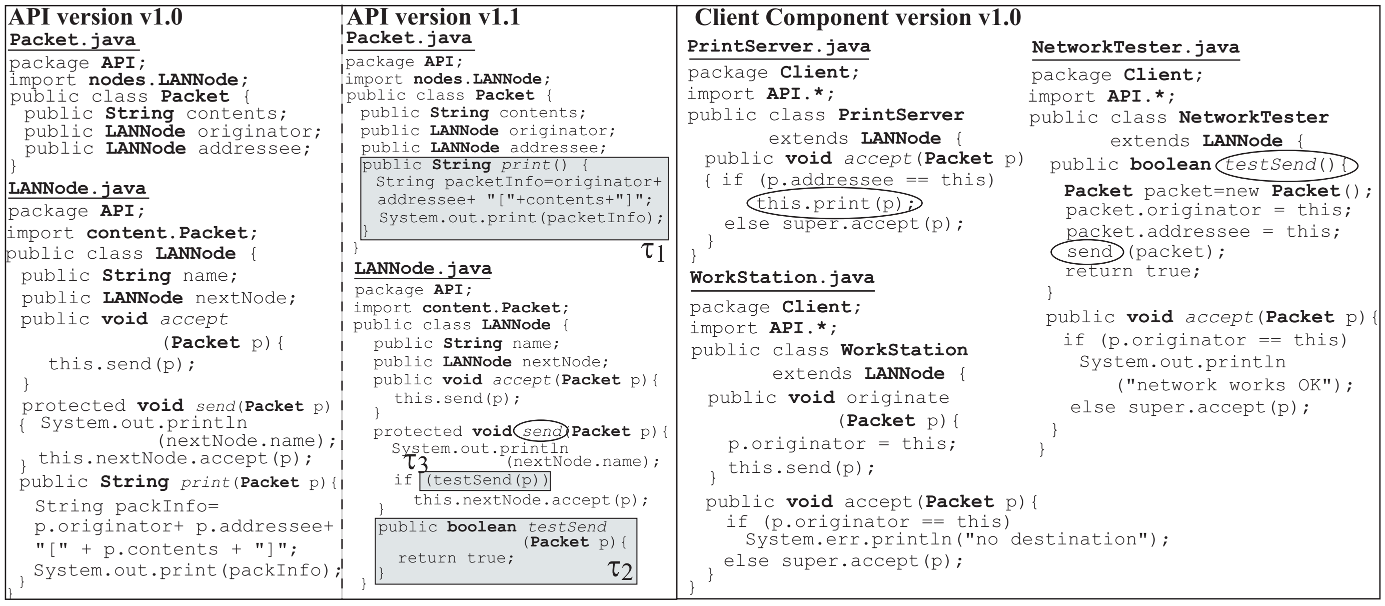
As a motivation example, let us closely examine the following LAN simulator in Figure 1. A developer uses an API package that contains two classes: `LANNode` and `Packet` (Figure 1). The `LANNode` represents a general LAN node behavior. It has two key methods: `send()` and `accept()` which can be used to send and receive packets. Furthermore, the packets are represented by the `Packet` class. The developer builds on the API by creating three classes that represent different types of nodes (Figure 1). The three classes: `Workstation`, `PrintServer` and `NetworkTester` inherit from the general `LANNode` class and use its `send` and `accept` methods. At this initial stage, the client code and API are semantically consistent and are marked with version *v1.0*.

Assume that the third-party API developers decide to extend the functionality of the API and to rewrite some parts for better readability. The changes are highlighted in the figure which shows the second API version (API version 1.1 in Figure 1). The first change (τ_1) moves the responsibility of printing the packet contents from the `LANNode` class to the `Packet` class, while τ_2 adds a method `testSend` to `LANNode` to test the network before sending. Finally, τ_3 makes use of the new testing method in the `send` method.

Let us now consider the effects of API evolution. Let us assume that in the first scenario, the developer has the source code of the new API. After compiling his client code with the new API, he can detect the error in `PrintServer`, which is caused by the move of `print` method from `LANNode` to `Packet`. He can easily recognize that error and change the method call `this.print(p)` into `p.print()`. Now, no other compiling error is reported. However, the program runs into an infinite loop and may crash without apparent reason. After debugging, this unexpected behavior was found to be caused by the added method `testSend` in `LANNode` which have the same signature as a method in the child class `NetworkTester`. Therefore, any call to `LANNode.testSend` is forwarded to `NetworkTester` if the object is of `NetworkTester`. Basically, starting from an object instantiated from `NetworkTester` (e.g. `network-tester`), a call to `network-tester.testSend()` will require a call to `LANNode.send()`. The new `LANNode.send()` contains a call to `testSend()`. However, this call is forwarded to `NetworkTester.testSend()` (rather than `LANNode.testSend`) because it is an object of `NetworkTester`. `NetworkTester.testSend()` in turn calls `LANNode.send()` and so on. This results in an infinite loop. In other words, it is a run-time error.

In both cases, human intervention is required to fix the compiling error and the run-time error. In the latter case of the infinite loop problem, the conflict was not uncovered until the program was executed. Now, let us assume that the source code for the old and the new APIs are *not* available and at some critical moment, the system needs to be updated without being shut down. In those cases, software components require *runtime (online) update*, which implies that software changes need to be temporarily coped while the application continues to run. Online update is especially desirable for mission-critical applications, for example, the flight control software on spacecraft or critical transaction processing servers. Such applications are expected to run continuously without any interruptions. Ultimately, the application developers will need to migrate to the new versions of APIs. Meanwhile, online update is a good compromise with changes. In this case, it is obvious that an automatic process is needed to resolve the conflicts in order to temporarily cope with the API changes.

The conflicts emerge due to compatibility-breaking refactoring changes to the API. Initially, the client code's developers constructed the application based on a certain understanding for the API structure. This understanding is manifested through the inheritance relation between the client and API components. However, when the API changed, the old understanding is no longer valid. Refactoring operations can change the API in such a way that invalidates the developers' understanding. For source code update, they can always intervene and make appropriate changes to fix the conflicts. However, for *automatic* online update, changes to API components should be *shielded* from client's code. The tool needs to seamlessly integrate changes to APIs into the running application if those changes do not affect the client program's behavior. The tool must be able to identify the cases in which conflicts cannot be automatically resolved.



API version v1.0

Packet.java
```
package API;
import nodes.LANNode;
public class Packet {
  public String contents;
  public LANNode originator;
  public LANNode addressee;
}
```
LANNode.java
```
package API;
import content.Packet;
public class LANNode {
  public String name;
  public LANNode nextNode;
  public void accept
              (Packet p){
    this.send(p);
  }
  protected void send(Packet p)
  { System.out.println
              (nextNode.name);
    this.nextNode.accept(p);
  }
  public String print(Packet p){
    String packInfo=
    p.originator+ p.addressee+
    "[" + p.contents + "]";
    System.out.print(packInfo);
  }
}
```

API version v1.1

Packet.java
```
package API;
import nodes.LANNode;
public class Packet {
  public String contents;
  public LANNode originator;
  public LANNode addressee;
  public String print() {
    String packetInfo=originator+
    addressee+ "["+contents+"]";
    System.out.print(packetInfo);
  }
}
```
LANNode.java
```
package API;
import content.Packet;
public class LANNode {
  public String name;
  public LANNode nextNode;
  public void accept(Packet p){
    this.send(p);
  }
  protected void send(Packet p){
    System.out.println
              (nextNode.name);
    if (testSend(p))
      this.nextNode.accept(p);
  }
  public boolean testSend
              (Packet p){
    return true;
  }
}
```

Client Component version v1.0

PrintServer.java
```
package Client;
import API.*;
public class PrintServer
        extends LANNode {
  public void accept(Packet p)
  { if (p.addressee == this)
      this.print(p);
    else super.accept(p);
  }
}
```
WorkStation.java
```
package Client;
import API.*;
public class WorkStation
        extends LANNode {
  public void originate
            (Packet p){
    p.originator = this;
    this.send(p);
  }
  public void accept(Packet p){
    if (p.originator == this)
      System.err.println("no destination");
    else super.accept(p);
  }
}
```
NetworkTester.java
```
package Client;
import API.*;
public class NetworkTester
        extends LANNode {
  public boolean testSend(){
    Packet packet=new Packet();
    packet.originator = this;
    packet.addressee = this;
    send(packet);
    return true;
  }
  public void accept(Packet p){
    if (p.originator == this)
      System.out.println
          ("network works OK");
    else super.accept(p);
  }
}
```

Figure 1. Motivation Example

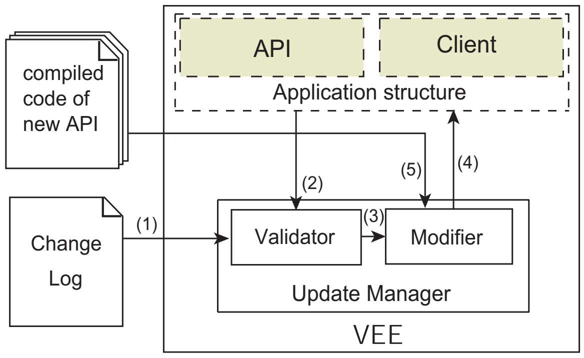


Figure 2. VEE-based, API Online Update

3 Overview of Our Approach

Our framework suggests the use of *change log*, provided by the API developers, recording changes to APIs including refactorings. This technique becomes popular as used in refactoring record-and-replay for source code update [11]. Eclipse from version 3.2 onward has provided this refactoring record feature. In the motivation example, the online update tool should be able to examine the API change log and then update the application while it is running. For example, since the method `print` has moved, the tool could re-direct the method calls to the right place. The method overriding conflict that caused the infinite loop can be resolved by redirecting the call to `testSend` from `LANNode` to its own `testSend` rather than the child's `testSend`.

Since online update modifies running applications, it is very natural to build the online update tool as an extension to Virtual Execution Environment (VEE). Several programming languages/frameworks rely on a VEE to execute, such as Java and Microsoft.Net. In these languages, the application is compiled into a standard intermediate instruction form (e.g. bytecodes) that can be understood by the VEE. The VEE then loads these instructions and executes them on the underlying machine. The VEE uses several data structures to represent the *application structure* (i.e. classes, methods and attributes). These structures are created when the VEE loads the application's binary files.

Figure 2 shows the general layout of our refactoring-aware, API online update framework. Initially, the VEE is presented with the change log that records changes to the API component including refactoring changes (step (1)). In order to effectively prevent unwanted impacts from API changes to running applications, the online update manager module needs to distinguish which classes belong to API or client. This information is also provided in the log file by developers. Later in the update process, the update manager will decide which API changes can be accessible from the client code based on the refactorings in the change log.

The update manager as an VEE extension first validates the refactorings in the change log with respect to the current state of the running application (i.e. step (2)) using our validation algorithm (Section 4). If the refactoring changes are valid, the updater presents the chain of refactorings (i.e. step (3)) to the modifier. If changes are invalid or require human intervention, the online update tool stops. Taking a valid chain, the modifier module (Section 5) applies each operation one-by-one to the running application. The modifier changes the application structure (step (4)), rather than fully loading the new API binaries. It only loads the needed part of the new API, e.g., a newly added method or class (step (5)). For example, for an `AddMethod` operation, the change log file will contain the name of the receiving class, and the method signature. From those information, the update manager will load the compiled code of the newly added method of the API from a predetermined location on disk. While for some other operations, no compiled code is necessary. For example, in the case of `MoveMethod`, we only need the name of the donating class, the receiving class, and the method name recorded in the change log file.

4 Refactoring Change Validation

During the development process, the developers may rely on some API classes through inheritance. In these situations, the client developer may rely on the already provided API methods or may override them. In general, the client component may attach itself to the API inheritance structure in several points. When the application is running, it must be in a consistent state in which correct inheritance relationships are hold between the client and the API code.

API developers continue to develop the API by refactoring it for different reasons, such as to enhance its maintainability or readability. However, as in the motivation example, although the refactorings are intended to be behavior-preserving, they can affect the inheritance structure. The change in the inheritance structure can cause unexpected behavior in the client code. This is due to the changed interface between the client and the API. For example, suppose that some feature in the client's code depends on an overriding method that is called on a certain event. If that method was renamed in a refactoring operation, then the feature is no longer accessible when the expected event occurs.

Inheritance structure in a client component or API component can be either a tree or a forest of trees (assume that there is no multiple inheritance). Figure 3 depicts a sample inheritance structure consisting of one tree T_0. The root of an inheritance tree in a client component can be an extended point of a class (i.e. node) in the API. These extensions may rely on API functionality directly, or may override some methods in the API to provide application-specific functionality. This implies that a change to one of the API entities (i.e. class, method, field, etc) can directly affect the client's classes that rely on the changed entities.

Beside defining the inheritance structure, we also need to define the inheritance conflict. An application is in a stable and a consistent state before refactoring. A *stable* state is achieved if the application can be compiled and executed without errors. While a *consistent* state is achieved if the application behaves as intended by the developer. It is important to distinguish these cases since after refactoring, the application might arrive at a non-stable state, or dangerously, at an inconsistent state. In general, in our framework, an *inheritance conflict* is defined as a deviation from the consistent and stable state due to changes to inheritance structure.

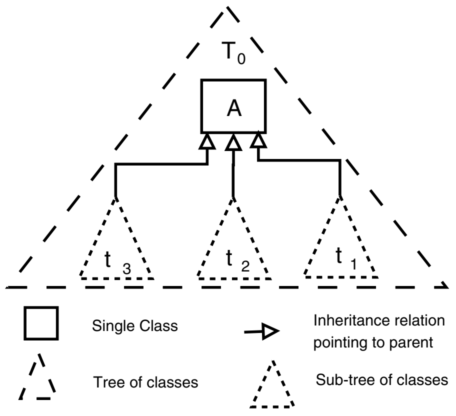


Figure 3. Inheritance Structure

In the rest of this section, we describe how different refactorings can affect the application's inheritance structure. The list of refactoring operation is adopted from a list of most popular refactorings in API evolution as noted in [5]. We classify the conflicts caused by refactorings into two major groups. The first is the set of conflicts that can be automatically solved, while the other is the set that requires human intervention to be resolved. Furthermore, we provide a general procedure to resolve conflicts that belong to the first group. Section 5 will illustrate how these procedures can be easily implemented within a VEE.

4.1 Resolvable Refactoring Effects

Refactorings can affect the inheritance structure and transfer the application into an inconsistent state, in which the application behaves in a different way than initially intended by the developers. Resolving these conflicts requires knowledge about the developers' intentions, which may be hard to determine in the context of an automatic tool. However, we will show that by examining refactoring change log, we can detect and resolve a certain set of inheritance conflicts. Let us discuss the effects of resolvable conflicts in the context of the following operations to the API.

4.1.1 Method Addition

Assume that a method is added into a super class in the API. This refactoring operation may cause a conflict if the new method signature exists in a subclass in the client code. This overriding causes the dynamic dispatch mechanism to forward method calls intended for the added method to the (accidentally) overriding method. However, the developer for client code did not intend for methods in the children classes to override the new method, since it did not exist before. Therefore, the application may have a different behavior after the refactoring operation.

To resolve this conflict, calls to the new method should forwarded to it specifically rather than its overrides in the client component. Since the method is newly added, calls to that method will be also recorded in the change log as a part of the refactoring sequence. In this case, we defer the conflict resolution until we encounter a “Add method call” or a “Replace method call” operation.

4.1.2 Method Deletion

The deletion of a method usually indicates that it was a *deprecated* method in the old API, and that the new API is no longer maintaining the method. However, the client code may contain calls or still override the deleted method. After the deletion, the application is no longer in a correct state, since it contains references to a non-existing entity (i.e. the deleted method). Although this is an expected side effect for using deprecated methods, the conflict can be resolved by ignoring the deletion operation. This keeps the original interface between the client and API, and will maintain the application in the previous correct state. Eventually, when the application can be shut down, it will need to be updated.

4.1.3 Method Moving

In the API, a method can be moved in different “directions” in the inheritance structure. It can be moved up to the super class (e.g. from t_2 to class A in Figure 3), or down to a subclass (e.g. from class A to t_1). It can also be moved to another tree in the inheritance structure (e.g. from t_1 to a class in t_3). Each of these cases may present different conflicts and require different resolution strategies.

Moving a method up has similar effects to method addition. The moved method will possibly become a “parent” to other methods. For example, in Figure 3 a method moved from a class in t_2 to class A might have new children in t_1 and t_3 in addition to its original children in t_2. Therefore, this case presents similar conflicts as in the case of method addition and can be resolved using the same strategy.

Moving a method down has similar effects to method deletion. Although in contrast, the moved method is still available, where in method deletion, the method is intended to be removed from the API. The moved method will not be visible from some parts of the inheritance structure. For example, moving a method down from class A to a sub class in t_2 will make it invisible for classes in t_1 and t_3. The conflict can be resolved by keeping a copy in the old class. This will preserve the initial inheritance structure of the client component without affecting the API component. Note that calls to the old method within the API are already replaced by the API developers to handle the moved method.

Moving a method to a class in another tree is one of the most recurring refactoring operation. The side effect for this operation is the combined effects of both method addition and method deletion, since the moved method can be considered to be deleted from the old class and added to the new one. Therefore, to resolve this conflict, calls to the new method should be forwarded to that method explicitly. This will ensure that the dispatch mechanism will not forward the calls to the accidental overriders. Furthermore, the method name should remain in the old class, so the methods that override it will be accessible from the client code.

4.1.4 Method Rename

Renaming a method in the API has two side effects on the inheritance structure. Firstly, the method with the old name may be overridden by child classes. By removing the old name, the overriding method is no longer visible from the parent. Therefore, a call to the method with the new name will not be forwarded to the original overriding methods of the child classes. The second effect is that after the name change, the method may accidentally be overridden by some methods of the child classes. This case happens if the new name already exists in the child classes, and because of dynamic dispatch, calls to the new name are forwarded to those methods in the subclasses. VEE can easily recognize these cases and direct them to the right method.

4.1.5 Method Call Replacing

This operation and its special case of adding a method call do not directly affect the inheritance structure. However, it can *reveal* conflicts caused by a method addition or rename at run time. For example, a newly added method can be accidentally overridden by a child class (as explained in the Method Addition section). The added call to the new method will be forwarded at runtime to the child method. Therefore, the only case that we need to consider is when a call is made to a new method (either via addition, renaming, or moving).

This problem can be resolved by forwarding the new call to the target method specifically in the cases that the target is an added or renamed method. In other words, the online updater recognizes this case and the new method is called, instead of its overrider. This strategy is based on the fact that the new call was intended for the new method or any method that “intentionally” overrides it (i.e. API method), rather than to its “accidental” overrider (i.e. client method).

4.1.6 Field Refactoring

We note that refactorings that change the API's `private` fields do not affect the client code in general. Therefore, we need to resolve only refactorings that change *publicly* accessible fields, i.e., to consider the cases where the refactored fields have public access. The first refactoring operation is to move a field. This operation is usually associated with moving multiple static fields in classes to a central place (e.g. a single class). The moved field is no longer accessible to child classes, especially, if a child uses its methods to access the field rather than the methods in the parent class. To keep the application in a stable state, all references to the moved field need to be replaced by either a call to the parent accessor, or to the new field location.

Fields can also be renamed. The new name will cause a conflict if the same name was used for another field in a child class. To resolve the conflict, one can either delegate the problem to the developers later, where they can rename the affected child fields. For temporarily coping with the change, our tool automatically performs field rename refactoring at run-time on the matching field in the client's code.

4.1.7 Class Moving and Renaming

Performing refactoring operations on a class such as moving a class or renaming a class will have effects on the methods or fields in that class, thus changing the visibility of those entities to the child classes in the client's code. The effects of a class moving operation on its fields and methods are equivalent to the effects of a series of field and method moving operations. Therefore, they can be resolved using the same strategies discussed for field and method moving. Note that renaming a class does not affect the inheritance structure of fields and methods, since the inheritance relations are maintained. Therefore, to accommodate the new class name, we only need to refactor the old name usages in the client's component by replacing it with the new name.

4.1.8 Method Object Refactoring

This operation renames a method intended to be overridden, and uses the old name as a factory method. The old method now returns an object, and that object can be used to invoke the new method name. For example, an API contains a method m_a where client code should override this method. This refactoring does the following: m_a is declared as final, so all overriding will be invalid. The old method's body is moved to another method m_b. The method m_a now returns an object of a special class. A special method in the returned object is used to invoke m_b.

The effect of this operation can be resolved by renaming the method that overrides the old name to the new name (i.e. from m_a to m_b) in child classes. Also, calls to the old method in child classes are modified to make use of the returned object from m_a. If the new name already exists in one of the child classes, then the method can not be renamed in that class. To resolve this problem, we have two strategies: we can rename the method invoked by the returned object (e.g. from m_b to m_c) and propagate the renaming to m_b's overriders. We could also rename the method with the new name (i.e. m_b to m_c) in the child class and replace all calls to it in the client component.

4.2 Unresolvable Refactoring Effects

4.2.1 Method Signature Change

In this section, we present refactoring changes that require human intervention to resolve them. The first one is "Method Signature Change". After this operation, a method in the client code that calls the affected method will no longer be valid, since the old method does not exist. Automatically changing call sites requires knowledge about how to map the new method arguments and return type to the old method signature, thus, requiring human intervention.

There is a special case where the change can be resolved. That is when the argument new type is the sub-type or super-type of the old argument type. For example, if the argument type changes from a primitive type to its wrapper class or visa versa, then the call sites can be changed to pass the new type. This is feasible because the mapping from the old type to the new type is known. However, for other cases this mapping can not be inferred so easily.

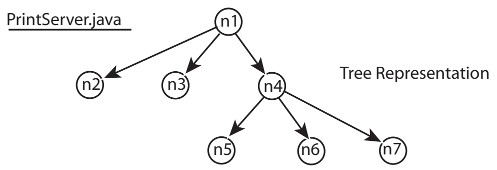
4.2.2 New Hook Method

A hook method is an abstract method in a parent class where all children are required to override this abstract method. Failing to do so is considered as a compile error. The hook method is intended to serve as a link between the API and client code. This hook can be called on certain events. Overriding this method provides the client with the chance to respond to the events. If a new hook method was added to the API, the classes in the client code inherited from the API need to implement that abstract method. A naive solution would consist of adding empty methods to provide the needed implementation. However, this solution may not preserve behavior correctness, since the call may indicate an event that needs special response. Therefore, developers are required to supply the hook method implementation.

5 Refactoring-aware, Online API Update Algorithm

5.1 Attribute Table

The Previous section describes individual cases of refactorings. This section describes our algorithm used in the



Attribute Table

	"NodeType"	"Name"	"parent"	"children"	"SuperType"	...
n1	CompiUnit	"PrintServer"	null	[n2,n3,n4]	undef	
n2	PackageDecl	"nodes"	n1	null	undef	
n3	ImportDecl	"content.Pac..."	n1	null	undef	
n4	TypeDecl	"PrintServer"	n1	[n5,n6,n7]	LANNode	
n5	MethodDecl	"print"	n4	null	undef	
n6	MethodDecl	"getPacketIn.."	n4	null	undef	
n7	MethodDecl	"accept"	n4	null	undef	

Figure 4. Attribute table for `PrintServer`

```
INPUT: An ordered set of refactor operations R
Algorithm: Refactoring-aware Online Updating
    while(R ≠ $\phi$) do
        $(r, c)$ = nextOp($R$);
        $changes$=apply($r, c$);
        $childs$= children($c$);
        for each $c' \in childs$ do
            if (isAPI($c'$))
                applyTab($changes, c'$);
                $childs$=$childs$ ∪ children($c'$);
        end
        Remove r from R
    end
end
```

Figure 5. Refactoring-aware Online Update

"modifier" module of our update tool (Figure 2). The algorithm deals with a valid *chain* of refactorings as its input.

To represent a program in run-time, we use a data structure called *attribute table*. The tree structure of a program is represented as follows. Each row corresponds to a node with a unique ID, representing for an abstract syntax tree (AST) node. Each column corresponds to an attribute, representing for a property associated with each node. Tree-structure information is encoded in "parent" and "children" attributes. The "parent" attribute maps a node to its parent node. The "children" attribute maps a node to a *sequence* of its children nodes. Cells of the attribute table can be in any data type, possibly a reference to a node, or a collection of nodes. For example, a method is represented by a node associated with a "NodeType" slot containing "MethodDecl" (see Figure 4). The "NodeType" slots are enumeration values of predefined AST node types. Furthermore, depending on the type of the AST node, the corresponding node has additional attributes modeling different semantic properties of the AST node. If an attribute is not applicable to a node, an *undef* value is used for the corresponding slot. If a node does not have a child, its children slot contains a *null* value.

5.2 Algorithm

Figure 5 shows the pseudo-code of our refactoring-aware, online API update algorithm. The algorithm uses the following functions: 1) **nextOp**(R) returns the first item in the chain R, 2) **apply(r,c)** applies a refactoring r on the class c and returns the changes to the attribute table, 3) **children(c)** returns a list of classes that inherit from c, 4) **isAPI(c)** returns true if c belongs to API, false otherwise, and 5) **applyTab(e,c)** applies the table changes e to class c.

For each operation r that targets a class c, the algorithm first calls `apply(r,c)`, which applies the refactoring operation to the class and change the attribute table accordingly. The changes to the attribute table are returned. Since the class c may have children, the changes should be propagated to them. For example, if a method was added to c, then all of c's children should have access to that method, which requires the algorithm to go through all children and descendants and modify the attribute table. The algorithm takes the children returned by `children(`c`)` and applies the required changes to each child c' by calling `applyTab(`e, c'`)`. The changes are propagated to all children except those that belong to a different component (i.e client classes should not be affected by API changes). The process will be repeated until all operations are processed.

The algorithm applies the changes to the currently loaded application rather than reloading the new version. By doing so, the algorithm keeps the running application in a stable state. Reloading the new application binaries instead of applying individual change can cause inconsistencies. For example, a class A can be loaded that requires a newly added method to class B. In this scenario, the application can reach a stage where class A is loaded while class B is not, causing the class A to complain about the undefined method in B. Also, the algorithm can keep the application in a consistent state. Because refactoring operations in a valid chain ensure their ordering dependence, a feature is always defined in the application before its use.

6 Tool Implementation with Jikes RVM

We have implemented this framework using Jikes RVM as the VEE. The update manager module was implemented as a thread. The updater thread is initially idle. However, upon user request, the thread wakes up and starts processing the change log file.

After the application binary files (i.e. Java bytecodes) are loaded by JikesRVM, the information about classes is stored in a special area within the virtual machine. These

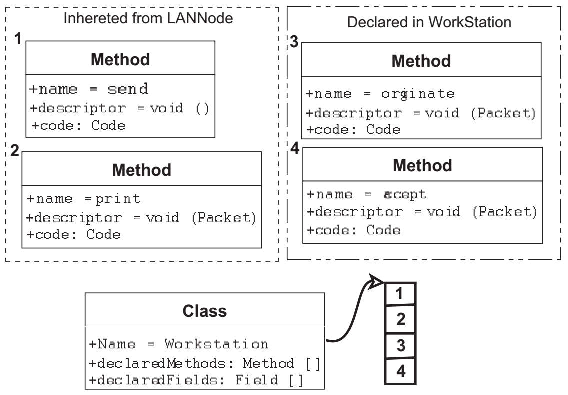


Figure 6. Internal Structure for a Class in JVM

collected information will be used throughout the application execution. Figure 6 shows a typical representation for a class. Each class representation structure holds an array that describes the class's fields and methods, where each array element points to the object representing the fields or the methods. Furthermore, each instantiated object has a pointer to its class representation. The code for methods is stored within the virtual machine in two format: in bytecodes and in native machine code. Bytecodes are loaded from the class files. The translation from bytecodes to native code is done by the Just In Time compiler (JIT).

The rest of this section describes how a running application is updated with JVM. Due to space limitation, we show only the details of the update involving four following refactorings.

Method Addition The new method bytecodes are read from a class file that resides in a predetermined place on disk. The new method may use constants, refer to fields or invoke methods, where these entities are stored in the constants' pool of the donating class (the class that contains the new method body). However, the constants' pool entries may not match the entries of the receiving class. Therefore, the modifier's first task is to change the method bytecodes to accommodate the differences between the constants' pools.

Each reference to the constants' pool in the new method is replaced with its equivalent in the receiving class constants' pool. If no matching entry is found, the modifier will create the needed entry in the receiving class constants' pool. Now the method can use the receiving class constants. The modifier adds the method to the list of declared methods and forwards the method to JIT to be compiled into the native code. The modifier will also traverse the inheritance tree of the receiving class to add the method to the declared method list of the child classes of the receiving class if the children belong to the same component as the parent.

Method Deletion By deleting the method entry from the class's method table, the deleted method is no longer visible to the application. Although the method body may still reside in memory. However, from the application point of view, the method does not exist since there is no reference to it in the method table. The only restriction for this operation is that the method can not be fully deleted if it is in the current call stack (invoked and waiting for a return), since the application will return to execute the remaining of the deleted method later on. When the method does not exist on the call stack, the JVM can safely remove it from the memory, since all references were deleted by the modifier.

Class Rename Within JVM, the class name is an attribute of that class. As seen in Figure 6, the `Class` entity holds a name attribute. Furthermore, JVM holds the mapping between classes and their names to facilitate searching. The name attribute and mapping can be changed to reflect the new class name. In addition, JVM accesses classes by using references to their entities. Therefore, the class name is not usually used. However, there is an exception: when a class is dynamically loaded, JVM needs to resolve its references to classes, methods, and fields. The loaded class refers to these entities by name, which requires JVM to search for the class associated with the requested name. Thus, the new name is added by the online updater to the list of name-to-class mappings to resolve newly loaded classes.

Replacing a Method Call This operation is implemented by deleting the old method body containing the call and replacing it with a method body containing the new call. The new method body is taken from the new version of the API.

After reading the modified method and replacing its constant references as discussed in AddMethod, the modifier instructs the JVM to compile the method into native code. The old method body's native code is still linked to the method entry in the class structure. However, instead of deleting the old method completely and then adding the modified method, we only need to replace the machine code of the target method with the new version. Now, any call to that method will use the new version. Similar to the case of method deletion, the old code will remain in memory until all references to it from the call stack are removed (by `return` statements).

7 Performance Evaluation

This section presents two performance evaluation studies for our refactoring-aware, online API update tool. In the first experiment, we measured the minimum overhead of the use of our online updater in the extended JVM versus without the updater in a regular JVM. This experiment reflects

Table 1. Overhead in extended JVM

Application	Exec time
compress (An LZW implementation)	1.01
jess (NASA's CLIPS-based Expert Shell System)	1.0
db (Memory Resident Database)	1.02
javac (Java Compiler)	1.01
mpegaudio (MPEG compression)	1.01
Mtrt (Multi-threaded Scene Renderer)	1.04
jack (Parser generator)	1.03
Geometric Mean	1.016

the cost of using the extended JVM in normal situations, where no update is required. We executed several applications using the extended JVM and using the regular JVM, then compared their execution time. The extended JVM was not presented with a change log file. Therefore, the difference in execution time reflects the overhead of maintaining the additional thread. For this experiment, we used SPECJVM98, which represents a variety of applications. These applications are described in Table 1.

The average execution time for each application was calculated by taking the average of eight executions without considering the minimum and maximum execution time. This procedure was conducted for both JVMs (i.e. the unmodified and the extended JVM). Table 1 shows the normalized execution time for the extended JVM compared to the regular JVM. Results higher than one means that the extended JVM is slower than the regular JVM. It is noticed that in general, the online updater increases the execution time compared to the regular JVM. The highest increase corresponds to `Mtrt`. This is due to the fact that `Mtrt` is a multi-threaded application while others are single-threaded. Since the online updater is implemented as an extra thread within the JVM, applications that use multi-threads will suffer from higher performance loss.

The second experiment shows the overhead of the actual update process. We analyzed the extended JVM performance while updating an application. The target application used is the motivation example in Figure 1. The application creates a network composed of ten nodes and connects them in a token ring. A packet sent through the network will visit all nodes and return to its sender. This is repeated for a sufficiently large number of loops.

Figure 7 shows the average time needed to complete a loop (in milliseconds). Before l_1, the application uses API v1.0. After l_1 the extended VEE notices the update request and starts updating the API. Meanwhile, the average loop time increases by almost 100%. The online update finishes at l_2 and the average loop time decreases. The increase in average loop time after l_2 compared to loop time before l_1 is due to the new API behavior. The new API

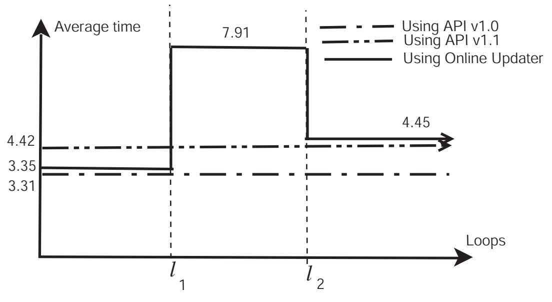


Figure 7. Performance Evaluation

added a method `testSend` and this method was called inside `LANNode.send`. This extra method invocation adds an overhead to the application using API v1.1 compared to using API v1.0. The dashed lines in Figure 7 depict the average loop times when the application is executed on the regular JVM. It is noted that applications executed using the extended JVM have a higher average loop time by 1%. Also, it is worthy to note that runs that use the same API version have comparable average loop time.

8 Related Work

Many tools were proposed to assist the developers in the update task. They can be divided into two categories: offline updating tools [6] and online updating tools [2, 13, 19, 21, 9, 4]. Offline updating tools rely on changing the application files, either in source code or binary form.

Online updating tools apply the changes to the running application, which allow the continuation of the execution. Online software update adds additional complexity to traditional software maintenance because in addition to updating the code, the program's state must also be updated to the new version. The two main methods of online software updating are *state transferring* and *state transformation*.

State transferring is performed between processes by serializing and transferring the program state. Examples of state transferring include [16, 10, 3, 14]. State transformation is the process of morphing the old program into the new version. Examples of state transformation include [1, 17, 13, 12, 7, 18, 15, 8, 4]. The motivation for state transformation is that program modifications can be done efficiently without having to update the entire program state. Another advantage of state transformation is its ability to perform without requiring special hardware or operating system. Ginseng [18] is a dynamic software update tool for C programs. In Ginseng, programs are compiled specially so that they can be dynamically patched, and generate most of a dynamic patch automatically. Our framework does not require a special compiler, but is limited to refactoring-aware API update. Proteus [22], a calculus for software update modeling, permits a program's type structure to change dynamically but guarantees that the updated program remains type-correct. BARBER [24] is a binary refactoring tool for Java. It refactors the application's binary files according to a pre-built catalogue of performance enhancement operations such as class splitting or merging.

Changing the code of an executing program is only one part. Another important part is to transform the program state. Previous research on program state updates includes [17, 12, 7, 8, 15]. The types of changes considered by previous research are adding, removing, and updating classes. There are two categories, those that allow class interfaces to be modified [12, 7, 8, 15], and those that do not [1, 17]. Previous mechanisms for state transformations use variations of proxies [1, 17, 13], one-to-one state transformation functions [12, 15] or use version adapters [7].

9 Conclusions

In current practice, the software updating process in a component-based software system with the presence of heavily refactored API code is still largely manual and disruptive to running, mission-critical applications. To address the online, API update issue, we have developed an online updating framework based on VEE. Our refactoring-aware, online API update tool, as an extension of Jikes RVM, takes the API change log, and seamlessly applies proper updates to the running system without human intervention.

Our key contributions include 1) a VEE-based, refactoring-aware, online update framework and tool that uses Jikes RVM, 2) detailed study and analysis of effects and potential conflicts caused by different types of refactorings on the inheritance structure of a client's application, and 3) a JVM-based resolution scheme and procedure for resolvable cases. In this paper, we showed that by simply examining refactoring change log, we can detect and resolve a large set of inheritance conflicts caused by API evolution. Thus, our VEE-based, online update framework is a good solution for mission-critical applications to temporarily cope with API changes. Our experimental study showed that our tool requires little overhead while maintaining the consistent state of the updated running application.

References

[1] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *USENIX ATC*, 2005.

[2] R. Bialek. *Dynamic Updates of Existing Java Applications*. Ph.d. thesis, Univ. of Copenhagen, Denmark, June 2006.

[3] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18:87–103, 2006.

[6] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. ACM Press, 2007.

[7] D. Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[8] D. Duggan and Z. Wu. Adaptable objects for dynamic updating of software libraries. In *Workshop on Unanticipated Software Evolution*, 2002.

[9] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems and Software*, 14(2):111–128, 1991.

[10] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[11] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proceedings of International Conference on Software Engineering*, 2005.

[12] M. Hicks. *Dynamic Software Updating*. PhD thesis, Dept. of Computer Science, University of Pennsylvania, 2001.

[13] G. Hjalmtysson and R. Gray. Dynamic c++ classes, a lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, number 98, 1998.

[14] C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, University of Maryland, 1993.

[15] I. Lee. *Dymos: A Dynamic Modification System*. PhD thesis, University of Wisconsin, Madison, 1983.

[16] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *ASPLOS*, October 2004.

[17] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP'00*, 2000.

[18] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *PLDI'06: ACM Conf. on Programming Language Design and Implementation*, 2006.

[19] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software, 2002.

[20] L. S. Sameer Ajmani, Barbara Liskov. Modular Software Upgrades for Distributed Systems. In *European Conference on OO Programming*, pages 452–476, 2006.

[21] J. Stanek, S. Kothari, T. N. Nguyen, and C. Cruz-Neira. Online Software Maintenance for Mission-Critical Systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006)*, pages 93–102, 2006.

[22] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe, flexible dynamic software updating. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 2006.

[23] E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *Proceedings of the 22st IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 2006.

[24] E. Tilevich and Y. Smaragdakis. Binary refactoring: improving code behind the scenes. In *Proceedings of the 27th international conference on Software engineering*, 2005.